□研究論文□

韓國軍事運營分析學會誌
第16卷, 第2號, 1990. 12. 31.

Ada 프로그램의 Visibility Graph 생성모델에 관한 연구 (A Study on Visibility Graph Generating Model of Ada Program)

鄭仲泳 · 金禧宙 · 尹昌燮*

Abstract

Programming-in-the-Large refers to software development environment and includes the organization and representation of a system structure, module decomposition, component dependence analysis, seperate compilation, subsystem and composition identification. The most intricate problem in this environment is the mastery of the structural complexity of large software systems. Ada programming language is tailored to the needs for building of large, integrated software systems from many program units.

The visibility graph generating model presented in this paper transforms Ada source program into a visibility graph with nodes for program units and edges for visibility relations among program units. The system description in terms of program units and their visibility relations produced by this model can be utilized for some apects of Programming-in-the-Large environment and also assists designeers, programmers, integrators and maintainers in defining, understanding and exploring the structure of evolving software systems.

The model designed and implemented in Ada progrmming language runs on PCs and will remain useful both in practice and as experimental tool.

* 國防大學院

제1장 서 론

Ada의 대형 소프트웨어 시스템은 다른 시간, 다른 장소에서 여러사람들에 의해서 작성된 프로그램 단위(program unit)들을 모아서 점진적으로 구성, 개발된다.

Programming-in-the-Large에서 사용되는 언어는 Programming-in-the-Small에서 사용되는 언어가 제공하지 못하는 모듈을 시스템으로 통합 구성할 수 있는 능력, 그리고 프로그래머의 의도를 정형적으로 기록하여 시스템의 구조를 볼 수 있는 능력, 또한 컴파일러를 통하여 시스템의 일관성을 검사할 수 있는 능력을 제공하는 것이 바람직하다[5, 16, 17].

Ada언어는 대형의 복잡한 소프트웨어 시스템의 설계와 구현에 적합한 언어이며, 하향식(top-down)과 상향식(bottom-up) 개발방법에 의하여 소프트웨어 시스템을 설계하고 구현할 수 있도록 이를 지원한다. Ada의 프로그램은 하나 이상의 프로그램 단위로 구성되며, 각 단위는 분리 컴파일이 가능하다. 또한 Ada의 프로그램 단위는 명세와 몸체로 이루어지며 이 두 부분은 서로 분리하여 컴파일 될 수 있다[1, 3].

Ada의 이러한 특성은 Programming-in-the-Small에서 필요한 도구들을 능가하는 도구들을 필요로 하게 한다. 소프트웨어 시스템을 구성하는 구성단위들의 계층적 구조를 표현하고, 분석하며, 구성단위를 관리하는 광범위한 지원을 제공하는 도구들이 필요하다[10, 15. 16]. 특히 소프트웨어의 개발 및 유지보수단계에서 소프트웨어의 시스템의 구조를 파악하고 이해하여야만 점진적인 개발 그리고 유지보수를 효율적으로 실행할 수 있다.

프로그램 단위간의 가시성 관계의 규명은 Ada를 이용한 프로그램 구조에 관한 전반적인 정보를 제공하기 때문에 프로그램 그 자체의 코드보다 프로그램의 구조를 이해하는 데 필수적인 자료이다. 가시성 관계의 규명은 시스템을 설계하고, 시스템을 점진적으로 구축할 때 유용하게 사용되며 특히, 유지보수단계에서 시스템에 관한 문서의 적절한 지원을 받지 못하기 때문에 Source Program으로부터 가시성을 규명한다는 것은 대단히 중요한 것이다.

본고의 목적은 Ada의 Source Program을 입력하고, 그 출력으로는 프로그램을 구성하고 있는 구성단위간의 가시성 관계를 생성하는 모델을 제안하는데 있다.

본고의 구성은 제2장에서 Ada언어의 특징과 가시성 개념 그리고 기존의 연구에 관하여 고찰하고, 제3장에서 Ada 프로그램의 가시성 그래프 생성 모델을 설계하고 구현하며, 제4장에서 검증한다.

제2장 Ada 언어의 특성과 가시성 개념 고찰

제1절 Ada 언어의 특성

Ada 언어는 여러사람이 여러장소에서 독립적

으로 분산 개발한 프로그램 구성단위를 조립하여 완전한 시스템으로 구축 가능하도록 하며, 이러한 분산 개발의 개념들은 Ada의 분리 컴파일(separate compilation), 일반화 프로그램단위(generic program unit) 등으로 지원되고 있다.

Ada 프로그램 단위는 명세부분(specification part)과 몸체부분(body part)으로 구성되며, 명세부분은 프로그램의 사용자에게 필요한 정보를 보여주는 부분이며, 몸체부분은 프로그램 구현의 세부사항을 포함하고 있고 사용자로부터 은폐되어 있는 부분이다.

2.1.1. Ada 프로그램 단위

Ada 소프트웨어 시스템은 하나 이상의 프로그램 단위(program unit)로 구성되고, 타스크를 제외한 각 단위는 분리 컴파일이 가능하다. 프로그램 단위는 서브프로그램(subprogram), 패키지(package), 타스크(task), 일반화 단위가 있다.

다음은 서브프로그램, 패키지, 일반화 단위에 대해 구체적으로 고찰하고자 한다. 타스크는 실시간 문제 해결의 Ada 프로그램이므로 본고의 범위에서 제외한다.

2.1.1.1. 서브프로그램

사용자의 관점에서, 프로그램이란 일련의 순차적 또는 병렬적 행위를 통하여 다른 대상과 상호작용을 하는 대상들의 집단이라 할 수 있다. 가장 훌륭한 프로그램의 형태란 이러한 순차적 또는 병렬적 행위가 실세계의 알고리즘 추상화를 직접 반영하고 있는 형태이다. 바로, 이러한 높은 수준의 알고리즘을 정의하기 위한 메카니즘이 Ada의 서브프로그램이다.

서브프로그램은 Procedure 그리고 Function으로 분류되는데, 기존 언어에서 사용되는 Procedure, Function과 구조적인 면에서 유사하다. 서브프로그램은 외부와의 접속관계(interface)를 위하여 매개변수(parameter)를 가질 수 있다. 서브프로그램도 다른 프로그램 단위와 마찬가지로 명세부분과 몸체부분으로 나누어져 있다.

명세부분은 서브프로그램과 외부와의 접속관계 및 호출규정을 정의하는 부분이며, 자료의 형식을 강조하기 위하여 매개변수와 변수의 형식을 정의하고 있다. 서브프로그램 명세는 서브프로그램을 선언하는 부분인데 반해, 서브프로그램 몸체는 알고리즘의 실행절차를 구현한 부분이다. 이 부분은 일련의 국부적인 선언(local declaration)및 내포된 서브프로그램을 가질 수 있다.

2.1.1.2. 패키지

패키지는 논리적으로 관련된 개체들(logically related entities) 또는 계산에 사용되는 자원들(computational resources)을 포장(encapsulate)하고 있으며, 자료형식, 자료대상, 서브프로그램, 타스크, 그리고 다른 패키지까지도 포함할 수 있다.

일반적으로 패키지는 명세와 몸체의 두 부분으

로 구성되어 있다. 명세는 패키지의 가시적인 부분이며, 사용자들에게 개체들을 보여주는 부분이다. 패키지의 사용자는 단지 이 가시적인 부분을 보고 패키지를 이용할 수 있고, 몸체가 포함하고 있는 구현의 세부사항은 알 필요가 없다. 패키지의 구조는 분할성, 추상화, 국부화 및 정보은폐를 직접적으로 지원해준다. 특히 이 두 부분은 분리 컴파일이 가능하므로, 이들을 물리적으로 분리시키므로써 소프트웨어로 문제를 해결하는 과정에서 복잡한 문제를 분할하여 해결하는 데 도움이 된다.

2.1.1.3. 일반화 프로그램 단위(generic program unit)

Ada 언어에서 프로그램 단위를 인수화(parameterization)하여 일반적인 사용이 가능하도록 하는 메카니즘이 일반화 프로그램 단위이며, 일반화 패키지와 일반화 서브프로그램으로 구분된다.

자료의 형식만이 다르고 알고리즘이 같은 경우 각각의 프로그램을 작성하지 않고 일반화 프로그램 단위라는 하나의 단위로 작성하였다가 실제의 프로그램 기능에 알맞도록 실례화(instantiation)하여 사용할 수 있다. 다시 말하면 Ada의 일반화 프로그램 단위들은 하나의 템플레이트(templated)이며 실행되는 프로그램이 아니고, Translation Time에서 실행 가능한 프로그램으로 변환된다.

2.1.2. 프로그램 단위들간의 접속관계

프로그램 단위의 접속관계는 내포구조와 'With'문과 'Separate'문으로 나눈다.

2.1.2.1. 내포 구조

Ada 언어에서는 어느 한 선언의 범위(scope)안에 하나의 선언을 내포하는 여러가지 방법들을 제공한다. 예를 들면 서브프로그램 몸체나 패키지 몸체의 선언부는 다른 서브프로그램 이나 패키지를 포함할 수 있다.

내포 구조는 여러 계층으로 심화 될 수 있다. 내부(inner)선언의 프로그램 부분은 직접적이거나 또는 간접적으로 외부(outer)선언의 프로그램 부분에 의해서 둘러싸인다. 이 때에 내부선언에 속하는 프로그램 단위는 외부선언에 속하는 프로그램 단위에 내포되었다 라고 한다[4].

2.1.2.2. 'With' 문

대형 시스템의 생성시 상향식 접근방법(bottom-up approach)은 많은 모듈들이 공유할 것으로 예상되는 것부터 먼저 생성하고 점차 시스템의 관점에서 이들을 통합 구축하는 접근방법인데 Ada에서는 라이브러리 단위(library unit)와 'With'문 사용이 상향식 접근방법(bottom-up approach)를 지원하는 구문이다.

이전에 이미 컴파일된 라이브러리 단위를 이용하고자 할 때 'With'문을 사용하여 이들 프로그램 라이브러리에 대한 가시성을 새로운 프로그램 단위가 갖게 된다. 이러한 접근 방법의 잇점은

프로그래머들이 필요한 만큼의 구성단위들을 선택적으로 선별하여 시스템을 구축하는데있다. 컴파일하고자 하는 구성단위가 'With'문으로 다른 라이브러리 단위를 명시하고 있다면 'With'문에 참조된 라이브러리 단위가 먼저 컴파일된 다음에야 비로소 'With'문을 사용한 구성단위의 컴파일이 가능하다. 'With'문의 선언은 'With'문으로 참조된 라이브러리 단위가 가지고 있는 모든 개체들이 'With'문을 선언한 구성단위에서 사용될 수 있다는 뜻이며, 구성단위들간에 가시성을 유지하는 하나의 방법이다. 가시성 관계는 제2절에서 다루기로 한다.

2.1.2.3. 'Separate'문

일반적으로 프로그램은 계층적 구조의 성격을 이용하여 구축된다. Ada에서도 하향식 설계방법을 사용한다. 시스템을 분할하고 분할된 부분을 명세로 우선 설계하고 그 구체적인 실행에 관한 구현은 차후에 실시한다. 하위단위는 상위단위에서 명시되고 상위단위가 정의된 다음에 그리고 컴파일이 끝난 다음에 점진적으로 구현되면서 구현시마다 컴파일이 가능하다. 이 때에 사용하는 문장이 'Separate'이며 'Separate'문은 그 하위단위를 내포하는 상위단위임을 뜻한다. 'Separate'문도 앞에서 언급하였던 'With'문과 같이 프로그램 구성단위들 간에 일어나는 가시성 관계를 명시하고 있다.

제2절 Ada 언어의 가시성 개념 고찰

전통적으로 개체의 가시성(entity visibility)은 선언문(declaration), 범위(scope)와 바인딩(binding) 등의 용어를 사용하여 정의되어 왔다[15]. 프로그래밍 언어에서 개체란 언어의 구성요소로서, 하나의 주어진 명칭을 갖는 것이다. 따라서 개체는 대상, 자료형식, 지시문 또는 서브프로그램들이라 할 수 있다. 선언문은 개체를 실제로 제시하는 것이고 개체에 명칭을 부여하는 것이다. 선언문의 범위는 프로그램 코드 중에서 그 선언문이 의미를 갖고 사용될 수 있는 프로그램의 영역을 의미한다. 바인딩이란 식별자의 사용이 프로그램의 어는 한 장소에서 특정한 선언문과 연관을 갖도록 하는 것이다.

하나의 개체는 범위내에서만 가시적 일 수 있다. 용어의 뜻을 보다 명확히 설명하기 위하여, 예를 든다면, Ada에서 부프로그램, 패키지 또는 타스크의 명세부분에 정의된 개체의 식별자들은 그 식별자 사용범위가 몸체의 끝까지이고 몸체부분에 정의된 식별자의 범위는 몸체에만 국한되며 명세부분에서는 가시적 일 수 없다.

Ada 프로그램의 범위와 가시성 규칙은 프로그램내에서 선언된 명칭을 어느 곳에서 참조할 수 있는가와 각 식별자의 사용과 특정한 선언문이 어떻게 관련되는가를 나타내준다. 프로그램에서의 선언문은 프로그램내의 특정부분에만 영향을 미치고 그 외의 곳에서는 영향을 미치지 못한다.

어떤 식별자의 개체가 관계되어 있는 선언문의 범위내에서, 식별자가 그 개체를 참조하지 않는 곳이 있을 수 있다. 프로그램에서 식별자가 나타날 수 있는 곳과 특정위치에서 그것의 의미가 무엇인가에 대한 모든 질문은 범위와 가시성이라는 기본적인 두가지 개념으로 해결할 수 있다.

제 3 절 기존의 가시성 모델에 관한 연구

소프트웨어 시스템을 구성하는 개체들 사이에 존재하는 가시성 관계를 광의의 의미로 내포가 논의되어 왔으며, 현대 프로그래밍 언어에서 제공하는 뛰어난 가시성 제어방법이었다. 그러나 Ada, Smalltalk, Euclid, Gypsy, Mesa, Modula-2와 같이 언어가 다양화되면서 내포만으로는 부족했던 면들을 충족하기 위해서 가시성 제어를 위한 선택적이며 보완적인 방법들이 제공되었다.

Wolf[15]는 개체간의 가시성 관계를 접근 요청(requisition of access)와 접근 제공(provision of access)으로 구분되는 개념으로 설명하고 있다. 개체에 대한 접근은 선언문에 의하여 개체를 사용할 수 있는 권한이다. 접근 요청은 내부 또는 외부의 한 개체가 잠재적으로 어떤 개체들의 집합을 참조할 수 있는 권한을 요청할 때 발생한다. 대부분 프로그래밍 언어에서 그 자체에 대한 접근(국부적으로 선언된 개체에 접근)과 비국부적인 개체에 대한 접근을 요청한다.

접근 제공은 외부 또는 내부의 한 개체가 어떤 다른 개체들의 집합에 대해 잠재적으로 참조할 수 있는 권한을 부여하는 것이다. 예를 들면 내포 구조의 경우 한 서브프로그램은 그 서브프로그램의 상위계층(parent), 동일계층(sibling), 하위계층(descendant)에게 자신으로 접근할 수 있는 권리, 즉 그 서브프로그램을 실행(invoke)할 수 있는 권리를 부여하는 것이다.

Wolf의 가시성 제어 모델은 설계과정에서 개체의 가시성 관계를 논리적으로 표현하고 이를 이용하여 Module Interconnection Language의 개발에 중점을 두었다. 개체간의 가시성 관계는 다음 정의에 의해 가시성 그래프로 표현된다.

정의 1) 가시성 그래프 $G=\langle N, Ar, Ap\rangle$는 방향성 그래프

여기서 N은 개체(entity)에 해당하는 노드의 유한집합

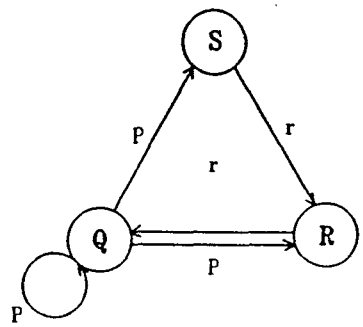
Ar은 노드$\langle n_i, n_j\rangle$라는 순서화된 쌍의 유한집합이고, $n_i, n_j \in N$이며 n_i가 n_j에게 접근을 요청하는(requisition) 관계이다.

Ap는 노드$\langle n_i, n_j\rangle$라는 순서화된 쌍의 유한집합이고, $n_i, n_j \in N$이며 n_i가 n_j에게 접근을 허용하는(provision) 관계이다.

예를 들면 <그림 2-1>은 정의 1)을 적용한 가시성 그래프이고 여기에서 노드는 프로그램 단위를 개체로 하며, p는 노드간의 Ap 관계를 r은 노드간의 Ar 관계를 나타내고 있다.

Gannon[7]은 가시성 관계를 논리적으로 표현하고 이들을 이용하여 Ada의 패키지가 Ada 프

로그램의 내부에서 사용되는 위치에 따른 프로그램의 복잡도 측정과 분석 모델로서 제시하였다.



N = {Q, R, S}
Ar = {⟨S, R⟩, ⟨R, Q⟩}
Ap = {⟨Q, R⟩, ⟨Q, S⟩, ⟨Q, Q⟩}

그림 2-1 Wolf의 가시성 그래프

⟨그림 2-2⟩는 논리적으로 표현된 가시성 그래프이지 Ada Source Program으로 부터 생성되는 것은 아니다.

Ada 언어는 특성상 많은 프로그램 단위로 시스템이 구성되므로 이들 프로그램 단위간의 가시성 관계를 찾아낸다는 것은 매우 어려운 작업이므로 Ada Source Program으로부터 자동화한 도구에 의해 가시성 관계를 생성하는 것이 필요하다.

Muller[9]는 Rigi 모델을 제안하였고 이는 전체 시스템의 관점에서 어느 한 모듈에서 발생한 변경에 의하여 다른 모듈에 미치는 영향을 분석

```
package P is
    procedure X is separate;
end P;
```

```
separate (P)
procedure X is
begin
---;
end X;
```

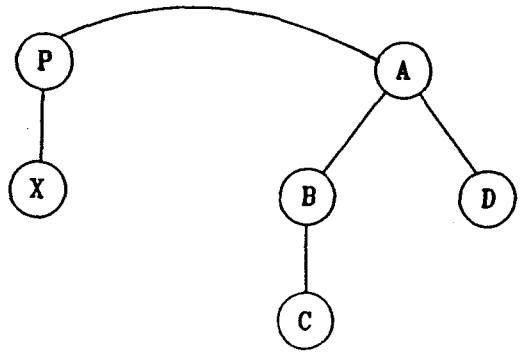
```
with P;
procedure A is
    procedure B is separate;
    procedure D is seperate;
begin
---;
P.X;
end A;
```

```
seperate (A)
procedure B is
procedure C is separate;
begin
---;
P.X;
end B;
```

```
separate (A.B)
procedure C is
begin
---;
end C;
```

```
separate (A)
procedure D is
begin
P.X;
end D;
```

가) 예제 프로그램-1



나) 가시성 그래프

그림 2-2 예제프로그램-1에 대한 가시성 그래프

할 수 있도록 하였다. 〈그림 2-3〉의 경우에 a, b를 명세부분이라 하고 a_1, a_2, a_3, b_1을 각각 a, b에 해당하는 몸체부분이라 하면 전통적인 컴파일 규칙에서는 a의 개체를 변경하게 되면, 변경에 의해 a를 사용하는 모든 접속관계와 관련된 구현부분들을 재컴파일해야 한다. 이와 같이 전통적인 컴파일 규칙에서는 한 모듈이 변경되면 그 모듈과 관련된 모든 모듈을 재컴파일해야 한다. 그러나 이런 재컴파일중에서 재컴파일하지 않아도 될 모듈을 재컴파일하는 경우도 있을 수 있다.

Compilation Dependency Graph CDG=〈V, A〉에서 $v \in V$는 정의모듈이고 $(v, w) \in A$는 w가 v에서 정의된 개체를 사용한다는 것을 뜻한다. $CDG_i = \langle V_i, A_i \rangle$, $1 \le i \le r$, $1 \le r \le |V|$이고

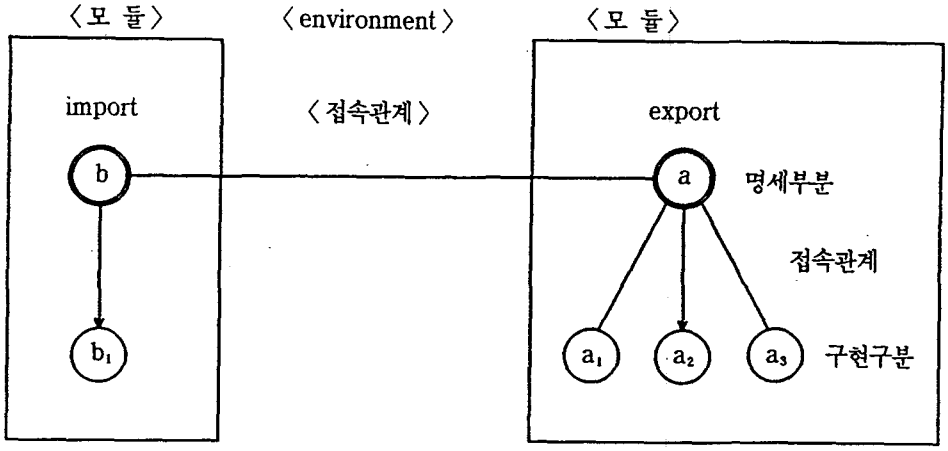


그림 2-3 a모듈을 이용하여 작성된 b모듈의 가시성

r은 CDG의 Strongly Connected Component 이다. 〈그림 2-4〉는 정의 모듈과 모듈간의 CDG 로부터 컴파일 순서를 판단하기 위하여 처리한 과정을 보여주고 있다.

CDG의 Reduced Graph를 ACDG (acyclic compilation dependency graph)라고 하며 ACDG=〈V', A'〉에서 V'은 Strongly Connected Component의 집합이며 A'={(v, w) | (v, w)∈A, $v \in V_i$, $w \in V_j$, $i \neq j$} 이다.

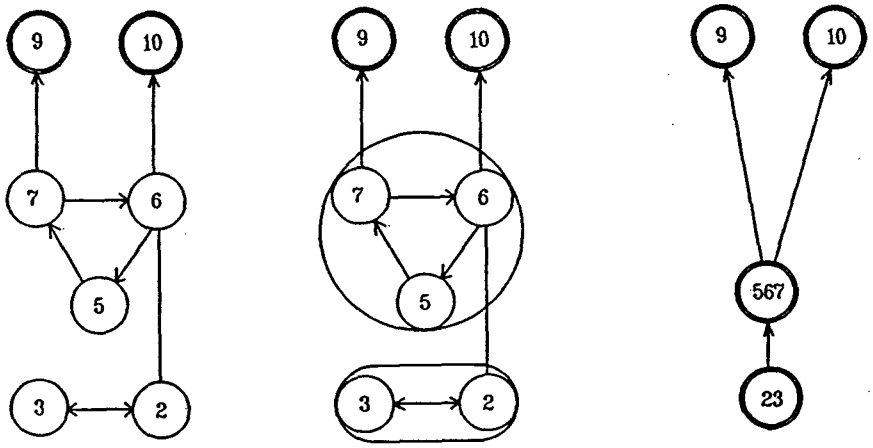
CDG_i의 SCC (strongly connected component)의 V_i집합은 단일블럭(monolithic block) 처럼 컴파일되어야한다. (v, w)∈A'은 v가 w보다 먼저 컴파일되어야 하는 것을 표시한다. CDG=〈V, A〉에서 임의의 노드를 Root로 지정하면 RCDG (root) = (V', A') 이다.

V'={v | v ∈ V and there is a path from root to v}

A'={(v, w) | (v, w)∈A, and v, w∈V'} 이다.

RCDG (root) = (V', A')의 Reduced Graph는 Acyclic Rooted Compilation Dependency Graph ARCDG (root) = (V", A") 이다. V' (root)은 Root가 변했다면 전통적인 컴파일 규칙에서 재컴파일이 필요한 정의모듈을 명시한다. RCDG (root)를 이용하여 위상순서(topology order)을 만들게되면 바로 그것이 지정한 Root의 변경으로부터 재컴파일을 하여야 할 순서를 의미한다.

전체 시스템의 기본 접속관계에 미치는 변경의 영향을 결정하기 위해서(예를 들어 영향받지 않은 단위로부터 영향받은 컴파일 단위를 분리하기 위해서) 모듈에 포함된 내용 뿐만 아니라 변경의



〈CDG〉 〈Strongly connected component〉 〈ACDG〉

그림 2-4 CDG의 변환과정

특성을 분석할 필요가 있다. 기본 접속관계의 변경은 접속관계에 있는 사용모듈의 Import와 Export List를 통해서 변경을 전파한다. 정의모듈 d의 영향받은 Objcet는 Object Set의 Inside(d)와 Outside(d)에 저장한다. 정의모듈 d의 Object Set Inside는 d의 내부에서 영향을 받은 Object를 표시한다. 이것은 Tichy[13]의 "Smart Recompilation" 알고리즘의 Chang Set과 Reference Set과 연관되는데 D의 구현 Variant에 대한 영향을 결정하기 위해서 사용된다. 정의모듈 d의 Object Set의 Outside는 d의 환경에 영향을 줄 수 있는, 예를 들어 정의모듈 d의 사용모듈에 변경을 줄 수 있는 Objcet을 표시한다. 각 정의모듈간의 변경 영향을 분석하는 변경전파 알고리즘은 Object Set을 계산하고, 공집합 여부를 검사하여 다음과 같이 분석을 한다.

○ Inside=0;

모듈은 영향를 받지 않는다(접속관계나 구현이 영향을 받지 않는다)

○ Inside≠0;

접속관계는 영향을 받으므로 재컴파일되어야 한다.

○ Outside=0;

접속관계에서 사용모듈은 영향을 받지 않는다.

○ Outside≠0;

접속관계에서 사용모듈은 영향을 받는다. 영향받은 Object들은 Import나 Export List을 통해서 전파된다.

Rigi 모델에서는 전체 시스템의 관점에서 어느 한 모듈에서 발생한 변경에 의하여 다른 모듈에 미치는 영향의 내역을 분석 결정하고 모듈의 변경에 의해서 발생하는 재컴파일을 감소시키는 일반적인 모델로 제시되었다. 변경에 의해 발생하는 영향을 모듈 내부와 외부로 연결된 환경에 미치는 영향을 분리하여 Ada를 포함하여 일반적인 프로그래밍 환경을 대상으로 접속관계의 분석을 하였다.

제 3 장 Ada프로그램의 가시성 그래프 생성모델 설계 및 구현

제 1 절 가시성 그래프 생성모델의 설계 배경

앞장에서 기술하였던 기존의 연구결과를 살펴보면, 가시성의 개념을 연구의 목적에 알맞도록 모델화하고 있음을 알 수 있다. 즉 Wolf[15]는 개체의 접근 요청과 접근 제공의 관계로, Gannon[7]은 내포구조와 분리(separate) 그리고 'With'관계로, Muller[9]는 컴파일 순서의 관계로 설명하고 있다. 그러나 하나의 관계만을 정의하고 이를 해석하는 과정에서 의미를 정확히 부여한다면 이러한 관계들을 규명할 수 있다. 예를 들면 Muller의 컴파일 순서는 Gannon의 접근 요청과 제공의 단계에서 접근 제공의 구성단위가 먼저 컴파일 된 다음에 접근 요청의 구성단위가 컴파일되는 순서로 해석할 수 있다.

정의 1) 내포구조에 의한 가시성 관계

프로그램 단위 n_i가 n_j를 내포하고 있을 경우에 n_j는 n_i에 대하여 내포(Nesting)구조에 의한 가시성 관계를 가지고 있다고 하고, n_i R_N n_j로 표기한다. 이 때에 n_j가 n_i로부터 분리되어 있는 구성단위도 포함한다.

정의 2) 'With'문에 의한 가시성 관계

프로그램 단위 n_i가 'With'문을 사용하여 다른 프로그램 단위 n_j를 참조하였을 경우 n_i와 n_j는 'With'문에 의한 가시성 관계를 가지고 있다고 하고 n_i R_W n_j로 표기한다.

위에서 정의한 가시성 관계를 기존의 연구에서 제시된 가시성 관계와 비교하여 그 의미를 부여한다면 n_i R_N n_j에서 프로그램 단위 n_i에 포함되어 있는 개체들은 n_j의 개체가 접근을 요청하게 되면 그 접근요청이 허용되어야 함을 의미하고 n_i는 n_j보다 먼저 컴파일 되어야 함을 의미한다. n_i R_W n_j에서 n_j에 포함되어 있는 모든 개체들은 n_i에서 참조될 수 있다는 의미이며 이는 n_i가 n_j의 모든 개체들을 접근요청 할 수 있고 n_j는 접근요청을 허용할 수 있다는 의미이다. 또한 n_j는 n_i보다 먼저 컴파일 되어야 함을 의미한다.

제 2 절 가시성 그래프 생성모델 설계

본 논문에서 설계하고자 하는 가시성 그래프 생성모델은 <그림 3-1>과 같다.

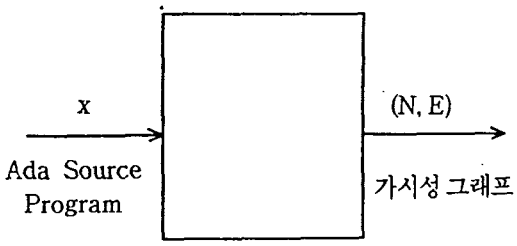


그림 3-1 가시성 그래프 생성모델

즉, 가시성 그래프 생성모델 T는 Ada Source Program x를 입력으로 받아들여 노드(N)와 연결선(E)으로 표현되는 가시성 그래프를 생성한다. 가시성 그래프에 대한 세부사항을 정의하면 다음과 같다.

정의 3) Ada 프로그램의 가시성 그래프 G(x)는 노드(N)와 연결선(E)으로 표현되는 단순 방향성 그래프(simple directed graph)이다.

$G(x) = (N, E)$

여기서,

X : Ada Sourece Program

N : Ada 프로그램 단위들의 집합

$E : E_N \cup E_W$

여기서, $E_N = \{\langle n_i, n_j\rangle \mid (n_i, n_j \in N \text{ and } (i \neq j) \text{ and } (n_i\ R_N\ n_j)\}$

$E_W = \{\langle n_i, n_j\rangle \mid (n_i, n_j \in N \text{ and } (i \neq j) \text{ and } (n_i\ R_W\ n_j)\}$

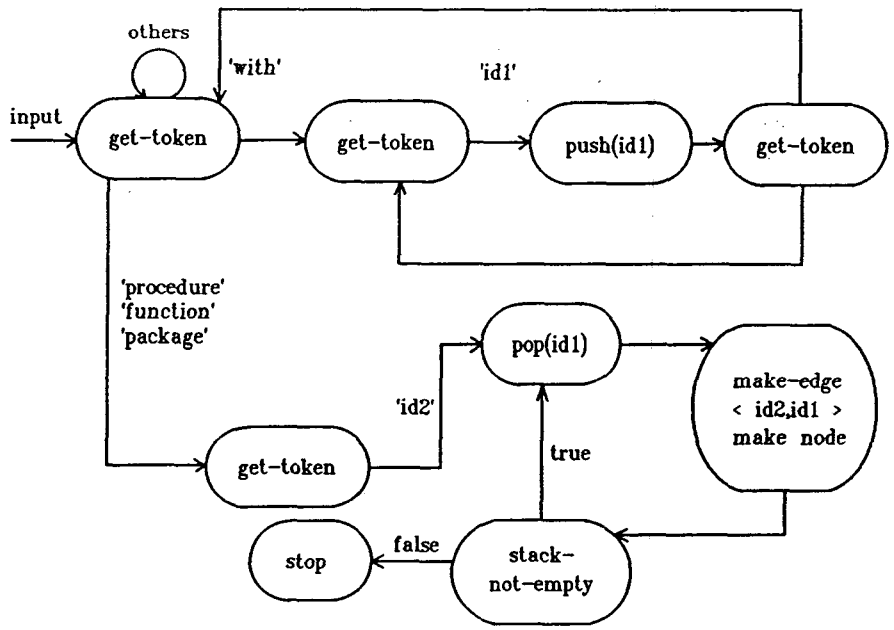
정의 3)에서 본 바와 같이 Ada 프로그램의 가시성 그래프에는 내포구조에 의한 가시성 관계를 나타내는 연결선(E_N)과 'With'문에 의한 가시성

관계를 타나내는 연결선(E_W)이 존재하므로 본 모델에서는 각각의 가시성 관계를 분석하는 알고리즘을 구분하여 설명한다.

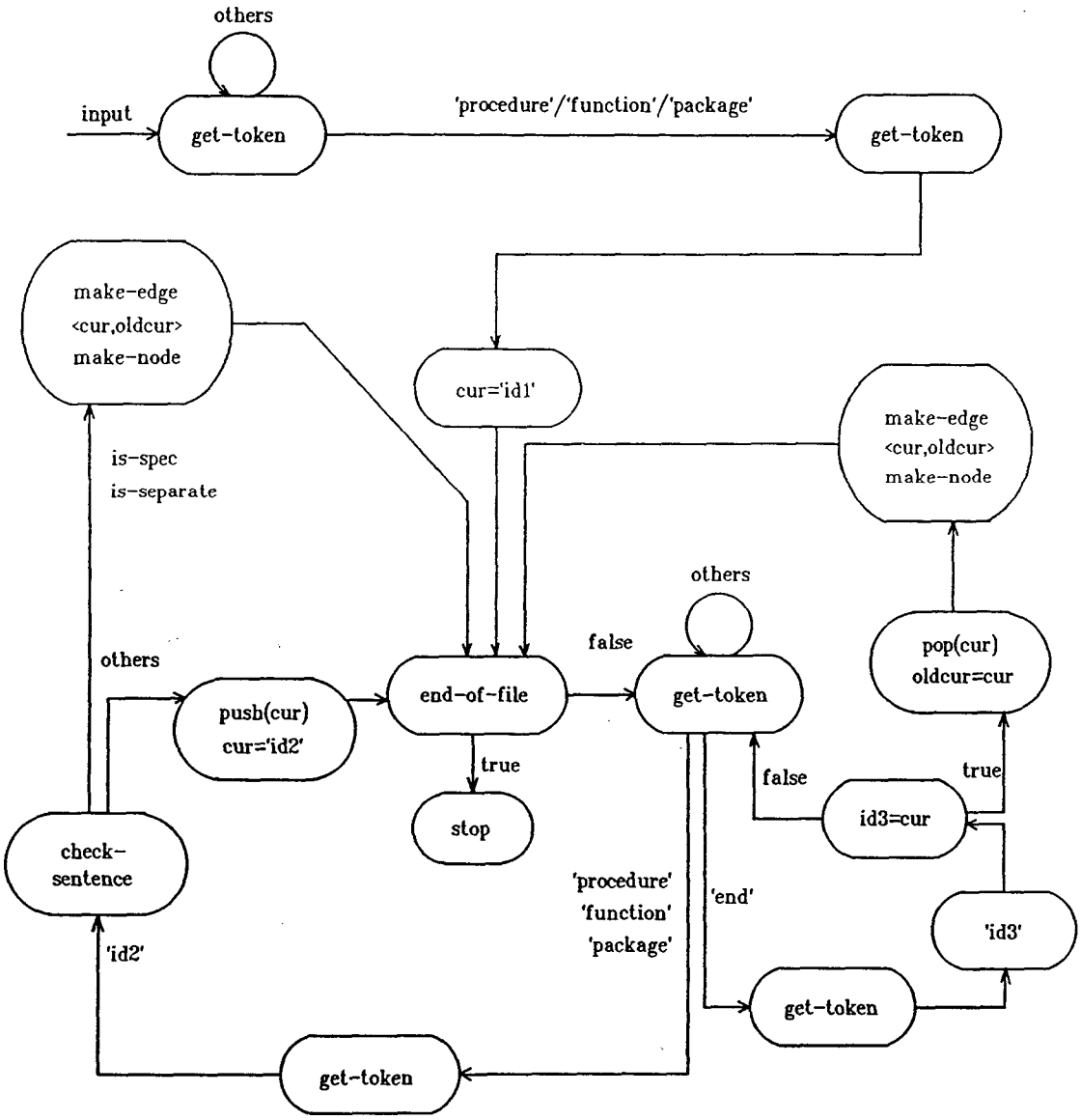
〈그림 3-2〉는 가시성 그래프를 생성하기 위한 알고리즘을 나타낸 것이다.

〈그림 3-2〉 (가)의 Algorithm 1은 'With'문에 의한 가시성 관계를 분석하는 알고리즘으로써 전체 가시성 그래프중에서 'With'문에 의한 부분적인 가시성 그래프, 즉 E_W와 E_W에 관계되는 N을 생성한다. (나)의 Algorithm 2는 내포구조에 의한 부분적인 가시성 그래프, 즉 E_N과 E_N에 관계되는 노드 N을 생성한다.

위와 같이 'With'문에 의한 가시성 그래프를 생성하는 과정과 내포구조에 의한 가시성 그래프를 생성하는 과정을 분리하는 이유는 설계자 및 유지 보수 프로그래머의 필요에 따라 가시성 그래프의 범위가 다를 수 있으므로 요구에 적합한 가시성 그래프를 생성하기 위하여 분리한다. 예를 들면 한 프로그램 단위내에서 내포된 프로그램 단위들간의 가시성이 요구될 경우는 Algorithm 2에 의하여 가시성 그래프를 생성하고, 특정 프로그램 단위가 'with'문을 사용하여 참조한 다른 프로그램 단위들을 파악하기 위해서는 Alogorithm 1에 의하여 가시성 그래프를 생성하는 것이 바람직하다.



가) Algorithm 1



나) Algorithm 2

그림 3-2 가시성 그래프 생성모델 구성도

제 3 절 가시성 그래프 생성모델 구현

가시성 그래프 생성모델을 구현하기 위하여 프로그래밍 언어는 Ada를 사용하였으며, 컴파일러는 IntegraAda 4.0.1(AETEC, Inc, 1988)을 사용하였고, 적용 기종은 IBM PC XT/AT 호환 기종으로 하였다. 개발방법은 Booch의 대상 중심 개발방법을 적용하여 구현하였다.

가) 대상의 식별 및 각 대상의 작업 식별

가시성 그래프 생성모델을 대상 중심 개발방법으로 구현하기 위하여 모델을 대상으로 분리하고, 각 대상의 작업을 식별하면 다음과 같다.

(1) Input-file : 시스템 라이브러리에 존재하는 프로그램 단위의 Text-file에 대한 작업들을 갖는다.
- Open-file : 입력화일을 개방한다.
- Close-file : 입력화일을 닫는다.
- End-of-infile : 입력화일의 current-pointer가 화일의 끝에 위치하고 있으면 "true", 그렇지 않으면 "false"를 반환한다.
- Get-token : 입력화일에서 문자를 읽어들여 토큰을 추출한다.

(2) Visi-file : 완성된 가시성 그래프를 저장하고 출력하는 작업을 갖는다.
- Make-form : 출력 양식을 생성한다.
- Make-edge : 프로그램 단위들간의 가시성 관계를 출력한다.
- Make-node : 프로그램 단위의 명칭을 출력한다.

(3) Visi-stack : alghorithm 화일에서 전달된 토큰을 저장하고 필요시 제공한다.
- Push : Visi-stack에 한개의 토큰을 저장한다.
- Pop : Visi-stack의 top에 존재하는 토큰을 제공한다.
- Set-token : Visi-stack을 초기화 한다.
- Is-empty : Visi-stack이 비었으면 "true", 그렇지 않으면 "false"를 반환한다.

(4) Node-que : Algorithm에서 찾아낸 프로그램 단위의 명칭을 일시적으로 저장하며 필요시 출력하는 작업을 갖는다.
- Node-queue : 프로그램 단위의 명칭을 저장하는 일을 수행한다.
- Get-node : 저장된 프로그램 단위의 명칭을 꺼내는 일을 수행한다.
- Queue-empty : Queue가 비었으면 "true", 그렇지 않으면 "false"를 반환한다.
- Set-queue : Queue를 초기화시킨다.

(5) Algorithm : 프로그램 단위간의 가시성 관계를 추출한다.
- Generate-visi-graph 1 : 'With'문 사용으로 설정된 프로그램 단위들간의 가시성 관계를 추출한다.
- Generate-visi-graph 2 : 내포된 프로그램 단위간의 가시성 관계를 추출한다.

나). 가시성 그래프 생성모델의 대상간의 가시성 설정

각 대상간의 가시성 관계를 Booch 도형을 이용하여 표기하면 <그림 3-3>과 같다.

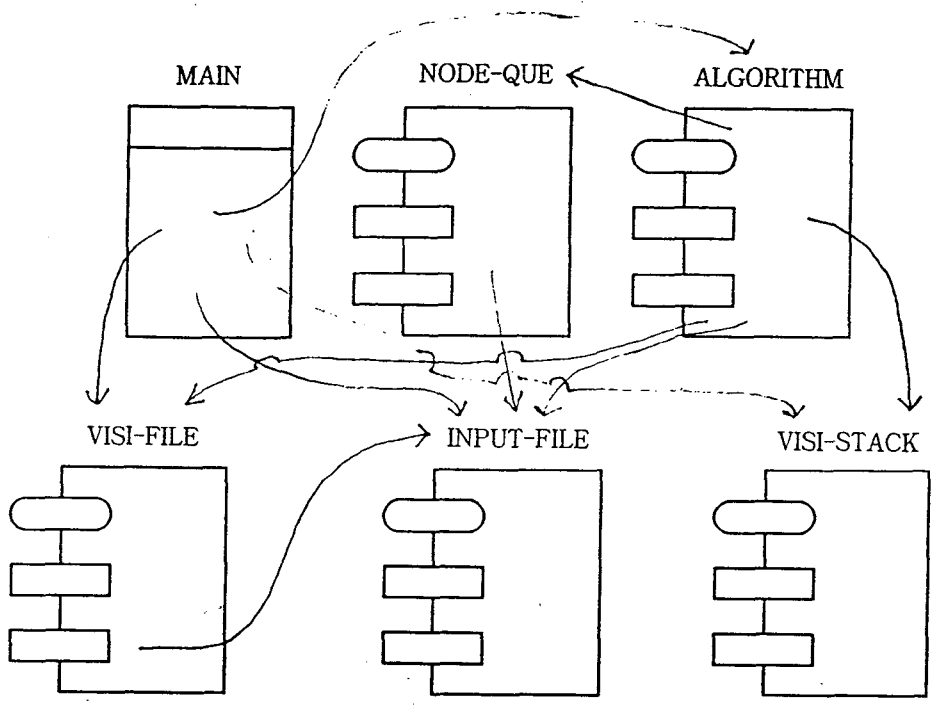


그림 3-3 Booch도형으로 표기된 대상간의 가시성 관계

제4장 검 증

본 모델의 실행 결과를 검증하기 위하여 Rajlich[10]가 제시한 <그림 4-1>의 Ada 프로그램을 입력으로 사용하였다.

```
With DATABASE_PACK, ITEM_PACK;
use DATABASE_PACK, ITEM_PACK;
procedure MAIN is
  begin
end MAIN;

with ITEM_PACK; use ITEM_PACK;
package DATABASE_PACK is
  procedure INITIALIZE_DATABASE;
  procedure SEARCH;
end DATABASE_PACK;
with SEQUENTIAL_IO;
package body DATABASE_PACK is
  procedure  INITIALIZE_DATABASE is
```

```
      begin
      end INITIALIZE_DATABASE;
    procedure SEARCH is
      begin
      end SEARCH;
    end DATABASE_PACK;

package ITEM_PACK is
    function EQUAL(T: in ITEM) return BOOLEAN;
    function LESS(T: in ITEM) return BOOLEAN;
    function DONE return BOOLEAN;
    procedure PRINT_INFO(T: in ITEM);
    procedure READ_INPUT;
    procedure PRINT_MESSAGE;
  end ITEM_PACK;
with TEXT_IO; use TEXT_IO;
package body ITEM_PACK is
    function EQUAL(T: in ITEM) return BOOLEAN is
      begin
      end EQUAL;
    function LESS(T: in ITEM) return BOOLEAN is
      begin
      end LESS;
    function DONE return BOOLEAN is
      begin
      end DONE;
    procedure PRINT_INFO(T: in ITEM) is
      begin
      end PRINT_INFO;
    procedure READ_INPUT is
      begin
      end READ_INPUT;
    procedure PRINT_MESSAGE is
      begin
      end PRINT_MESSAGE;
end ITEM_PACK;
```

그림 4-1 예제 프로그램-2

본 모델이 생성한 가시성 그래프 G(main) = (N, E)는 다음과 같다.

$N = \{MAIN_{pro},\ DATABASE\text{-}PACK_{pkg},\ ITEM\text{-}PACK_{pkg},\ TEXT\text{-}IO_{pkg},\ SEQUENTIAL\text{-}IO_{pkg},\ INITIALIZE\text{-}DATABASE_{pro},\ SEARCH_{pro},\ EQUAL_{fun},\ LESS_{fun},\ DONE_{fun},\ PRINT\text{-}INFO_{pro},\ READ\text{-}INPUT_{pro},\ PRINT\text{-}MESSAGE_{pro},\}$

$E_V = \{\langle MAIN_{pro}, DATABASE\text{-}PACK_{pkg}\rangle,\ \langle MAIN_{pro}, ITEM\text{-}PACK_{pkg}\rangle,\ \langle DATABASE\text{-}PACK_{pkg},\ ITEM\text{-}PACK_{pkg}\rangle,\ \langle DATABASE\text{-}PACK_{pkg}.\ SEQUENTIAL\text{-}IO_{pkg}\rangle,\ \langle ITEM\text{-}PACK_{pkg},\ TEXT\text{-}IO_{pkg}\rangle\}$

$E_V = \{\langle DATABASE\text{-}PACK_{pkg},\ INITIALIZE\text{-}DATABASE_{pro}\rangle,\ \langle DATABASE\text{-}PACK_{pkg},\ SEARCH_{pro}\rangle\ \langle ITEM\text{-}PACK_{pkg},\ EQUAL_{fun}\rangle,\ \langle ITEM\text{-}PACK_{pkg},\ LESS_{fun}\rangle,\ \langle ITEM\text{-}PACK_{pkg},\ DONE_{fun}\rangle,$

〈ITEM-PACK$_{pkg}$, PRINT-INFO$_{pro}$〉,
〈ITEM-PACK$_{pkg}$, READ-INPUT$_{pro}$〉,
〈ITEM-PACK$_{pkg}$,
PRINT-MESSAGE$_{pro}$〉}

입력된 프로그램의 가시성 관계와 생성된 가시성 그래프를 비교해 보면 Procedure MAIN과 MAIN에서 'With'문을 사용하여 참조한 프로그램 단위인 Package DATABASE-PACK, ITEM-PACK과의 가시성 관계는 $E_{\#}$로 표현되어 있고 Package DATABASE-PACK과 내포된 프로그램 단위인 Procedure INITIALIZE-DATABASE, SEARCH와의 가시성 관계는 $E_{\#}$으로 표현되어 있다. 그리고 그 외의 'With'문을 사용한 프로그램 단위들간의 가시성 관계와, 내포된 프로그램 단위간의 가시성 관계를 모두 표현하고 있으므로 생성된 가시성 그래프는 주어진 입력 프로그램의 가시성 관계를 정확히 나타내고 있음을 보여주고 있다.

제5장 결 론

Ada 언어는 특성상 Programming-in-the-Small에서 필요한 도구들을 능가하는 도구들을 필요로 하게 된다. 소프트웨어 시스템을 구성하는 구성 단위들을 표현하고, 분석하며, 구성 단위를 구성하고 관리하는 광범위한 지원을 제공하는 도구들이 필요하다.

프로그램 단위간의 가시성 관계의 규명은 Ada를 이용한 프로그램 구조에 관한 전반적인 정보를 제공하기 때문에 프로그램 그 자체의 코드보다는 프로그램의 구조를 이해하는데 필수적인 자료이다. 가시성 관계의 규명은 시스템을 설계하고, 시스템을 점진적으로 구축할 때 유용하게 사용되며 특히 유지보수 단계에서 Source Program으로부터 가시성 관계를 규명한다는 것은 대단히 중요한 것이다.

본 논문에서 제시한 가시성 그래프 생성모델은 Ada의 Source Program을 입력으로 하고, 그 출력으로 Ada의 Source Program을 구성하고 있는 구성단위간의 가시성 관계자료를 생성하는 모델이다. 본 모델의 사용분야는 소프트웨어 유지보수 단계에서 구성단위간의 파급효과와 슬라이스(slice)부분을 분석할 수 있는 도구 생성 및 소프트웨어 구성단위간의 변경효과분석을 위한 접근 요청과 접근 제공을 분석하는 데에도 사용될 수 있고, 개발과정에서 프로그램 단위들간의 컴파일 순서를 파악하는 데에도 사용될 것으로 기대된다.

Ada 언어로 작성된 소프트웨어의 대부분이 방대한 규모임을 감안할 때, 본 논문의 모델에서 대상으로 다룬 부분은 타스크를 제외한 부분적인 것이므로 이를 더욱 개선하고 보완하여, 보다 일반적인 소프트웨어 분석도구가 되도록 하는 연구가 필요하다.

참고문헌

〔1〕 이광건, 윤창섭, Ada언어의 재사용을 이용한 소프트웨어의 구현에 관한 연구, 석사학위논문, 국방대학원, 1989.

〔2〕 J. G. P. Barnes, *Programming in Ada*, 2nd Ed., Addision-Wesley Publishing Company, 1984.

〔3〕 Grady Booch, *Software Engineering with Ada*, 2nd Ed., The Benjamin/Cummings Publishing Company, Inc., 1987.

〔4〕 Norman H. Cohen, *Ada as a second language*, McGraw-Hill Book Company, 1986.

〔5〕 Frank DeRemer and Hans H. Krow, "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Trans. on Software Eng.*, Vol. SE-2, No. 2, Jun. 1976, pp. 80~86.

〔6〕 Narain Gehani, *Ada : An Advanced Introduction*, 2nd Ed., Prentice-Hall International, Inc., 1989.

〔7〕 J. D. Gannon, E. E. Katz, and V. R. Basill, "Metrics for Ada Package : An Initial Study," Department of Computer Science, University of Maryland, Mar. 1986.

〔8〕 Geoff Gilpin, *Ada A Guided Tour and Tutorial*, Prentice Hall Press, 1986.

〔9〕 Hausi A. Muller, Rigi - A Model for Software System Construction, Integration, and Evolution based on Module Interface Specification, Ph. D. thesis, Department of Computer Science, Rice University, Huston Texas, 1986.

〔10〕 Vaclav Rajlich, "Paradigms Design and Implementation in Ada" *Cammunication of the ACM*, Vol. 28, No. 7, Jul. 1985, pp. 718~727.

〔11〕 Jan Skansholm, *Ada from the begining*, Addison-Wesley Publishing Company, 1988.

〔12〕 Daniel L. Stock, *JANUS/ADA Compiler User Manual*, RR Software Inc., Version 4.4, Jan. 1988.

〔13〕 Walter F. Tichy, "Smart Recompilation," *ACM Trans. on Programming Languages and System*, Vol. 8, No. 3, Jul. 1986, pp. 273~291.

〔14〕 Peter Wegner, *Programming with Ada An Introduction by Means of Graduated Examples*, Prentice-Hall, Inc., England Cliffs, New Jersey, 1980.

[15] A. L. Wolf, L. A. Clarke, and J. C. Wileden, "A Model of Visibility Control," IEEE Trans. on Software Eng. Vol. 14, No. 4, Apr. 1988, pp. 512~520.

[16] _____, "The AdaPIC Tool Set : Supporting Interface Control and Analysis Throughout The Software development Process," IEEE Trans. on Software Eng. Vol. 15, No. 3, Mar. 1989, pp. 250~263.

[17] _____, "Ada-Based Support for Programming-in-the-large," IEEE Software, Mar. 1985, pp. 58~68.