一种静态分析 C++ 异常处理的方法*)

姜淑娟[1,2] 徐宝文[1] 史 亮[1] 肖 洋[2]

(东南大学计算机科学与工程系 南京 210096)[1] (中国矿业大学计算机科学与技术学院 徐州 221008)[2]

摘 要 异常处理是现代程序设计语言提供的用来提高软件健壮性的一种机制。由于在 C++ 的函数界面中并不要求声明该函数所能传播出的异常的类型,所以要想提高系统的健壮性,必须清楚在程序的执行过程中可能引发的异常、异常的传播路径等。然而在大型系统中,要想确定这些信息是非常困难的。本文针对 C++ 的异常处理机制,首先提出了一个描述 C++ 异常结构信息的模型,并把该模型应用于递归函数中。然后,描述了一个基于该模型的分析 C++ 程序异常结构信息的工具 CETool。该工具能提供所有显式引发异常的有关信息,为系统中异常处理结构的改进和程序的结构测试提供有价值的信息。最后给出了该工具的实现方法和应用实例。

关键词 异常处理,静态分析,异常传播,结构测试

An Approach to Static Analysis of Exception Handling in C++

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

(Department of Computer Science & Engineering, Southeast University, Nanjing 210096)[1]

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

Abstract Exception handling in modern programming languages is a mechanism that can improve software robustness. Since the signature of an C++ function may not specify the set of exceptions that the function can propagate, it is necessary to figure out the exceptions that may be raised during executing program, the origins of the exceptions and their exception propagation paths. Unfortunately, in large programs, this exceptional information can be difficult, if not impossible, to determine. Firstly, the paper presents a model that can describe the exception handling information for C++ exception handling mechanism, and applies the model to recursive functions. Then describes CETool, a static analysis tool we have developed to provide exception-flow information for C++ systems based on this model. The CETool provides the information related to the explicit exceptions. The information is helpful to support the improvements to the exception handling structure of a system and structure testing of a program. It also presents the implementation of CETool and its application.

Keywords Exception handling, Static analysis, Exception propagation, Structure testing

1 引言

随着软件规模的不断扩大、程序复杂度的提高,软件的健壮性越来越被人们所重视。否则,一个很小的异常便极有可能造成整个系统的崩溃。为了避免异常或错误的出现,软件开发人员不得不编制大量代码。在一些软件系统中大约有超过三分之二的程序代码用于系统的异常处理[1]。自从 John Goodenough1975 年首次提出了异常处理的概念以来[2],现有的高级语言大都提供了专门结构来进行异常处理。如果异常处理机制使用得当,确实能提高软件的健壮性[6]。要想提高软件的健壮性,就要清楚在程序的执行过程中可能引发哪些异常?异常控制流的走向如何?异常如何传播?然而,在大型系统中,要想确定这些信息是非常困难的。因为在 C++ 的函数界面中并不一定要声明该函数所能传播出的异常的类型,一个函数可能并没有准备处理在调用树中低于它几层的调用函数中传播出的异常。所以,计算一个函数所能引发和传播出的异常就是一个必不可少的任务。

本文首先针对 C++ 的异常机制建立了异常结构模型;然后在此基础上设计并开发了一个静态分析工具 CETool,该工具可以提供 C++ 程序中显式引发的异常的类型、引发异常的位置、异常的传播路径以及处理程序中的异常类型与它所捕获的异常类型的关系等信息,并用图形化的方式显示给用户,为异常处理结构的改进和程序的结构测试提供了依据。

2 C++ 中异常处理机制

本节简单地介绍 C++ 的异常处理机制,详细的内容请参阅文[4]。

C++ 的异常处理结构如下:

```
try {……
    throw assignment-expressionopt;
}
catch( exception-declaration ){…处理代码 1…}
catch( exception-declaration ){…处理代码 2…}
catch(…){…默认的处理代码…}
```

*)本文得到国家自然科学基金(60373066)、国家 973 重大基础研究(2002CB312000)、教育部跨世纪杰出人才基金、教育部博士基金(20020286004)、江苏省计算机信息处理技术重点实验室(苏州大学)基金资助。**姜淑娟** 副教授、在职博士生,主要从事程序设计语言、软件工程等方面的教学与研究工作;**徐宝文** 教授、博士生导师,主要从事程序设计语言、软件工程、并行程序设计等方面的教学与科研工作;**史 亮** 博士生,研究方向为程序设计语言、软件测试;**肖 洋** 硕士生,研究方向为程序设计语言、编译技术。

如果程序引发了一个异常，则正常的执行被终止，控制转移到异常处理机制。异常通过隐式引发(在默认情况下，只有C++语言预定义的异常才能隐式引发)或通过执行 throw 语句显式引发。异常处理代码紧跟在 try 块后的 catch 语句之后，try 块与 catch 块之间不可插入任何的语句，try 块可以嵌套。如果在与 try 块相关联的 catch 子句中找不到相匹配的处理程序，则异常开始向外层的 try 块传播，并进一步沿函数的调用链反向传播。

还有一种不常见的异常的传播，重新抛出 function-try-block[3]。把 try/catch 处理代码放在函数中初始化列表的周围，用来捕获由基类或成员变量构造函数抛出的任何异常。在构造函数和析构函数中，如果执行到达了处理程序的末尾，在 function-try-block 的处理程序中捕获的异常必须重新抛出。

异常传播的路径主要取决于 try 块的嵌套结构和程序动态调用的顺序。当一个异常传播出一个 try 块时，则到与上一层 try 块相关联的 catch 块中继续寻找相应的异常处理代码。函数内的异常传播到上一级的调用函数，异常沿函数调用链反向传播。如果该函数调用在 try 块之内，则在与 try 块相关联的 catch 块中继续寻找相应的异常处理代码；如果该函数调用不在 try 块之内，则再继续向上一层的调用函数传播。如果异常传播到主程序还没有被处理的话，则调用 C++ 标准库中定义的函数 terminate()。terminate()的缺省行为是调用 abort()，使程序非正常结束。但 terminate()的行为是可以定制的，用户可根据需要来设计它的行为。

3 异常处理分析模型

我们对 C++ 的异常结构的分析主要从宽度和深度两个方面来考虑。

从宽度方面主要解决如下问题：一个函数所引发的异常类型；异常处理程序所能捕获到的异常类型；传播出该函数的异常类型；不可达的异常处理程序等。

从深度方面主要分析异常的传播路径。

3.1 异常处理分析模型

为了解决上面提出的问题，我们对 C++ 的异常处理结构进行建模。该模型借鉴了 Schaefer [11] 关于 Ada 语言、Robillard[15]关于 Java 语言的模型，并对其进行了改进。下面给出相关的定义。

定义 3.1 一个范围 s (scope)是一个复合语句，该复合语句内至少定义了一个 try 块，即

$s=(I, G)$，I 是语句的集合，G 是 try 块的集合。

它表示了异常处理结构的一个单元，这个单元可以是一个程序、一个函数或一个类。

定义 3.2 一个保护范围 g (guarded scope)由一个 try 块 t_1 和与该 try 块关联的一组 catch 子句的有序表 C 组成，即 $g=(t_1, C)$，$c\in C$ 是一个 catch 子句。

由于 try 块可以嵌套，所以 try 块内部又是一个范围，它也由一组语句和一组直接定义在里面的保护范围组成。

由于 C++ 的异常处理模型是终止异常模型，在处理程序中引发的异常不能传播回到保护块，而是传播到上层嵌套范围内继续寻找匹配的处理程序。所以处理程序中的语句属于上层范围内的语句，即与保护范围 g 关联的处理程序中的语句属于语句集合 I，而不属于 t_1。

定义 3.3 函数 encounters(s)→E：在一个范围 s 内遇到的异常类型的集合。

根据定义 3.1，由范围 s 内的语句集合 I 在执行过程中所引发的异常的类型和保护范围 G 传播出的异常类型组成，可表示为：

$$\text{encounters}(s)\equiv \text{generates}(I)\cup \text{raises}(I)\cup \text{propagates}(I)\cup \text{propagates}(G)$$

下面来分别定义这几个函数。

定义 3.4 函数 generates(i)→E：语句 i 执行过程中隐式引发的异常类型的集合，如零作除数。

对于在一个范围内的任何语句 $i\in I$，generates(i)集合是在 C++ 实时环境中所引发的异常的类型，它的定义依赖于 C++ 语言的语法和所使用的例库。为了表示方便，我们定义：

$$\text{generates}(I)\equiv \bigcup_{\forall i\in I}\text{generates}(i)$$

定义 3.5 函数 raises(i)→E：语句 i 执行过程中显式引发的异常类型的集合。

在 C++ 中，用关键字 throw 来显式引发异常，在编译时可以静态地获得其类型 T。可以定义：$\text{raises}(I)\equiv \bigcup_{\forall i\in I}\text{raises}(i)$

如果语句 i 不包括 throw 语句，则 raises(i)= Φ。

定义 3.6 函数 propagates(i)→E：语句 i 中所调用的函数传播出的异常类型的集合。

假设语句 i 所调用的函数的集合是 f_i，为了表示方便，可以定义：

$$\text{propagates}(I)\equiv \bigcup_{\forall i\in I}\text{propagates}(f_i)$$

如果语句 i 不包括函数调用，则 $propagates$(i)=Φ。

定义 3.7 函数 catches(g,c)→E：在一个保护范围 g 内遇到的异常且被与 g 关联的 catch 子句捕获到的异常类型的集合。

在 C++ 语言中，catch 子句的排列是有序的，查找处理程序时选择最先匹配的 catch 子句(父类异常的处理程序可以捕获子类的异常)，而不是选择最适合的 catch 子句。

定义 3.8 假设把一个类型 c_1 与另一个类型 c_2 相同或是 c_2 的子类表示为：$c_1\leqslant c_2$，对一个保护范围 $g=(t_1, C)$，$C=(c_1,\cdots,c_n)$，我们定义：

$$\text{catches}(c_i)\equiv\{e\mid e\in \text{encounters}(t_1)\wedge e\notin \bigcup_{j=1,\cdots i-1}\text{catches}(c_j)\wedge e\leqslant c_i\}$$

所以，与一个保护范围关联的所有的 catch 子句所捕获的异常的类型为：

$$\text{catches}(C)\equiv \bigcup_{\forall c\in C}\text{catches}(c)$$

定义 3.9 函数 uncaughts(g)→E：在保护范围 g 内遇到的、但与 g 关联的 catch 子句捕获不到的异常的类型的集合。

根据上面的定义，则可以得出下列的引理：

对一个保护范围来说，它引发的所有异常，减去它的处理程序所捕获到的异常，所得的差即是此范围捕获不到的异常，也就是传播出的异常。

引理 3.1 对一个保护范围 $g=(t_1, C)$来说，

$$\text{uncaught}(g)\equiv \text{encounters}(t_1)-\text{catches}(C)$$

$$\text{propagates}(g)\equiv\text{uncaught}(g)$$

可以定义：在一组保护范围 G 内：$\text{uncaught}(G)\equiv \bigcup_{\forall g\in G}\text{uncaught}(g)$

$$\text{propagates}(G)\equiv \bigcup_{\forall g\in G}propagates(g)$$

该模型适用于单个的函数，但大多数的 C++ 系统都是由

多个函数组成的。对于多个函数的系统，我们按照函数调用图[23]的拓扑顺序逐个计算各个函数所传播出的异常。

函数调用图的偏序序列就是函数的计算顺序。这可保证在计算任一函数时，它所调用的函数所传播出的异常是已经计算过的。

3.2 递归函数的异常传播类型的计算

对于计算某个范围内所传播出的异常，如果没有递归调用，利用上面的模型可以直接计算出。但如果在该范围内存在递归调用，则前面的模型就需要进行如下的特殊处理。

递归调用分为直接递归和间接递归。直接递归在函数调用图中表现为某一节点上有一个环，而间接递归则表现为一条回路。拓扑排序不能消除函数调用图中的函数调用环。

下面分两种情况来进行计算。

```
输入：递归函数的源代码
输出：递归函数传播出的异常类型集合
EL=Φ; //递归函数传播出的异常类型集
EL'=Φ;
F() 是递归函数
F = (I, G); I = ⋃i; G = ⋃g;
for  I中的每一个语句 i  do
      if  语句i是递归调用语句 then
          if 是直接递归 then
              EL'=Φ;
          else // 间接递归
              设调用顺序为:  F, A1,A2...An, F
              for j=1 to n do
                  计算 propagates(Aj);
                  EL'=EL' ∪ propagates((Aj);
              end for
          end if
          propagates(i) = EL';
          计算集合 generates(i), raises(i);
      else // 不是递归调用
          计算集合 generates(i), raises(i), propagates(i); //
      end if
end for
for  G中的每一个g  do  //对每一个 try 块
      计算 propagates(g);
end for
EL = generates(I) ∪ raises(I) ∪ propagates(I) ∪ propagates(G);
```

图 1　计算递归函数的传播异常类型的算法

对于直接递归调用函数 F 来说，当计算到达递归调用点时，我们首先设置 F 所传播出的异常类型的集合为空值(NULL)，则得出的结果就是函数 F 所传播出的异常类型的集合。

对于间接递归调用函数来说，假设调用顺序为：$F, A_1, A_2, \cdots, A_n, F$。当计算到达调用点 F 时，我们初始化 F 所传播出的异常类型的集合为空值(NULL)，我们就可以计算出 A_n 所传播出的异常类型的集合，再计算出 A_{n-1} 所传播出的异常类型的集合，以此类推，就可以计算出 F 所传播出的异常类型的集合。具体算法如图 1 所示。

对于直接递归调用函数来说，我们计算函数所传播出的异常类型集合时，是对函数内的所有语句进行计算的，所以首先设置 F 所传播出的异常类型的集合为空值(NULL)，并不影响最终的结果。对于间接递归函数来说，假设 $EL(A_i)$ 是函数 A_i 中除了 CALL A_{i+1} 语句以外的所有语句所传播出的异常，$EL'(F)$ 是函数 F 中除了 CALL A_1 语句以外的所有语句所传播出的异常，则 $EL(F) = EL'(F) \cup EL(A_1) \cup EL(A_2) \cdots \cup EL(A_n) \cup EL(F)$。我们首先假设方程右边的 $EL(F)$ 为 NULL，并不影响整个方程的值。这种计算方法要比 Schaefer 等人的方法简单：他们的方法需要扫描两遍，而我们的方法只需扫描一遍。

3.3 模型应用

用一个实例来说明上面模型的应用，如图 2 所示。一个 C++ 函数对应于一个范围 $s=(I,G)$，语句 I 包括在函数内但不在保护范围 G 内的所有语句。G 包括在函数内直接声明的所有的 try 块以及与 try 块相关联的 catch 子句。

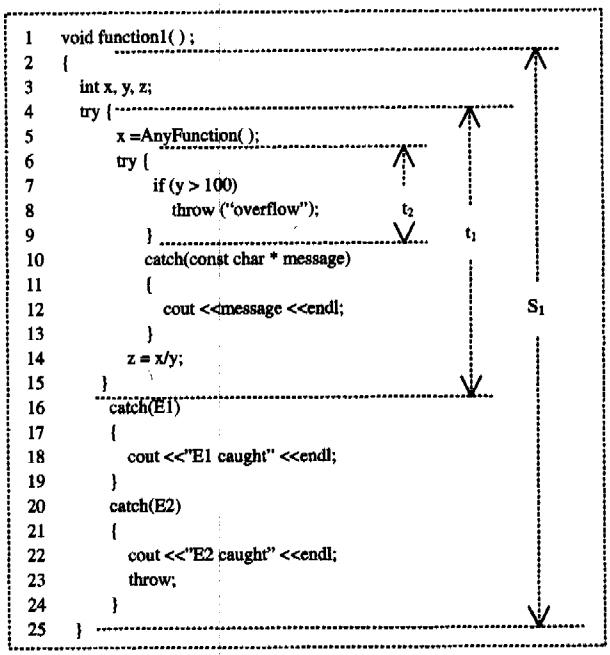

图 2　一个简化的 c++ 的异常处理结构的例子

图 2 是一个简单的 C++ 的异常处理结构的例子。在这个例子中，function1 包含了两个 try 块：一个是顶级的(4～15)，另一个是嵌套在里面的(6～9)。在 10 行的 catch 子句声明的异常类型为“const char”。如果在程序的执行过程中，第 8 行语句抛出异常，由于其类型是“const char”，则会被第 10 行的 catch 子句捕获到，否则就传播到外层的范围 s_1。如果在第 5 行的函数 AnyFunction()传播出的异常、或第 12 行和 14 行的语句中引发的异常，如果是 E1 类型或 E2 类型或是它们的子类，则会被第 16 行或第 20 行的 catch 语句所捕获，否则会传播出范围 s_1，到 function1 的调用函数中。

对于图 2 中的例子进行建模(右边的竖条指出了每一个 scope 的范围)，结果如表 1 所示。为了表示方便，假设函数 AnyFunction()可能传播出异常类型为 E2 的异常，根据前面的定义和引理 3.1 与引理 3.2，可以很容易地计算出每个范围内所遇到的异常和传播出的异常，计算结果如表 2 所示。

表 1　function1()中各个范围所包含的语句

范围	语句	范围	语句
s_1	(I_1, G_1)	G_2	{6～9,10}
I_1	{3,18,22,23}	t_2	(I_3, G_3)
G_1	{4～15,16,20}	I_3	{7,8}
t_1	(I_2, G_2)	G_3	Φ
I_2	{5,12,14}		

表 2　function1()中各个函数的值

函数	异常类型集合
encounters(t_2)	{ const char }
propagates(G_2)	Φ
encounters(t_1)	{divide by zero, E_2}
propagates(G_1)	{divide by zero }
encounters(s_1)	{ E_2, divide by zero }

由表 2 可知，范围 s_1 传播出两种类型的异常：E2 和 divide by zero，其中 divide by zero 类型的异常是由范围 G_1 传播出的，E2 类型的异常是在范围 s_1 内引发并捕获到的异常，但在处理程序中又重新抛出了。

通过这种形式化的描述，我们就可以设计工具来计算每个范围内所遇到的异常，为程序的分析和测试提供有价值的信息。

3.4 模型的局限性

为了减少模型的复杂度，我们并没有做数据流分析和可达性分析，这就使该模型得出的结果比较保守。如对于显式引发的异常，只要有 throw 语句，该模型就认为它能引发异常。但如果该 throw 语句不可达，实际上并不会引发该异常。

如果函数包含异常界面，可以简化该模型，函数传播出的异常类型的集合就可以通过异常界面直接得出，不需计算。但在 C++ 中，如果函数给出了异常界面，但又引发了不在异常界面中指定的异常类型时，则系统会调用一个特殊的函数 unexpected()。

由于该模型是对源代码进行分析，所以如果某一个函数的源代码无法获得的话，异常信息只能通过函数的异常界面获得；如果该函数不包含异常界面，我们假设该函数不向外传播异常。实际上，C++ 的异常界面和 Java 的异常界面有明显的不同：在 Java 的异常界面中没有声明的异常类型（检查类型异常）是严禁传播出该方法的；而在 C++ 中，并不强制要求函数有异常界面，在异常界面中声明了异常类型，则表明该函数可能传播出这种类型的异常，没有异常界面的函数可能传播出任何类型的异常。所以，在没有源代码的情况下计算出的结果并不很准确。

如果函数是多态调用时，由于没有做别名分析，所以得到的结果比较保守。

4 分析工具

静态分析是程序分析中一个很重要的方法。如果能设计一个工具来自动进行，可以避免大量重复性的工作。为了帮助开发者更好地理解和利用 C++ 的异常处理机制，针对上面的模型，我们开发了一个分析 C++ 程序中异常处理结构的静态分析工具 CETool。

4.1 功能及实现

静态分析工具 CETool 可以计算出 C++ 源程序中的异常处理结构信息，并用图示的方式显示出来。具体功能如下：

(1)根据前面的模型，CETool 可以计算出在范围 $s=(I, G)$ 内：propagates(I)、raises(I)和 uncaught(G)。并显示出所有可能显式引发的异常的位置、异常的类型、从保护范围 G 中传播出异常的类型以及传播异常的函数名等。

(2)计算 catch 语句所捕获的异常的类型，检测不可达的 catch 语句，检测 catch 语句所捕获的异常类型与引发的异常类型是完全相同？还是通过包含的关系捕获的？

(3)对选定的异常用图示的方式显示其传播路径。

我们首先对源程序进行词法分析和语法分析，保存与异常相关的信息，分析异常的传播路径，并设计工具使异常的传播路径可视化。系统结构图如图 3 所示。

由于篇幅所限，异常传播路径的分析详见参考文[22]，在此不做详述。

4.2 应用

对程序中的异常结构进行分析，能够帮助程序设计人员检测捕获不到的异常，使处理异常更准确。同时，对程序中的异常流进行分析，有利于软件的测试和维护工作，如结构测试。

结构测试技术是通过设计测试用例，使程序执行并覆盖程序中的不同结构元素。如路径测试是设计测试用例使其执行特定的路径。异常处理结构引入了新的结构元素，在进行结构测试时应该包含这些结构元素。根据我们的工具，可以分析出程序中所有的 throw 抛出的异常类型、catch 子句中的异常类型以及异常的传播路径。根据这些信息就可以帮助我们更有针对性地设计测试用例，使测试更加彻底。通常情况下，异常处理代码很难测试到，并且很容易出现错误。根据这些信息就可以很容易地测试异常处理代码，这也是我们下一步要研究的方向。

图 4 是我们开发的工具的界面，图的左边是以层次结构表示的语法结构图，以函数为单位显示出该模块内的 try 语句、catch 语句和 throw 语句的信息。图的中间是源代码，图的右边是指定异常的传播路径。图中是选定的异常类型 E2 的传播路径。

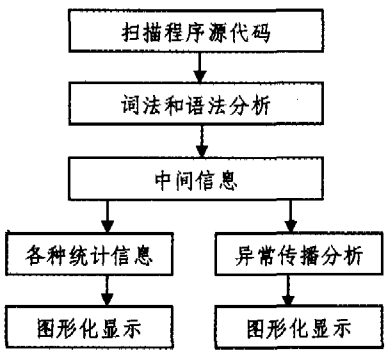

图 3 系统结构图

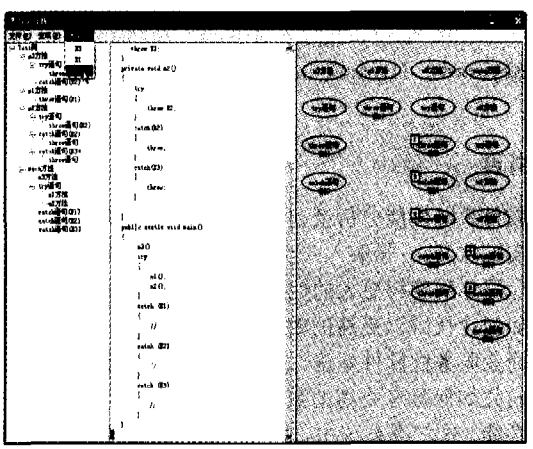

图 4 异常分析的可视化工具

5 相关工作比较

Robillard 和 Murphy[13~15]设计了一个分析 Java 程序中的异常流信息的工具 Jex，该工具在处理递归函数中所传播出的异常时采用了异常界面的信息，但这种方法并不适用于 C++ 程序。因为 Java 中的异常界面与 C++ 中的异常界面有很大的不同：在 Java 语言中，在异常界面中没有声明的异常

（下转封四）

(上接第 291 页)

类型(检查类型异常)是严禁传播出该方法的。而在 C++ 中,并不强制要求函数有异常界面。如果在异常界面中声明了异常类型,则表明该函数可能传播出这种类型的异常;如果没有异常界面,则该函数可能传播出任何类型的异常。所以 Robillard 和 Murphy 的方法并不适合于计算 C++ 内递归函数中所传播出的异常。

Schaefer 等[11]也开发了一个静态分析 Ada 程序中异常信息的工具。但他们在计算递归子程序所传播出的异常的集合时,需要扫描两遍;我们则只需要扫描一遍,而结果是一样的。

Brennan[12]只做了过程间的异常传播分析,没有具体到异常的类型和引发异常的语句。

结束语 为了更好地描述 C++ 程序中的异常处理结构,我们给出了一个描述模型。根据这个模型,既可以获得异常结构的局部信息,也可以获得异常处理结构的全局信息,这对于检测和改进异常处理结构都有很大的帮助。并对该模型应用于递归函数进行了讨论。我们也描述了一个静态的分析工具 CETool,它可以提供 C++ 程序的异常处理结构信息,包括异常引发的位置、异常类型、处理程序的类型、异常的传播路径等信息。但该工具目前还只能给出显式引发的异常的有关信息,对隐式引发的异常的有关信息还没有实现,这也是我们下一步研究的方向。

参 考 文 献

1 Garcia A F, Rubira C M F, Romanovsky A, et al. A comparative study of exception handling mechanisms for building dependable object-oriented software. The Journal of Systems and Software, 2001, 59: 197～222

2 Goodenough J B. Exception handling: issues and a proposed notation. Communications of the ACM, 1975, 18(12): 683～696

3 Claar J. Rethrowing function-try-blocks, Software Development China Machine press, 2003(in Chinese)

4 Stroustrup B. The C++ Programming Language (Special Edition). Addison-Wesley, 2000

5 Choi J D, Grove D, Hind M, et al. Efficient and precise modeling of exceptions for the analysis of Java programs. In: Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (Sept). ACM, 1999. 21～31

6 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

7 Pessaux F, Leroy X. Type-based analysis of uncaught exceptions. In: Proceedings of the 26th Symposium on the Principles of Programming Languages (Jan). ACM, 1999. 276～290

8 Guzm'An J C, Su'Arez A. A type system for exceptions. In: Proceedings of the 1994 ACM SIGPLAN Workshop on ML and Its Applications (June). ACM, 1994. 127～135

9 Fahndrich M, Foster J, Cu J, et al. Tracking down exceptions in standard ML programs: [Tech Rep]. CSD-98-996. Berkeley: University of California, 1998

10 Lang J, Stewart D B. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. ACMTrans Program Lang Syst, 1998, 20(2): 274～301

11 Schaefer C F, Bundy G N. Static analysis of exception handling. Ada Softw Pract Exper, 1993, 23(10): 1157～1174

12 Brennan P T. Observations on program-wide Ada exception propagation. In: Proceedings of the Conference on TRI-Ada '93 (Sept). ACM, 1993. 189～195

13 Robillard M P, Murphy G C. Analyzing exception flow in Java programs. In: Proceedings of the Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering. Lecture Notes in Computer Science (Sept) vol. 1687. New York, NY: Springer-Verlag, 1999. 322～337

14 Robillard M P, Murphy G C. Designing robust Java programs with exceptions. In: Proceedings of the 8th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (Nov). New York, NY: ACM Press, 2000. 2～10

15 Robillard M P, Murphy G C. Static analysis to support the evolution of exception structure in object-oriented systems. ACM Transactions on Software Engineering and Methodology, 2003, 12(2): 191～221

16 Sinha S, Harrold M J. Criteria for testing exception-handling constructs in Java programs. In: Proceedings of the International Conference on Software Maintenance (Sept). Los Alamitos, CA: IEEE Computer Society Press, 1999. 265～274

17 Sinha S, Harrold M J. Analysis and testing of programs with exception handling constructs. IEEE Trans on Softw Eng, 2000, 26(9): 849～871

18 Ryder B G, Smith D, Kremer U, et al. A static study of Java exceptions using JESP: [Tech Rep]. DSC-TR-406. Department of Computer Science, Rutgers University. 1999

19 Chang B-M, Jo J-W, Her S H. Visualization of exception propagation for Java using static analysis. In: Proceedings of the Second International Workshop on Source Code Analysis and Manipulation. IEEE, CA, 2002. 173～182

20 Howell C, Mularz D. Exception handling in large Ada systems. In: Washington Ada Symposium Proceedings, June 1991. 90～101

21 Howell C, Mularz D, Bundy G. Exception handling, or "when bad things happen to good programs". 14th International Conference on Software Engineering Tutorial, May 1992

22 Jiang Shujuan, Xu Baowen, Shi Liang. An approach to analysis exception propagation. In: the 8th IASTED International Conference on Software Engineering and Applications. November 9-11, 2004. MIT Cambridge, MA, USA, 2004. 300～305

23 徐宝文,张挺,陈振强. 递归子程序的依赖性分析及其应用. 计算机学报, 2001, 24(11): 1278～1284

计算机科学

(1974 年 1 月创刊)

第 33 卷第 4 期 (月刊)

2006 年 4 月 25 日出版

国际标准连续出版物号 ISSN 1002－137X

国内统一连续物出版号 CN50—1075/TP

定价: 30.00 元 国外定价: 5 美元

邮发代号: 78—68

发行范围: 国内外公开

主管单位: 国家科学技术部

主办单位: 国家科技部西南信息中心

编辑出版: 《计算机科学》杂志社

重庆市渝中区胜利路 132 号 邮政编码: 400013

电话: (023) 63500828 E-mail: jsjkx@swic.ac.cn

网址: www.jsjkx.com

社 长: 牟炳林

总 编: 彭 丹

主 编: 朱宗元

主编助理: 徐书令

印 刷 者: 重庆科情印务有限公司

总发行处: 重庆市邮政局

订 购 处: 全国各地邮政局

国外总发行: 中国国际图书贸易总公司 (北京 399 信箱)

国外代号: 6210—MO