Analysis of Object-Oriented Programs with Exception-Handling Constructs

Yanmei Zhang, Shujuan Jiang, Xuefeng Zhao
School of Computer Science and Technology, China University of Mining and Technology
Xuzhou, 221116, Jiangsu Province, China, ymzhang@cumt.edu.cn

Abstract

In the dependence analysis of object-oriented programs, if the influence of exception propagation on the dependency is not considered, the information will be inaccurate. At present, the existing methods with exception propagation mostly aimed at intra-class analysis, which cannot meet the needs of actual software development. This paper proposes an approach to analyzing inter-class control dependence of programs with exception-handling constructs. The approach takes account of constructors and destructors, which makes the analysis more accurate. The approach is carried on by constructing inter-class control flow graph. According to the relationships of different classes, such as inheritance, aggregation and association, we incrementally construct an inter-class control flow graph. Finally, we apply the analysis method to C++ program slicing. The experiment results show that our approach is more accurate and can bring about the improvement of the slice accuracy through analyzing the influence of exception-handling constructs.

Keywords: Exception Propagation, Inter-Class, Control Flow, Program Slicing

1. Introduction

Program dependency analysis is essential to program analysis, software understanding and software maintenance, and it has a wide range of applications in software maintenance, program testing, program optimization and parallel analysis [1, 2]. Lu et al [8] have already proposed object-oriented program dependence analysis techniques at class cluster-level, but they ignored the effect of the exception-handling constructs on the program.

However, exception-handling mechanism is necessary parts in high-level programming language (such as Ada, C++, Java, etc.). If the exception handling mechanism can be used correctly, it can improve the software robustness. However, exception propagation can change the execution path of the program without considering exception-handling constructs; besides, the corresponding control flow and data flow information would be changed. Failure to account for the effects of exception-handling constructs in performing analyses can result in incorrect analysis information. When this inaccurate information is used for software engineering tasks such as structural tests, regression test and program slicing, it may cause serious failure [3, 18].

Therefore, the importance of exception propagation causes the wide attention of software developers and testers. At present, many researchers are engaging in this research. A large number of literatures have proposed many intra-procedural and inter-procedural techniques of control dependency [1, 2, 3, 4, 10, 11, 16, 17]. Although the exception propagation of the programs in these literatures are analyzed, they are merely procedure-level or class-level, which obviously can't meet the need of actual development of object-oriented software.

So far, the literatures on inter-class control dependence analysis for programs with exception-handling constructs are relatively rare: For example, Sinha and Harrold [12] have proposed an approach of analysis and testing of programs with exception-handling constructs. However, the exception analysis is not precise enough. The techniques proposed in their paper applied only to explicitly raised exceptions.

None of above researches, however, took account of constructors and destructors or considered the influences of exceptions on analysis techniques- program slicing.

In this paper, on the basis of our previous work [7], we present an approach to analyzing inter-class control dependence of programs with exception-handling constructs, and give a corresponding construction method of inter-class control flow graph for programs with exception-handling (EICCFG). Finally, we apply our analysis information to an application:
program slicing. We discuss the problems and solutions for C++ exception-handling constructs; constructs in other languages, such as Java, can be analyzed similarly. The experiment results show that our approach brings about the improvement of the program slicing accuracy through analyzing the influence of exception-handling constructs.

This paper is divided into six sections, and the organization of the paper is as follows. Section 2 introduces the inter-class control dependency analysis. Section 3 gives a case study. Section 4 gives some experiments. Section 5 reviews the related works on inter-procedural and inter-class analyses. Finally, section 6 draws conclusions and points out future research work.

2. Inter-class dependency analysis

In this section, we propose a cluster-level control dependency analysis method. Cluster is a group of cooperative classes. Cluster testing is mainly to detect the interaction of a group of cooperating classes. The cluster-level analysis method in this paper takes into account the exception-handling constructs and inter-class relationships.

In order to give an accurate definition to inter-class control dependence relationship, we introduce the concept of class control flow graph (CCFG) as follows:

Definition 1 Class Control Flow Graph (CCFG): suppose C is a class program segment, Class Control Flow Graphs CCFG (C) = (N, E) is a composite graph. Where:
(1)\(N = \{CC, M_{\text{Inter}}, M_{\text{Exit}}, M_{\text{Call}}, M_{\text{Return}}, NS\}\), \(CC\) represents class control vertex set; \(M_{\text{Inter}}\) and \(M_{\text{Exit}}\) are function entry vertex set and exit vertex set respectively; \(M_{\text{Call}}\) and \(M_{\text{Return}}\) are function call vertex set and return vertex set respectively; \(NS\) is program statement vertex set.

(2)\(E = \{EB, EF\}\), \(EB\) is undirected edge set; \(EF\) is attributive edge set, and it expresses the attributive relationship of function (or attribute) vertexes and the class control vertexes. \(EF\) is directed edge set, expresses the edge set of program control flow.

Inter-class dependency analysis is based on inter-class control flow graph, which is based on CCFG. However, when there are exception-handling constructs, in order to facilitate inter-class dependency analysis, we need to construct inter-class control flow graph of program with exceptions to reflect inter-class dependency intuitively.

2.1. Construct inter-class control flow graph with exceptions (EICCFG)

In this subsection, we propose a construction method of EICCFG.

Definition 2 Inter-class control flow graph of program with exception-handling constructs (EICCFG): EICCFG is composed by combining the CCFG of each object, adding the edge set and vertex set that represent exception-handling constructs. EICCFG = \(<C, N, E1, E2, Sentry, Sexit, Norm_{\text{exit}}, exceType_{\text{exits}}><C, N, E1, E2, Sentry, Sexit, Norm_{\text{exit}}, exceType_{\text{exits}}>\). Where,
\(C = \{c\mid c\) represents the classes composed a system or sub-system\};
\(N = \{n\mid n\) represents statements in a program\};
\(E1 = \{<n1, n2>\mid n1, n2 \in N, and n2 may be executed immediately after statements n1 executed\};
\(E2 = \{<c1, c2, I> or <c1, c2, A_g> or <c1, c2, A_s>\mid c1, c2 \in C, and c2 inherits c1 or aggregates c1 or associate with c1\};
Sentry \(\in N\), represents the entrance of a function;
Sexit \(\in N\), represents the exit of a function;
Norm_{\text{exit}} represents the normal exit vertex of a function;
ExceType_{\text{exits}} represents that each thrown exception type in a function has an exception exit vertex.

The task is to add exception-handling constructs to construct complete EICCFG. In this paper, the steps for building an EICCFG are given below. The first step is to construct CCFG for each class, that is, inter-class function callings are ignored temporarily; the second step is to connect the separated class control graph CCFG to form the EICCFG. The third step is to add exception-handling constructs.

Step 1 (build CCFGs): it can be constructed according to the strategy in reference [6].
Step 2 (connect the CCFGs to form inter-class CCFG): This is the key step to resolve. In this step,
we first deal with the inter-class function callings without exceptions.

Step 3 (add exception-handling constructs): the step is to add exception-handling constructs that can bring the propagation of exception in inter-class and inter-procedural to complete EICCFG.

The first step is to construct Object Relation Diagrams (ORD) for all of the classes, which can describes the static dependence relations (i.e. inheritance, aggregation and association) between two different classes. That is, inter-class function callings are ignored temporarily.

The second step is to connect each CCFG to the corresponding class in ORD, and adds control dependence edges from the call vertex to the called function M’s enter vertex for each call to a function M.

According to the influences of exception propagation on dependency of program, we know that inter-class exception propagation analysis is based on the inter-class control flow graph of programs. In order to handle exception-handling constructs and facilitate the exception propagation analysis, in this section, the paper presents an approach to analyze exception propagation of object-oriented program.

First, we make changes on traditional intra-procedural control flow in the following aspects:

1. Add an exceType_exit (exception exit’ type) vertex for each type of exception which may be thrown in a procedure, and add a norm_exit (normal exit) vertex for each procedure. In addition, S_exit vertex as the successor vertex of exceType_exit and norm_exit vertex;
2. Each try/catch statement is corresponding to a try/catch vertex, and it is connected with the first vertex of try/catch block;
3. Each throw statement is corresponding to a throw vertex. If the throw statement is in try block, and there exists catch block which is matched with the try block, then the outgoing edge of throw vertex is connected to the catch vertex, otherwise it is connected to the corresponding exceType_exit vertex.

Then, we make some changes on inter-procedural control flow in the following aspects:

1. If a procedure which may throw an exception is called within a try block, and there exists catch vertex which is matched with exceType_exit vertex in the called procedure, then the outgoing edge of exceType_exit vertex is connected to the catch vertex; otherwise, for the other conditions, it is connected to the corresponding exceType_exit vertex of the called procedure. Then exception will back propagate along this path;
2. The norm_exit vertex of called procedure is connected to the corresponding return vertex.

2.2. Construction algorithm of inter-class control flow graph with exceptions (EICCFG)

In this subsection, according to the construction approach of EICCFG, we give its corresponding construction algorithm. It is shown in Figure 1.

In the algorithm, the main process is as follows.

First, build CCFGs and construct Object Relation Diagrams (ORD) for all of the classes (line 1).

Then, for each CCFG in CCFGs, connect CCFG to the corresponding class vertexes of ORD (lines 2-4).

Next, we deal with the inter-class function callings without exceptions (lines 5-41). For each function M of class C, if M call the function M1 of another class C1, and M1 is a new defined member function of class C1 or M1 override function of its parent class, then connect the call edge and return edge of M to the corresponding call vertex and return vertex of M1 in C1(lines 5-10). Otherwise search the parent class of C1 (lines 11-21). If C is associated with C2 in M, add calling vertex C and return vertex R for the constructor of C2 at the define vertex of object variable for C2; meanwhile, add the edge that from the calling vertex of M to the entry vertex of the constructor of C2 and the edge from the exit vertex of the constructor of C2 to the return vertex of M (lines 24-27). In addition, add calling vertex C and return vertex R for destructor of C2 at the end of M; meanwhile, add the edge that from the calling vertex of M to the entry vertex of the destructor of C2 and the edge from the exit vertex of the destructor of C2 to the return vertex of M (lines 28-31). If C aggregates C2, add calling vertex C and return vertex R for constructor of C2 at the end of constructor of C; add the edge from the calling vertex of the constructor of C to the entry vertex of the constructor of C2 and the edge from the exit vertex of the constructor of C2 to the return vertex of the constructor of C (lines 33-36). In addition, add calling vertex C and return vertex R for destructor of C2 at the end of the destructor of C; meanwhile, add the edge from the calling vertex of the destructor of C2 and the edge from the exit

...
Finally, add exception-handling constructs (lines 42-64). First, add a control dependency edge from calling vertex in function M to M’s entrance vertex and a control dependency edge from function M’s norm exit vertex to M’s norm return vertex (lines 43-44). In the case that the calling vertex of each called function M1 is in function M and in a try block, we try to find if there exists catch block matching with exceType_exit for each exceType_exit vertex of function M1. If the catch block matching with exceType_exit can be found, then add a control dependency edge from exceType_exit vertex to corresponding catch vertex in M (lines 45-48); else while function call vertex is in a nested try block, we need to find catch vertex that matched with the type of exceType_exit from outer catch block. If matched, then add a control dependency edge from exceType_exit vertex to the catch vertex; in addition, add a control dependency edge from exceType_exit vertex to M’s exceType_exit vertex (lines 49-58). In the case that the calling vertex of each called function M1 is in function M but not in a try block, add a control dependency edge from exceType_exit vertex to M’s exceType_exit vertex for each exceType_exit vertex of function M1 (lines 59-63). Finally delete edges that have nothing to do with exception-handling structure of EICCFG, simplify EICCFG (lines 65-67).

The algorithm bases on inter-class control dependence analysis, therefore it refers to functional callings between different classes.

Algorithm: Construction of EICCFG

Input: class cluster C={C1,...,Cm}, EICFGs
Output: EICFG(C)

1. CCFG(C) = CCFG of each class in C;
2. EICFG(C) = object relation diagram of C;
3. EICFG(C)=EICFG of C;
4. C is the upper class of C;
5. M(D) if M is a new defined method of class C, return true, else return false;
6. Override(M) if class C overrides method M in base class or returns false;
7. ECD(E)=ECD of E, i.e., the entry vertex of E is the entry vertex of M;
8. ECD(E)=ECD of E, i.e., the exit vertex of E is the exit vertex of M.
9. begin
   1. construct OBD and CCFG(G)=ORICG;
   2. for each C in C do
      3. connect CCFG(C) to the corresponding class vertices in G;
      4. (Nondisjunct class) add the new class to establish CCFG for each class;
   5. G=G+CCFG(C);
   6. end
   7. for each C in C do
      8. for each M in C do
         9. if M is a new defined method of class C or M overrides method of
            10. (then connect the call edge and return edge of M with the corresponding call vertex
                11. and return vertex of M in C;
                12. if M is a new defined method of C or M overrides method of C's
                   13. then connect the call edge and return edge of M with the corresponding call vertex
                      and return vertex of M in C;
                      14. if M is a new defined method of C or M overrides method of C's
                         then connect the call edge and return edge of M with the corresponding call vertex
                            and return vertex of M in C;
                            15. if M is a new defined method of C or M overrides method of C's
                               then connect the call edge and return edge of M with the corresponding call vertex
                                  and return vertex of M in C;
                                  16. if M is a new defined method of C or M overrides method of C's
                                     then connect the call edge and return edge of M with the corresponding call vertex
                                        and return vertex of M in C;
                                        17. if M is a new defined method of C or M overrides method of C's
                                           then connect the call edge and return edge of M with the corresponding call vertex
                                              and return vertex of M in C;
                                              18. if M is a new defined method of C or M overrides method of C's
                                                 then connect the call edge and return edge of M with the corresponding call vertex
                                                    and return vertex of M in C;
                                                    19. if M is a new defined method of C or M overrides method of C's
                                                       then connect the call edge and return edge of M with the corresponding call vertex
                                                          and return vertex of M in C;
                                                          20. if M is a new defined method of C or M overrides method of C's
                                                             then connect the call edge and return edge of M with the corresponding call vertex
                                                                and return vertex of M in C;
                                                                21. if M is a new defined method of C or M overrides method of C's
                                                                   then connect the call edge and return edge of M with the corresponding call vertex
                                                                      and return vertex of M in C;
                                                                      22. if M is a new defined method of C or M overrides method of C's
                                                                         then connect the call edge and return edge of M with the corresponding call vertex
                                                                            and return vertex of M in C;
                                                                            23. if M is a new defined method of C or M overrides method of C's
                                                                               then connect the call edge and return edge of M with the corresponding call vertex
                                                                                  and return vertex of M in C;
                                                                                  24. if M is a new defined method of C or M overrides method of C's
                                                                                     then connect the call edge and return edge of M with the corresponding call vertex
                                                                                        and return vertex of M in C;
                                                                                        25. if M is a new defined method of C or M overrides method of C's
                                                                                           then connect the call edge and return edge of M with the corresponding call vertex
                                                                                             and return vertex of M in C;
                                                                                             26. if M is a new defined method of C or M overrides method of C's
                                                                                               then connect the call edge and return edge of M with the corresponding call vertex
                                                                                                 and return vertex of M in C;
                                                                                                 27. if M is a new defined method of C or M overrides method of C's
                                                                                                     then connect the call edge and return edge of M with the corresponding call vertex
                                                                                                         and return vertex of M in C;
                                                                                                         28. if M is a new defined method of C or M overrides method of C's
                                                                                                            then connect the call edge and return edge of M with the corresponding call vertex
                                                                                                                and return vertex of M in C;
                                                                                                                29. if M is a new defined method of C or M overrides method of C's
                                                                                                                   then connect the call edge and return edge of M with the corresponding call vertex
                                                                                                                        and return vertex of M in C;
                                                                                                                        30. if M is a new defined method of C or M overrides method of C's
                                                                                                                           then connect the call edge and return edge of M with the corresponding call vertex
                                                                                                                               and return vertex of M in C;
                                                                                                                               31. end
   8. end
   9. end
   10. end

Figure 1. Construction Algorithm of EICCFG
2.1.1. Complexity of the construction algorithm

The cost of the construction algorithm in Figure 1 is bounded by the number of methods that it processes and the cost of processing each method. If \( M \) is the number of methods in a program, in the worst case, the algorithm may process \( O(M) \) methods. For each method, the algorithm examines all call sites in the method, and for a given call site, the algorithm iterates over the exceptional-exit nodes in the called method. For each exceptional-exit node, it searches for a catch handler and processes finally blocks in the calling method. If \( C \) and \( X \) are the numbers of call sites and the number of exceptional-exit nodes, \( H \) and \( F \) are the number of catch handlers and finally blocks, respectively, in a method, the cost of processing a method is \( O(C \times X \times (H + F)) \). Therefore, the cost of EICCFG construction is \( O(M \times C \times X \times (H + F)) \).

3. Case study

In this section, we explain and validate the method proposed in this paper through an example C++ program, denoted as P1. Figure 2 shows its code. The statements of P1 are labeled as 1 ~ 36.

From the analysis of above sections, we can conclude that exception propagation can change the original control flow and data flow of a program, so it influences control and data dependencies of programs. If the influence of exception propagation is not considered when perform the program slicing, it will be inaccurate. Currently, there is no effective method can solve the problems brought by the inter-class exception propagation. The proposed construction method of EICCFG in this paper can accurately describe the influences of inter-class exception propagation on the control flow of program. So it can be applied to the slicing of programs with exception-handling constructs. Therefore, in this section, we apply the proposed construction method to program slicing.

Program slicing is a technique introduced by Mark Weiser [15] that is useful for many applications, including program understanding, testing, debugging, and maintenance. A lot of scholars have carried on a large amount of research to it. In this paper, the discussion is focused on backward slicing. A backward program slicing contains all parts of the program that might directly or indirectly affect the program slicing criterion.

Given a program \( P \), program interest point \( g \) and variable \( s \), the slice \((s, g, P)\) is composed of parts of statements and predicates in program \( P \). The execution of these statements and predicate may influence the value of variable \( s \) in interest point \( g \), then \((g, s)\) is called the slicing criterion [15].

Given a slicing criterion \<(g, s)\> of a program \( P \), it is equivalent to \(<s, \text{Use}(s) \cup \text{Def}(s)>\). Here, Use \((s)\) is the representation of the use of all the variables on statement \( s \); Def \((s)\) represents the definition of all the variables on statements [15]. Because of the transitivity of dependency of the sequential program, inter-class slicing algorithm based on program dependency graph is essentially a graph reach ability problem.

The two-pass graph reach ability algorithm [3] is a slicing algorithm that efficiently computes slices on the SDGs. The two-pass graph reach ability algorithm works in two phases. For the backward slicing, during the first pass, the algorithm traverses backward along all edges, and marks all vertices that it can reach; then during the second pass, it traverses backward from all the vertices that were marked during the first pass along all edges except call edges, and again marks all vertices reached. Then the slice is the union of the vertices marked during pass one and pass two [3]. Thus with the same method, we can get the slicing of program which contains exception-handling constructs.

The example program P1 shown in Figure 2 is a program with exception-handling constructs. The program also illustrates the situation of dealing with inter-class function callings to construct EICCFG according to the three relationships of different classes respectively.

From the program P1, we can get the following information. There exists inheritance relationships between the class B/C and class A, class D is aggregate of class A and B, class D and C is association relationships. The control dependency information is represented with control flow graph; it is mainly reflected by path and the control relationship. In object-oriented program, control strategy is mainly used to describe intra-function (or intra-class) control dependency currently [7]. However, it actually...
can also describe the characteristics of inter-class control flow through the inter-class interaction and the interface calling. That is, the method involves the mutual information between the classes. Therefore, the computation of inter-class control dependency information is much more complex than the computation of inter-procedural.

![Figure 2. Source Code of Program P1](image)

However, exception propagation can cause the changes of some assignment vertexes of variables in the program, that is, the original define-use chains of the program are changed. For example, if there is no exception raised in the program, the assignment vertex on variable ‘g’ of statement 23 is in statement 16, and its value is 1; but if there raises an exception in statement 6, the assignment vertex on variable ‘g’ in statement 23 is in statement 22, and its value is 2.

On the other hand, exception propagation can reduce the define-use chains of the program. For example, if there is no exception raised in program P1, statement 13 is a assignment vertex on variable ‘z’, and statement 17 is its use vertex; but if there raises an exception in statement 6, the use vertex on ‘z’ of statement 17 would disappear.

According to the proposed approach, the EICCFG of the instance program P1 is shown in Figure 3. For the example program P1, we choose (23, g) as the slice criterion. The corresponding slice path is 23→22→21→19→18→16→6→5→15→14→12→10→1. Conversely, if we do not consider the particularity of throw and catch, only deal with them as general statements, its corresponding slice path is 23→22→21→20→19→18→16→8→7→6→5→15→14→12→11→10→1.

The slices on variable ‘g’ of the 23 line are the statements with + symbol in front, and it can be seen that the total number of these statements is 13; if we do not consider the particularity of throw and catch, only deal with them as general statements, the slices on variable ‘g’ of 23 line are the statements with stars in front, and the total number of these statements is 17. Therefore, the slice accuracy is improved by 23.5%.
4. Experiments

In this section, we validate the performance of our method through a group of experiments.

4.1. Experiments tool—CETool

The previous proposed approaches only deal with procedural programs; it is not suitable for object-oriented language in general. In this paper, we implement a static analysis tool (CETool) in order to analyze object-oriented programs with exception-handling constructs automatically, which is based on fully research of the exception handling mechanism of C++ programming language, careful analysis and comparison of the advanced software analysis and maintenance tools domestic and international. In particular, the tool is mainly suitable for C++ program. It not only can provide information on call/control/data dependencies, but also analyze the exception propagation information, including the local information and global information of exception handling constructs. Once we have determined a slice criterion of a program, and input it, then the tool can automate the Arabic numerals of the sliced statements.

The system is first to compile source code with GCC (GNU Compiler Collection) compiler and get intermediate results AST (Abstract Syntax Tree), on the basis of which to analyze the program. The AST generated by the GCC compiler source documents is the intermediate results through lexical, grammatical and semantic analysis, which is a text AST. Therefore, the analysis based on this can save the process of lexical, grammatical and semantic analysis, which can enhance the analysis efficiency and accuracy.

The system framework graph of CETool is shown in Figure 4. The input is the C++ source code. It first conduct the lexical, grammatical analysis to get AST documents; For the C++ program on Windows platform, it is transplanted to Linux platform to compile; then, standard the AST documents; furthermore, on the basis of standardized AST documents to conduct intra-procedural dependence analysis, intraprocedural exception-handling analysis, procedure call chain analysis, virtual function call analysis, class hierarchy structural analysis, interprocedural dependence analysis, as well as exception propagation analysis; next, we can obtain the exception propagation path and object-oriented system dependence information; finally, it is applied to the programs slicing and can obtain the slicing statements according to the slicing criterion.
4.2. Experiments description

To evaluate the effectiveness of the proposed approach and the developed tool, we have applied the approach to program slicing in some programs written in C++. The first system used for our experiment is the program P2 in Figure 5, which has more complex function invoking relations. The second is CppSQLite - a very thin C++ wrapper around the public domain SQLite database library, which is an open source system and is provided http://www.codeproject.com/KB/database/CppSQLite.aspx. The detailed descriptions of the two programs are shown in Table 1.

![System Framework Graph](image)

**Figure 4. System Framework Graph**

4.3. Results

Table 2 shows the results of the two systems in our experiments. Where, # slice statements represents the number of slice statements; # slice statements' represents the number of slice statements when do not consider the particularity of throw and catch, only deal with them as general statements.

For the example program P2, we choose the 61st line in 1.cpp as the slice criterion. The corresponding slice path is 61 (1.cpp) → 59 (1.cpp) → 52 (1.cpp) → 51 (1.cpp) → 50 (1.cpp)

However, if we not consider the particularity of throw and catch, only deal with them as general statements, its corresponding slice path is 61 (1.cpp) → 60 (1.cpp) → 57 (1.cpp) → 56 (1.cpp) → 54 (1.cpp) → 51 (1.cpp) → 50 (1.cpp)

Note: the statements without brackets represent those of in 2.cpp.

For the program P2 in Figure 5, when we choose the 61 line in 1.cpp as the slice criterion, the slices on the 61 line are the statements with + symbol in front, which are 29, 40, 47, 49, 50, 51, 52, 59, 61 in
1.cpp and 61, 67, 69, 71, 100, 103 in 2.cpp, the total number of these statements is 16; however, if we do not consider the particularity of throw and catch, only deal with them as general statements, the slice on the 61 line are the statements with stars in front, which are 29, 40, 47, 49, 50, 51, 54, 56, 57, 60, 61 in 1.cpp and 61, 67, 69, 70, 100, 103, 104, 106, 107 in 2.cpp, and the total number of these statements is 21. So slice accuracy is improved by 23.8%.

Similarly, for the program CppSQLite, we choose the 102 line in CppSQLite3DemoMT.cpp as the slice criterion. The corresponding slice are 102 (void ReadLockThreadProc (void* p) ) in (CppSQLite3DemoMT.cpp) → 77 (void ReadLockThreadProc (void* p) ) → 82 (void ReadLockThreadProc (void* p) ) → 83 (void ReadLockThreadProc (void* p) ) → 1120 → 1122 (CppSQLite3.cpp) → 1124 → 1127 → 88 → 1183 → 1185 → 1261 → 1263 → 1265 → 97 → 578 → 580 → 582 → 584 → 587; where, the statements without brackets represent those of in CppSQLite3.cpp. However, if we do not consider the particularity of throw and catch, only deal with them as general statements, the slice on the 102 line in CppSQLite3DemoMT.cpp are 102 (void ReadLockThreadProc (void* p) ) in (CppSQLite3DemoMT.cpp) → 77 (void ReadLockThreadProc (void* p) ) → 82 (void ReadLockThreadProc (void* p) ) → 83 (void ReadLockThreadProc (void* p) ) → 1120 → 1122 (CppSQLite3.cpp) → 1124 → 1127 → 88 → 1183 → 1185 → 1261 → 1263 → 1265 → 97 → 578 → 580 → 582 → 584 → 587. Similarly, the statements without brackets represent those of in CppSQLite3.cpp.

From the comparison results, we can find that the two results that consider exception-handling constructs and ignoring it are different: the proposed approach can bring about the improvement of the slicing accuracy through analyzing the influence of exception-handling constructs.

<table>
<thead>
<tr>
<th>No.</th>
<th>System</th>
<th>Slice criterion</th>
<th># Slice statements</th>
<th># Slice statements*</th>
<th>Improved accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>P2</td>
<td>61 line in CppSQLite3DemoMT.cpp</td>
<td>16</td>
<td>21</td>
<td>23.8</td>
</tr>
<tr>
<td>2</td>
<td>CppSQLite</td>
<td>102 line in CppSQLite3DemoMT.cpp</td>
<td>20</td>
<td>23</td>
<td>13.04</td>
</tr>
</tbody>
</table>
5. Related work

Many researchers have investigated the analysis of control dependency for the program with exception-handling constructs.

Ottenstein et al [11] introduced program dependence graphs in a program slicing technology, which simplified the complexity and improved the efficiency of program slicing. But the algorithm was fit for intra-procedural slicing that uses the program dependence graph (PDG). Although that algorithm was designed for procedural languages like C, the ideas apply to a subset of Java that excludes threads and exceptions. Choi et al [16] presented an efficient and precise modeling of exceptions for the analysis of Java programs. But it is merely a representation of a program's intraprocedural control flow.

From the above analysis, although the exception propagation is analyzed in those references, it is merely the intra-procedural techniques and algorithm of control dependency, which obviously can't meet the need of actual development of object-oriented software.

In addition, there are several other papers on inter-procedural techniques and algorithm of control dependency.

and computing slices on SDGs, that accommodates programs with arbitrary inter-procedural control flow.

Robillard [17] proposed a model that encapsulates the minimal concepts necessary for a developer to determine exception flow for object-oriented languages that define exception as objects. Sinha et al [1] presented a systematic and structured approach, for supporting software-engineering tasks, based on the static and dynamic analyses of constructs that cause implicit control flow. Jo [10] proposed a static analysis of Java programs that estimates their uncaught exceptions independently of the programmer’s declarations. Later, Fu et al [2] presented a static exception-chain analysis approach, which computed chains of semantically-related exception-flow links, and thus reported entire exception propagation paths, instead of just discrete segments of them.

In the above analysis, although the exception-handling constructs are included in the programs, it is merely the inter-procedural analysis, which obviously can't meet the need of actual development of object-oriented software. In addition, the exception analysis is not precise enough.

As for the analysis on inter-class control dependence for programs with exception-handling constructs, the literatures are relatively rare: Harrold et al [13] extended traditional system dependence graphs using class dependence graphs to make object-oriented C++ program can be represented, and then computed C++ program slicing using improved two-pass traversal graph accessibility algorithm. Later, they also presented techniques to construct representations for programs with explicit exception occurrences —exceptions that are raised explicitly through throw statements —and exception-handling constructs [12]. The empirical studies results are preliminary in that they did not indicate the actual differences in the control dependences; besides, they did not study the effects of the differences in control dependences on slicing.

Our approach can overcome the problems in the existing solutions above. Compared with intra-procedural analysis techniques, especially the existing interprocedural analysis methods in reference [12], the exact difference is that their method can not reflect the inter-class relationships, but the proposed method is able to. In addition, the proposed analysis method can show the effects of the differences in control dependences on the slicing techniques, which use control dependences [10].

6. Conclusions

The paper proposes an inter-class control-flow representation that takes into account exceptional flow of control, which is on the basis of studying the effects of the exception propagation on the dependency in the cluster-level testing of object-oriented programs. The approach takes account of constructors and destructors, which makes the analysis more accurate. According to the relationships of different classes, such as inheritance, aggregation and association, we incrementally construct an inter-class control flow graph and present an efficient algorithm. We also consider the effects of exceptions on analysis techniques- program slicing. The experiment results show that our approach brings about the improvement of the slice accuracy through analyzing the influence of exception-handling constructs.

It should be noted that our experimental tool is only suitable for C++ program; besides, we can only obtain control/data dependencies information but not ICCFG or SDGs. In our future study, we will do some further improve to the tool and apply it to Java programs to validate its effectiveness. In addition, how to efficiently visualizing the ICCFG and SDGs for a program is also our future research work.

7. Acknowledgment

This work was supported in part by awards from National Natural Science Foundation under 60970032, Natural Science Foundation of Jiangsu Province under BK2008124, Qing Lan Project and Graduate Training Innovative Projects Foundation of Jiangsu, China under CX10B_157Z.
8. References


