Compiler에서의 Cause-Effect Graph 방법을 이용한 Syntax Analysis에 관한 연구

김 영 철°
연세대학교 산업대학원

김 재 희
연세대학교 전자공학과 교수

A Cause-Effect Graph approach to the Syntax Analysis Stage in a Compiler Design

Young Chul Kim
Graduate of Eng., Yonsei Univ.

Jaihie Kim
Dept. of Electronic Engineering, Yonsei University

In this paper, in order to make the compiler that the corrective detection of errors and expression of more understandable error messages are provided, the usage of cause-effect testing method and case study (simulation COBOL compiler) using this methodology are illustrated.

1. 서론

Compiler 는 프로그램내에 존재하는error 를 정확하게 찾아내어, 프로그래머가 error를 쉽고 빠르게 수정할 수 있도록error message 를 알기쉽고 정확하게 제공해야 한다. 만일 compiler 가 error를 추상적 또는 부정확하게 지적한다면 프로그래머가 error 를 찾고 수정하는데 많은 노력과 시간을 투입해야 하며, 개발기간의 지연은 물론 프로그램의 질을 저하시키게 되므로 compiler 가 의미없는 error message 를 제공하는 것은 피하여야 한다.[1,2]

본 논문에서는 cuase-effect graph 방법을 compiler 구성중의 Syntax analysis 단계에서 적용시켜 error 를 검출하는 방안을 연구하고, 이 부분에 대한 모의 프로그램을 작성하여 test 함으로써 cuase-effect graph 방법이 프로그래머에게 error 검출에 대한 훨씬 유용한 정보를 제공해주는 도구로 사용할 수 있음을 보이고자 한다.

본 논문의 구성은 2장에서는 testing 의 정의와 cause-effect graph 방법론을 간략히 소개하고, 3장에서는 이 방법론을 COBOL 의 "MOVE" statement 에 적용하여 cause-effect graph 를 작성하고, 최적의 test data 를 얻기 위해 decision table 을 작성하여 4장에서는 추출된 test data 를 source statement 를 입력했을때 error 를 완벽하게 찾아내어 알기쉽게 message 를 제공해주는 모의 compiler 를 작성함으로써, 본 논문의 타당성을 입증하고자 한다.

2. Testing Strategy

(1) 정의

모든 프로그램은 error 를 반드시 포함하고 있기 때문에, 프로그램의 신뢰성을 최대한 보장하기 위해서는 testing 과정을 반드시 거쳐서 error 가 실제로 존재하고 있다는 것을 증명해야 한다.[3] 그러나 error가 실제로 전혀 존재하지 않는다는 것을 증명할 수 있는 testing방법은 현실적으로 불가능하다. 왜냐하면 근본적으로 exhaustive testing방법이 존재할 수 없기 때문이다. 따라서 error를 가능한 조기에 발견하고, 최소한의 test data 로 최대한의 error 를 발견하게 함으로써 프로그램개발에 투입되는 비용과 인력을 줄일 수있는 경제적 testing 방법이 필요한데, 이것의 역할은 전체 test data 를 대표할 수 있는 test data 를 추출해 내는 것이다.[4],[5]

본 논문에서 적용하고자 하는 cause-effect graph 방법은 이러한 testing방법론 중의 하나이다.

(2) Cause-effect graph

Graph 를 구성하기 위해서 이 방법론에서는 도표 2-1 및 도표 2-2와 같은 기본적인 표현을 사용한다.

Function	Notation	Meaning
IDENTIFY	ⓐ — ⓑ	a=1 then b=1
NOT	ⓐ ~ ⓑ	a=1 then b=0
OR	ⓐ, ⓑ, ⓒ → ⓓ	a or b or c=1 then d=1
AND	ⓐ, ⓑ → ⓒ	a and b=1 then c=1

<그림 2-1> BASIC notation

FUNCTION	NOTATION	MEANING
EXCLUSIVE	E<-- ⓐ ⓑ	AT MOST ONE I a,b중 많아도 1개는 1이다
INCLUSIVE	I<-- ⓐ ⓑ ⓒ	AT LEAST ONE I a,b,c중 최소1개는 1이어야 한다.
ONE & ONLY ONE	O<-- ⓐ ⓑ	ONE & ONLY ONE I a,b중 오직 1개만 1이어야 한다.
REQUIRES	R ⓐ ⓑ	a=1이면 b는 항상 1이어야 한다.
MASK	M ⓐ ⓑ	effect a=1 then effect b=0 이어야 한다

<그림 2-2> Constraints notation

3. Cause-effect graph에 의한 Modeling

Testing은 software의 정확성을 검증하는 과정이나, exhaustive testing 이 근본적으로 불가능하므로 testing 방법론을 적용하여 경제적인 testing 을 하기 위한 최적의 test data 를 추출해 내는 것이다. 따라서, compiler 의 정확성을 검증하기 위해서는 프로그램을 구성하는 각 statement 를 test data 로 입력을 시켜야 함으로 이를 위해 본 논문에서는 COBOL compiler 를 택하였고, 실제로는 모든 statement 에 적용해야하나 일반적으로 실수를 범하기 쉬운"MOVE" statement 를 model 로 선정하고자 한다.

(1) MOVE statement 문법

```
MOVE { data-name-1 } TO data-name-n
     { literal-1   }
```

① data를 보내는 field 는 data-name-1 과 literal-1중에서 선택 사용할 수 있다.

② data 를 받는 field 는 반드시 data-name 일것

③ data-name 은 숫자로만 구성이 되거나 hypone 으로 시작 또는 끝나면 안되고 30자를 초과할 수 없으며, 영문자, 숫자, hypone 등으로 구성되어야 한다.

④ literal 은 18자를 초과할 수 없으며 숫자, +, -, . 등으로 구성이 되어야 한다.

(2) Causes 와 Effects 의 서술

① 상기 문법에 의거하여 "MOVE" statement 가 error 가 되지 않기 위한 조건을 각operator 및 operand 별로 서술하여 그들 각각에 고유번호를 부여함으로써 cause 를 결정한다.

CAUSE NO.	CAUSE 내 용
1.	1st operator는 MOVE이다.
2.	1st operand 는 存在한다.
3.	1st operand 는 data-name-1이다.
4.	1st operand 는 literal-1이다.
5.	data-name-1은 Alphabet로 始作해야 한다.
6.	data-name-1은 30자 以內로 構成되어야 한다.
7.	literal-1은 numeric 또는 Quotation Mark로 始作해야 한다.

(표 3-1) CAUSES

② 각 operator 및 operand 자체, 또는 상호 조합에 의해 발생할 수 있는 모든 error 와 정상적인 경우를 찾아내어 그들 각각에 고유번호를 부여함으로써 effect 를 결정한다.

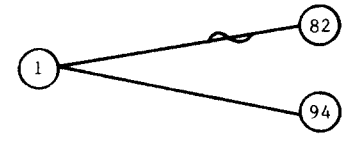
EFFECT NO.	EFFECT 내 용
82.	1st operator가 MOVE가 아니다.
83.	1st operand 가 없다.
84.	Data-name-1 이 data name이나 literal이 아니다.
85.	Data-name-1 이 30자를 超過한다.
86.	literal-1 이 18자를 超過한다.
87.	2nd operator가 To가 아니다.
94.	모든것이 正常이다.

(표 3-2) EFFECTS

(3) Cause 와 Effect 간의 Graph

Causes 와 Effects 사이의 관계를 Graphing 할때 크게 2가지 경우가 발생한다.

① Cause 가 effect 에 직접 영향을 주는 경우

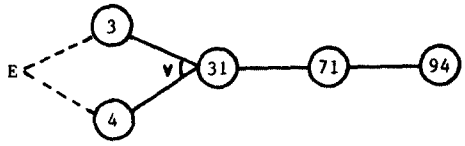


(도표 3-3) Direct Cause-Effect Graph

Effect 82 가 존재한다(Value =1)는 것은 Cause 1이 존재하지 않음(Value = 0)을 의미하므로 Cause 1과 Effect 94는 NOT 관계이다.

반면에 Effect 94가 존재한다는 것은 Cause 1이 존재함을 의미하므로 이들의 관계는 Identify 관계이다.

② 여러개의 Cause 가 상호 결합하여 effect 에 영향을 주는 경우



(도표 3-4) Intermediate Cause-Effect Graph

"MOVE" statement의 문법에 의해, Cause 3과 4만으로는 effect 94에 직접 영향을 줄 수 없다.

왜냐하면 effect 94는 "MOVE" statement 가 정상임을 의미하는데, 정상이 되기 위해서는 모든 cause 의 상호결합에 의해서만 가능하기 때문이다. 따라서 이때는 cause 들간에 intermedinate cause가 발생된다.

상기 graph에서 cause 3과 4의 관계에서 intermediate cause 31이 발생되며, intermediate cause 71은 도표 3-5에서 보듯이 cause 2와 intermediate cause 31, 51, 34와의 관계에 의해 결정되어 진다.

또한 cause 3과 4는 문법에 의해 결코 동시에 존재할 수는 없으나, 그들중 어느 cause 도 존재하지 않을 수도 있는 exclusive 관계에 있음을 알 수 있다.

상기와 같은 과정을 거쳐 "MOVE" statement 를 구성하는 각 cause 와 effect 사이에 발생할 수 있는 관계를 규정하여 최종적으로 만들어진 완전한 cause-effect graph 는 도표 3-5와 같다.

(4) Decision table 작성

Cause 와 effect 사이의 관계를 규정한다는 것은 상호 결합에 의해 만들 수 있는 test data 의 경우의 수를 의미한다. 도표 3-5에서 보듯이 "MOVE' statement 에 대하여 만들 수 있는 test data 는 무수히 많다.

그러나 현실적으로 이 수많은 test data 로 testing 을 하는것은 불가능하기 때문에 최적의 test data 를 추출할 필요가 있다. 이를 위해 decision table 을 적용한다.

decision table 작성은 cause-effect graph 방법을 적용하는 과정중 가장 어렵고 시간이 많이 소요되는 작업으로서 이 table을 잘못 작성하면 원하는 test data 를 얻을 수 없다.

effect 72를 예를들어 decision table을 작성하는 방법을 설명하고자 한다.

effect 72의 값 이 1(=존재)이 되기 위해서는 cause 12,13과 intermediate cause 35, 36의 값 이 모두 1 이어야 한다. intermediate 35의 값이 1이 되기 위해서는 cause14, 15가 모두 1이어야 한다.

또한 intermediate36의 값이 1이 되기 위해서는 cause 16, 17 이 각각 0, 1이거나 1, 0의 값을 가져야 한다.

이렇게 effect 의 값을 갖기 위해 각 cause 및 intermediate cause 값을 거꾸로 추적하여 값을 결정하는 것을 "Back-through trace"라고 한다.

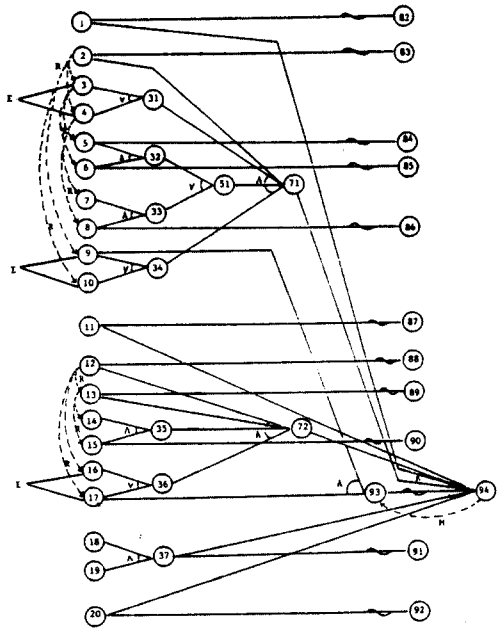
도표 3-6은 완료된 decision table 이며, 도표 3-7은 decision table 에 의해 추출된 test case data 이다.

		TEST CASE													SELECTION		
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C A U S E	1	0															1	1	1
	2		0														1	1	1
	3																0	1	1
	4																1	0	0
	5			0													0	1	1
	6				0												0	1	1
	7																1	0	0
	8					0											1	0	0
	9														1		0	1	0
	10																1	0	1
	11						0										1	1	1
	12							0									1	1	1
	13								0								1	1	1
	14															0	1	1	1
	15									0							1	1	1
	16																0	1	1
	17														1		1	0	0
	18										0	1	0				1	1	1
	19										1	0	0				1	1	1
	20													0			1	1	1
중간 CAUSE	31																1	1	1
	32																0	1	1
	33																1	0	0
	34																1	1	1
	35																1	1	1
	36																1	1	1
	37																1	1	1
	51																1	1	1
	71																1	1	1
	72																1	1	1
EFFECT	82	1																	
	83		1																
	84			1															
	85				1														
	86					1													
	87						1												
	88							1											
	89								1										
	90									1									
	91										1	1	1						
	92													1					
	93														1		0	0	0
	94																1	1	1
	95															1			

<도표 3-6> Decision Table

Decision table의 종축의 번호는 cause 와 intermediate cause 및 effect 에 부여된 고유번호를 의미하며, 횡축의 번호는 이 table 에 의해 추출하고자 하는 test data 가 몇가지 인가를 의미한다. 이때 종축과 횡축이 만나는곳의 "0"은 "absence" 를 나타내고 "1"은 "presence" 를 의미한다.

예를들어 effect 82가 존재하기 위해서는 cause 1이 존재하지 않는 경우를 test data 로 선정함을 의미하는 것이다.



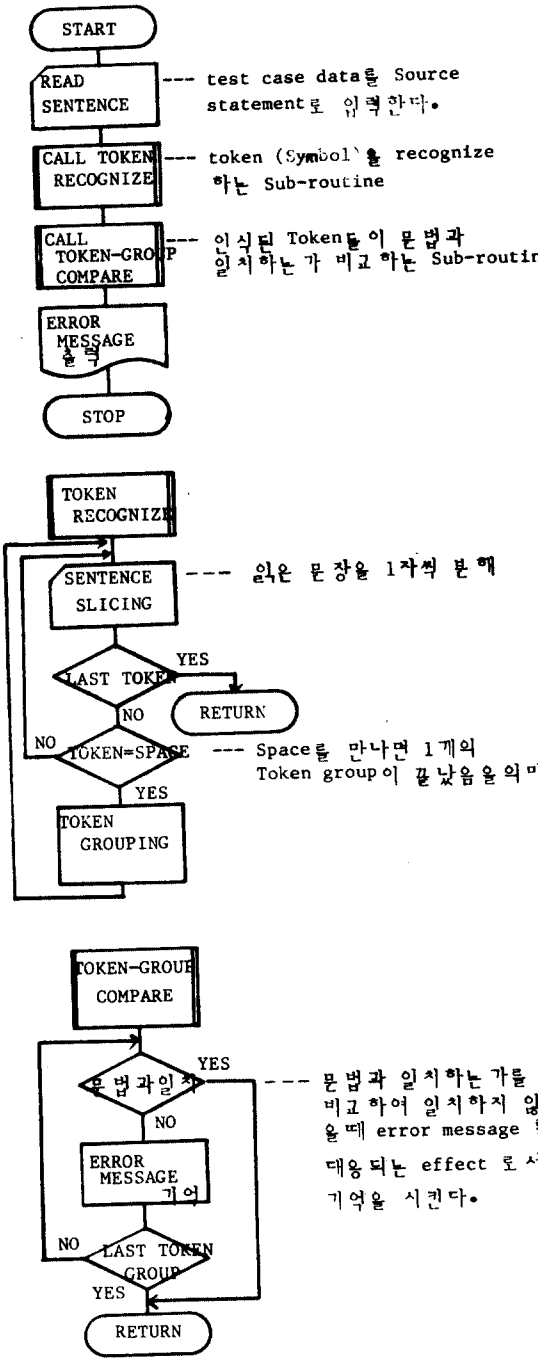
(도 표-5) Complete Cause-Effect Graph

TEST CASE NO.	TEST DATA
1	MVOE DNAME TO B.
2	MOVE TO B.
3	MOVE -M15 TO B.
4	MOVE KIMYOUNGCHULYERYEXCELLENTPAPERS TO B.
5	MOVE 19E508161986080830783 TO B.
6	MOVE DNAME TOO B.
7	MOVE DNAME TO
8	MOVE DNAME TO 999.
9	MOVE DNAME TO KIMYOUNGCHOLEXCELLENTTHESIS250816.
10	MOVEDNAME TO B.
11	MOVE DNAME TO B.
12	MOVEDNAMETOB.
13	MOVE "19E50816" TO C.
14	MOVE DNAME TO D.
15	MOVE DNAME TO "KIM".
16	MOVE 1985 TO E.
17	MOVE DNAME TO F.
18	MOVE YOUNG TO CHUL.

(표 3-7) Test case data

4. 모의 프로그램에 의한 test결과

(1) 프로그램 개요



(2) Test 결과

모의 프로그램에 의한 test 결과는 아래와 같다.

(입력된 문장)

문장번호	문 장 내 용
1	MVOE DNAME TO B.

(Error message 출력내용)

문장번호	Error Message	effect번호
1	First operator is not "MOVE" or ABSENT	82

즉, 프로그래머가 프로그램에서 사용한 단어를 사용하여 error message 를 표현해줌으로써, 보다 수정하기에 쉽도록 message 를 출력할 수 있다.

5. 결론

국히 제한적이기는 하지만 Cause-effect graph 방법을 compiler 설계시 적용하여 만든 모의 프로그램에서는 error 를 정확하게 찾아내어 error message 를 출력함으로써 프로그래머가 error 에 대한 수정을 쉽게 할 수 있도록 제공해 줌을 알 수 있었다.

따라서 같은 원리에 의해, 다른 명령어 뿐만 아니라 다른compiler, Assembler, interpreter등을 설계할때 본 논문에서 적용한 방법론을 적용한다면 우수한 성능을 가진것을 만들 수 있을 것이라고 생각된다.

다만 이러한 방법으로 만들었을때 compiler 자체가 차지하는message 가 커지고, 기존 compiler 에 비해 compile 시간이 더 많이 소요될 가능성이 있다.

또한 기존의 compiler 제작시 각 공급 회사들이 어떠한 방법론을 적용했는지 참고자료가 없어서 조사하지 못한점이 아쉽다.

따라서 향후 이러한 점들을 철저히 조사, 분석하여 문제점들을 해결하고, test data의 추출을 보다 쉽고 빠르게 할 수 있는 방법에 대한것을 연구과제로 삼고자 한다.

참 고 문 헌

1. A.V.AHO, J.D.ULLMAN, principles of compiler design, Addison-Wesley,

2. M.L.SHOOMAN, Software Engineering, McGraw-Hill, 1983

3. G.J.Myers, The Art of software testing, Wiley-interscience, 1979

4. M.H.Van Emden, programming with Verification conditions, IEEE Trans. on software engineering, Vol SE-5. NO2, pp148~159, March, 1979

5. Tsun S. Chow, testing software design modeled by finite-state machines,IEEE Trans on software engineering Vol SE-4, NO3, pp178~186. May, 1978

6. L.A. Clarke, Johnette, Hasel, D.J. Richardson, A close look at domain testing, IEEE Trans on software engineering Vol SE-8, NO4, pp380-390, July. 1982