

Approach to evaluating exception handling of programs

Jiang Shujuan¹ Xu Baowen²

(¹School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221116, China)

(²School of Computer Science and Engineering, Southeast University, Nanjing 210096, China)

Abstract: To solve the problems that the exception handling code is hard to test and maintain and that it affects the robustness and reliability of software, a method for evaluating the exception handling of programs is presented. The exception propagation graph (EPG) that describes the large programs with exception handling constructs is proposed by simplifying the control flow graph and it is applied to a case to verify its validity. According to the EPG, the exception handling code that never executes is identified; the points that are the most critical to controlling exception propagation are found; and the irrational exception handling code is corrected. The constructing algorithm for the EPG is given; thus, this provides a basis for automatically constructing the EPG and automatically correcting the irrational exception handling code.

Key words: software robustness; exception handling; exception propagation; evaluating program; control flow graph

Most modern programming languages provide an exception handling mechanism. Syntactically, an exception handling mechanism provides a means to raise an exceptional condition explicitly, and a means to express a block of code to handle one or more exception conditions^[1-2]. In some systems, the codes developed for error detection and handling are numerous and complex. As a consequence, up to two-thirds of a program can be for error handling^[3-4]. Ideally, robust software has to be able to recover from faults without substantially increasing the code complexity. Although the purpose of the exception handling code is to improve the robustness of the software, people have noticed that the exception handling code contains more errors than the other parts of the software. Indeed in a case-study^[5], more than 50% of the operational failures of a telephone switching system were due to faults in exception handling and recovery algorithms. The Ariane 5 launch vehicle was lost due to an unhandling exception destroying \$ 400 million of scientific payload^[4].

It is difficult to test the exception handling by usual methods because of its particularity^[6-7]. If we evaluate the exception handling code of the program

before it runs, and correct the irrational parts, the robustness of the program is improved. This is a useful work for large programs.

We have made some achievements in analyzing exception propagation^[8-9]. The work of this paper is based on our former work. This method overcomes the limitations of previous methods, too complex in describing large programs, and also provides a basis for automatically evaluating the exception handling code.

1 Describing Method of Exception Handling Constructs

Now, the methods of describing programs with exception handling constructs are mainly the method of Sinha and colleagues and the method of Choi and colleagues. Sinha^[10] proposed a method of describing intra-procedure and inter-procedure control flow of Java programs with exception handling constructs. The representations explicitly show the exception types that can be raised at throw statements, and exception types that are propagated across methods. In the describing method, there are throw nodes, catch nodes, final nodes and their corresponding edges excepting the nodes that a usual control flow graph has.

Choi et al.^[11] described an intra-procedural control-flow representation called the factored control-flow graph (FCFG) to efficiently analyze programs written in Java that contain exception handling constructs. The FCFG represents exceptional control flow caused by both explicit and implicit exceptions. But the potentially exception-throwing instruction (PEI) presented by

Received 2007-03-16.

Foundation items: The National Natural Science Foundation of China (No. 60503020), the National Basic Research Program of China (973 Program) (No. 2002CB312000), the Natural Science Foundation of Jiangsu Province (No. BK2006094), the Science Research Foundation of China University of Mining and Technology.

Biographies: Jiang Shujuan (1966—), female, doctor, professor, shjjiang@cumt.edu.cn; Xu Baowen (1961—), male, doctor, professor, bwxu@seu.edu.cn.

them decomposes an exception statement into too small instructions. This does harm to the understanding and testing of a program.

The two methods solve the problem of describing exception handling constructs in large part, but they are not applied to large programs because of their complexity. Now, we will present a simple method of describing large programs with exception handling constructs.

1.1 Construction of exception propagation graph

Exception handling constructs change the control flow of programs, but the effect is only related to some special elements of a program rather than all statements. The special elements are the places where the exception is raised, the type of the raised exception, the type and place of exception handling, and the function call chain. To analyze exception propagation, all these facilities must be represented in a simple and precise way without losing useful information. For convenience, we construct a simple exception control flow graph (ECFG) for each function. The ECFG is similar to a program control flow graph.

Definition 1 An exception control flow graph of a function F is a direct graph $G_E = \langle S_S, S_E, s_1, s_F \rangle$, where S_S is a node set and S_E is an edge set. Nodes in S_S represent try statements, throw statements, catch handlers, function call statements, and exception-exit nodes that are created dynamically. $S_E = \{(s_i, s_j) \mid s_i, s_j \in S_S \wedge s_j \text{ may be executed after } s_i\}$. Two special nodes s_1 and s_F represent the beginning and the end of F . If $(s_i, s_j) \in S_E$, s_i is an immediate predecessor of s_j , then s_j is an immediate successor of s_i . In the ECFG, if it can reach s_j from s_i , $S_{\text{path}}(s_i, s_j)$ is the set of nodes that pass through from s_i to s_j .

An ECFG can be automatically constructed from a program source code. For a try statement, a catch block and a throw statement, a try node, a catch node with a catch-list property which records exception types in the same order of catch statements, and a throw node with an exception-type property are created. Comparisons are made between the exception-type of throw node with the catch-list of the catch node. If matching, a handler node is created; otherwise, an exit node is created, which is the immediate successor of the catch node. The detailed construction algorithm is shown as follows:

Algorithm 1 ConstructECFG

Input: Program source code;

Output: Exception control flow graph.

for the i -th try statement, create a try _{i} node ($i = 1, 2, \dots, n$)

for each catch block that is matched with the i -th try statement {

 create a catch _{i} node that contains a catch-list _{i} property,
 which records exception types in the same order of catch statements}
 for the j -th throw statement within the i -th try block {
 create throw _{ij} node which contains an exceptiontype property;

 add an edge from try _{i} node to throw _{ij} node;
 type inference for throw _{ij} node;
 add an edge from throw _{ij} node to catch _{i} node;
 compare exceptiontype of throw _{ij} with catch-list _{i} of catch _{i}
 node
 if matched
 create a “handler(exceptiontype)” node as successor of
 catch _{i} node
 else create an “exit function name(exceptiontype)” node as
 successor of catch _{i} node}
 if throw statement is within a catch block {
 create throw node as successor of “handler(exceptiontype)”
 node
 if the catch statement is within nested try block
 add an edge from throw node to nesting catch node and
 continue to compare
 else create an “exit function name(exceptiontype)” node as
 successor of throw node}
 else {create throw node that contains an exceptiontype property;
 create an “exit(function name)” node as successor of throw
 node; }
for a function call statement within the i -th try block {
 create a “function name _{i} ” node;
 add an edge from try _{i} node to “function name _{i} ” node}
for a function call statement not within try block
 create a “function name” node;

The ECFG only expresses the intra-function exception handling constructs. The propagation of exceptions on the call stack creates inter-function exceptional control flow. In order to analyze exception propagation, we construct an exception propagation graph (EPG) of program P on the basis of the ECFG. In the exception propagation graph, the nodes are the same as the nodes in the ECFG except for adding exception handling nodes and exception exit nodes; the edges are the same as the edges in ECFG except for adding function call edges and exception return edges.

An exception propagation graph for a program P consists of ECFGs for each function in P ; at each call site, the call node is connected to the entry node of the called function by a call edge, and the exceptional-exit node of the called function is connected, by an exceptional-return edge, to the corresponding return node. If the function call site is in a try block, the corresponding return node is the catch node that is matched with the try block; otherwise, it is the exit node. Algorithm ConstructEPG shows the construction algorithm of the EPG.

Algorithm 2 ConstructEPG

Input: Exception control flow graph for each function in P ;

Output: Exception propagation graph for P .

```

Declare functionlist: Functions that are processed iteratively
initialize functionlist with functions in  $P$ ;
while functionlist is not empty {
  remove function  $N$  from functionlist;
  for each call site in try $i$  block in function  $M$  that calls  $N$  {
    create a call edge;
    for each exit( $N$ ) node in function  $N$  {
      add an exception-return edge that connects to catch $i$  node
in function  $M$ ;
      if exceptiontype in exit( $N$ ) can match with exception-
type in catch-list $i$ ;
        create a handler node;
      else {
        while catch $i$  node in a nesting try block {
          compare exceptiontype in exit( $N$ ) with catch-list of
nesting catch node
          if match
            create a handler node; }
        create an exit( $M$ ) node and add  $M$  to functionl-
ist; }
    }
  }
  for each call site not within try block in function  $M$  that call  $N$ 
    create a call edge;
  for each exit( $N$ ) node in function  $N$ 
    create an exception-return edge and connect exit( $N$ )
node to exit( $M$ ) node;
}

```

In the EPG, an exception propagation path is a sequence of nodes that are from throw node to exception handling node,

$$S_{\text{path}}(s_{\text{throw}}, s_{\text{handler}}) = \langle s_{\text{throw}}, s_1, s_2, \dots, s_{\text{handler}} \rangle$$

1.2 Complexity of algorithm analysis

Let N be the number of the nodes in the ECFG that only contains try nodes, throw nodes, catch nodes, function call statements, exception handler nodes that dynamically create, and S be the time that looks for these nodes in the program. The cost of ConstructECFG is linear in the size of a function if the function contains no throw statements. To process a throw statement, ConstructECFG performs type inferencing and determines its successors based on the inferred types. The cost of type inferencing depends on which approach type inferencing is used^[12]. Let $wcc(TI)$ be the worst-case complexity of type inferencing and P be the number of inferred exception types. Let T be the number of throw statements in a function, and let H be the number of catch handlers, respectively, that enclose a call site in a function. For each type that is inferred for a throw statement, ConstructECFG searches for a target among the enclosing catch handler. Therefore, the cost of processing a throw statement is $O(wcc(TI) + P \times H)$. The overall cost of the ECFG construction is $O(N + S + T \times (wcc(TI) + P \times H))$.

The cost of ConstructEPG is bounded by the num-

ber of functions that it processes and the cost of processing each function. If M is the number of functions in a program, in the worst case, ConstructEPG may process $O(M^2)$ functions. For each function, the algorithm examines all call sites, and for a given call site, the algorithm iterates over the “exit function-name (exceptiontype)” node in a called function. For each “exit function-name (exceptiontype)” node, ConstructEPG searches for a catch handler in the calling function. If C and X are the numbers of call sites and the number of “exit function-name (exceptiontype)” nodes, respectively, in a method, the cost of processing a method is $O(C \times X \times H)$, where H is the number of catch handlers that enclose a call site. Therefore, the cost of EPG construction is $O(M^2 \times C \times X \times H)$.

1.3 Case study

We take the method described in section 1.1 as an example. The following is a simplified C++ program code.

```

void m1()
{ try {...
1      throw new E2();
      ...}
2  catch (E2) {...
3      throw; }
4  catch (E3) {...}
}
void m2()
{ ...
  try {...
5      throw new E3();
      ...}
6  catch (E2) {...}
}
void main()
{ ...
7  m2();
  try {...
8      m1();
      ...}
9  catch(E1) {...}
10 catch(E2) {...}
11 catch(E3) {...}
}

```

Fig. 1 shows its EPG that is constructed according to the algorithms. The exception E2 raised in function m1() is caught in statement 2, so it creates a handler (E2) node. Because it is re-thrown in statement 3, it creates a throw(E2) node and then creates an “exit m1 (E2)” node. The exception E3 raised in function m2() propagates because there is no matching handler in catch block, and it then creates an “exit m2 (E3)” node.

In Fig. 1, since function m1() has an “exit m1

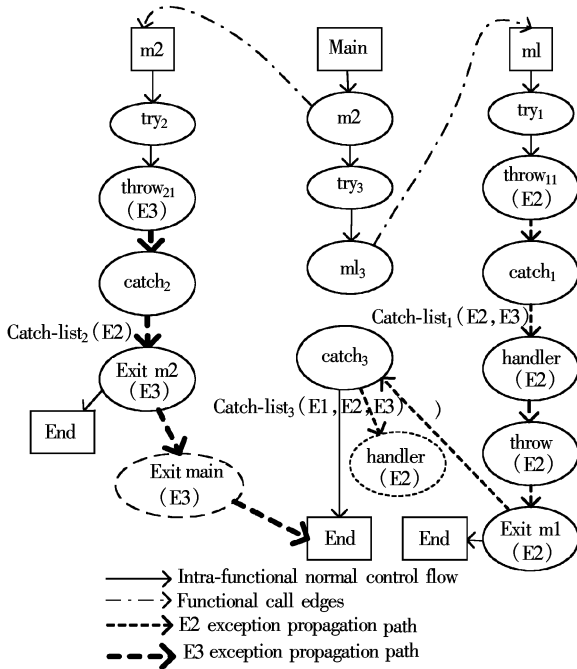


Fig. 1 Exception propagation graph

(E2)” node and the $m1()$ is called within the try block, the “exit $m1(E2)$ ” node is connected, by an exception-return edge, to the $catch_3$ node in the ECFG of $main()$. Comparing E2 with $catch_list_3$, it creates a $handler(E2)$ node as the successor of the $catch_3$ node because of matching. Although function $m2()$ has an “exit $m2(E3)$ ” node, it creates an “exit $main(E3)$ ” node because the function call statement $m2()$ is not in the try block. The exception-return edge is from the “exit $m2(E3)$ ” node to the “exit $main(E3)$ ” node.

In Fig. 1, the dashed lines denote function call edges and exception-return edges because of exception propagation. We can easily find the two exception propagation paths from the EPG.

$$S_{path}(E2) = \{throw_{w11}(E2), catch_1, handler(E2), throw(E2), exit\ m1(E2), catch_3, handler(E2)\}$$

$$S_{path}(E3) = \{throw_{w21}(E3), catch_2, exit\ m2(E3), exit\ main(E3)\}$$

2 Evaluating of Exception Handling Code

According to the EPG, we can evaluate the exception handling code of programs. This can provide threads for improving the design and implementation of the exception handling code.

First, we can find some catch statements that never execute. In the C++ program, when an exception is thrown, the exception-handling system looks through the “nearest” handlers in the order they appear in the source code. When it finds a match, the exception is

considered handled and no further searching occurs. Matching an exception does not require a perfect correlation between the exception and its handler. An object or reference to a derived-class object will match a handler for the base class. The advantage of this matching method is efficient, but it can result in some catch statements never executing. If a handler for a base class is located before a handler for the derived-class in the order they appear in the source code, respectively, within a try block, the later handler (a handler for the derived-class) will never execute. We can find this error according to the catch-list property of the catch node and correct it. This makes the exception being handled more exactly. For example, in Fig. 1, we can obtain the relationship among E1, E2 and E3 according to the program code and find whether a handler for a base class is located before a handler for the derived-class according to the $catch_list_3(E1, E2, E3)$.

Secondly, we can find the origins of the exceptions propagation according to EPG. For example, a frequently called function that propagates just one exception can quickly result in a proliferation of continued propagations, so the developers should pay more attention to the frequently called functions and assure that no exception propagates from them.

Thirdly, we can find that some function call statement is not within the try block from the EPG. For example, $m2()$ raises an exception E3 and propagates it in Fig. 1. Because the statement $m2()$ is not within the try block of $main()$, the exception E3 propagated from $m2()$ cannot be matched with the catch block in $main()$ although there is a handler for exception E3 in the catch block of $main()$. We can remove the function-call statement into the try block, so it can search for an exception handler in a catch block.

Fourthly, when exceptions are continually propagated, even if eventually handled, the question must be addressed whether the exception handlers take appropriate corrective action. This does not imply that immediately handling exception guarantees sufficient corrective action, only that it is more likely. For example, we can find the suitable points in exception propagation paths that are the most critical to controlling exception propagation from the EPG. For example, locating more handlers in the frequently called functions, and so on.

Finally, if many exceptions propagate out of the main program and the program abnormally terminates, it means that there is no matching handler for a raised exception. This is finally handled by the default han-

andler of the system. This implies that the ability to handle exceptions is lacking.

3 Conclusion

In this paper, we present a new approach, an exception propagation graph, to describing large programs with exception handling constructs. In this method, the EPG consists only of statements that relate to exception handling, which is different from the traditional control flow graph. We can evaluate the exception handling code according to the EPG and correct the irrational part code. The information can guide programmers to put exception handlers at appropriate places by tracing exception propagation. The paper also gives the ECFG and EPG construction algorithms, and this provides support for automatically constructing the EPG.

In future work, we are planning to develop a visualization tool that visualizes the exception propagation path and automatically correct part of the irrational exception handling codes.

References

[1] Goodenough J B. Exception handling: issues and a proposed notation[J]. *Communications of the ACM*, 1975, **18**(12): 683 – 696.

[2] Robillard M P, Murphy G C. Static analysis to support the evolution of exception structure in object-oriented systems [J]. *ACM Transactions on Software Engineering and Meth-*

odology, 2003, **12**(2): 191 – 221.

[3] Garcia A F, Rubira C M F, Romanovsky A, et al. A comparative study of exception handling mechanisms for building dependable object-oriented software[J]. *The Journal of Systems and Software*, 2001, **59**(2): 197 – 222.

[4] Tracey N, Clark J, Mander K, et al. Automated test-data generation for exception conditions [J]. *Software-Practice and Experience*, 2000, **30**(1): 61 – 79.

[5] Toy W N. Fault-tolerant design of local ESS processors[J]. *Proceedings of the IEEE*, 1978, **66**(10): 1126 – 1145.

[6] Jiang Shujuan, Zhang Yongping, Yan Dashun, et al. An approach to automatic testing exception handling [J]. *ACM SIGPLAN Notices*, 2005, **40**(8): 34 – 39.

[7] Fu C, Ryder B G, Wonnacott D G. Robustness testing of Java server applications [J]. *IEEE Transactions on Software Engineering*, 2005, **31**(4): 292 – 311.

[8] Jiang Shujuan, Xu Baowen, Shi Liang. An approach of data-flow analysis based on exception propagation analysis[J]. *Journal of Software*, 2007, **18**(1): 74 – 84. (in Chinese)

[9] Jiang Shujuan, Xu Baowen, Shi Liang, et al. An approach to analyzing dependence based on exception propagation analysis[J]. *Journal of Software*, 2007, **18**(4): 832 – 841. (in Chinese)

[10] Sinha S, Harrold M J. Analysis and testing of programs with exception-handling constructs[J]. *IEEE Transactions on Software Engineering*, 2000, **26**(9): 849 – 871.

[11] Choi J D, Grove D, Hind M, et al. Efficient and precise modeling of exceptions for analysis of Java programs [C]//*Proceedings of PASTE’ 99 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Toulouse, France, 1999: 21 – 31.

一种异常处理策略的评测方法

姜淑娟¹ 徐宝文²

(¹ 中国矿业大学计算机科学与技术学院,徐州 221116)
(² 东南大学计算机科学与工程学院,南京 210096)

摘要:针对程序中异常处理代码难以测试和维护、影响软件的健壮性和可靠性的问题,提出了一种评测程序中异常处理策略的方法.通过简化程序的控制流图,得到一种描述大型程序中异常处理结构的方法——异常传播图,并用实例验证了其有效性.根据程序的异常传播图,可以检测出程序中不可达的异常处理代码、找到控制异常传播的最佳位置、修正不合理的异常处理策略等.并给出了异常传播图的构造算法,为该方法实现自动化处理提供基础.

关键词:软件健壮性;异常处理;异常传播;评测程序;控制流图

中图分类号:TP311