Journal of The Korea Society of Computer and Information
Vol. 17 No. 7 , July 2012

2012-17-7-1-3

GML파일을 이용한 검증조건의 시각화

허 혜 림*, 김 제 민*, 박 준 석*, 유 원 희*

Visualization of Verification Condition by GML file

Hye-Lim Hu*, Jemin Kim*, Joonseok Park*, Weonhee Yoo*

요 약

프로그램 검증을 위해 사용되는 방법으로 프로그램을 검증조건으로 변환하여 정리 증명기를 통해 프로그램의 유효성을 확인하는 방법이 있다. 검증조건 생성을 통한 프로그램의 검증의 경우 검증조건은 프로그램을 검증하기 위한 충분하고 정확한 정보를 가지고 있어야한다. 하지만 프로그램의 변환을 통해 생성된 검증조건의 경우 논리식만으로 구성되어 있어 사용자가 쉽게 그 내용을 파악할 수 없다. 본 논문에서는 가독성이 떨어지는 검증조건을 시각화하는 프로그램을 구현하였다. 프로그램을 통해 검증조건을 구성하고 있는 논리식간의 관계 등을 비롯한 정보를 보다 쉽게 확인할 수 있다.

▸ Keyword : 검증조건(VC), 그래프, 시각화, 비구조화 프로그램

Abstract

There is a method which identifies validity of program by transforming program to verification condition to verify program. If program is verified by generating verification condition, verification condition must have enough and accurate information for verifying program. However, verification condition is consisting of logical formulas, so the user cannot easily identify the verification condition. In this paper, we implemented program that visualize the poorly readable verification conditions. By the program, the users can easily identify information, such as the relationship between logical formulas that represent verification condition

▸ Keyword : Verification Condition(VC), graph, visualization, unstructured program

• 제1저자 : 허혜림 • 교신저자 : 유원희
• 투고일 : 2011. 07. 09, 심사일 : 2012. 03. 07, 게재확정일 : 2012. 05. 23.
* 인하대학교 컴퓨터 · 정보공학과(Dept. of Computer Science & Information Technology, Inha University)
※ 이 논문은 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(No. 2011-0026495)

I. 서 론

프로그램의 신뢰성이 중요하게 여겨져 감에 따라 프로그램의 신뢰성을 높이기 위한 연구도 다양하게 진행되고 있다. 프로그램 테스팅이나 검증 등과 같은 연구가 이에 포함된다. 프로그램을 실행하기 전 프로그램의 결함을 가능한 많이 발견하여 제거할 수 있다면 프로그램의 신뢰성을 높일 수 있고 프로그램의 유지보수를 위한 비용과 노력을 절감할 수 있다. 프로그램의 신뢰성을 높이기 위해 연구된 방법은 여러 가지가 있다. 그 중 일반적으로 실행시간에만 찾을 수 있는 오류들을 추가적으로 정적으로 발견하는 것을 목표로 하는 방법이 확장 정적 검증(Extended static checking)이다[1, 2].

확장 정적 검증 방법은 검증조건(Verification Condition)을 생성하여 검증하는 방법을 사용한다. 확장 정적 검증에서는 프로그램을 검증조건으로 변환하여 정리증명기(Theorem Prover)를 이용해 프로그램의 유효성을 확인한다. 검증조건은 논리식으로 표현되는데 정리증명기는 논리식을 풀어 유효한지 아닌지를 확인해 준다. 문제가 없는 프로그램의 경우 결과는 유효하다고 나오지만 그렇지 않을 경우에는 유효하지 않다는 결과가 나온다. 정리증명기에 따라서는 반례(Counter Example)를 제공하기도 한다. 제공된 반례를 이용하여 프로그램의 문제를 보다 쉽게 파악할 수 있다.

그림 1은 바이트 코드의 검증을 위한 도구인 JBVF[3]의 일부인 Birs의 검증조건 생성기의 구조와 검증조건 생성기 구현을 위한 시각화 프로그램의 구조이다. Birs는 바이트 코드의 검증을 위한 중간언어로 검증에 필요한 정보를 많이 포함하고 있다[3]. 검증조건 생성기는 보통 그림 1의 검증조건 생성기와 비슷한 구성을 보인다. 바이트 코드를 중간표현으로 변환한 결과를 검증조건 생성기에 입력으로 넣어 보호 명령 생성 단계와 패시브 폼 생성 단계를 걸친 후 최약 전조건(weakest precondition) 계산 규칙을 이용하여 검증조건을 생성하는 구조이다. 그림 1에서 보이는 것과 같이 검증조건 생성기는 여러 단계의 변환과정을 거치게 된다. 검증조건 생성 단계 중 어느 한군데서라도 문제가 생길 경우 원하는 검증조건을 얻지 못한다. 그래서 검증조건 생성기를 구현 할 때는 생성된 검증조건이 검증을 위한 정보를 제대로 가지고 있는지, 구현에 문제가 있어 잘못된 검증조건이 생성되지는 않았는지 확인할 필요가 있다.

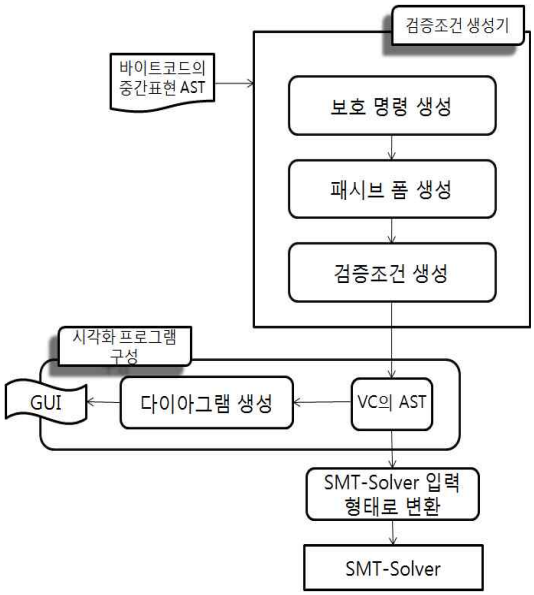

그림 1 Birs의 검증조건 생성기와 시각화 프로그램
Fig. 1 Verification condition generator of Birs and visualization program

생성된 검증조건은 논리식으로 표현된다. 프로그램의 전체를 논리식으로 구성해 표현할 경우 가독성이 떨어져 내용을 파악하기 어렵다. 검증조건 생성과 관련된 연구와 개발에 있어서 가독성이 떨어지는 검증조건은 일일이 확인하는데 시간이 많이 소모된다. 큰 프로그램의 검증조건은 더욱 확인하기 어렵다. 그래서 Birs의 검증조건 생성기를 구현할 때 검증조건이 제대로 생성되었는지 쉽게 확인할 수 있도록 검증조건의 시각화 프로그램을 이용하였다. 시각화로 표현된 검증조건은 간선 모양과 노드 관계를 통하여 보호 명령으로의 변환 결과에 대해서 확인할 수 있고 노드 이름을 통하여 패시브 폼 변환이 적용된 결과도 간단히 확인 할 수 있어 검증조건 생성기의 구현시간을 줄이는데 도움이 된다.

본 논문에서는 그림 1의 시각화 프로그램의 구현에 대해서 서술하고 검증조건을 GML(Graph Modeling Language)로 표현하는 방법을 제시한다. 본 논문에서 사용하고 있는 검증조건 문법과 같은 문법을 사용하는 모든 검증조건 생성기는 본 논문에서 제시하는 방법을 이용하여 검증조건을 GML로 변환할 수 있고 생성된 GML 파일을 통해 동일한 시각화 프로그램으로 검증조건을 시각화 할 수 있어 확장 정적 검증방법을 사용하는 다양한 도구에서 활용할 수 있다. 사용자는 시각화 된 검증조건을 통하여 검증조건이 프로그램의 정보를 제대로 포함하고 있는지 확인할 수 있다.

본 논문은 다음과 같이 진행된다. 2장에서는 논문에 필요한 기본지식 및 관련 연구에 관하여 서술한다. 3장에서는 비구조화 프로그램에 대한 검증조건생성을 간단히 설명하고 4장에서는 생성된 검증조건을 시각화하는 방법을 보인다. 5장에서는 논문의 결과와 향후 연구방향에 대해서 설명한다.

II. 관련 연구

1. 검증조건 생성

명세 된 프로그램을 검증조건으로 바꿔 검증하는 방법은 프로그램 검증에 많이 사용된다. 하지만 프로그램은 다양한 언어로 만들어지기 때문에 언어나 구조에 따라 검증조건 생성 방법에 약간씩 변화가 필요하다. 구조화 프로그램과 비구조화 프로그램도 검증조건 생성 방법이 조금 다르다.

일반적으로 검증조건을 생성할 경우 그림 2와 같은 단계를 거친다. 소스 단편을 보호 명령(Guarded command)이라는 검증조건의 정보를 가지고 있는 중간표현으로 변환한다. 보호 명령에 대한 자세한 내용은 다른 논문에 서술되어 있다[4, 5, 6]. 예전에는 보호 명령단계가 끝난 후 바로 검증조건으로 변환하였지만 배정문 등의 원인으로 검증조건이 중복 생성되어 프로그램 크기에 비해 너무 많은 검증조건이 생성되는 문제가 있었다. 그래서 검증조건을 생성하기 전 패시브 폼으로 변환하는 단계를 집어넣어 중복을 유발하는 원인들을 제거하도록 한다. 패시브 폼의 필요성과 방법에 대해서는 [7]에 서술되어 있다. 패시브 폼으로의 변환까지 완료된 프로그램은 검증조건 생성 단계로 들어간다. 검증조건 생성단계에서는 프로그램을 최약 전조건 계산 규칙을 이용하여 검증조건으로 변환한다.

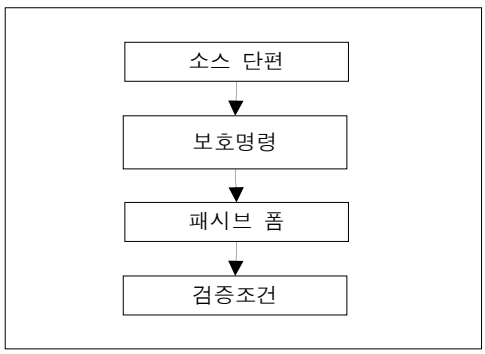

그림 2. 검증조건생성 과정
Fig. 2. The creation of verification conditions

비구조화 프로그램은 구조화 프로그램과 다르게 goto문과 같은 분기문이 있다. 그래서 구조화 프로그램과 달리 비구조화 프로그램은 검증조건을 생성하기 위해 보호 명령으로 변화할 때 명령어가 추가된다. goto와 같은 명령문을 처리하기 위해서 프로그램의 분기 정보를 포함할 수 있어야 하기 때문에 프로그램의 블록 정보를 포함할 수 있어야 한다. 이 후 패시브 폼으로 변환한 후 검증조건 생성단계에서 goto문으로 분기되는 블록들을 and연산자를 이용하여 연결한다[8]

그림 2와 같은 과정으로 생성된 검증조건은 정리 증명기를 이용하여 검증조건의 유효성을 판단한다.

2. GML

GML(Graph Modeling Language)[9] 은 그래프를 묘사하기 위해 계층 ASCII를 기반으로 하는 파일 형태이다. 정보를 그래프로 표현할 때 사용하는 언어로서 그래프를 구성하고 있는 노드와 간선의 정보를 표현할 수 있는 언어이다. GML 파일을 이용하여 그래프의 정보를 표현하고 GML 파일을 시각화 해주는 프로그램에 입력으로 넣으면 그래프를 그릴 수 있다.

일반적인 프로그램은 필요한 정보를 그래프 형태로 가진다. 하지만 각각 자신만의 그래프 형태를 가짐으로써 다른 프로그램간의 그래프 교환은 거의 불가능하다. 그래프를 통한 간단한 데이터 교환과 같은 작업은 더 어렵다. 다른 프로그램간의 그래프 교환을 가능하게 하기 위해 만들어진 언어가 GML이다. GML 파일은 단순한 구조를 가지고 있고 그래프의 정보를 가지고 있는 GML 파일을 이용하여 다른 프로그램의 그래프와의 정보 교환이 용이하다. 또한 큰 프로젝트에서 여러 단계에서 다른 정보를 그래프로 표현할 경우 GML 파일로 그래프의 정보를 저장한다면 하나의 시각화 프로그램만으로도 시각화가 가능하기 때문에 확장성이 좋고 유연성이 있다.

```
computationfunction TF :( int x ) -> int{
                read( x )
                block 0 : 0 : ifd  ¬(x<10)  2
                block 1 : 1 : vreturn  1
                block 2 : 2 : vreturn  0

                from 0 to 1 when x<0
                from 0 to 2 when ¬ x<10
                returnblock 1, 2
        ensure result =1 ∨ result = 0
}
```

그림 3. 비구조화 프로그램
Fig. 3. Unstructured program

III. 검증조건 생성과 검증조건의 시각화

그림 3은 비구조화 프로그램의 예이다. 그림 4는 그림 3의 비구조화 프로그램을 검증조건생성 방법[8]을 적용하여 검증조건으로 변환한 결과를 보여준다. 그림 3의 비구조화 프로그램은 바이트 코드의 검증을 위해 사용되는 Birs[10]라는 중간언어형태이다. 비구조화 프로그램의 검증조건 생성 과정은 그림 2의 방법을 따르지만 몇몇 단계에서 구조화 프로그램에서의 방법과 차이가 있다. 이 장에서는 그림 3의 비구조화 프로그램이 검증조건으로 변환되는 과정에 대해서 간단히 설명하고 검증조건의 시각화의 필요성과 시각화 프로그램의 구현에 대해서 설명한다.

$(x_0<10 \Rightarrow result_0=1 \Rightarrow result_1 = result_0 \Rightarrow$
$(result_1 = 1 \vee result_1 = 0) \wedge true) \wedge$
$(\neg(x_0<10) \Rightarrow result_0=0 \Rightarrow result_1 = result_0 \Rightarrow$
$(result_1 = 1 \vee result_1 = 0) \wedge true)$

그림 4. 비구조화 프로그램의 검증조건 생성 결과
Fig. 4. Results of the verification conditions generation for unstructured programs

1. 검증조건의 생성

1.1 보호명령

비구조화 프로그램은 goto문과 같은 명령문에 의한 비구조적인 문제로 인해 구조화 프로그램에서 사용되는 보호명령 문법을 그대로 사용할 수 없다. 그래서 비구조화 프로그램의 보호 명령 문법은 구조화 프로그램과는 조금 다르다. 비구조화 프로그램의 보호명령은 goto문과 같은 명령문에 의한 프로그램 흐름 변화가 발생하고, 이를 표현하기 위해 블록단위로 프로그램을 나누어 표현한다. 블록단위의 표현으로 인해 프로그램의 흐름을 표현할 수 있도록 하여 보호명령 역시 비구조적 형태를 가진다는 것이 구조화 프로그램의 보호명령과는 다른 점이다.

Startbe : Startok ≡ Thenok ∧ Elseok
Thenbe : Thenok ≡ $x_0<10 \Rightarrow result_0=1 \Rightarrow result_1 = result_0$ ⇒ Endok
Elsebe : Elseok ≡ $\neg(x_0<10) \Rightarrow result_0=0 \Rightarrow result_1 = result_0$ ⇒ Endok
Endbe : Endok ≡ Afterok
Afterbe : Afterok ≡ $(result_1 = 1 \vee result_1 = 0) \wedge true$

그림 5. 최약 전조건 변환을 적용한 후의 각 block의 검증 조건
Fig. 5. The VC of each block after weakest precondition transformation

1.2 패시브 폼

패시브 폼 변환을 통하여 변수가 두 번 이상 배정되지 않도록 함으로써 검증조건의 중복을 줄여 검증조건의 크기를 줄일수 있다. 비구조화 프로그램의 보호명령을 패시브 폼으로 변환하는 과정은 일반적인 구조화 프로그램의 검증조건생성 과정에서 패시브 폼을 생성하는 방법과 같은 방법을 사용한다.

1.3 검증조건생성

구조화 프로그램의 검증조건생성 단계에서는 검증조건의 중복을 막기 위해 최약 전조건 변환법을 그대로 사용하는 것이 아닌 패시브 폼 문장의 결과를 예측하여 결과별로 검증조건생성 규칙을 적용한다. 하지만 비구조화 프로그램에서는 최약 전조건 변환법을 그대로 적용한다. 그래서 비구조화 프로그램의 검증조건은 구조화 프로그램의 검증조건보다 중복이 더 발생하게 된다.

또한 비구조화 프로그램은 블록별로 나누어져있는 프로그램의 검증조건을 생성하기 위해 그림 5와 같이 블록단위로 최약 전조건 변환법을 적용하여 블록단위로 검증조건을 생성하고 다음 식을 이용하여 최종적인 검증조건을 생성한다.

Startbe ∧ Thenbe ∧ Elsebe ∧ Endbe ∧ Afterbe ⇒ Startok

그림 4에서 보이는 것과 같이 생성된 검증조건은 가독성이 매우 떨어진다. 따라서 다음 장에서 가독성이 떨어지는 검증조건의 가독성을 높이기 위해 검증조건의 시각화에 대해서 설명하고 이 장에서 보인 예제를 이용하여 시각화 예를 보인다.

2. 검증조건의 시각화와 설계

2.1 검증조건 시각화의 필요성

프로그램의 신뢰성을 높이기 위한 일환으로 검증조건을 이용한 프로그램 검증 방법은 많이 연구되어지고 있다.

그런데 연구자들이 실제 검증조건 생성기를 연구 및 구현할 경우 생성된 검증조건의 내용을 확인해야 하는 경우가 많이 있다. 예를 들어 검증조건 생성기를 실제 구현할 때 잘못된 코드 생성으로 인한 검증조건의 오류등과 같은 문제가 발생할 수 있다. 그래서 연구자들은 생성된 검증조건이 문제가 있는지 확인할 필요가 있다. 하지만 생성된 검증조건은 그림 4에서 보이는 것과 같이 텍스트 형태로 되어 있어 검증조건이 길어지거나 복잡할수록 가독성이 떨어지는 문제가 있다. 검증조건의 가독성을 높인다면 검증조건 생성과 관련된 연구를 진행하는 연구자 및 학생들에게 도움이 될 수 있다. 그래서 본 논문에서는 검증조건의 시각화를 통해 가독성을 높이는 방법

에 대해 제안하고 구현하였다.

이 장에서는 그림 4와 같이 생성된 검증조건을 확인하기 쉽도록 시각화하는 방법을 논한다.

2.2 검증조건의 그래프 표현 규칙

이 장에서는 검증조건의 시각화 방법에 대해서 설명한다. 비구조화 프로그램의 검증조건을 시각화 하는 것이기 때문에 그림 5와 같이 블록 정보를 포함하고 있는 형태의 검증조건이 시각화를 통해 정보를 얻기에 좋다.

본 논문에서 사용한 검증조건 생성기에서는 그림 5의 검증조건을 그림 8의 BNF에 맞게 표현한다. 그림 8의 BNF를 사용하여 생성된 모든 검증조건들은 본 논문의 시각화프로그램을 이용하여 시각화 할 수 있다.

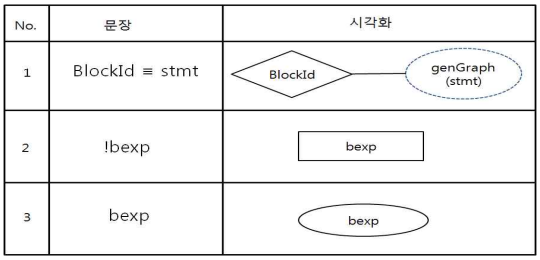

그림 6. 검증조건의 노드 표현 규칙
Fig. 6. Rule of node expression in verification condition

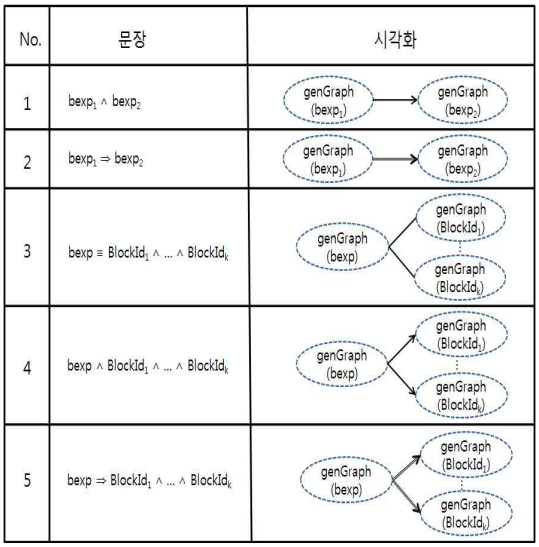

그림 7 간선 변환 규칙
Fig. 7. Rule of edge translation

시각화 프로그램은 검증조건을 노드와 간선으로 이루어진 그래프로 시각화한다. 노드는 각각의 블록 아이디와 표현식들이고 간선은 관계를 표현한다. 검증조건을 그래프로 시각화하는 규칙은 그림 6과 그림 7에 해당한다. 그림 6은 노드 표현 규칙이다. 검증조건이 블록 정보를 포함하고 있기 때문에 블록과 표현식을 구별해서 노드 모양을 다르게 표현한다. 블록 아이디는 마름모 모양의 노드로 표현하고 블록에 해당되는 검증조건들을 실선을 이용하여 연결한다. 점선으로 그려진 타원은 함수의 인자에 따라 다른 결과가 나올 수 있다. 그림 6의 genGraph는 입력받은 문장을 가지고 그래프를 생성하는 함수를 의미하며 인자로 들어온 문장에 적합한 노드 모양과 간선을 결정하고 생성한다. genGraph가 노드를 생성할 때는 그림 6의 규칙을 사용하고 간선을 생성할 때는 그림 7의 규칙을 이용한다. 기본적인 표현식은 타원을 이용하여 표현한다. 표현식에 Not이 있을 경우에는 직사각형 모양의 노드를 이용하고 노드 안에는 Not을 제외한 표현식만을 넣는다.

간선은 표현식을 연결하고 있는 연산자에 따라 다른 모양으로 표현된다. 각 블록의 검증조건들은 각 블록 아이디 노드에 실선으로 연결된다. 그림 7은 간선 표현 규칙이다. 그림 7의 bexp는 표현식을 의미한다. 표현식들이 and 연산자로 연결되어 있으면 화살표 모양의 간선으로 표현하고 함축 연산자로 연결되어 있으면 선이 이중으로 되어있는 화살표 모양으로 표현한다. 그림 5와 같은 검증조건은 비구조화 프로그램이기 때문에 goto문을 가지므로 각 블록의 검증조건의 마지막은 goto문에 의해 이동하는 블록아이디의 정보가 들어간다. 검증조건 마지막에 나열되어있는 블록아이디들은 그림 7의 규칙 4, 규칙 5를 이용하여 간선 모양을 결정하고 노드를 연결한다. 블록아이디 바로 다음에 표현식 없이 블록아이디들의 나열이 올 수 있는데 이 경우에도 마찬가지로 그림 7의 규칙 3처럼 실선을 이용하여 해당 블록 아이디에 하나씩 연결해준다.

```
<VC> -> <blockId> <expression>
<expression> -> true
              | false
              | <arithExpr><Relop><arithExpr>
              | <expression><Boolop><expression>
              | !<expression>
              | id
<blockId> -> id
```

그림 8 검증조건의 BNF
Fig. 8. BNF of verification condition

검증조건은 원래부터 표현식에 존재했던 불 표현식과 assert와 assume에 의해서 생성된 불 표현식이 있다. 최약전조건은 최우단유도 형태이므로 검증조건 생성하기 전에 원래 존재했던 불 표현식과 검증조건 생성으로 인해 assert와 assume에서 나온 불 표현식을 구별할 수 있다. 검증조건을

그래프로 변환할 때는 검증조건을 생성할 때 나타난 and와 함축 연산자를 가지고 간선의 모양을 결정한다. 그래서 시각화 된 그래프를 통해 검증조건의 assert와 assume관계를 확인할 수 있다. 또 노드 안에 쓰여진 표현식으로 노드의 패시브 폼 변환 결과도 확인하기가 쉽다.

2.3 예제

이 장에서는 표현 규칙을 이용하여 생성하려는 그래프의 형태를 보인다. 그림 6과 그림 7의 노드와 간선 변환 규칙을 이용하여 그림 5의 검증조건을 그래프로 변환하였을 때 최종적으로 생성되는 그래프는 그림 9와 같다.

Start 블록은 이후 두 개의 블록으로 이동하는 흐름만을 정보로 가지고 있기 때문에 그림 7의 3번 규칙을 이용하여 그린다. 그래서 마름모 형태의 노드의 Start 노드에 Start노드의 흐름 정보인 Then과 Else 노드를 실선으로 연결하고 있다. Then과 Else노드는 각각 블록의 아이디이기 때문에 해당 블록의 검증조건을 포함하고 있고 그 검증조건의 첫 번째 노드를 실선을 이용하여 연결하고 있다. 각 검증조건들은 그림 6의 2번과 3번 규칙에 따라 노드의 타원과 직사각형 모양의 노드를 가지고 그림 7의 1번과 2번 규칙에 따라 화살표와 이중화살표 형태의 간선을 가진다.

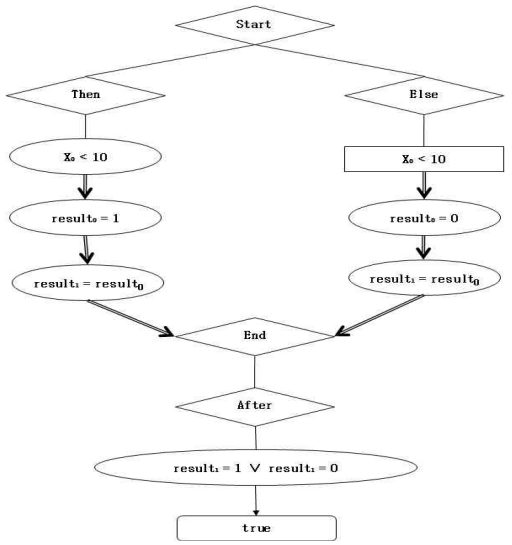

그림 9. 그림 5의 검증조건을 그래프로 변환한 결과
Fig. 9. The result of converting VC of Fig. 5 to the diagram

그림 9의 그래프에서 보이는 것처럼 각 검증조건이 포함되어 있는 블록과 프로그램을 구성하고 있는 블록들의 정보를 간단한 노드만으로 확인할 수 있다. 검증조건에 패시브 폼 변환이 제대로 되었는지도 노드에 적혀있는 검증조건으로 확인할 수 있다. 또한 간선의 모양으로 각 노드가 assume과 assert 중 어디에 영향을 받는지도 확인할 수 있다. 그림 9와 같은 형태라면 구현의 문제로 인해 assume과 assert가 잘못 적용되었거나 패시브 폼이 잘못 되었거나 하는 문제를 한눈에 알 수 있다. 또한 프로그램 흐름 정보를 제대로 가지고 있는 검증조건인지도 확인이 가능하다.

3. 시각화의 구현

3.1 구현환경

프로그램의 구현은 리눅스 2.6.35 버전의 우분투 10.10 버전에서 수행되었으며 구현을 위해 사용한 언어는 OCaml 버전 3.12로 Sawja 라이브러리 1.2 버전와 이클립스 helios 버전을 이용하여 구현하였다.

3.2 시각화 프로그램의 구현

이 장에서는 시각화 프로그램의 구현에 대해서 서술한다. 이 장에 나오는 모든 함수들은 OCaml을 이용하여 구현하였다. 검증조건의 시각화를 위해 그래프 생성단계에서 그래프의 정보를 GML로 표현한다. 그리고 검증조건이 변환된 형태의 그래프 정보를 포함하고 있는 GML파일을 생성한다. GML 파일은 그래프의 노드와 간선의 정보를 포함하고 있으며 필요에 따라 더 많은 정보를 추가할 수 있다. 그래프 생성단계에 생성한 GML파일의 정보를 프로그램을 이용하여 사용자에게 제공한다. 사용자는 프로그램을 이용하여 간단하게 GML파일의 정보를 그래프로 그려진 형태로 확인할 수 있다.

```
module StringIntString = struct (*node*)
 type t = string* int *string (* label, name ,shape *)
          let compare = compare
  let hash = Hashtbl.hash
  let equal = (=)
end

module String = struct (*edge*)
  type t = string
  let compare = compare
  let hash = Hashtbl.hash
  let equal = (=)
  let default = ""
end

module G =
Imperative.Digraph.ConcreteBidirectionalLabeled(StringIntSt
ring)(String)
```

그림 10. GML의 노드와 간선 구조 생성
Fig. 10. generating shape node and edge of GML

GML을 이용하여 그림 6과 그림 7의 규칙에 따라 노드와

간선을 표현하기 위해 노드와 간선이 포함하고 있는 정보를 담을 수 있는 틀을 생성해야 한다. 그림 10은 각각의 노드와 간선의 추가하고 싶은 정보를 GML로 표현하기 위해 구현한 틀이다. 그림 10으로 인해 만들어진 틀에서 노드는 추가적으로 문자열 2개와 정수 하나의 정보를 입력받을 수 있고 그 정보는 각각 노드의 label과 이름과 모양을 나타낸다. 간선은 간선의 모양을 나타내는 문자열의 정보를 추가적으로 입력받을 수 있다. 노드는 노드의 id정보를 기본으로 포함하고 있어 GML파일을 생성할 때 자동으로 id를 설정해주고 간선은 목적노드와 소스노드의 정보를 기본으로 입력시켜 주어야한다. 시각화에 더 많은 정보가 필요할 때는 그림 10의 틀을 수정시켜 보다 많은 정보를 포함시킬 수 있다.

```
node [
    id 1
    label "IF"
    number 0
    shape "diamond"
 ]

edge [
  source 2
  target 5
  label "D_arrow"
]
```

그림 11. 노드와 간선의 GML 표현
Fig. 11. Expression of node and edge in GML

그림 11은 그림 10으로 인해 생성되는 GML의 간단한 형태이다.

```
function : graph generation
Input : block ID, statement in block
output : graph
---
let create_gml_bi g (VC( Id(bi) , stmt))=
            let n = G.V.create(bi, 0, "diamond") in
G.add_vertex g n;
            let arw = "Line" in
            let (vertex, g) = create_node g stmt "Null"
        in
            let graph = create_edge n arw vertex g in
        graph
```

그림 12. 그래프 생성 함수
Fig. 12. Graph generation function

비구조화 프로그램의 검증조건을 시각화 할 경우 그림 5와 같이 블록으로 나뉘어진 형태의 검증조건이 시각화했을 때 좀 더 정보를 알기 쉽다. 그래서 그림 5와 같이 블록별로 구분할 수 있는 정보를 시각화에 반영하였다. 그림 6의 Id를 마름모 형태로 변환한 것은 블록 정보를 반영하기 위한 것이다. 그림 12는 각 블록의 Id를 마름모로 표현하여 블록 정보를 포함시키고 그 블록내의 문장들의 정보를 시각화하기 위한 함수를 호출하는 함수이다. create_node함수는 G.V.create를 통하여 블록의 이름 정보를 포함하는 노드를 마름모 형태로 생성하여 그래프의 정보에 포함시키고 create_node 함수를 이용하여 블록내의 문장들의 노드를 생성한다. 그리고 create_edge 함수를 이용하여 생성된 노드들을 연결시켜주는 간선을 생성한다. 생성된 그래프는 현재 블록의 id와 블록 내에 포함되는 모든 문장들의 정보와 흐름을 이전에 생성된 정보에 추가로 포함한다. 그림 12의 함수를 모든 블록에 적용하면 모든 시각화 정보를 포함하고 있는 최종 그래프가 생성된다.

```
function : node generation
input : statement, graph
output : node list, graph
---
let rec create_node g stmt n=
 match stmt with
 |VC_header.RelExpr (a1, op, a2) ->
        let s = make_string a1 op a2 in
        let l = check_name s in
        if(n = "Not")
                then let v =
            G.V.create( s, l, "rectangle") in
            G.add_vertex g v; ([(v,n)], g)
                else let v =
            G.V.create( s, l, "ellipse") in
            G.add_vertex g v; ([(v, n)], g)
 |VC_header.NotExpr a ->
        let ( v, g ) = create_node g a "Not" in (v, g)
 | VC_header.BoolExpr (b1, op, b2) ->
        let d_op = determine_arrow op in
    if( d_op = "And" || d_op = "Or" || d_op = "IM")
            then
             let (v, g) =
               create_one b1 b2 d_op n g in (v, g)
            else(
             let (v1,  g1) = create_node g b1 n in
             let (n1, a1) = get_element v1 in
                 let (v2, g2) = create_node g1 b2 n in
                 if(a1 = "Line") then (v1@v2, g2)
              else  ([(n1,d_op)]@v2, g2))
 |VC_header.TT ->let s = check_name "True" in
         let v = G.V.create( "True" , s , "ellipse" ) in
         G.add_vertex g v; ([(v, n)], g)
 |VC_header.FF ->
         let v =G.V.create( "False" , 0 , "ellipse" ) in
         G.add_vertex g v; ([(v, n)], g)
 |VC_header.Id s ->
         let v = G.V.create( s, 0, "diamond" ) in
G.add_vertex g v; ([(v, "Line")], g)
```

그림 13. 노드 생성 함수
Fig. 13. Node generation function

블록내의 각 문장들의 노드를 생성하는 함수인 create_node 함수는 그림 13과 같다. create_node함수는 블록 Id를 제외한 모든 문장들의 노드를 생성한다. 그림 6에 따르면 Not이 붙은 표현식은 사각형으로 표현하고 그렇지 않은 일반 표현식들은 타원 형태로 표현된다. 그래서 create_node 함수는 노드의 형태와 문장정보를 포함하는 노드 정보를 생성하고 생성한 노드와 다음 노드 사이의 간선 모양 정보를 쌍으로 만들어 리스트에 저장한다. 그리고 저장한 리스트와 그래프를 반환해준다. 간선 정보는 determine_arrow 함수를 통해 얻는다. create_node 함수에 의해 생성된 노드 정보와 간선정보를 이용하여 그림 12의 create_edge 함수를 이용하여 최종 그래프 정보를 생성한다. 마지막으로 생성한 그래프 정보를 GML파일로 생성해 준다.

시각화의 마지막 단계로 GML파일의 시각화 프로그램을 이용하여 실제 시각화를 시도한다. 실제 구현된 시각화 프로그램을 이용하여 생성된 실제 GML파일의 일부가 그림 14이다.

그림 14는 4.3절의 예제를 이용하여 GML파일을 생성한 예이다. 그림 14에서 보는 것과 같이 GML 파일은 node와 edge로 구성되어 있고 각각의 노드가 표현하는 문장의 정보 등을 가지고 있다. edge는 간선정보를 포함하고 있으며 간선의 형태와 목적노드 소스노드에 대한 정보 등을 가지고 있다.

그림 14의 GML파일을 시각화 프로그램을 이용하여 시각화 한 결과가 그림 15와 같다. 그림 15의 그래프는 그림 9와 같은 형태로, 그림 9의 텍스트 형태의 검증조건보다 내용을 확인하기 쉽다.

```
node [
  id 12
  label "True"
  number 2
  shape "ellipse"
]
node [
  id 13
  label "True"
  number 4
  shape "ellipse"
]
node [
  id 14
  label "Then"
  number 0
  shape "diamond"
]
node [
  id 15
  label "result0=1 Or result0=0"
  number 0
  shape "ellipse"
]
edge [
  source 1
  target 9
  label "Line"
]
edge [
  source 1
  target 14
  label "Line"
]
edge [
  source 2
  target 5
  label "D_arrow"
]
```

그림 14. 그림 5의 내용을 그래프로 변환한 GML 파일 일부

Fig. 14. Some of the GML file for transformation Fig. 5 to graph

Ⅳ. 결 론

본 논문에서는 생성된 검증 조건의 시각화를 통해 검증 조건의 가독성을 높이는 시도를 하였다. 최종적으로 생성되는 검증조건은 제어구조가 명확하지 않아서 검증조건 생성 시 발생하는 문제나 검증조건의 내용을 확인하기 어렵다. 하지만 제어구조를 파악하기 쉬운 중간단계의 블록단위의 검증조건에 대해 시각화하게 되면, 시각화된 내용을 토대로 검증조건 생성기 구현시간을 단축할 수 있다. 본 논문에서는 그래프를 이용한 시각화를 통해서 검증조건의 가독성을 높여 검증조건을 쉽게 확인할 수 있도록 한다. 이를 통해 검증조건 생성기 개발 및 구현 등에 있어 구현시간을 단축할 수 있다. 검증조건의 요소가 되는 블록 Id와 블록내의 식들을 구분해서 다른 모양의 노드로 표현하고 검증조건의 요소들을 연결하고 있는 논리 연결자를 표현하기 위해 간선을 사용한다. 검증조건의 시각화 정보는 GML로 표현되어 파일로 저장되고 GML 파일은 간단한 프로그램으로 그래프를 그려준다. GML파일은 필요에 따라 정보를 쉽게 추가할 수 있기 때문에 시각화에 용이하다. 시각화 된 검증조건은 보호명령 형태에 대한 정보 및 패시브 폼으로 변환 되었을 때의 형태도 확인할 수 있기 때문에 문제를 발견하기 쉽다.

향후 연구과제로는 검증조건의 크기가 커짐에 의해 그래프가 복잡하게 커지는 점을 고려하여 블록 단위로 그래프를 확장 축소할 수 있는 기능을 추가할 예정이다.

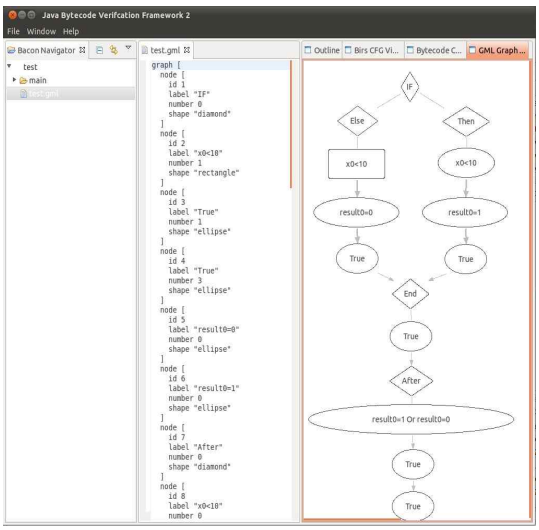

그림 15. GML시각화 프로그램을 이용한 시각화
Fig. 15. Visualization VC using GML visualization program

참고문헌

[1] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. “An overview of JML tools and applications.” Int. J. Softw. Tools Technol. Transf. 7, 3, pp.212-232, June 2005.

[2] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. “Extended static checking for Java.” SIGPLAN Not. 37, 5, pp.234-245, May 2002.

[3] JeMin Kim, JoonSeok Park, WeonHee Yoo, “A Design of Verification Framework for Java Bytecode”, The Korea Society of Digital Industry & Infromation Management , June 2011

[3] K. Rustan M. Leino, Todd Millstein, and James B. Saxe. "Generating error traces from verification-condition counterexamples." Sci. Comput. Program. 55, 1-3, pp.209-226, March 2005.

[4] Edsger Wybe Dijkstra. "A Discipline of Programming (1st ed.)." Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[5] Greg Nelson. "A generalization of Dijkstra's calculus." ACM Trans. Program. Lang. Syst. 11, 4, pp.517-561, October 1989.

[6] Ralph-Johan J. Back, Abo Akademi, J. Von Wright. "Refinement Calculus: A Systematic Introduction (1st ed.)." F. B. Schneider and D. Gries (Eds.). Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[7] Cormac Flanagan and James B. Saxe. "Avoiding exponential explosion: generating compact verification conditions." In Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '01). ACM, New York, NY, USA, pp.193-205, 2001.

[8] Mike Barnett, K. Rustan M. Leino, “ Weakest-precondition of unstructured programs”, ACM SIGSOFT Software Engineering Notes, v.31 n.1, pp.82-87, January 2006

[9] GML: A portable Graph File Format http://www.lkn.ei.tum.de/arbeiten/faq/guidelines/gml-tr.html

[10] SeonTae Kim, JeMin Kim, JoonSeok Park, WeonHee Yoo, “BIRS ; ByteCode Intermediate Representation With Specification” The 35th Conference of the KIPS, 18, 1, p.265-268 May 2011

[11] Andreas Gal. "Efficient Bytecode Verification and Compilation in a Virtual Machine." Ph.D. Dissertation. University of California at Irvine, Irvine, CA, USA. Advisor(s) Michael Franz. AAI3243940, 2006.

[12] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. "Checking Java Programs via Guarded commands." In Proceedings of the Workshop on Object-Oriented Technology, Ana M. D. Moreira and Serge Demeyer (Eds.). Springer-Verlag, London, UK, pp.110-111, 1999.

[13] Mike Barnett, Bor-Yuh Evan Chang, Robert

DeLine, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs." In Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005

[14] Jean-Christophe Filliatre, "Why : a multi-language multi-prover verification tool," LRI-CNRS UMR 8623, Université Paris Sud, March 2003

[15] The Coq Proof Assistant. http://coq.inria.fr/.

[16] The PVS Specification and Verification System. http://pvs.csl.sri.com/.

[17] John Harrison. HOL Light. http://www.cl.cam.ac.uk/users/jrh/hol-light/.

[18] The Mizar project. http://mizar.uwb.edu.pl/.

[19] The Simplicify decision procedure(part of ESC/Java). http://research.compaq.com/SRC/esc/simplify/.

[20] Silvio Ranise and David Déharbe. The haRVey decision procedure. http://www.loria.fr/~ranise/haRVey/.

저 자 소 개

허 혜 림

2009년 8월: 인하대학교 정보공학부 공학사.

2012년 2월: 인하대학교 컴퓨터·정보공학과(공학석사)

관심분야: 프로그램 분석&검증

Email : lena03@naver.com

김 제 민

2006년 2월 : 인하대학교 컴퓨터공학부 (공학사)

2008년 2월 : 인하대학교 정보공학과(공학석사)

2008년 3월~현재 : 인하대학교 컴퓨터정보공학과 박사과정

관심분야 : 프로그램 분석, 프로그래밍 언어, 컴파일러

Email : jeminya@hanmail.net

박 준 석

2000년 2월 : 미국 남가주대학교 컴퓨터과학과(공학석사)

2004년 7월 : 미국 남가주대학교 컴퓨터과학과(공학박사)

2004년 8월~2006년 : 삼성전자 SoC 연구소

2006년 3월~현재 : 인하대학교 컴퓨터정보공학부 부교수

관심분야 : 고성능 컴퓨팅, 병렬 컴파일러, 컴퓨터 구조

Email : joonseok@inha.ac.kr

유 원 희

1975년 2월 : 서울대학교 응용수학과(이학사)

1978년 2월 : 서울대학교 계산학과(이학석사)

1985년 2월 : 서울대학교 계산학과(이학박사)

1979년~현재 : 인하대학교 컴퓨터 정보공학과 교수

Email : whyoo@inha.ac.kr