Constructing Control Flow Graph for Java by Decoupling Exception Flow from Normal Flow*

Jang-Wu Jo[1] and Byeong-Mo Chang[2]

[1] Department of Computer Engineering
Pusan University of Foreign Studies
Pusan 608-738, Korea
jjw@pufs.ac.kr

[2] Department of Computer Science
Sookmyung Women's University, Seoul 140-742, Korea
chang@sookmyung.ac.kr

Abstract. A control flow graph represents all the flows of control that may arise during program execution. Since exception mechanism in Java induces flows of control, exception induced control flow have to be incorporated in control flow graph. In the previous research to construct control flow graph, they compute exception flow and normal flow at the same time while computing control flow information. In this paper, we propose a method to construct control flow graph by computing separately normal flow and exception flow. We show that normal flow and exception flow can be safely decoupled, hence these two flows can be computed separately. We propose the analysis that estimates exception-induced control flow, and also propose exception flow graph that represents exception induced control flows. We show that a control flow graph can be constructed by merging an exception flow graph onto a normal flow graph.

Keywords: control flow graph, exception flow, normal flow

1 Introduction

A control flow graph(CFG) is a static representation of the program and represents all alternatives of control flow. The CFG is essential to performing many program-analysis techniques, such as data-flow and control-dependence analysis, and software-engineering techniques, such as program slicing and testings. For these program analyses and software engineering techniques to be safe and useful, the CFG should incorporate all the flows of control that may arise during execution of the program. Exception mechanism in Java may induce flow of control during program execution[1]. When an exception is thrown during program execution, an exception flow occurs from the statement that throws the exception to the handler block or the exit of the main method. So, these exception flow must be incorporated in CFG.

* This work was supported by grant No. R01-2002-000-00363-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

Recently, several works on the effects of exception flow have been proposed. The first one is to construct CFG that incorporates exception flow[2].The second one is to modify the program analysis techniques [2,3,4] and software engineering technique in order to consider the effect of exception flow[2]. However, in constructing CFG in [2], they compute normal flow and exception flow at the same time. This is due to that Java program's normal flow and exception flow are mutually dependent.

In this paper, we propose a method to construct CFG by computing separately normal flow and exception flow. We investigated Java programs and found that such cases as normal flow and exception flow are mutually dependent rarely happened. This suggests that, in most cases, normal flow analysis can be done independent of exception flow analysis. This does not mean that we don't guarantee the safety of these two flow analyses. For such cases when these two flows are mutually dependent, we use the type information instead of the result of exception flow analysis, believing that this approximation would be rarely detrimental to the accuracy of normal flow analysis. We propose an *exception flow analysis* that estimates exception-induced control flow, and also propose *exception flow graph* that represents exception-induced control flow. The CFG that represents both normal and exception flow can be constructed by merging an exception flow graph onto a normal flow graph (CFG with only normal flow).

The advantages of decoupling these two flow analyses are two folds. The first one is that when only one flow information (normal flow or exception flow) is needed, the desired flow can be computed solely, instead of computing two flows at the same time. The second one is that we can use already existing normal flow graph that is constructed by a former technique. Moreover, since our exception flow graph represents the information about thrown exceptions, such as origins, handlers, and propagation paths, it can be used to guide programmers to put handlers at appropriate places.

The rest of this paper is organized as follows. Section 2 exemplifies the mutual dependence between normal control flow analysis and exception flow analysis, and describes how these two analyses can be decoupled. Section 3 describes a static analysis to compute exception flow of Java programs. Section 4 describes how to construct control flow graph that incorporates exception flow. Section 5 contains our conclusions.

2 Decoupling Exception Flow Analysis from Normal Flow Analysis

Normal flow analysis and exception flow analysis are mutually dependent: computing normal flow requires the information on exception flow, and computing exception flow also requires the information on normal flow.

The Java code in Figure 1 illustrates this situation where these two analyses are mutually dependent. Consider the call to m() in line 1. The call may induce an exception flow because exceptions may be propagated by the called method m(). The flow of propagated exceptions is in the reverse order of method call chain. So, computing the flow induced by propagated exceptions requires the method call graph, which is a result of normal flow analysis. For the situation where computing normal control flows requires the information on exception induced control flows, consider the call to m() in line 3 which uses the catch parameter `x`. The method m() may be overridden in its subclasses. In order to determine which m() among overridden methods may be called during execution, the type of the exception that are caught by catch block is required, which is a result of exception flow analysis.

```
  try {
1:   x.m( );
2:} catch (Exception x) {
3:   e.m();
  }
```

Fig. 1. An example code

We conducted a study of frequency with which the case of mutual dependence between these two flows appeared in real Java programs. We examined a suite of fourteen Java programs, which covers a wide range of application areas, including language processors, a compression utility, an artificial intelligence system, a simulation utility, and a servlet container. We found that 0.3% of method calls in catch block require the exception flow information. Thus we can separate exception flow analysis from normal flow analysis. This does not mean that we don't guarantee the safety of normal flow analysis. For such cases when the information of exception flow is required, the type information of catch parameter is used instead. We believe that this approximation would be rarely detrimental to the accuracy of normal flow analysis.

3 Exception Flow Analysis

This section presents an exception flow analysis which estimates exception induced control flows of Java programs. Our analysis is based on set-based framework[5], which consists of three phases: designing construction rules, collecting set constraints and solving them.

3.1 Source Language

As will be explained in 3.2, our exception flow analysis collects set constraints at exception related constructs, such as method declaration, `throw` statement `try-catch` statement, and method call. For presentation brevity we define abstract syntax of these constructs and their semantics are same as in [1].

[Throw]	$\texttt{throw}\ e$
[Try-catch]	$\texttt{try}\ \mathit{block}\ \texttt{catch}\ \texttt{(}c\ x\texttt{)}\ \mathit{block}$
[MethCall]	$e\texttt{.}m\texttt{(}e\texttt{)}$
[MethDecl]	$m(x) = \mathit{block}\ \ [\texttt{throws}\ c^*]$

3.2 Set Constraints

Each set constraint is of the form $\mathcal{X} \supseteq se$ where $\mathcal{X}$ is a set variable and se is a set expression. The meaning of a set constraint $\mathcal{X} \supseteq se$ is intuitive: set $\mathcal{X}$ contains the set represented by set expression se. Multiple constraints are conjunctions. We write $\mathcal{C}$ for such conjunctive set of constraints.

In case of our analysis, the set expression is of this form:

$$
\begin{array}{rll}
se \rightarrow & \mathcal{X} & \text{set varaible} \\
& \mid se \cup se & \text{union} \\
& \mid \langle c, \ell \rangle & \text{thrown exception from } \ell \\
& \mid se - \{c_1, ..., c_n\} & \text{exceptions escaping from } \texttt{try-catch} \\
& \mid se \cap \{c_1, ..., c_n\} & \text{exceptions caught by } \texttt{catch}\text{-block} \\
& \mid se \cdot \ell & \text{exception propagation}
\end{array}
$$

The thrown exception from a `throw` statement labelled with ℓ is represented by $\langle c, \ell \rangle$ where c is the class name of the exception. The set expression $se - \{c_1, ..., c_n\}$ is for representing the exceptions that escape from `try-catch` statement. The set expression $se \cap \{c_1, ..., c_n\}$ is for representing the exceptions that is caught by `catch` block. The set expression $se \cdot \ell$ records an exception propagation path by appending a label ℓ to se.

The formal semantics of set expressions is defined by an interpretation $\mathcal{I}$ that maps from set expressions to sets of values in $V = ExnName \times Trace$, where $ExnName$ is the set of exception names, and $Trace = Label^*$. A trace $\tau \in Trace$ is a sequence of labels in $Label$, which is an exception propagation path.

$$
\begin{array}{l}
\mathcal{I}(se \cdot \ell') = \mathcal{I}(se) \cdot \ell' \text{ where } \mathcal{I}(se) \cdot \ell' = \{\langle c, \ell_1 \cdots \ell_n \ell' \rangle | \langle c, \ell_1 \cdots \ell_n \rangle \in \mathcal{I}(se)\} \\
\mathcal{I}(se - \{c_1 \cdots c_n\}) = \{\langle c, \tau \rangle | \langle c, \tau \rangle \in se, c \notin \{c_1 \cdots c_n\}\} \\
\mathcal{I}(se \cap \{c_1 \cdots c_n\}) = \{\langle c, \tau \rangle | \langle c, \tau \rangle \in se, c \in \{c_1 \cdots c_n\}\}
\end{array}
$$

We call an interpretation $\mathcal{I}$ a *model* (a solution) of a conjunction $\mathcal{C}$ of constraints if, for each constraint $\mathcal{X} \supseteq se$ in $\mathcal{C}$, $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se)$.

3.3 Set Constraints Construction

The basic idea of our exception flow analysis is that exception flows are traced by recording the labels of statements that exceptions are propagated through. The constructs that we record the labels of are `throw` , `catch` block of `try-catch`, and method declarations, which are necessary for constructing CFG. We assume this kind of constructs s has a label ℓ, which is denoted by $\ell : s$.

Our analysis increases the cost-effectiveness by enlarging the analysis granularity. Instead of defining a set variable for each statement or expression, our analysis defines a set variable for each methods and `try` blocks. For each method m, the set variable $\mathcal{X}_m$ represents the flows of exceptions escaping from method m. For each `try` block b_1 of `try` b_1 `catch` $(c\ x)$ b_2, the set variable $\mathcal{X}_{b_1}$ represents the flows of exceptions escaping from `try` block b_1. This approach of enlarging the analysis granularity is addressed in [6] and is applied to uncaught exception analysis successfully [7].

Figure 2 has the rules to generate set-constraints. The left-hand-side m in relation $m \triangleright s\ : \mathcal{C}$ indicates that a method or a try-block m contains the statement

$$[\text{Throw}]_m \quad \frac{m \rhd e : \mathcal{C}_1}{m \rhd \ell : \texttt{throw}\, e : \{\mathcal{X}_m \supseteq \langle c, \ell \rangle,\ c = Class(e)\} \cup \mathcal{C}_1}$$

$$[\text{Try-catch}]_m \quad \frac{m \rhd b_1 : \mathcal{C}_{b_1} \quad m \rhd b_2 : \mathcal{C}_{b_2}}{m \rhd \texttt{try}\, b_1\ \ell : \texttt{catch}(c\ x)\ b_2 : \{\mathcal{X}_{b_1} \supseteq (\mathcal{X}_{b_1} \cap \{c\}^*) \cdot \ell,\ \mathcal{X}_m \supseteq (\mathcal{X}_{b_1} - \{c\}^*)\} \cup \mathcal{C}_{b_1} \cup \mathcal{C}_{b_2}}$$

$$[\text{MethCall}]_m \quad \frac{m \rhd e_1 : \mathcal{C}_1 \quad m \rhd e_2 : \mathcal{C}_2}{m \rhd e_1.m'(e_2) : \{\mathcal{X}_m \supseteq \mathcal{X}_{c.m'} | c \in Class(e_1), m'(x) = e_{m'} \in c\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{MethDecl}]_m \quad \frac{m \rhd b : \mathcal{C}_m}{m \rhd \ell : m(x) = b :\ \{\mathcal{X}_m \supseteq \mathcal{X}_m \cdot \ell\} \cup \mathcal{C}_m}$$

Fig. 2. Set-constraints construction rules

s the constraints $\mathcal{C}$ are generated from statement s. The $class(e)$ represents the classes that the expression e's object belongs to, which is the result of normal control flow analysis.

Consider the rule $[\text{Throw}]_m$. It throws an exception e, which is represented as $\langle c, \ell \rangle$ where $c = class(e_1)$ is the class name of the exception and l is the label of the `throw` statement, which is an origin of the exception.

Consider the rule $[\text{Try-catch}]_m$. Among the exceptions escaping from `try` block b_1, the same class or subclasses of class c in `catch`$(c\ x)$ are caught. The label l of `catch`$(c\ x)$ is appended to the flows of caught exceptions in order to record the flow to the exception handler. Hence, $\mathcal{X}_{b_1} \supseteq (\mathcal{X}_{b_1} \cap \{c\}^*) \cdot \ell$, where $\{c\}^*$ represents all the subclasses of a class c including itself. The exceptions escaping from `try-catch` statement have to be contained in the set variable of the method or `try` block that contains this statement. Hence, $\mathcal{X}_m \supseteq (\mathcal{X}_{b_1} - \{c\}^*)$

Consider the rule $[\text{MethCall}]_m$. The method m'`(`x`) = `$e_{m'}$ is declared inside the classes $c \in class(e_1)$ of e_1's objects. Hence, $\mathcal{X}_m \supseteq \mathcal{X}_{c.m'}$ for uncaught exceptions. (The subscript $c.m$ indicates the index for the method m of class c.)

Consider the rule $[\text{MethDecl}]_m$. The set variable $\mathcal{X}_m$ includes the uncaught exceptions from the method m. The label l of method m is appended to the flows of uncaught exceptions in order to record the exceptions propagate through the method m.

```
1: public static void main() throws E2
     try {
2:      m1( );
3:   } catch (Exception x) {
4:      ;
     }
5:   m3( );
  }
6: void m1( ) throws E1{
7:   m2( );
  }

8: void m2( ) throws E1{
9:   if(...)
10:      throw new E1();
11: void m3( ) throws E2 {
12:   if (...)
13:      throw new E2();
14:   if (...)
15:      m3( );
    }
```

Fig. 3. An example program for exception propagation

Example 1. We can construct a collection $\mathcal{C}$ of constraints by applying the construction rules in Figure 2 to a program in Figure 3.

$$\begin{array}{lll}
\mathcal{X}_{main} \supseteq \mathcal{X}_{try} - \{Exception\}^*, & \mathcal{X}_{m1} \supseteq \mathcal{X}_{m2}, & \mathcal{X}_{m3} \supseteq \{\langle E2, 13\rangle\}, \\
\mathcal{X}_{try} \supseteq (\mathcal{X}_{try} \cap \{Exception\}^*) \cdot 3, & \mathcal{X}_{m1} \supseteq \mathcal{X}_{m1} \cdot 6, & \mathcal{X}_{m3} \supseteq \mathcal{X}_{m3}, \\
\mathcal{X}_{try} \supseteq \mathcal{X}_{m1}, & \mathcal{X}_{m2} \supseteq \{\langle E1, 10\rangle\}, & \mathcal{X}_{m3} \supseteq \mathcal{X}_{m3} \cdot 11, \\
\mathcal{X}_{main} \supseteq \mathcal{X}_{main} \cdot 1, & \mathcal{X}_{m2} \supseteq \mathcal{X}_{m2} \cdot 8 &
\end{array}$$

3.4 Solving the Set Constraints

A collection $\mathcal{C}$ of constraints for a program guarantees the existence of its least solution (model) because every operator is monotonic (in terms of set-inclusion) and each constraint's left-hand-side is a single variable [5]. We write $lm(\mathcal{C})$ for the least model of a collection $\mathcal{C}$. The least model can be computed by iterative fixedpoint method because the solution space is finite: exception classes and labels in the program.

Example 2. We can compute the solution $lm_S(\mathcal{C})$ of set-constraints $\mathcal{C}$ in *Example 1* by iterative fixpoint method.

$$\begin{array}{llll}
\{\ \mathcal{X}_{main} \supseteq \{\langle E2, 13\cdot 11\cdot 1\rangle\}, & \mathcal{X}_{try} \supseteq \{\langle E1, 10\cdot 8\cdot 6\cdot 3\rangle\}, & \mathcal{X}_{m1} \supseteq \{\langle E1, 10\cdot 8\cdot 6\rangle\}, \\
\ \ \mathcal{X}_{m2} \supseteq \{\langle E1, 10\cdot 8\rangle\}, & \mathcal{X}_{m3} \supseteq \{\langle E2, 13\cdot 11\rangle\} & \}
\end{array}$$

Theorem 1. *Let P be a program and $\mathcal{C}$ be the set-constraints constructed by the rules in Figure 2. Every exception trace of P is included in the solution $lm_S(\mathcal{C})$.*
Proof sketch. We first have to lift the standard semantics to a collecting semantics called set-based approximation so as to collect sets of concrete traces, because a static program point can be associated with a set of traces. Correctness proofs can be done with respect to this collecting semantics by the fixpoint induction over the continuous functions that are derived from our constraint system as in [8]. □

We can see exception flow by defining the *exception flow graph* of the solution $lm_S(\mathcal{C})$.

Definition 1. Let $\mathcal{C}$ be the set-constraints constructed for a program P. *Exception flow graph* of the solution $lm_S(\mathcal{C})$ is defined to be a graph $\langle V, E\rangle$ where V is the set of labels in P and $E = \{\ell_1 \rightarrow^c \ell_2, \ell_2 \rightarrow^c \ell_3, \cdots, \ell_{n-1} \rightarrow^c \ell_n | \langle c, \ell_1\ell_2\cdots\ell_n\rangle \in lm_S(\mathcal{C})(\mathcal{X})$ for a set variable $\mathcal{X}$ in $\mathcal{C}\}$ where $\ell_i \rightarrow^c \ell_{i+1}$ denotes an edge from ℓ_i to ℓ_{i+1} labelled with c.

4 Construction of CFG

A CFG which includes both normal flow and exception flow can be constructed by merging exception flow graph onto normal flow graph. We show the construction of CFG by using example Java program in Figure 3. The statement-level normal flow graph for Figure 3 is shown in Figure 5. The normal flow graph in

Figure 5 does not represent exception flow yet. As in the Figure 5, there are two exception flow paths, which are caused by a throw statement: the path starting from node 10 and node 13 (nodes that are shown as double circles in this figure). The exception flow graph for the example program is shown in Figure 4. We label each edge with the type of exception. By merging exception flow graph in Figure 4 onto normal flow graph in Figure 5, we can construct CFG in Figure 6 which incorporates both normal flow and exception flow. The CFG in Figure 6 contains exceptional-exit node, to model the propagation of exceptions between methods. An exceptional-exit node represents the propagation of an exception of type T by the corresponding method.

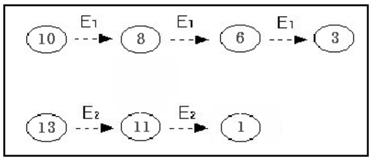

Fig. 4. Exception Flow Graph

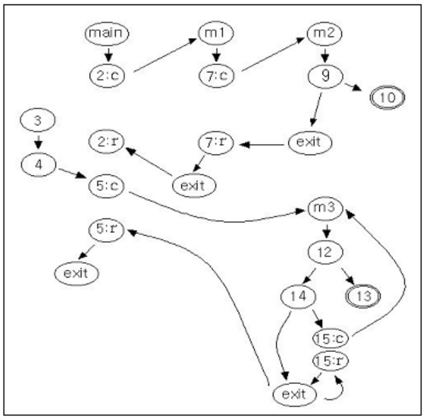

Fig. 5. Normal Flow Graph

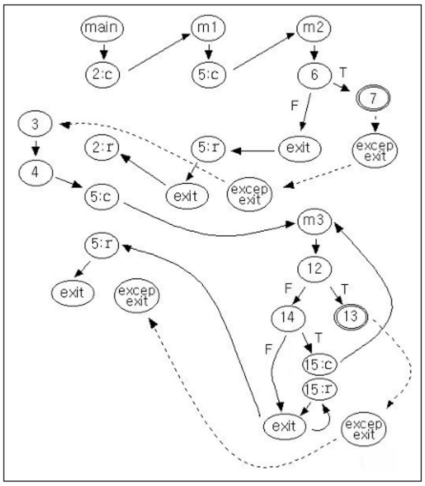

Fig. 6. Control Flow Graph

5 Conclusions

The contributions of this paper are two-folds. First, we showed that while computing control flow information, normal flow and exception flow can be computed separately, and also showed that the approximation from this separation is not detrimental to the accuracy of each flow analysis. Second, We presented an analysis that estimates exception-induced control flow, and also proposed exception flow graph that represents exception induced control flows. We showed that a control flow graph can be constructed by merging an exception flow graph onto a normal flow graph.

References

1. J. Gosling, B. Joy, and G. Steele, *The Java Programming Language Specification*, Addison-Wesley Longman, 1996.
2. S. Sinha and M. Harrold, Analysis and Testing of Programs With Exception-Handling Constructs, *IEEE Transations on Software Engineering* vol. 26, no. 9, pp. 849-871, 2000.
3. R. K. Chatterjee, B. G. Ryder, and W. A. Landi, Complexity of concrete type-inference in the presence of exceptions, *Lecture notes in Computer Science*, vol. 1381, pp. 57-74, Apr. 1998.
4. J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, Efficient and precise modeling of exceptions for analysis of Java programs, *Proceedings of '99 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 21-31, Sep. 1999.
5. N. Heintze, Set-based program analysis. Ph.D thesis, Carnegie Mellon University, 1992.
6. Jang-Wu Jo, B.-M. Chang, Granularity of Constrain-Based Analysis for Java, *Proceedings of ACM SIGPLAN Conference on Principles and Pracice of Declarative Programming*, pp. 94-102, Sep. 2001.
7. Jang-Wu Jo, B.-M. Chang, K. Yi, and K. Choe, An Uncuaght Exception Analysis for Java, *Journal of Systems and Software*, accepted for publication.
8. Patrick Cousot and Radhia Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. *Lecture Notes in Computer Science*, volume 939, pp. 293-308. Springer-Verlag, *Proceedings of the 7th international conference on computer-aided verification*, 1995.