예외 흐름 분석을 정상 흐름 분석과 분리하여 Java프로그램에 대한 제어 흐름 그래프를 생성하는 방법

(A Method to Construct Control Flow Graphs for Java Programs by Decoupling Exception Flow Analysis from Normal Flow Analysis)

조 장 우 † 창 병 모 ††
(Jang-Wu Jo) (Byeong-Mo Chang)

요 약 제어 흐름 그래프는 자료 흐름 분석과 제어 종속 분석과 같은 프로그램 분석 분야와 프로그램 슬라이싱과 테스팅과 같은 소프트웨어공학 분야에서 필요로 하는 정보이다. 이러한 분석들이 안전하고 유용하기 위해서는 제어 흐름 그래프는 예외 흐름을 포함해야 한다. 기존의 방법은 예외 흐름과 정상 흐름의 상호 의존적인 관계로 인해 두 흐름을 동시에 계산하면서 제어 흐름 그래프를 생성한다. 그러나 실제 Java 프로그램을 조사해 본 결과 두 흐름이 상호 의존적으로 필요한 경우는 거의 발생하지 않음을 알 수 있었다. 그러므로 정상 흐름과 예외 흐름을 분리해서 계산할 수 있음을 알았고, 예외 흐름을 계산하는 예외 흐름 분석을 제안한다. 그리고 예외 흐름을 표현하는 예외 흐름 그래프를 제안한다. 그리고 제어 흐름 그래프는 예외 흐름 그래프와 정상 흐름 그래프를 합병함으로써 생성될 수 있음을 보인다.

키워드 : 제어 흐름 그래프, 제어 흐름 분석, 예외 흐름, 정상 흐름

Abstract Control flow graph is used for 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 analyses to be safe and useful, the CFG should incorporate the exception flows that are induced by exceptions. In previous research to construct control flow graph, normal flows and exception flows are computed at the same time, since these two flows are known to be mutually dependent. By investigating realistic Java programs, we found that the cases when these two flows are mutually dependent rarely happen. So, we can decouple exception flow analysis from normal flow analysis. In this paper we propose an analysis that estimates exception flows. We also propose exception flow graph to represent exception flows. And we show that the control flow graph that accounts for exception flows can be constructed by merging exception flow graph onto normal control flow graph.

Key words : control flow graph, control flow analysis, exception flow, normal flow

1. 서 론

제어 흐름 그래프(Control Flow Graph)는 프로그램의 실행 중에 발생 가능한 제어 흐름을 표현하는 방법이다. 제어 흐름 그래프는 프로그램 분석과 소프트웨어 공학 분야에서 필요한 정보이다. 예를 들어, 자료 흐름 분석(data flow analysis), 제어 종속 분석(control dependence analysis), 예외 분석(exception analysis), 리그레션 테스팅(regression testing), 프로그램 슬라이싱, 테스트 데이터 생성 등을 위해서 제어 흐름 정보를 필요로 한다. 이러한 프로그램 분석들과 소프트웨어 공학에서의 작업들이 안전하고 정확하기 위해서는 제어 흐름 그래프가 실행 중에 발생 가능한 모든 제어 흐름들을 포함해야 한다.

Java 언어는 예외 메커니즘(exception mechanism)을 지원하므로 프로그램 실행 중에 예외로 인한 제어 흐름이 발생한다. 그리고 Java 프로그램에서 예외와 관련된 구문이 자주 사용된다는 연구 결과가 있다. Ryder[1]와 Sinha[2]는 Java 프로그램에서 예외 관련 구문들의 사용 빈도에 대한 실험을 하였는데, 이 실험에 따르면 전

· 본 연구는 한국과학재단 목적기초연구(R01-2002-000-00363-0) 지원으로 수행되었음.
† 종신회원 : 부산외국어대학교 컴퓨터공학부 교수
jjw@pufs.ac.kr
†† 종신회원 : 숙명여자대학교 컴퓨터과학과 교수
chang@sookmyung.ac.kr
논문접수 : 2003년 9월 18일
심사완료 : 2004년 2월 20일

체 클래스들 중에서 23.3%가 try 구문을 사용하고, 24.5%가 throw 구문을 사용하고, 그리고 메소드들 중의 16%가 예외 관련 구문들을 사용함을 보였다. 이 실험으로 Java 프로그램에서 예외가 자주 사용되므로 예외로 인한 제어 흐름이 자주 발생함을 알 수 있었다.

최근에 자바 프로그램에 대한 분석들에 대해 예외의 효과를 고려하도록 하는 연구들이 진행되고 있다. 이 연구들은 두 가지로 구분할 수 있는데, 첫 번째는 예외의 효과를 고려하도록 기존의 분석을 수정 또는 재정의 하는 작업이고, 두 번째는 본 논문의 주제로 예외의 흐름을 포함한 제어 흐름 그래프를 생성하는 분야이다. 첫 번째 연구의 예로 Choi[3]는 예외를 고려한 자료 흐름 분석을 수행하였고, Ryder[4,5]는 예외를 고려한 points-to 분석과 자료 흐름 분석을 수행하였고, Sinha[2]는 예외를 고려한 제어 종속 분석과 슬라이싱을 제안하였다. 두 번째 연구의 예로 Sinha[2]는 예외 흐름을 포함하는 제어 흐름 그래프를 생성하는 방법을 제안하였다.

그러나 Sinha[2]의 방법은 정상 흐름과 예외 흐름을 동시에 계산하면서 제어 흐름 그래프를 생성한다. 두 흐름을 동시에 계산하는 이유는 두 흐름이 계산하는 과정에서 서로의 계산 결과를 필요로 하는 상호 의존적인 관계이기 때문이다. 그러나 실제 Java 프로그램을 조사해 본 결과 두 흐름이 상호 의존적으로 필요한 경우는 거의 발생하지 않음을 알 수 있었다. 그러므로 본 논문에서는 예외 흐름 분석을 정상 흐름 분석과 분리하여 Java에 대한 제어 흐름 그래프를 생성하는 방법을 제안한다. 두 흐름 분석을 분리해서 얻어지는 장점은 다음 두 가지이다. 첫 번째는 두 흐름 정보 중 하나의 정보만 필요한 경우 두 흐름을 동시에 계산해야 되는 대신 하나의 흐름만을 분리해서 구할 수 있다. 두 번째는 기존의 예외를 고려하지 않은 제어 흐름 그래프를 사용할 수 있다. 기존의 제어 흐름 그래프에 예외 흐름을 추가하여 예외 흐름을 포함한 제어 흐름 그래프를 생성할 수 있다.

본 논문의 구성은 다음과 같다. 먼저 2장에서는 정상 흐름과 예외 흐름간의 상호 의존성과 이 두 흐름의 계산을 안전하게 분리하는 근거와 과정을 기술한다. 3장에서는 예외 흐름 분석을 제안한다. 4장에서는 3장의 결과인 예외 흐름 정보를 이용하여 예외를 포함한 제어 흐름 그래프를 생성하는 과정을 예로 보이고, 5장에서 결론을 맺는다.

2. 정상 흐름 분석과 예외 흐름 분석의 분리

제어 흐름 분석은 프로그램의 각 문장 s에 대해서, s 다음에 수행될 수 있는 문장들을 결정하는 것이다. 정상 흐름은 기본적으로 프로그램의 쓰여진 순서대로 수행이 되고, 제어 구조를 바꾸는 구문은 조건문, 반복문, 그리고 메소드 호출 등이 있다. 예외 흐름은 예외를 발생시키는 throw 구문에서 시작한다. throw 문장 다음에 수행되는 문장은 발생된 예외를 처리하는 catch 블록이다. throw 문을 포함하는 try-catch 문장에서 발생된 예외의 catch 블록이 존재하면 해당 catch 블록이 수행되고, 존재하지 않으면 중첩된 try-catch 블록과 메소드 호출 관계의 역순으로 발생한 예외의 catch 블록을 찾아간다. catch 블록 다음에 수행되는 문장은 예외가 발생한 문장의 다음 문장이 아닌 수행된 catch 블록의 다음 문장이다.

정상 흐름 분석과 예외 흐름 분석은 상호 의존적인 관계로 인해 두 흐름을 분리해서 구하지 않고 동시에 구해 왔다[2]. 두 흐름의 상호 의존성이란 정상 흐름을 구하기 위해서 예외 흐름 정보를 필요로 하고, 역으로 예외 흐름을 구하기 위해서 정상 흐름 정보를 필요로 하는 것을 의미한다.

그림 1은 두 흐름 분석이 상호 의존적인 경우를 보여준다. 라인 1의 메소드 호출 x.m()은 예외 흐름 분석에서 정상 흐름 정보를 필요로 하는 경우이다. 호출된 메소드에서 처리되지 않은 예외가 전달된다면 예외 흐름이 발생한다. 처리되지 않는 예외가 전달되는 순서는 메소드 호출 순서의 역순이므로, 예외 흐름 분석 시 정상 흐름 정보인 메소드 호출 정보를 필요로 한다. 다음으로 정상 흐름 분석에서 예외 흐름 정보를 필요로 하는 경우는 라인 3의 메소드 호출 e.m()이다. 메소드 m()이 Exception 클래스의 하위 클래스들에서 재정의 되었다면, 정상 흐름 분석은 재정의된 메소드들 중에서 실행될 메소드를 결정하는 것이다. 이를 위해서는 catch 변수 e에 바인딩 되는 객체의 타입을 필요로 하는데, 이는 예외 흐름 정보로써 라인 2의 catch 블록에서 처리되는 예외의 타입이다.

그러나 실제 Java 프로그램을 대상으로 두 흐름간의 상호 의존적인 경우가 얼마나 자주 발생하는지에 대한 실험을 수행한 결과 상호 의존적인 경우가 아주 낮은 비율로 발생함을 알 수 있었다[7]. 실험은 14개의 다양한 크기(70 KByte~3.8 MByte)와 다양한 분야(언어처

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

그림 1 정상 흐름 분석과 예외 흐름 분석이 상호 의존적인 예

리기, 압축 프로그램, 인공지능 시스템, 서블릿, 그리고 시뮬레이션)의 Java 프로그램들을 대상으로 수행하였다. 실험 결과 catch 블록 내의 963개의 메소드 호출 중에서 0.3%인 3개의 메소드 호출만이 예외 흐름 정보를 필요로 함을 알 수 있었다.

두 흐름간의 상호 의존적인 경우가 없다면 두 흐름 분석을 분리 수행하더라도 동시에 수행한 결과와 같은 결과를 제공하는데, 실제 Java 프로그램에서 상호 의존적인 경우가 아주 적게 발생함으로 각 분석의 정확도(precision)에서 많은 근사(approximation)없이 두 흐름 분석을 분리할 수 있음을 알 수 있다. 위 실험에 사용된 Java 프로그램에 대해 두 분석을 분리한 경우 catch 블록내의 메소드 호출 중에서 0.3%인 3개의 메소드 호출에 대해서는 정상 흐름과 예외 흐름 분석에서 근사의 결과를 제공한다. 이는 예외 흐름 정보인 catch 변수에 바인딩되어 있는 예외 객체의 타입을 알지 못하므로, 메소드 호출에서 실행될 메소드가 재정의된 메소드들 중의 하나라고 결정할 수 없으므로 정상 흐름 분석에서 근사의 결과를 제공한다. 그리고 정상 흐름 정보가 근사 결과이므로 이 흐름을 따라서 발생한 예외 흐름에 대해서도 근사의 결과를 제공한다.

그러나 두 분석을 분리하더라도 각 분석의 안전성(safety)을 보장할 수 있다. 정상 흐름 분석에서 예외 흐름 정보를 필요로 하는 경우에는, catch 변수의 타입 정보를 이용함으로써 안전성을 보장할 수 있다. 예를 들어 위와 같이 예외 흐름 정보를 필요로 하는 메소드 호출의 경우, catch 변수에 바인딩되어 있는 객체의 타입 대신에 catch 변수의 타입을 적용하면, 메소드 호출에서 실행될 메소드가 catch 변수의 타입에서 정의된 메소드와 그 하위 타입에서 재정의된 메소드들이라는 결과를 제공한다. 이는 예외 흐름 분석과 정상 흐름 분석이 실행 중의 모든 가능한 흐름들을 포함하면 안전성이 보장되는 가능-분석(may-analysis) 이기 때문이다.

3. 예외 흐름 분석

3.1 ExnJava

본 연구에서 예외 흐름 분석의 명료한 설명을 위해 예외 처리와 관련된 자바 언어의 구문들로 구성된 자바 언어의 부분 언어인 가상의 자바 언어 ExnJava를 정의하였다. ExnJava에 대한 추상 구문(abstract syntax)은 그림 2에 정의되어 있고, 의미구조는 [8]에 정의된 것과 유사하므로 본 논문에서는 생략한다.

3.2 예외 흐름 분석 설계

본 연구에서는 집합-기반 분석[6]을 이용하여 예외 흐름 분석을 설계하였다. 집합-기반 분석을 이용한 분석의 설계는 집합-관계식을 생성하는 생성 규칙(construction rule)을 정의하는 것이다. 집합-기반 분석은 생성 규칙을 이용해서 집합-관계식들(set constraints)을 구성하는 단계와 집합-관계식을 만족하는 해(solution)를 구하는 두 단계로 구성된다.

P	::=	$C*$	program
C	::=	**class** c **ext** c {**var** $x*$ $M*$}	class definition
M	::=	$m(x)$ = e [throws c*]	method definition
e	::=	id	variable
	|	id := e	assignment
	|	**new** c	new object
	|	**this**	self object
	|	e ; e	sequence
	|	**if** e **then** e **else** e	branch
	|	**throw** e	exception raise
	|	**try** e **catch** (c x) e	exception handle
	|	$e.m(e)$	method call
id	::=	x	method parameter
	|	$id.x$	field variable
c			class name
m			method name
x			variable name

그림 2 ExnJava의 추상구문

예외 흐름 분석의 기본 아이디어는 발생된 예외에 대해 예외가 발생한 곳의 레이블과 예외가 전달되면서 거치는 구문들의 레이블들을 기록하는 것이다. 예를 들어 예외 e가 메소드 m에서 처리되지 않고 전달된다면 예외 e가 메소드 m으로 전달되면서 거쳐온 경로에 메소드 m의 레이블을 추가하는 것이다. 그러므로 예외 흐름 분석의 결과는 예외 이름 × Label+의 형태를 가지고, Label+는 레이블들의 연속으로 예외가 거쳐온 경로를 표현한다.

본 논문에서는 제어 흐름 그래프를 생성하는데 필요한 정보인 예외의 발생지, 예외의 처리기, 그리고 처리기를 찾는 동안 거쳐 온 메소드에 대한 정보를 구하기 위해서 throw 문, try-catch 구문의 catch 구문, 메소드 선언과 같은 구문들의 레이블들만을 기록한다. 만약 분석에 따라서 다른 예외 흐름 정보가 필요하면, 레이블을 기록하는 구문을 조절함으로써 필요한 예외 흐름 정보를 구할 수 있다. 예를 들어, 메소드 호출과 try 블록과 같은 구문들의 레이블을 기록한다면, 예외를 발생시키는 정확한 메소드 호출과 try 블록에 대한 정보를 알 수 있다.

그림 3은 ExnJava 프로그램의 각 구문에 대한 집합 관계식을 생성하는 생성 규칙들이다. 프로그램의 모든 구문 e에 대해 집합 관계식 $X_e \supseteq se$를 생성한다. X_e의 아래 첨자 e는 규칙이 적용되는 현재 구문을 의미한다. 관계식 "$e \triangleright C$"는 "집합 관계식 C가 구문 e로 부터 생성된다"는 의미이다. X_e는 구문 e안에서 처리되지 않은 예외의 흐름 정보를 나타내는 집합 변수이다. 집합

관계식 $X_e \supseteq se$에서 집합-식(set-expression) se의 형태는 발생한 예외의 경로를 기록하기 위해서 다음과 같은 형태이다.

$se \rightarrow X \mid se \cup se \mid \langle c, l\rangle \mid se \cdot l \mid se - \{c_1, ..., c_n\} \mid se \cap \{c_1, ..., c_n\}$

위의 집합-식에서 X는 집합 변수이고 $se \cup se$는 집합-식들 간의 합집합을 의미한다. 그리고 예외 이름과 레이블의 쌍의 형태인 집합-식 $\langle c, l\rangle$은 예외 c가 레이블 l에서 발생한다는 의미이다. 집합-식 se에 레이블 l을 연결하는 집합-식 $se \cdot l$은 집합 se의 원소에 레이블 l을 기록하는 것으로 의미는 다음과 같다. $se \cdot l = \{\langle c, l_1 \ldots l_n l\rangle \mid \langle c, l_1 \ldots l_n\rangle \in se\}$. 다음으로 집합-식 $se - \{c_1, ..., c_n\}$은 예외 이름과 레이블의 쌍들의 집합에서 예외 이름들의 집합의 차집합을 표현하는데 의미는 다음과 같다. $se - \{c_1, ..., c_n\} = \{p \mid p \in se, e\text{Name}(p) \notin \{c_1, ..., c_n\}$, 여기서 $e\text{Name}(\langle c, l_1 \ldots l_n\rangle) = c\}$. 마지막으로 집합-식 $se \cap \{c_1, ..., c_n\}$는 예외 이름과 레이블의 쌍들의 집합과 예외 이름들의 집합의 교집합을 표현하는데, 의미는 다음과 같다. $se \cap \{c_1, ..., c_n\} = \{p \mid p \in se, e\text{Name}(p) \in \{c_1, ..., c_n\}\}$.

그림 3의 주요 생성규칙에 대한 의미는 다음과 같다.

[Throw] $\dfrac{e_1 \triangleright C_1}{l : \text{throw } e_1 \triangleright \{X_e \supseteq \langle c, l\rangle \cup X_{e_1}\} \cup C_1}, c = class(e_1)$

예외를 발생시키는 throw 구문에서는 발생된 예외와 예외가 발생된 구문의 레이블을 나타내는 $\langle c, l\rangle$과 throw 구문을 실행하기 전에 e_1을 계산하는 과정에 예외가 발생할 수 있으므로 e_1으로 인해 발생한 예외와 예외의 흐름을 나타내는 X_{e1}을 포함하는 집합-관계식 $X_e \supseteq \langle c, l\rangle \cup X_{e_1}$을 생성한다. 여기서 $class(e_1)$은 e_1의 클래스 분석의 결과로 발생된 예외의 클래스 이름을 나타내고, l은 throw 구문의 레이블이다.

[Try] $\dfrac{e_0 \triangleright C_0 \quad e_1 \triangleright C_1}{\text{try } e_0\ l : \text{catch } (c_1\ x_1)\ e_1 \triangleright \{X_e \supseteq (X_{e_0} - \{c_1\}^*) \cup X_{e_1},\ X_{e_0} \supseteq (X_{e_0} \cap \{c_1\}^*) \cdot l\} \cup C_0 \cup C_1}$

try 블록 e_0에서 발생한 예외 중에서 catch 구문에서 처리되는 예외는 catch 구문으로의 흐름을 발생시키므로 catch 구문의 레이블인 l을 추가하는 집합-관계식 $X_e \supseteq (X_{e_0} \cap \{c_1\}^*) \cdot l$ 을 생성한다. 여기서 $X_{e_0} \cap \{c_1\}^*$은 try 블록 e_0에서 발생한 예외 중에서 catch 구문에서 처리되는 예외를 의미하고, $\{c\}^*$은 c자신을 포함한 모든 하위 클래스들을 나타낸다. 그리고 try-catch 구문에서 처리되지 않는 예외는 try-블록에서 발생하여 catch 구문에서 처리되지 않는 예외와 catch 블록 e_1에서 발생한 예외이므로 이를 위한 집합-관계식 $X_e \supseteq (X_{e_0} - \{c_1\}^*) \cup X_{e_1}$을 생성한다. 본 논문에서는 try-catch 구문에 대해서는 레이블을 기록하지 않으므로 이 구문을 위한 집합-관계

[New] $\text{new } c \triangleright \emptyset$

[FieldAss] $\dfrac{e_1 \triangleright C_1}{id.x := e_1 \triangleright \{X_e \supseteq X_{e_1}\} \cup C_1}$

[ParamAss] $\dfrac{e_1 \triangleright C_1}{x := e_1 \triangleright \{X_e \supseteq X_{e_1}\} \cup C_1}$

[Seq] $\dfrac{e_1 \triangleright C_1 \quad e_2 \triangleright C_2}{e_1 ; e_2 \triangleright \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup C_1 \cup C_2}$

[Cond] $\dfrac{e_0 \triangleright C_0 \quad e_1 \triangleright C_1 \quad e_2 \triangleright C_2}{\text{if } e_0 \text{ then } e_1 \text{ else } e_2 \triangleright \{X_e \supseteq X_{e_0} \cup X_{e_1} \cup X_{e_2}\} \cup C_0 \cup C_1 \cup C_2}$

[FieldVar] $\dfrac{id \triangleright C_{id}}{id.x \triangleright C_{id}}$

[Throw] $\dfrac{e_1 \triangleright C_1}{l : \text{throw } e_1 \triangleright \{X_e \supseteq \langle c', l\rangle \cup X_{e_1}\} \cup C_1}\ c = class(e_1)$

[Try] $\dfrac{e_0 \triangleright C_0 \quad e_1 \triangleright C_1}{\text{try } e_0\ l : \text{catch } (c_1\ x_1)\ e_1 \triangleright \{X_e \supseteq (X_{e_0} - \{c_1\}^*) \cup X_{e_1},\ X_{e_0} \supseteq (X_{e_0} \cap \{c_1\}^*) \cdot l\} \cup C_0 \cup C_1}$

[MethCall] $\dfrac{e_1 \triangleright C_1 \quad e_2 \triangleright C_2}{e_1.m(e_2) \triangleright \{X_e \supseteq X_{c.m} \mid c \in \text{Class}(e_1), m(x) = e_m \text{ is a method of class } c\} \cup \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup C_1 \cup C_2}$

[MethDef] $\dfrac{e_m \triangleright C}{l : m(x) = e_m \triangleright \{X_m \supseteq X_{e_m} \cdot l\} \cup C}$

[ClassDef] $\dfrac{m_i \triangleright C_i \quad i = 1, \ldots, n}{\text{class } c = \{\text{var } x_1, \ldots, x_k, m_1, \ldots, m_n\} \triangleright C_1 \cup \ldots \cup C_n}$

[Program] $\dfrac{c_i \triangleright C_i \quad i = 1, \ldots, n}{c_1, \ldots, c_n \triangleright C_1 \cup \ldots \cup C_n}$

그림 3 예외 흐름 분석을 위한 구문 단위의 생성 규칙

식에는 레이블을 기록하는 식은 없다.

[MethCall] $\dfrac{e_1 \triangleright C_1 \quad e_2 \triangleright C_2}{e_1.m(e_2) \triangleright \{X_e \supseteq X_{c.m} \mid c \in \text{Class}(e_1), m(x) = e_m \text{ is a method of class } c\} \cup \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup C_1 \cup C_2}$

메소드 호출 $e_1.m(e_2)$에 대한 처리되지 않는 예외들은 구문 e_1 및 e_2에서 처리되지 않는 예외들과 호출된 메소드 m을 통해서 전달된 예외들이므로 집합-관계식 $X_e \supseteq X_{e_1} \cup X_{e_2}$과 예외 분석의 결과인 $X_{c.m}$을 포함하는 집합-관계식 $X_e \supseteq X_{c.m}$을 생성한다. 본 논문에서는 메소드 호출 구문에 대한 레이블은 기록하지 않으므로, 집합-관계식 내에 레이블을 기록하는 식은 없다.

[MethDef] $\dfrac{e_m \triangleright C}{l : m(x) = e_m \triangleright \{X_m \supseteq X_{e_m} \cdot l\} \cup C}$

메소드 m에서 처리되지 않은 예외는 메소드 m을 거치게 되므로, 처리되지 않는 예외의 경로에 이 구문의 레이블인 l을 추가하는 집합-관계식 $X_{c.m} \supseteq X_{e_m} \cdot l$을 생성한다.

3.3 분석 단위의 조절

3.2절에서 설계된 예외 흐름 분석은 각 구문마다 하나의 집합 변수를 사용하는 구문-단위의 분석이다. 구문-단위의 분석은 이론적으로는 정확한 분석을 제공하지만 집합 변수의 수가 구문의 수만큼 필요하므로 분석 속도 면에서 실용적이지 못한 것으로 알려져 있다[9]. 집합 기반 분석의 시간 복잡도는 집합 변수의 수가 N이라고 할 때 N^3이므로 실제적인 크기가 큰 자바 프로그램의 분석에는 적절하지 않다. 이러한 문제를 해결하기 위해서 분석 단위를 조절하는 연구가 진행되어 왔다[9-12]. 따라서 본 절에서는 분석 속도 면에서 실용적인 분석을 위해서 메소드와 try 블록에만 집합-변수를 정의하여 집합-변수들의 수를 줄임으로써 실용적인 예외 흐름 분석기를 설계한다. 이와 같이 분석의 단위를 조절하는 기법은 [11,13,14]의 처리되지 않는 예외의 분석에 성공적으로 적용된 바 있다.

그림 4는 그림 3의 생성규칙을 메소드와 try 블록 단위의 분석으로 변환한 생성 규칙이다. 그림 4에서는 각 메소드 또는 try-블록 m에서 처리되지 않는 예외에 대한 흐름 정보를 위한 집합 변수 X_m를 정의하고 $X_m \supseteq se$ 형태의 집합-관계식을 생성한다. 그림 4에서 $m \triangleright e$는 구문 e가 메소드 또는 try 블록 m 내에 포함되어 있는 구문이라는 것을 의미한다.

그림 4의 생성 규칙들을 적용하면 그림 3과 비교해서 생성되는 생성규칙들의 수가 적음을 알 수 있다. 이는 각 구문 e와 구문 e를 구성하는 부분 구문 e_0, e_1, e_2에 대한 집합 변수가 따로 정의되지 않고 그 구문을 포함하고 있는 메소드 또는 try-블록의 집합 변수를 사용함으로써 생성되는 집합-관계식이 생략 가능하기 때문이다. 예를 들어 구문 e가 if e_0 then e_1 else e_2이고 메소드 m에 포함되어 있다고 가정할 때, 구문 e에 대해 그림 3의 생성 규칙을 적용하면 집합-관계식 $X_e \supseteq X_{e_0} \cup X_{e_1} \cup X_{e_2}$이 생성되나, 그림 4의 규칙을 적용하

[New]m	$m \triangleright \text{new } c : \emptyset$
[FieldAss]m	$\dfrac{m \triangleright e_1 : C_1}{m \triangleright id.x := e_1 : C_1}$
[ParamAss]m	$\dfrac{m \triangleright e_1 : C_1}{m \triangleright x := e_1 : C_1}$
[Seq]m	$\dfrac{m \triangleright e_1 : C_1 \quad m \triangleright e_2 : C_2}{m \triangleright e_1 ; e_2 : C_1 \cup C_2}$
[Cond]m	$\dfrac{m \triangleright e_0 : C_0 \quad m \triangleright e_1 : C_1 \quad m \triangleright e_2 : C_2}{m \triangleright \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : C_0 \cup C_1 \cup C_2}$
[FieldVar]m	$\dfrac{m \triangleright id : C_{id}}{m \triangleright id.x : C_{id}}$
[Throw]m	$\dfrac{m \triangleright e_1 : C_1}{m \triangleright l : \text{throw } e_1 : \{X_m \supseteq \langle c, l \rangle\} \cup C_1} \; c = class(e_1)$
[Try]m	$\dfrac{m \triangleright e_g : C_g \quad m \triangleright e_1 : C_1}{m \triangleright \text{try } e_g \; l : \text{catch } (c_1 \; x_1) \; e_1 : \{X_m \supseteq (X_g - \{c_1\}^*), \; X_g \supseteq (X_g \cap \{c_1\}^*) \cdot l\} \cup C_g \cup C_1}$
[MethCall]m	$\dfrac{m \triangleright e_1 : C_1 \quad m \triangleright e_2 : C_2}{m \triangleright e_1.m'(e_2) : \{X_m \supseteq X_{c.m'} \mid c \in Class(e_1), m'(x) = e_{m'} \text{ is a method of class } c\} \cup C_1 \cup C_2}$
[MethDef]m	$\dfrac{m \triangleright e_m : C_m}{l : m(x) = e_m : \{X_m \supseteq X_{e_m} \cdot l\} \cup C_m}$
[ClassDef]m	$\dfrac{m_i : C_i, \quad i = 1, \ldots, n}{\text{class } c = \{\text{var } x_1, \ldots, x_k, m_1, \ldots, m_n\} : C_1 \cup \ldots \cup C_n}$
[Program]m	$\dfrac{\triangleright C_i : C_i, \quad i = 1, \ldots, n}{\triangleright C_1, \ldots, C_n : C_1 \cup \ldots \cup C_n}$

그림 4 예외 흐름 분석을 위한 메소드와 try-블록 단위의 생성 규칙

면 구문 e, e_0, e_1, e_2에 대한 집합 변수가 X_f이므로 집합-관계식 $X_f \supseteq X_f \cup X_f \cup X_f$이 생성되고 이 식은 항상 성립하는 식으로 생략 가능하다. 그리고 그림 4의 [FieldAss], [ParamAss], [Seq], [Cond], [MethCall] 구문들에 대해서도 항상 성립하는 집합-관계식이 생성되므로 생략 가능하다.

그림 4에서 레이블을 기록하는 생성 규칙의 의미는 다음과 같다.

[Throw]m

$$\frac{m \rhd e_1 : C_1}{m \rhd l : \text{throw } e_1 : \{X_m \supseteq \langle c, l\rangle\} \cup C_1} \quad c = class(e_1)$$

throw 구문이 메소드 또는 try-블록 m 내에 위치하므로 throw 구문과 부분 구문 e_1에 대한 집합 변수는 X_m이다. 그러므로 집합-관계식 $X_m \supseteq \langle c, l\rangle \cup X_m$을 생성하고 이 식을 단순화하면 $X_m \supseteq \langle c, l\rangle$이다.

[Try]m

$$\frac{m \rhd e_g : C_g \quad m \rhd e_1 : C_1}{m \rhd \text{try } e_g \; l : \text{catch } (c_1\ x_1)\ e_1 : \{X_m \supseteq (X_g - \{c_1\}^*),\ X_g \supseteq (X_g \cap \{c_1\}^*) \cdot l\} \cup C_g \cup C_1}$$

try-블록 e_g에 대한 집합 변수는 X_g이고, try-catch 구문과 부분 구문 e_1에 대한 집합 변수는 X_m이다. 그러므로 집합-관계식 $X_g \supseteq (X_g \cap \{c_1\}^*) \cdot l$과 $X_m \supseteq (X_g - \{c_1\}^*)$ ($X_m \supseteq (X_g - \{c_1\}^*) \cup X_m$의 단순한 형태)를 생성한다.

[MethDef]m $\quad \dfrac{m \rhd e_m : C_m}{l : m(x) = e_m : \{X_m \supseteq X_m \cdot l\} \cup C_m}$

메소드 m에서 처리되지 않은 예외는 메소드 m을 거치게 되므로, 처리되지 않는 예외의 경로에 이 구문의 레이블인 l을 추가하는 집합-관계식 $X_m \supseteq X_m \cdot l$을 생성한다.

구문 단위와 메소드와 try-블록 단위의 분석은 집합 변수의 수에 차이가 있다. 구문 단위의 분석은 각 구문마다 하나의 집합 변수를 정의함으로써 각 구문 마다 분석 정보를 제공하는 반면, 메소드와 try-블록 단위의 분석은 각 메소드와 try-블록 마다 분석 정보를 제공한다. 그러므로 메소드와 try-블록 단위의 분석은 메소드 또는 try-블록 내의 특정 구문에서의 분석 정보를 제공하지 못한다.

[예제 1] 그림 5의 간단한 예제 프로그램에 대해 그림 4의 생성 규칙을 적용하면 표 1과 같은 집합-관계식들이 생성된다. 생성된 집합-관계식들 중에서 4), 6), 8), 11) 은 그림 4의 [MethDef] 규칙을 적용한 것이데, 메소드의 레이블은 메소드의 시작 레이블인 1, 6, 8, 11을 사용한다.

3.4 분석의 해 구하기

분석의 해를 구하기 위해서 생성된 집합 관계식들의 해를 구하는 규칙을 정의한다. 그림 6은 해를 구하는 규칙으로, 규칙의 기본적인 의미는 프로그램에서 가능한

```
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();
   }
```

그림 5 예제 프로그램

표 1 집합-관계식의 생성 예

집합-관계식	적용규칙
1) $X_{main} \supseteq X_{try} - \{Exception\}*$	[Try]m
2) $X_{try} \supseteq (X_{try} \cap \{Exception\}*) \cdot 3$	[Try]m
3) $X_{try} \supseteq X_{m1}$	[MethCall]m
4) $X_{main} \supseteq X_{main} \cdot 1$	[MethDef]m
5) $X_{m1} \supseteq X_{m2}$	[MethCall]m
6) $X_{m1} \supseteq X_{m1} \cdot 6$	[MethDef]m
7) $X_{m2} \supseteq \{<E1, 10>\}$	[Throw]m
8) $X_{m2} \supseteq X_{m2} \cdot 8$	[MethDef]m
9) $X_{m3} \supseteq \{<E2, 13>\}$	[Throw]m
10) $X_{m3} \supseteq X_{m3}$	[MethCall]m
11) $X_{m3} \supseteq X_{m3} \cdot 11$	[MethDef]m

1) $\dfrac{X \supseteq X_1 \cup X_2}{X \supseteq X_1}$ 2) $\dfrac{X \supseteq X_1 \cup X_2}{X \supseteq X_2}$ 3) $\dfrac{X \supseteq Y \quad Y \supseteq \{\langle c, \tau\rangle\}}{X \supseteq \{\langle c, \tau\rangle\}}$

4) $\dfrac{X \supseteq X_1 \cdot l \quad X_1 \supseteq \{\langle c, \tau\rangle\}}{X \supseteq \{\langle c, \tau \cdot l\rangle\}}$

5) $\dfrac{X \supseteq X_1 - \{c_1, \ldots, c_k\} \quad X_1 \supseteq \{\langle c, \tau\rangle\} \quad c \notin \{c_1, \ldots, c_k\}}{X \supseteq \{\langle c, \tau\rangle\}}$

6) $\dfrac{X \supseteq X_1 \cap \{c_1, \ldots, c_k\} \quad X_1 \supseteq \{\langle c, \tau\rangle\} \quad c \in \{c_1, \ldots, c_k\}}{X \supseteq \{\langle c, \tau\rangle\}}$

그림 6 해를 구하는 규칙 S

자료 흐름 경로를 따라서 값들을 전달하는 것이다. 그림 6의 4)번 규칙은 예외 전달 경로를 기록하는 규칙으로 집합 변수 X_1의 처리되지 않는 예외는 레이블 l을 거치게 되므로, 지금까지의 경로 τ에 레이블 l을 추가한다. 그림 4의 규칙 S를 적용함으로써 집합-관계식들의 집합 C의 해 $lm_s(C)$를 계산할 수 있다.

[예제 2] 예제 1에서 생성된 생성 규칙들의 집합 C

에 그림 6의 해를 구하는 규칙 S를 적용하여 구한 해 $lm_s(C)$는 다음과 같다.

$\{X_{main} \supseteq \{<E2, 13 \cdot 11 \cdot 1>\}, X_{try} \supseteq \{<E1, 10 \cdot 8 \cdot 6 \cdot 3>\}$, Xm1 $\supseteq \{<E1, 10 \cdot 8 \cdot 6>\}$,

$X_{m2} \supseteq \{<E1, 10 \cdot 8>\}, X_{m3} \supseteq \{<E2, 13 \cdot 11>\}$ $\}$

예외 흐름을 나타내기 위해 다음과 같이 해 $lm_s(C)$에 대한 예외 흐름 그래프를 정의한다.

[정의 1] 예외 흐름 그래프

C를 입력 프로그램 P에 대해 생성된 집합-관계식의 집합이고 $lm_s(C)$는 이 집합의 해라고 할때, 해 $lm_s(C)$에 대한 예외 흐름 그래프는 그래프 $<V, E>$로 정의된다. V는 P내의 레이블들의 집합이고 $E = \{l_1 \rightarrow^c l_2, l_2 \rightarrow^c l_3, \cdots, l_{n-1} \rightarrow^c l_n \mid \langle c, l_1 l_2 \cdots l_n \rangle \in lm_s(C)(X)$, X는 C 내의 집합 변수$\}$이다. $l_i \rightarrow^c l_{i+1}$는 c 레이블을 갖는 l_i에서 l_{i+1}로의 에지를 나타낸다.

4. 예외를 포함하는 제어 흐름 그래프의 생성 예

본 절에서는 예외를 포함하는 제어 흐름 그래프를 생성하는 과정을 예로 보인다. 예외를 포함하는 제어 흐름 그래프는 정상 흐름 그래프와 예외 흐름 그래프를 합병함으로써 생성할 수 있다.

그림 6은 그림 5의 프로그램에 대한 문장 단위의 정상 흐름 그래프이다. 그림 6의 정상 흐름 그래프는 [2]에서 정의된 표현 방법을 사용하였다. 그림 6에서 하나의 메소드 호출문은 호출 노드와 리턴 노드로 구성된다. 그리고 각 메소드에 대해서 시작 노드와 exit 노드가 추가되었다. 메소드 호출문에 대해서 호출 노드에서 호출된 메소드의 시작 노드로의 흐름이 발생하고, 메소드가 정상적으로 종료할 때는 메소드의 exit 노드에서 리턴 노드로의 흐름이 발생한다. 그림 7에는 예외 흐름인 throw 문장(노드 10과 노드 13)으로부터 시작하는 두 개의 예외 흐름은 포함되지 않았다.

예제 프로그램에 대한 예외 흐름 그래프는 그림 8과 같다. 그림 8에서 메소드 정의 구문의 레이블은 메소드의 시작 레이블로 하였다. 메소드의 시작 레이블로 들어오는 에지 $l_i \rightarrow^c l_{i+1}$의 의미는 l_{i+1}의 메소드가 예외 c로 인한 예외 상황으로 종료하는 것이다. 그러므로 그림 8은 예외 E1이 라인 10에서 발생하여 메소드 m2()와 m1()이 예외 상황으로 종료하고 라인 3의 catch 블록에서 처리됨을 보여주고 있다. 그리고 예외 E2가 라인 13에서 발생하여 메소드 m3()와 main()가 예외 상황으로 종료함을 보여주고 있다.

그림 7의 정상 흐름 그래프와 그림 8의 예외 흐름 그래프를 합병함으로써, 정상 흐름과 예외 흐름을 포함하는 제어 흐름 그래프를 생성할 수 있다. 두 그래프를 합병한 그래프에 excep-exit 노드가 추가되었는데, 메소드가 예외의 발생으로 종료되는 경우를 표현한다. 그러므로 예외 흐름 그래프에서 메소드의 시작 레이블에 해당하는 노드는 합병한 그래프의 exception-exit 노드에 대응된다. 그림 8은 두 그래프를 합병한 결과이다. 그림 8

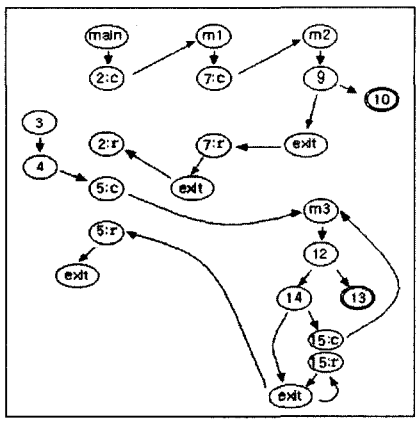


그림 7 정상 흐름 그래프

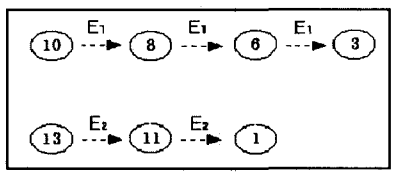


그림 8 예외 흐름 그래프

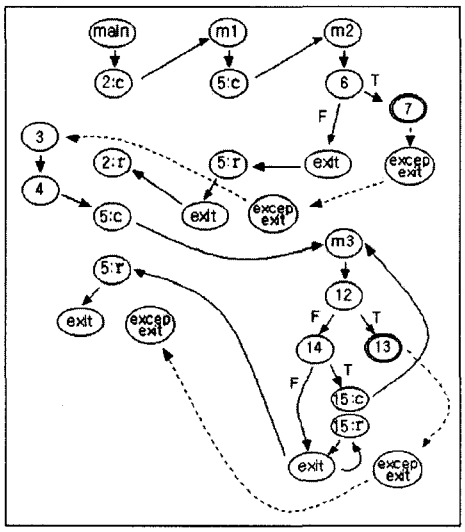


그림 9 정상 흐름과 예외 흐름을 포함하는 제어 흐름 그래프

에서 정상 흐름은 실선으로 예외 흐름은 점선으로 표현하였다.

5. 결 론

본 논문에서는 예외 흐름을 포함하는 제어 흐름 그래프를 생성하는 방법을 제안하였다. 기존의 방법은 예외 흐름과 정상 흐름의 상호 의존적인 관계로 인해 두 흐름을 동시에 계산하면서 제어 흐름 그래프를 생성한다. 그러나 실제 Java 프로그램을 조사해 본 결과 두 흐름이 상호 의존적으로 필요한 경우는 거의 발생하지 않음을 알 수 있었고, 정상 흐름과 예외 흐름을 분리해서 계산할 수 있음을 알았다. 그러므로 본 논문에서는 정상흐름 분석과 분리하여 예외 흐름을 계산하는 예외 흐름 분석을 제안하였고, 예외 흐름을 표현하는 예외 흐름 그래프를 제안하였다. 그리고 제어 흐름 그래프는 예외 흐름 그래프와 정상 흐름 그래프를 합병함으로써 생성될 수 있음을 보였다. 두 흐름 분석을 분리해서 얻어지는 장점은 다음 두 가지이다. 첫 번째는 두 흐름 정보 중 하나의 정보만 필요한 경우 두 흐름을 동시에 계산해야 되는 대신 하나의 흐름만을 분리해서 구할 수 있다. 두 번째는 기존의 예외를 고려하지 않은 제어 흐름 그래프를 사용할 수 있다. 기존의 제어 흐름 그래프에 예외 흐름을 추가하여 예외 흐름을 포함한 제어 흐름 그래프를 생성할 수 있다.

참 고 문 헌

[1] B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah, "A static study of Java exceptions using JESP," Tech. Rep. DCS-TR-403, Rutgers University, Nov. 1999.

[2] S. Shinha and M. J. Harrold, "Analysis and Testing of Programs With Exception-Handling Constructs," IEEE Trans. on Software Engineering, Vol. 26, No. 9, 2000.

[3] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, "Efficient and precise modeling of exceptions for analysis of Java programs," in *Proceedings of PASTE '99 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Sep. 1999, pp. 21-31.

[4] R.K. Chatterjee, et al., "Complexity of concrete type-inference in the presence of exceptions," *Lecture Notes in Computer Science*, vol. 1381, pp. 57-74, Apr. 1998.

[5] R Chatterjee and B. G. Ryder, "Data-flow-based testing of object-oriented libraries," Tech. Rep. DCS-TR-382, Rutgers University, Mar. 1999.

[6] N. Heintze, "Set-based program analysis," Ph.D thesis, Carnegie Mellon University, Oct, 1992.

[7] 이정수, 조장우, "클래스분석에서 예외분석의 필요성," 프로그래밍언어 논문지, 제16권, 제1호, 2002.2.

[8] S. Drossopoulou and T. Valkevych, "Java type soundness revisited," Technical Report, Imperial College, November 1999.

[9] F. Tip and J. Parlsberg, "Scalable Propagation-Based Call Graph Construction Algorithms," In Proceedings of 15th Annual Conference on Objected-Oriented Programming Systems, Languages, and Applications, pages 281-293, Oct. 2000.

[10] D. Bacon and P. Sweeney, "Fast Static Analysis of C++ Virtual Function Calls," In Proceedings of 11th Anual Conference on Objected-Oriented Programming Systems, Languages, and Applications, pages 324-341, Jan 1994.

[11] B.-M. Chang, J.-W. Jo, K. Yi, and K. Choe, "Interproceduaral Exception Analysis for Java," In Proceedings of 2001 ACM Symposium on Applied Computing, pages 620-625, Mar. 2001.

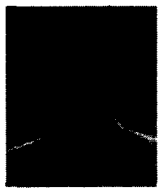
[12] B. Chang and Jang-Wu Jo, "Granularity of Constraint-Based Analysis for Java," in *Proceedings of ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, Sep. 2001. pp. 94-102.

[13] K. Yi and S. Ryu, "Towards a cost-effective estimation of uncaught exceptions in SML programs," In Lecture Note in Computer Science, volume 1302, pages 98-113, Springer-Verlag, Proceedings of 4th Static Analysis Symposium, Sep, 1997.

[14] J.-W. Jo, B.-M. Chang, K. Yi, and K. Choe, "An Uncaught Exception Analysis for Java," The Journal of Systems and Software, accepted for publication.

조 장 우

1992년 서울대학교 계산통계학과 졸업(이학사). 1994년 서울대학교 대학원 전산과학과(이학석사). 2003년 한국과학기술원 전산학과 박사(공학박사). 1997년~현재 부산외국어대학교 컴퓨터공학부 부교수. 관심분야는 프로그램 분석, 최적화 컴파일러, 모바일 프로그래밍, 개발환경



창 병 모

1988년 서울대학교 컴퓨터공학과 졸업(공학사). 1990년 한국과학기술원 전산학과(공학석사). 1994년 한국과학기술원 전산학과(공학박사). 1994년~1995년 한국전자통신연구소 박사후 연수 연구원. 1995년~현재 숙명여자대학교 컴퓨터과학과 부교수. 관심분야는 컴파일러 구성론(정적분석, 코드최적화), 논리 프로그래밍, 모바일 프로그래밍