韓國軍事運營分析學會誌
第17卷, 第2號, 1991. 12. 31.

Ada 프로그램에서 패키지 활용의 국부화 모델에 관한 연구
-A Study on Localization Model of Package Usage in Ada Program-

金先鎬, 尹昌燮*

Abstract

Software system is a hierarchical structure with collection of program units. Software system can import external packages globally or locally depending on the usage within a system. If the imported package is used globally, the software system can be influenced globally by any change of package and programmer's debugging time for the program maintenance will be greater. To solve these problems, it is desirable to use the imported package locally right on the usage point within the system.

The model presented in this paper analyzed entity usage of package in structure of program, identified the usage level to obtain localization and provided information for restructure of the program to localize package usage. To obtain localization, it identified declared entities inside the imported package and analyzed the specification and body part of program unit to identify entities referenced from the imported package.

The proposed model can be used to improve the maintainability of software system and contributed to reduction of programmer's debugging time in program maintenance.

* 國防大學院

I. 서론

대상 중심 개발(object-oriented development)은 시스템을 대상(object)의 개념으로 분할하고, 분할된 대상간의 관계를 규명하여 소프트웨어를 개발하는 방법으로서 Ada에서는 대상을 패키지로 구현한다[3]

패키지 명세부에서 선언된 개체(entity)들은 서로 관련된 개체들의 모임으로서 프로그램 단위간의 가시성에 의하여 다른 컴파일 단위(compilation unit)가 import하여 사용할 수 있다. 소프트웨어 시스템은 프로그램 단위의 집합으로서 계층구조에 의하여 구성되어 있다. 하나의 프로그램 단위가 패키지 대상을 import하여 사용할 때에 global 또는 local로 사용할 수 있다. 전체 시스템에 대하여 global로 사용한다면 패키지의 변화에 대하여 모든 프로그램 단위에 영향을 줄 수 있으며, 유지보수시 프로그래머의 디버깅(debugging)시간을 많이 소모할 수 있으며, 이러한 문제를 해결하기 위하여 전체적인 시스템의 구조를 정확하게 기록할 수 있는 어떠한 수단을 제공하는 것이 중요하다.

with문을 사용한 패키지의 가시성은 패키지가 프로그램 단위의 시작부분에서 with문장을 사용했을 때이며, 프로그램 구조 특성을 나타내는 데 사용된다. 프로그래머는 패키지가 각 프로그램 단위에서 어떻게 사용되었는가를 보다 잘 이해하기 위하여 프로그램 단위에 있어서 참조된 패키지의 개체를 찾아내는 것이 우선이다. 이것은 패키지의 개체가 활용되는 영역을 최소화함으로써, 즉 패키지의 개체활용을 국부화(localization)함으로써 쉽게 이해할 수 있다. 패키지의 개체활용을 국부화하기 위해서는 그 개체를 활용하는 프로그램 단위의 구조와 구조 내에서 패키지의 개체활용을 분석하고 분석할 결과를 이용하여 구조의 재구성이 이루어져야 한다.

본 논문의 목적은 Ada 프로그래밍 환경에서 패키지의 가시성 부분이 with문의 사용으로 Ada 프로그램 단위에서 사용되었을 경우, 프로그램 단위의 구조 내에서 패키지의 개체활용을 분석하여, 각 계층별로 패키지로 부터 참조한 개체가 존재하는 프로그램 단위를 식별하며 국부화를 위한 구조의 재구성에 관한 자료를 제공하는 모델를 제시하는 데에 있다. 본 논문의 모델에서는 Ada의 표준화된 입/출력 패키지와 실시간 처리를 다루는 타스크는 제외했다.

본 논문의 제2장에서는 Ada 프로그램의 일반적 특성과 가시성 관계를 고찰하고, 제3장에서는 프로그램 구조 분석을 위한 모델을 설계하고 이에 대한 알고리즘 기술 및 원형을 구현한다. 제4장에서는 검증의 결과를 제시하고자 한다.

2. Ada의 프로그램 단위 및 기존의 가시성 관계 고찰

2.1 프로그램 단위

Ada 소프트웨어 시스템은 하나 이상의 프로그램 단위로 구성되고, 타스크를 제외한 각 단위는 분리 컴파일이 가능하다. 프로그램 단위

는 서브프로그램(subprogram), 패키지(package), 타스크(task), 일반화(generic)가 있다[6].

다음은 서브프로그램, 패키지, 일반화에 대해 구체적으로 고찰하고자 한다. 타스크는 실시간 문제 해결의 Ada 프로그램 단위이므로 본고의 범위에서 제외한다.

2.1.1 패키지(package)

패키지는 명세부와 구현부로 구성된다. 패키지의 명세부는 패키지의 외부에서 사용되어야 할 개체를 선언하는 부분이며, 패키지는 이 개체들을 export하게 된다. 이 부분에서는 자료형식 선언(type), 대상(object), 서브프로그램의 명세부가 포함된다. 패키지 구현부는 사용자로 부터 은폐되는 부분으로, 패키지 명세부에서 선언된 각 프로그램 단위에 대한 알고리즘의 구현을 다루고 있다. 패키지 구현부는 명세부에서 선언된 패키지를 사용하는 사용자에게 필요한 정보만을 알려 주고 자세하고 불필요한 정보를 은폐하는 부분이며, 명세부에서 정의된 서브프로그램 구현, 패키지의 국지변수의 선언, 가시성있는 서브프로그램 간에 공유되는 보조 서브프로그램(auxiliary subprogram) 등으로 구성된다. 이 두 부분은 각각 별개의 파일에 저장되며, 분리 컴파일이 가능하다.

2.1.2. 서브프로그램(subprogram)

사용자의 관점에서, 프로그램이란 일련의 순차적 또는 병렬적 행위를 통하여 다른 대상과 상호 작용을 하는 대상들의 집단이라 보는데 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 패키지와 같이 명세부와 구현부로 구분되며, 프로시쥬어 및 함수의 두 형태를 가진다. 이것은 기존 언어에서 사용하는 프로시쥬어와 함수의 구조적인 면에서 유사하다. 프로시쥬어는 문장들의 집합으로 세개의 매개변수 모드(mode)를 가진다. 함수는 프로시쥬어와 같으나 단지 계산된 값을 반환하는 형태로 한개의 모드만 존재한다.

2.1.3 일반화(Generic)

Ada 언어에서 프로그램 단위를 인수화(parameterization)하고 유사한 프로그램 단위를 재사용할 수 있도록 하는 메카니즘이 일반화 프로그램 단위이며, 일반화 패키지와 일반화 서브프로그램으로 구분된다. 단순히 자료의 형식이 다르고 알고리즘이 같은 경우 각각의 프로그램을 작성하지 않고 일반화 프로그램 단위로 작성하여 대상의 부류를 표현한다. 일반화는 하나의 틀(template)로써 직접 호출할 수가 없으며 단지 실례화(instantiation)를 하여 사용 가능하다. 일반화 서브프로그램과 일반화 패키지는 라이브러리(library)에 존재하고 실례화(instantiation)에 의하여 프로그램 단위의 일부가 된다.

2.2. 기존의 가시성 관계 고찰

전통적으로 소프트웨어 시스템을 구성하는 개체(entity)의 가시성은 협의의 의미로 선언문(declaration), 범위(scope), 바인딩(bin-

ding) 등의 용어를 사용하여 정의되어 왔다. 그러나 Ada, Clu, Euclid, Gypsy, Mesa, MODULA-2, 그리고 Smalltalk와 같이 언어가 다양화되면서 뛰어난 가시성의 관계는 광의의 의미로 내포가 논의되어 왔으며, 이것만으로 부족했던 면들을 충족하기 위하여 가시성 제어를 위한 선택적이며 보완적인 방법들이 제공되었다[7].

Wolf[7]는 개체간의 가시성 관계를 접근요청(requisition of access)과 접근제공(provision of access)으로 구분되는 개념으로 설명하고 있다. 개체에 접근하는 것은 선언이나 명령문에서 개체를 참조하거나 사용하기 위한 권리이다. 개체가 다른 개체를 참조하려는 권리를 요구할 때 접근요청이 발생한다. 접근제공은 외부 또는 내부의 한 개체가 어떤 다른 개체들의 집합에 대해 잠재적으로 참조할 수 있는 권한을 부여하는 것이다. Wolf의 가시성 제어 모델은 개체의 가시성 관계를 논리적으로 표현하고 이를 이용하여 Module Interconnection Language의 개발에 중점을 두었다.

Gannon[4]은 가시성 관계를 논리적으로 표현, 이들을 이용하여 Ada의 패키지가 Ada 프로그램 내부에서 사용되는 위치에 따른 프로그램의 복잡도 측정과 분석 모델을 제시하였다.

Ada 언어는 특성상 많은 프로그램 단위로 시스템이 구성되므로, with문을 사용한 패키지가 프로그램 단위에서 어떻게 사용되었는가를 이해하기 위하여 프로그램 단위에서 참조된 개체를 식별하는 것이 필요하다.

Müller[5]는 Rigi 모델을 제안하였고 이는 전체 시스템의 관점에서 어느 한 모듈에서 발생한 변경에 의하여 다른 모듈에 미치는 영향을 분석할 수 있도록 하였다. 이렇게 함으로서 대형 시스템에서 한 모듈의 변경에 의해서 발생하는 재컴파일 노력을 감소시키는 일반적인 모델로 제시하였으며, 변경에 의해 발생하는 영향을 모듈 내부에서 그리고 그 모듈과 연결된 환경에 미치는 것으로 분리하여 일반적인 프로그래밍 환경을 대상으로 접속관계를 분석하였다. Rigi 모델의 적용은 프로그램 구조 분석을 필요로 하지만 이에 관한 연구는 제시되지 않았다.

3. 모델 설계 및 구현

3. 1. 개요

Ada 컴파일 단위에서 이전에 컴파일된 라이브러리 단위를 이용하고자 한다면 with문을 사용하여 프로그래머들이 필요한 만큼의 구성 단위들을 선별, 가시성을 유지하여 이용할 수 있다. 라이브러리 단위내의 개체들이 전체 시스템의 일부분인 하나의 프로그램 단위에서만 참조될 수도 있기 때문에 라이브러리 단위와 프로그램 구성 단위간의 분석을 통하여 참조된 개체가 존재하는 프로그램 단위의 계층 구조를 식별하여야 하며, 이것을 이용하여 국부화의 관점에서 프로그램의 구조를 개선할 수 있다.

본 논문에서는 프로그램의 구조 개선을 위한 기본개념을 설명하고, 프로그램 단위를 분석하여 패키지에서 참조한 개체가 존재하는 프로그램 단위를 각 계층(level)별로 구하며, 구해진

프로그램 단위로 패키지의 활용을 국부화하는 모델을 제시한다.

3.1.1. 국부화 과정

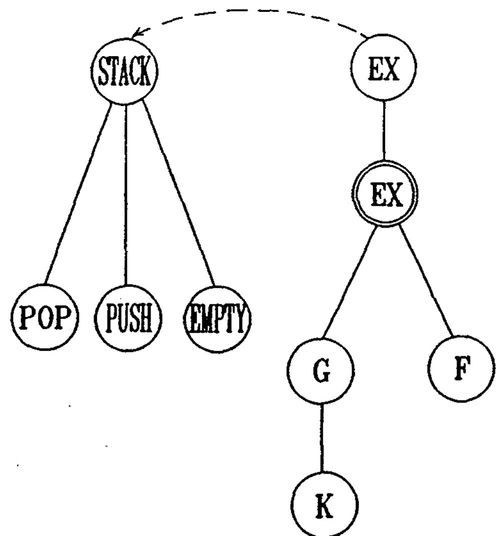
국부화는 물리적인 모듈에 있는 논리적으로 관련된 계산자원을 수집함으로써 보다 강한 응집도를 갖게 되는 것이며, 주어진 모듈이 충분히 독립적이고 보다 약하게 결합된 구조를 만드는 것을 의미한다[2].

<그림 3-1>과 <그림 3-2>의 예제, 프로그램을 이용하여 국부화가 잘 된 경우와 그렇지 못한 경우의 예를 들어 보면, <그림 3-1>은 국부화가 안된 경우로서, 패키지 stack의 개체는 최하위 계층에 존재하는 프로그램 단위 k에서만 참조되었으나, 최상위 계층에 존재하는 프로그램 단위 ex에서 import되었기 때문에 ex의 전체 영역에서 global로 사용되었다. 이것은 패키지의 변화에 대하여 모든 프로그램 단위에 영향을 줄 수 있다. <그림 3-2>는 국부화가 잘 이루어진 경우로서, 패키지 stack의 개체(entity) push는 프로그램 단위 g, f, k

```
package stack is
  procedure push  (h : integer);
  procedure pop    (h : out integer);
  function empty return boolean;
end stack;

with stack;use stack;
package ex is
  procedure g;
  function f(x : integer) return boolean;
end ex;
package body ex is
procedure g is
  char : integer;
 procedure k is
  begin
   for i in 1..20 loop
   push(i);put(i);end loop;
  end k;
 begin
  get(char);
  If char<0 then
   put(error);
   else
    k;end if;
  end g;
function f(x : integer)return boolean is
 begin
  if x>20 then
   return false;
   else return true;
  end if;
 end f;
end ex;
```

(가) 예제 프로그램



(나) 가시성 그래프

<그림 3-1> 비 국부화된 예제 프로그램과 가시성 그래프

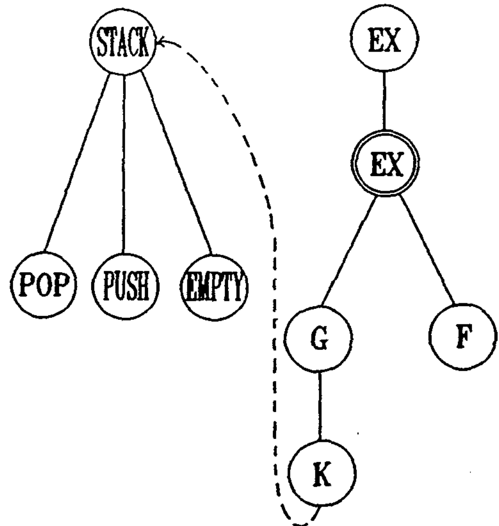
중 최하위 계층에 존재하는 k에서만 참조되었기 때문에 import된 패키지 stack은 프로그램 단위 ex에서 global하게 사용되기 보다는 프로그램 단위 k에서만 local하게 사용되었으며, 이것은 패키지가 프로그램 단위에서 어떻게 사용되었는가를 보다 쉽게 이해할 수 있다.

그러므로, 패키지의 개체가 참조된 프로그램 단위에서 국부화를 위한 구조의 재구성이 이루어 져야 하며, 재구성시 패키지의 개체가 참조된 계층의 프로그램 단위로 국부화[4]한다. 만일, 패키지의 개체를 참조한 프로그램 단위들 간의 내포(nested)관계가 존재하면 이들 중 최상위 계층(level)의 프로그램 단위로 국부화한다.

국부화하는 과정은 첫째, 비교해야 하는 두 개의 원시 프로그램 즉, 패키지에서 선언된 개체와 패키지를 참조하는 프로그램 단위가 외부에서 참조한 개체들을 모두 구한다. 둘째, 두 개의 원시 프로그램으로 부터 구해진 개체들을 서로 비교하여 패키지에서 참조한 개체가 존재하는 프로그램 단위를 각 계층별로 구한다. 셋째, 구해진 각 계층의 프로그램 단위로 패키지

```
package body ex is
 procedure g is
  char : integer;
procedure k is separate;
 begin
  get(char);
  if char<0 then
   put(error);
   else
   k;
  end if;
 end g;
function f(x : integer) return boolean is
 begin
  if x>20 then
   return false;
  else return true;
 end if;
 end f;
end ex;
with stack;use stack;
separate(ex, g)
procedue k is
 begin
  for i in 1..20 loop
   push(i);
   put(i);
  end loop;
end k;
```

(가) 예제 프로그램



(나) 가시성 그래프

<그림 3-2> 국부화된 예제 프로그램과 가시성 그래프

의 활용을 국부화한다.

국부화함으로써 시스템을 개발할 때 소프트웨어의 유지보수성을 증진시킬 수 있고, 유지보수시에는 프로그래머의 디버깅(debugging) 시간을 절약할 수 있으나, 모듈 사이즈 및 컴파일 시간이 증가되는 단점이 있을 수도 있다. 이와 같은 문제는 향후 전체적인 시스템의 관점에서 해결되어야 할 것이다.

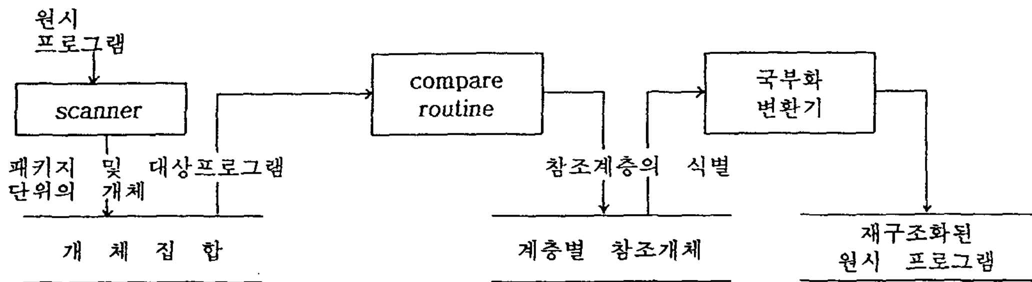
3.2. 모델 설계

3.2.1. 기본 모델

설계하고자 하는 모델은 〈그림 3-3〉에서 보는 바와 같으며, scanner에서는 두 개의 원시 프로그램 즉, 패키지와 패키지를 참조한 프로그램 단위를 분석하여 개체를 비교하기 위한 정보를 추출한다.

개체집합에서는 scanner에 의해 추출된 패키지 및 대상 프로그램 단위의 개체들을 저장한다.

compare routine에서는 두 개의 원시 프로그



〈그림 3-3〉 국부화 기본 모델

```
procedure  package_spec_set (declared  token : out  set_of_entities;
                             referenced_token : in  out  list_of_set)  is
  token : entity;
  begin
     loop
      get_next_token;
    if  token=package  then
        basic_declaration (token) :
        get_declared_token;
        for  token  in  declared_token  loop
         get_referenced_token;
        end  loop;
    elsif  token=subprogram  then
        add  token  to  entity_set  with  attributes
    elsif  token=private  then
          exit;
        end  if;
      end  loop;
  end  package_spec_set;
```

〈표 3-1〉 : 〈알고리즘1〉 명세부 분석 알고리즘

그램으로 부터 구해진 각각의 개체들을 서로 비교함으로써 패키지로 부터 참조된 개체가 존재하는 프로그램 단위를 구하며, 또한 프로그램 단위에 대한 각 계층을 구한다.

국부화 변환기에서는 compare routine에서 추출된 정보를 이용하여 패키지의 활용을 국부화한다.

3.2.2. 알고리즘

가. 명세부 분석

〈표 3-1〉은 명세부 분석에 관한 알고리즘을 나타낸 것이다. 여기에서 basic_declaration은 자료형식, 변수등에 대한 선언된 개체집합을 구하며, package의 경우에는 (token=package에 해당) 선언문내에 나머지 식별자들을 참조된 개체집합에 추가된다. 서브프로그램의 경우에는 (token=subprogram에 해당) 중복사용 문제 해결을 위하여 속성(매개변수의 수와 자료형)과 함께 기체집합에 추가한다.

나. 구현부 분석

〈표 3-2〉는 구현부 분석에 관한 알고리즘을 나타낸 것이다. 여기에서 package_spec_set은 구현부에 국지적인 선언부가 존재할 경우 〈표 3-1〉의 알고리즘을 이용하여 선언부내의 국지적인 개체집합과 그것에 대하여 참조된 개체집합을 구한다. program_unit에서는 각 프로그램 단위에 대한 참조된 개체집합을 구한다. 프로그램 단위에서 선언되지 않은 개체가 사용되었다면 그것은 외부로 부터 참조된 것이며, 이것을 판별하기 위하여 국지적으로 선언된 개체들과 우선 비교가 이루어져야 한다. 이때, 내포된 프로그램 단위(nested program unit)의 경우에는 그 프로그램을 내포하고 있는 하위내포수준(lower nested level)의 프로그램 단위에서 국지적으로 선언된 개체들과도 비교가 이루어져야 한다. 이러한 문제는 stack 구조를 사용하여 해결하였는데 프로그램 단위에서 국지적으로 선언되는 개체들을 모두 개체 stack에 저장하면서 프로그램 단위 내에 사용된 개체들을 검사한다. 만일 stack에 저장되어 있지 않은 개체가 사용되었다면 그 개체는 외부에서 참조해 온 것이므로 참조집합에 그 개체를 추가하며, 프로그램 단위가 끝나는 순간 stack에 저장되었던 국지선언 개체들을 모두 빼낸다.

```
procedure  package_body_set (declared_token : out  set_of_entities;
                             referenced_token : in  out  set_of_entities) is
  begin
    package_spec_set (declared_token,  referenced_token) ;
   for  each  program_unit  loop
        start_point : =top;
      process_specification (referenced_token) ;
      process_declarative_part (referenced  token) ;
      process_executable_part (referenced_token) ;
       top : =start_point;
   end  loop;
 end  package_body_set;
```

〈표 3-2〉 : 〈알고리즘2〉 구현부 분석 알고리즘

여기에서 내포된 프로그램 단위가 나타날 경우는 〈표 3-2〉의 알고리즘에서 program_unit를 재귀적(recursively)으로 호출하여 내포된 프로그램 단위에 적용한다. 그러므로 내포되 프로그램 단위에 사용된 참조개체들이 집합에 추가된다. process_executable_part는 프로그램 단위를 실행하는 부분으로 수행부분 내부에서 사용된 개체들 중 식별자를 차례로 검사하여 외부에서 참조된 것을 개체집합에 추가한다.

다. 패키지의 개체를 참조한 프로그램 단위 식별

〈표 3-3〉은 패키지의 개체를 참조한 프로그램 단위의 식별에 관한 알고리즘을 나타낸 것이다. 여기에서 process_program_unit는 패키지의 개체를 참조한 프로그램 단위가 외부에서 참조한 개체들의 집합을 구하며, 이 집합들과 패키지 가시성 부분에서 선언된 모든 개체 집합간의 교집합을 구하면, 외부에서 제공한 개체들의 집합을 구할 수 있다.

〈그림 3-1〉의 예제 프로그램을 이용하여, 지금까지 기술된 알고리즘을 적용하면 프로그램 단위가 참조한 패키지의 개체를 구할 수 있다. 먼저, 〈그림 3-1〉의 프로그램 구조를 살펴보면 패키지를 참조한 프로그램 단위인 ex는 명세부와 구현부로 구성되어 있으며, 구현부 내에는 function f, procedure g가 존재하며, procedure g의 내부에는 procedue k가 내포되어 있다. 또한, 패키지 stack은 ex의 명세부에서 import되었다.

프로그램 구조 내에서 패키지의 개체참조 여부를 살펴보면, 패키지 stack내부에 선언된 모든 개체들은 〈표 3-1〉의 알고리즘을 적용하여 다음과 같이 구할 수 있다.

- $STACK_{declared_token}$; {push, pop, empty, h}

프로그램 단위인 명세부와 구현부는 〈표 3-2〉의 알고리즘을 적용하여 외부에서 참조된 개체집합을 아래와 같이 구할 수 있다.

- $PACKAGE_SPEC_{referenced_token}$: {integer, boolean}
- $PACKAGE_BODY_{referenced_token}$: {}
  - $F_{referenced_token}$: {integer, boolean}
  - $G_{referenced_token}$: {get, integer, put}
  - $K_{referenced_token}$: {push, put}

패키지 및 패키지를 참조한 프로그램 단위에서 구해진 각각의 개체집합들은 〈표 3-3〉의 알고리즘을 적용하여 패키지의 개체를 참조한 프로그램 단위를 식별할 수 있다.

```
procedure intersection(referenced_set : in out set_of_entities) is
   import : set_of_entities;
   declared_in_spc : set_of_entities;
  begin
   for with unit loop
    process_program_unit(import);
    referenced_set : =declared_in_spc ∩ import
   end loop;
 end intersection;
```

〈표 3-3〉 : 〈알고리즘3〉 패키지의 개체를 참조한 프로그램 단위 식별 처리

-프로그램 단위가 참조한 패키지의 개체

· $K_{referenced_set}$: {push}

3.2.3. 계층(level) 식별

프로그램 구조의 계층식별은 stack을 이용하여 프로그램 단위 명칭과 관련되는 계층의 번호를 부여하며, 번호가 높을수록 최하위 계층을 나타낸다. 만일, 동일계층이 나타날 때에는 각 계층에서 프로그램 단위간의 관계를 명시하기 위하여 Ada의 가시성 그래프[9]를 이용하여 표시한다. 〈그림 3-1〉의 예제 프로그램을 이용하여 프로그램 단위간의 계층 식별은 다음과 같이 할 수 있다.

-프로그램 단위별 계층 식별

· $EX_{package_spec}$ ==⇒ level : 0 (spec)

· $EX_{package_body}$ ==⇒ level : 0

· F ==⇒ level : 1

· G ==⇒ level : 1

· K ==⇒ level : 2

-프로그램 단위간의 가시성 그래프

· 〈EX, F〉

· 〈EX, G〉

· 〈G, K〉

3.3. 구현

본 논문의 모델을 구현하기 위하여 Ada 언어를 사용하였으며, 국부화를 위한 변환기는 본 구현에서 제외한다. 컴파일러는 Integer Ada 4.0.1 (AETEC INC, 1988)을 사용하였고, 대상 기종은 IBM PC XT/AT 호환 기종으로 하였다. 그리고 시스템의 개발방법은 Booch의 대상중심 개발방법을 적용한다[2].

3.3.1. 대상의 식별 및 작업

각 대상의 작업을 식별하면 〈그림 3-4〉와 같다.

(1) Main : 주 프로그램으로서 프로그램이 실행되는 부분이다.

(2) Scanner : 원시 프로그램을 분석하여 모든 개체를 추출하며, get_next_token 작업을 수행한다.

(3) Compare_routine : 〈표 3-3〉의 알고리즘과 관련된 대상으로써, 각 개체들에 대한 compare작업을 하며, intersetion copy, insert, dispose, move_if_present 작업을 수행한다.

(4) File_mgt : 입력 및 출력에 대한 파일관리를 하며, end_of_infile, open_file, close_file 작업을 수행한다.

(5) Stack : 패키지의 개체들에 대한 push, pop, in_stack의 작업을 수행한다.

(6) Level_unit : 프로그램의 계층 식별 및 가시성 그래프를 위한 level, visibility_graph 작업을 수행한다.

(7) Reserved_word : Scanner에 의하여 추출된 식별자의 상태를 검사하며, is_keyword의 작업을 수행한다.

(8) Subpro_name : 〈표 3-1〉과 〈표 3-2〉의 알고리즘과 관련된 대상으로써, 프로그램 단위의 개체를 식별하며, push_id_list, declared_in_package, process_formal_part, process_declarative_part, process_program_unit의 작업을 수행한다.

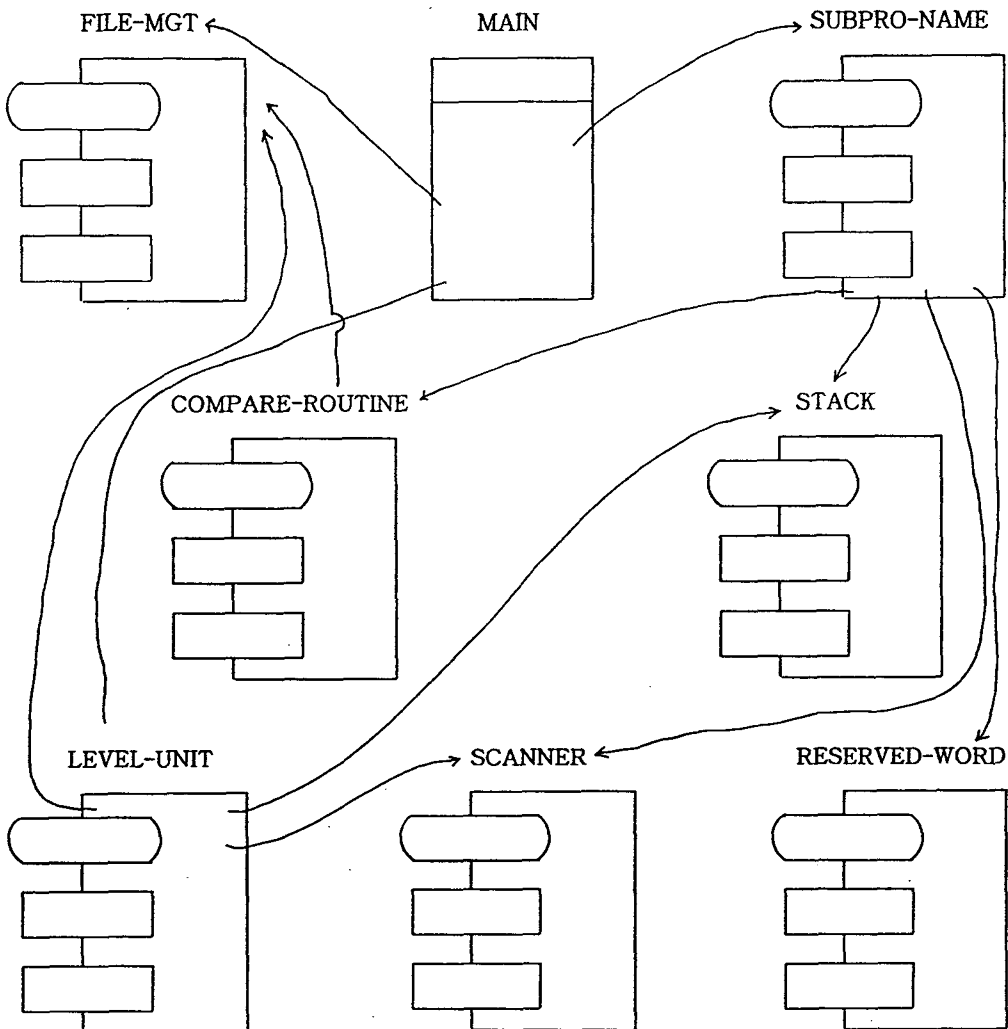
3.3.2. 가시성 설정

각 대상간의 가시성은 Booch도형을 이용하여 표기하면 〈그림 3-4〉와 같다.

4. 검 증

본 논문의 검증은 하나의 예제 프로그램을 이용하여 본 모델에서 제시한 방법에 의하여 패키지의 개체를 참조한 프로그램 단위를 계층별로 나타냈으며, 이것을 이용하여 Ada 패키지가 하나의 프로그램 단위에서 global 및 local로 사용되는 위치에 따른 프로그램 구조를 비교하여 패키지의 척도를 측정하여 검증하였다.

본 논문의 예제 프로그램은 〔2〕에서 제시된



〈그림 3-4〉 Booch도형을 이용한 모델의 가시성 설정

것이고, 본 모델에서 나타난 결과는 다음과 같다.

-프로그램 단위별 계층 표시

· Body ==⇒ LEVEL : 0

· rotate ==⇒ LEVEL : 1

· scale ==⇒ LEVEL : 1

· trans ==⇒ LEVEL : 1

-프로그램 단위가 참조한 개체집합

· rotate : {cos, sin}

-프로그램 단위간의 가시성 그래프

· 〈Body, rotate〉

· 〈Body, scale〉

· 〈Body, trans〉

이것에 대한 패키지의 척도는 〔4〕을 적용하여 측정하였으며, 이것의 UP(P)는 패키지의 개체를 참조한 프로그램 단위로 국부화가 잘 되어 있느냐를 측정하는 것으로, *UP(P)*의 비율이 높을수록 패키지가 local하게 사용된 경우이며, 낮을수록 패키지가 global하게 사용되었음을 나타낸다.

따라서, 패키지 척도의 결과로써 시스템의 가시성 비율을 비교하면, 패키지가 local하게 사용된 프로그램은 global하게 사용된 프로그램보다 가시성 비율이 낮다는 것을 알 수 있다.

5. 결론

본 논문에서는 패키지의 가시성 부분이 with문의 사용으로 Ada 프로그램 단위에서 사용되었을 경우, 패키지로 부터 참조된 개체를 식별하기 위하여 프로그램 구조 내에서 패키지의 개체활용을 분석하며 참조한 개체가 존재하는 프로그램 단위의 계층 구조를 식별하여 국부화를 위한 구조의 재구성에 관한 자료를 제공하는 모델을 제시하였다. 패키지에서는 패키지 내에서 선언된 모든 개체들을 식별하였고, 패키지를 참조한 프로그램 단위에서는 외부에서 참조한 개체를 식별하기 위하여 프로그램 단위인 명세부와 구현부를 분석했다. 명세부에서는 선언된 개체와 이것에 대한 참조된 개체들을 식별하였고, 구현부에서는 프로그램 구조가 내포구조로 되어 있으므로 프로그램 단위별로 외부에서 참조된 개체를 식별하였다.

본 논문에서는 패키지의 개체활용을 국부화함으로써 시스템을 개발할때 소프트웨어의 유지보수성을 증진시킬 수 있고, 유지보수시에는 프로그래머의 디버깅(debugging)시간을 절약할 수 있다. 반면에, 모듈 사이즈 및 컴파일 시간은 증가될 수 있기 때문에 이와 같은 문제를 해결하기 위한 앞으로의 연구가 요구되며, 또한 Ada의 표준화된 입/출력 패키지 및 task를 제외하였기 때문에 완전한 프로그램 구조분석을 위하여 Ada의 표준화된 입/출력 패키지 및 task를 포함하여 분석하는 것이 필요하다.

참 고 문 헌

〔1〕 Baker, T., "A One-pass Algorithm for Overload Resolution in Ada," *ACM Trans. Program Lang. and Sys.*, Vol. 4, No. 4, Oct. 1982, pp. 601-614

〔2〕 Booch, G., *Software Engineering with Ada*, 2nd Ed., The Benjamin/Cummings Publishing Company, Inc., 1986.

〔3〕 Booch, G., "Object-Oriented Development," *IEEE Trans. on Software Eng.*, Vol. SE-12, No. 2, Feb. 1986.

〔4〕 Gannon, J., E. Katz, and V. Basili, *Metrics for Ada Package : An Initial Study*, Department of computer Science, University of Maryland, Mar. 1986.

〔5〕 Müller, H., Rigi-A Model for Softwae System Construction, Integration and Evolution based on Module Interface Specifications, Ph. D. thesis, Department of Computer Science, Rice University, Huston Texas, 1986.

〔6〕 United States Department of Defense, *Reference Manual for the Ada Programming*, Jul. 1980.

〔7〕 Wolf, A., Clarke, L., and J. Wileden, "A Moel of Visibility Control," *IEEE Trans. on Software Eng.*, Vol. 14, No. 4, Apr. 1988, pp. 512-520.

〔8〕 정중영, 윤창섭, Ada 프로그램의 Visibility Graph 생성 모델에 관한 연구, 석사학위논문, 국방대학원, 1990.