객체지향 프로그램 표현을 위한 객체지향 프로그램 종속성 그래프

류 희 열[†]·박 중 양[††]·박 재 흥[†††]

요 약

소프트웨어 공학 기법과 툴들은 제어 흐름 그래프, 프로그램 종속성 그래프, 시스템 종속성 그래프와 같은 프로그램의 그래픽한 표현에 의존한다. 본 논문은 기존의 객체지향 프로그램의 표현방법의 복잡하고 중복된 표현의 문제점을 개선하여 새로운 객체 지향 프로그램 종속성 그래프를 제안한다. 객체지향 프로그램 종속성 그래프는 클래스 종속성 그래프와 클래스 계층구조 그래프, 그리고 프로시듀어 종속성 그래프로 구성된다. 제안된 객체지향 프로그램 종속성 그래프는 기존의 방법에 비해 그 표현이 간결할 뿐 아니라, 쉽게 확장이 가능하여 프로그램이 점진적으로 개발되는 경우에도 사용할 수 있다. 뿐만 아니라 프로그램에 대한 동적 정보를 제공할수 있도록 확장될 수 있다.

A Object-oriented Program Dependency Graph for Object-oriented Program Representation

Hee-Yeol Ryu[†] · Joong-Yang Park[††] · Jae-Heung Park[†††]

ABSTRACT

Many software engineering tools and techniques rely on graphic representations of software, such as control flow graphs, program dependence graphs, or system dependence graphs. Existing graphic representations for object-oriented programs are complicated, reduplicated. We thus propose a new graphic representation for object-oriented programs, Object-oriented Program Dependency Graph (OPDG). An OPDG consists of class dependence graph, class hierarchy graph and procedure dependence graph. Other features of OPDG are (1) the representation is compact; (2) the representation is easy to extend for the incremental development of a program; and (3) the representation can be extended to provide dynamic information.

1. 서 론

많은 소프트웨어 공학 툴과 기법들은 프로그램의 분석 정보를 요구한다. 예를들면, 테스팅(testing), 디버깅(debugging), 역공학(reverse engineering), 재공학(reengineering), 재사용(reuse), 슬라이싱(slicing)과 같은 분야에서 사용되는 툴들은 자료흐름(data flow)과 제어흐름(control flow)과 같은 프로그램 분석 정보를 요구한다. 이러한 분석 정보를 표현하는 방법으로 프로그램 언어와 자료구조에 독립적이며, 단지 알고리즘에 기반이된 표현으로 제어 종속성 서브그래프(CDS : Control Dependence Subgraph)와 자료 종속성 서브그래프(DDS : Data dependence Subgraph)로 구성된 프로그램 종속성 그래프(PDG : Program Dependency

† 준 회 원 : 동의공업대학 사무자동화과 교수
†† 정 회 원 : 경상대학교 통계학과(정보통신연구센터) 교수
††† 정 회 원 : 경상대학교 컴퓨터과학과(정보통신연구센터) 교수
논문접수 : 1998년 2월 19일, 심사완료 : 1998년 8월 6일

Graph)가 알려져있다. PDG 표현은 프로그램의 이해를 쉽게하고, 프로그램 버전들 사이의 차이를 식별하고, 이전에 개발된 프로그램의 일부분을 추출과 병합을 통하여 새로운 프로그램의 생성을 용이하게 한다. 따라서 다른 프로그램 언어로 재설계하고 재개발 할 필요가 없는 장점을 갖는 PDG는 절차적 프로그래밍 언어의 분석 정보 표현에 널리 사용되고 있다[1,2].

단일 프로시듀어를 표현하는 PDG를 확장하여 다중 프로시듀어간의 제어흐름 정보와 자료흐름 정보를 동시에 표현하는 시스템 종속성 그래프(SDG : System Dependence Graph)가 있다[3,4,5]. 그러나, SDG는 클래스(class), 객체(object), 상속(inheritance), 다형성(polymorphism), 동적 바인딩(dynamic binding)과 같은 객체지향 패러다임을 표현할 수 없다[6,7,8,9].

객체지향 프로그램의 표현방법에 대한 연구가 몇몇 있으나 다음과 같은 문제점들이 있다.

Loren Larsen[7,8]의 방법은 기존의 SDG에 클래스, 파생 클래스, 클래스간의 상호작용, 다형성등과 같은 객체지향 패러다임을 표현할 수 있도록 클래스 종속성 그래프(CLDG : CLass Dependence Graph)를 추가하여 객체지향 프로그램을 표현하는 새로운 SDG 이다[7]. 프로시듀어간의 호출문맥을 표현하기 위해서 실매개변수와 형식 매개변수의 중복된표현과 여러종류의 자료흐름 종속성 간선들의 사용은 그래프 표현을 매우 복잡하게 하며, 또한 기저 클래스(base class)의 파생 클래스(deriverd class)가 많아지면 클래스간의 계층구조 표현이 대단히 복잡해지며, main() 프로그램부분과 클래스 부분의 식별 및 분류가 어렵고, 객체지향 프로그램의 특징인 점진적인 개발과 갱신을 적절히 반영하지 못하는 단점이 있다.

Brian A. Malloy[6]의 방법은 기존의 PDG에 CLDG를 추가하고 매개변수 표현방법을 개선하여 새로운 객체지향 표현방법이다. 동일한 제어 종속성을 갖는 문장들의 그룹을 나타내는 범위정점을 추가하여 제어 종속성은 적절히 표현하고 있으나 다소 복잡해지는 단점과 자료 종속성의 표현이 생략되어 프로그램의 분석 정보를 정확하게 표현하지 못하는 단점이 있다. 특히, 객체 생성시 멤버변수들간의 자료 종속성을 표현하지 못한다.

Marry Jean Harrold[9]의 방법은 기존의 PDG에 CLDG, 클래스 계층구조 그래프(CHG : Class Hierarchy Graph), 클래스 호출 그래프(CCG : Class Call Graph), 클래스 제어 흐름 그래프(CCFG : Class Control Flow Graph), 클래스간 호출 그래프(ICCG : Inter-Class Call Graph), 클래스간 제어흐름 그래프(ICCFG : InterClass Control Flow Graph), 클래스간 종속성 그래프(InterClass Dependence Graph)등을 추가하는 객체지향 프로그램의 표현방법으로 프로그램의 분석정보를 정확하게 표현할 수 있는 장점이 있다. 그러나, PDG를 기반으로 하기 때문에 매개변수 정점과 자료흐름 간선의 표현으로 많은 정점들과 간선들의 중복 표현 때문에 복잡하고, 또한 많은 서브 그래프의 중복표현은 메모리의 낭비와 그래프의 복잡도를 높이는 주요 원인이 된다.

본 논문에서 제안하는 객체지향 프로그램 종속성 그래프(OPDG : Object-oriented Dependency Graph)) 방법은 위의 문제점들을 다음과 같이 개선하여 객체지향 프로그램의 분석정보를 정확히 표현한다.

① 프로시듀어간의 호출문맥 표현을 위해 프로시듀어 호출간선과 클래스 호출간선의 의미를 확장하여 제어의 흐름과 자료의 흐름을 나타내어 매개변수 정점의 수를 최소화한다.

② 점진적인 개발과 부분적인 표현을 할 수 있도록 각 클래스의 표현이 main() 프로그램과 독립되어 CLDG 표현하며, 호출간선과 객체생성 간선으로 main() 프로그램의 프로시듀어 종속성 그래프(PRDG : Procedure Dependence Graph)와 연결된다.

③ 각 부분 그래프를 최소화하여 중복 표현을 줄이기 위해 CLDG, CHG, PRDG로 이용하여 전체 OPDG를 구성한다.

④ 객체의 멤버변수 리스트를 이용하여 객체 사이의 자료흐름 정보를 표현한다.

제안하는 OPDG는 객체지향 프로그램의 분석정보를 표현하기 위해 최소한의 정점들을 사용하여 프로시듀어간의 호출 문맥을 정확하게 표현하고, 점진적인 개발 및 유지보수를 위해 부분적인 표현을 적절히 반영할 수 있으며, 또한 각 부분 그래프의 수를 줄여서 중복 표현을 최소화하고, 객체의 멤버변수들 간의 자료 종속성을 표현할 수 있는 장점이있다.

본 논문의 구성은 다음과 같다. 2 장에서는 객체지향 프로그램 표현을 위한 표기법과 각 정점과 간선들을 정의하고, 다형성과 동적 바인딩의 표현 방법의 정의와 OPDG를 정의하고, 응용 프로그램에 적용하여 OPDG로 표현하고, 3 장에서는 OPDG의 크기를 분석

한다. 끝으로 4 장에서는 결론 및 향후 연구과제를 제시한다.

2. 객체지향 프로그램 종속성 그래프(OPDG)

2.1 매개변수 전달 표현

매개변수 전달을 위해서 형식 매개변수와 실 매개변수를 각각 한 개의 정점으로 표현하여 호출지역에서 메소드 또는 프로시듀어를 호출할 때, 메소드 또는 프로시듀어에서 호출지역으로 제어가 되돌아올 때 양 방향으로 매개변수를 1 : 1 대응하여 교환함으로써 기존의 formal_in, formal_out, actual_in, actual_out 정점을 이용하는 방법보다 정점의 수를 감소시킨다. 또한 그들간의 연결 간선들이 감소된다. 단, 매개변수의 값에 변화가 발생하면 각각을 호출간선의 오른쪽에 표현한다. 아래 (그림 1)의 프로그램을 Susan Horwitz[3]방법과 본 논문에서 제안하는 매개변수 전달에 대한 표현 방법을 각각 (그림 2), (그림 3)에 비교하여 보인다.

1에서 10까지의 합을 1개의 main 프로그램과 3개의 프로시듀어를 이용하여 구하는 간단한 예제 프로그램과 시스템 종속성 그래프이다.

```
Program Main        Procedure A(x, y)    Procedure Add(a, b)  Procedure Increment(z)
 sum := 0;           call  Add(x, y)      a := a + b;           call Add(z, 1)
 i := 1;             call  Increment(y)   return               return
 while i < 11 do    return
   call A(sum, i)
 od
end(sum, i)
```

(그림 1) 예제 프로그램
(Fig. 1) Example program

위의 예제 프로그램에 대한 linkage grammar의 production과 이행적 종속성(transitive dependence)은 다음과같다.

Main→A　　　A→Add Increment
Add→ε　　　Increment→Add

x_in y_in A x_out y_out

(그림 2) 예제 프로그램의 SDG
(Fig. 2) The System Dependence Graph of the example program

(그림 3) 예제 프로그램에 대한 제안하는 그래프 표현
(Fig. 3) The proposed graph representation for the example program

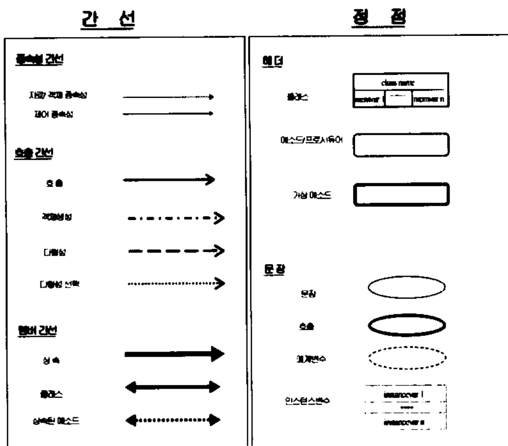
a_in b_in Add a_out b_out

z_in Inc z_out

본 논문에서 제안하는 방법은 x_in→x_out, y_in→y_out, a_in→a_out, b_in→b_out, z_in→z_out의 이행적 종속성은 프로시듀어 호출간선에 암시적으로 표현되어 있고 y_in→x_out, b_in→a_out에 대한 이행적 종속성도 표현하고 있다. 따라서 구조가 단순하게되어 프로그램의 표현이 간결하며, 그래프 표현을 위한 메모리의 낭비를 줄이며, 프로그램의 이해가 용이하도록 하며, 프로그램을 정확하게 표현할 수 있다.

2.2 표기법

제안하는 OPDG에서 사용하는 정점과 간선들에 대한 기호는 종속성 간선, 호출간선, 멤버간선, 헤더, 문장과 매개변수로 분류하여 (그림 4)에서 정의한다.



(그림 4) 각 정점과 간선에 대한 기호법 정의
(Fig. 4) The definition of notations for each edges and vertices

2.3 클래스 종속성 그래프(CLDG)

(1) 기저 클래스 표현

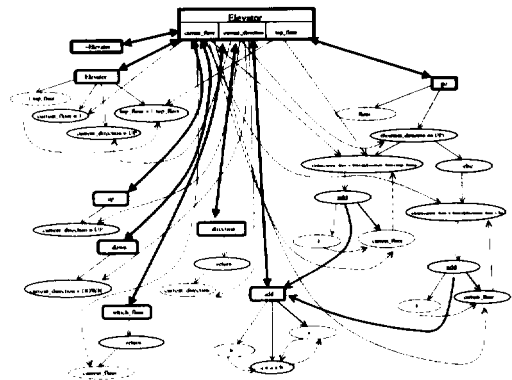
프로그램 분석과 재사용을 쉽게하기 위해서 프로그램의 각 클래스를 각각의 독립된 CLDG로 표현하여 OPDG 정보저장소에 저장함으로써 프로그램의 점진적인 개발과 갱신을 용이하게 한다. CLDG는 클래스의 메소드에 대한 각각의 PRDG로 구성되고, 또한 각 메소드는 클래스 멤버 간선으로 연결된다. 또한 각 메소드의 헤더에 표현되어 있는 접근권한과 자료속성들의 관계를 분석하여 자료 종속성 간선을 표현한다. 클래스 정점과 클래스 멤버 간선은 클래스가 다른 시스템과 결합될 때, 다른 클래스와 결합될 때 클래스에 대한 메소드 정보를 제공한다.

CLDG는 한 번 생성되면 그 클래스가 수정, 삭제등의 변경이 발생하기 전에는 더 이상의 재구성이 발생하지 않는다. 따라서, CLDG는 재사용이 용이하고, 클래스의 추가, 갱신, 삭제가 발생할 때 해당 클래스에 대해서만 CLDG를 구성하여 사용하기 때문에 OPDG를 구성할 때 전체 OPDG를 재구성할 필요가 없으며, 해당 클래스의 CLDG만 추가하여 표현할 수 있다.

(그림 5)는 클래스 Elevator의 C++ 소스코드이고 이 클래스의 CLDG는 (그림 6)과 같다.

```
class Elevator {
    public :
        Elevator(int l_top_floor) {
                                current_floor = 1;
                                current_direction = UP;
                                top_floor = l_top_floor;};
        ~Elevator(){};
        virtual void up(){
                            current_direction = UP;};
        virtual void down(){
                            current_direction = DOWN;};
        int which_floor(){
                            return current_floor;};
        Direction direction(){
                            return current_direction;};
        virtual void go(int floor){
            if(current_direction == UP) {
                while((current_floor != floor) && (current_floor <= top_floor))
                                add(current_floor, 1);}
            else {
                while((current_floor != floor) && (current_floor > 0))
                                add(current_floor, -1);}}
    private :
        add(int &a, const int &b){
                                a = a + b;};
    protected :
            int current_floor;
            Direction current_direction;
            int top_floor;
        };
```

(그림 5) 클래스 Elevator의 C++ 소스 코드
(Fig. 5) The C++ code of class Elevator



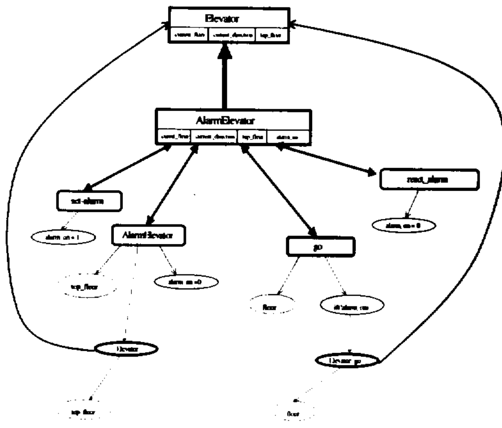
(그림 6) 클래스 종속성 그래프
(Fig. 6) Class Dependence Graph

(2) 파생 클래스 표현

기저 클래스로부터 상속된 모든 메소드들을 재사용하는 의미로 상속간선으로 연결하며, 파생 클래스에서 정의된 각각의 메소드들에 대한 표현을 함께 구성하여 파생 클래스의 CLDG를 생성한다. 상위 클래스 정점과 하위 클래스 정점을 상속간선으로 연결하여 클래스 간의 계층구조를 나타내며, 또한 상속받는 상위 클래스의 메소드와 하위 클래스 정점을 상속된 메소드 간선으로 연결하여 CHG에 나타낸다. 파생 클래스 정점에는 상위 클래스의 자료속성과 파생 클래스 자신의 자료속성과 전역변수를 함께 포함한다. 다음 (그림 7)는 클래스 Elevator로부터 파생된 AlarmElevator의 C++ 소스코드이고, 그 파생클래스의 CLDG는 (그림 8)과 같다.

```
AlarmElevator(int top_floor) : public Elevator {
    public :
        AlarmElevator(int top_floor) : Elevator(top_floor)
        {
         alarm_on = 0;};
         void set_alarm(){
         alarm_on = 1;};
         void reset_alarm(){
                         alarm_on = 0;};
        void go(int floor){
                         if(!alarm_on)
                                  Elevator::go(floor);};
  protected :
         int alarm_on;
         };
```

(그림 7) 클래스 AlarmElevator의 C++ 소스 코드
(Fig. 7) The C++ code of class AlarmElevator



(그림 8) 파생 클래스 AlarmElevator의 CLDG
(Fig. 8) CLDG of derived class AlarmElevator

(3) 클래스의 상호작용 표현

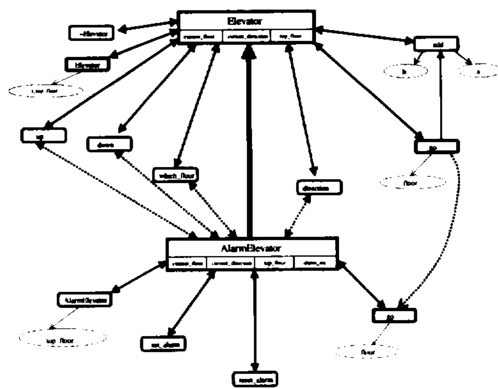
객체지향 소프트웨어에서 클래스는 new 연산자, 선언등을 통해서 클래스의 객체를 생성한다. 예를들면, Elevator elevator_object(10) 또는 elevator_pointer = new Elevator(10)와 같은 문장을 사용해서 Elevator 클래스의 객체를 생성한다. 클래스 C1이 클래스 C2의 인스턴스 클래스라하면 클래스 C1에서 클래스 C2의 생성자에 대한 호출이 있다. 이러한 호출은 인스턴스의 위치인 C1의 생성자 정점에서 C2 클래스 헤더 정점으로 호출간선을 연결하여 표현한다. 또한, C2의 공용영역에 있는 메소드 M2에 대하여 C1에서 메소드 M1내에 호출문장이 있으면 메소드 M1의 호출 정점에서 C2 클래스 헤더 정점으로 호출간선을 연결하여 표현한다. 클래스의 상호작용을 표현하기 위한 이들 간선은 위 (그림 8)에서 클래스 AlarmElevator의 생성자에서 클래스 Elevator 정점으로 호출간선을 연결하고 클래스 AlarmElevator의 메소드 go에서 클래스 Elevator 정점으로 호출간선을 연결하여 표현한다.

(4) 다형성 표현

CLDG는 다형성을 갖는 메소드 호출을 표현해야만 한다. 다형성을 갖는 메소드는 클래스에 대한 간접참조가 포인터를 통해서 만들어지고 참조된 객체의 형식은 여러개의 클래스에 대한 참조를 나타내기 때문에 실행시간에 결정된다. 다형성은 상위 클래스의 가상 메소드와 하위 클래스의 가상 메소드의 이름이 같기 때문에 다형성 호출이 발생하면 이들 가상 메소드들 중에서 동적으로 바인딩되는 한 개의 메소드만 결정한다. 따라서, CLDG는 동적으로 바인딩되는 가상 메소드를 정적인 분석동안에 표현하기위해 다형성을 갖는 상위 클래스의 가상 메소드에서 하위 클래스의 모든 가능한 목적지를 다형성 선택 간선으로 표현한다. 다형성 호출이 발생하면 해당 클래스의 가상 메소드 헤더와 다형성 선택 간선으로 연결된 모든 가능한 목적지들의 집합에서 특정한 호출의 하나만 선택한다. 다형성을 갖는 호출에대한 모든 가능한 목적지를 정적으로 줄이기 위한 알고리즘에 대한 연구들이 있으나 형추론 문제에 대한 정확한 해결은 NP-hard이다[11,12].

다음 (그림 9)는 위의 Elevator, AlarmElevator 클래스간의 계층구조와 하위 클래스가 상속 받는 상위 클래스의 메소드 표현, 가상 메소드들간의 다형성 표현 등을 간선으로 연결하여 표현하는 CHG이다. CHG는

상위 클래스 Elevator의 up(), down(), which_floor(), direction()등의 메소드가 하위 클래스 AlarmElevator로 상속되는 메소드 관계를 상속 메소드 간선으로 연결하여 표현하고, Elevator 클래스의 go() 가상 메소드와 AlarmElevator 클래스의 go() 가상 메소드간에 다형성 선택 간선으로 연결하여 다형성 호출이 발생할 때 이들 중 하나의 메소드만 선택될 수 있는 동적 바인딩을 표현한다.



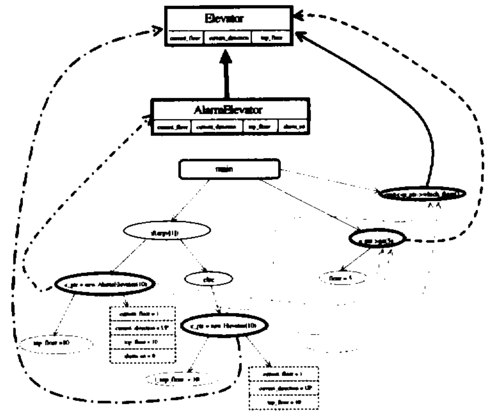
(그림 9) 클래스 계층구조 그래프
(Fig. 9) Class Hierarchy Graph

2.3 프로시듀어 종속성 그래프(PRDG)

PRDG는 클래스의 각 메소드, 독립된 프로시듀어, main()함수등을 표현한다. PRDG는 각 문장정점에 대하여 제어 종속성 간선과 각 문장의 변수들에 대한 자료 종속성 간선으로 표현하는점에서는 단일 프로시듀어의 PDG와 같으나 차이점을 들자면 클래스 객체의 생성을 표현하기 위한 객체생성 호출 간선, 다형성 호출을 표현하기 위한 다형성 호출간선, 클래스 메소드 호출을 표현하기 위한 메소드 호출등의 간선은 해당 클래스의 헤더 정점과 호출 정점간을 연결하여 표현한다. (그림 10)는 main()함수에 대한 C++ 소스코드이고, (그림 11)은 호출 클래스 정점에 대한 호출 간선이 포함되어 있는 PRDG 이다.

```
main(int argc, char **argv)
      {
          Elevator *e_ptr;
          if(argv[1])
                e_ptr = new AlarmElevator(10);
          else
                e_ptr = new Elevator(10);
          e_ptr->go();
          cout<<"\nCurrently on floor:"
                     <<e_ptr->which_floor()<<"\n";
                              }
```

(그림 10) main()함수의 C++ 소스코드
(Fig. 10) The C++ code of main() function



(그림 11) main() 함수의 PRDG와 클래스 관계
(Fig. 11) Class relation and PRDG of main() function

3. 그래프 크기 분석

객체지향 프로그램의 표현을 위해 본 논문에서 제안하는 OPDG는 개별 클래스에 대한 각각의 CLDG, 클래스 상호작용과 다형성 표현을 위한CHS, 각 메소드 및 독립 프로시듀어와 main()함수 표현을 위한 PRDG로 구성되어 기존의 방법보다 더욱 간결하고 정확하게 표현할 수 있다. 또한 OPDG로 구성할 때 전체 크기도 감소된다. OPDG의 크기에 영향을 주는 항목들은 <표 1>에 나타낸다.

OPDG의 크기는 main() 함수의 Size(PRDG), 독립된 프로시져의 Size(PRDG), 클래스의 Size(CLDG)의 합이다. 각각의 크기를 결정하는 상한경계의 계산식은 다음과 같다[7,8].

$$Size(PRDG) = n * ((1 + v) + c * (1 + p * 2) + i * (1 + mv))$$
$$Size(PRDG) = mn + (n * v) + (c * (1 + p * 2)) + (p * 2$$

$$Size(CLDG) = d * ((cn + m * (1 + p * 2)) + m * (mn * n * v + c * (1 + p * 2)))$$

$$Size(OPDG) = Size(PRDG) + Size(PRDG) + Size(CLDG)$$

〈표 1〉 OPDG의 크기에 영향을 주는 요소
〈Table 1〉 Elements affecting the size of OPDG

항목	내 용
v	단일 메소드/프로시저에 대한 문장들의 최대수
e	단일 메소드/프로시저에 대한 간선들의 최대수
p	형식 매개변수의 최대수
c	호출정점의 최대수
cl	클래스의 수
m	클래스의 멤버인 메소드의 최대수
i	클래스 객체 생성 호출의 최대 횟수
mv	클래스 멤버 변수의 최대수
n	한 문장을 표현할 수 있는 최대크기
mn	한 메소드/프로시듀어 헤더를 표현할 수 있는 최대크기
cn	한 클래스 헤드를 표현할 수 있는 최대크기

위의 Size(OPDG)는 OPDG의 상한 경계값을 계산하는 수식이다. 그러나, 실제 OPDG의 크기는 계산식보다 상당히 적게 나타난다. 왜냐하면, 메소드/프로시듀어 호출에서 전체 매개변수의 값이 변경된다고 가정하였다[7,8]. (그림 5)의 예제 프로그램의 최대 상한 경계값과 실제 크기간의 차이는 아래 <표 2>와 같다.

(가정 : n = cn = mn = 1)

〈표 2〉 상한 경계값과 실제 크기의 차이
〈Table 2〉 differences of between real size and upper bound size

알고리즘	상한 계산값	실제 표현값
Loren D. Larsen	464	114
OPDG	325	89

따라서, 본 논문에서 제안하는 OPDG는 상한 경계값과 실제 표현값 양쪽에서 메모리의 공간 효율이 좋다는 것을 알 수 있다.

4. 결 론

객체지향 프로그램의 분석정보를 표현할 수 있는 새로운 OPDG를 제안하고 실제 예제 프로그램에 적용하여 표현하였으며, 또한 그 크기의 상한 경계값을 구할 수 있는 수식으로 나타내어 실제 OPDG로 표현한 크기와 상한 경계의 계산값을 기존의 알고리즘과 비교 분석하여 메모리 공간 효율이 훨씬 좋다는 것을 보였다. 제안하는 OPDG는 개별 클래스를 각각의 독립된 CLDG, 클래스의 상속관계를 나타내는 CHG, main() 함수와 독립 프로시듀어/메소드를 표현하는 PRDG로 구성되어 각 그래프는 필요시 재사용되며 각 프로그램의 내용이 수정될 때에만 재구성된다.

제안하는 OPDG의 장점은 기존의 방법보다 표현이 간단하고 정확하며, 호출문맥을 정확하게 표현하고, 점진적인 개발과 부분 표현을 할 수 있으며, 이미 구성한 CLDG, CHG, PRDG의 재사용과, 정적인 분석방법으로 다형성에대한 동적 바인딩 표현등을 들수있다.

향후연구 과제로는 동적 객체지향 프로그램 종속성 그래프로의 확장, OPDG의 구현등을 들 수 있다.

참 고 문 헌

[1] Ottenstein, K. J. and Ottenstein, L. M., "The program dependence graph in a software development environment," ACM SIGPLAN Not. Vol.19, No.5, pp.177-184, May. 1984.

[2] Ferrante, J., Ottenstein, K., and Warren,J., "The program dependence graph and its use in optimization," ACM Trans. Progrm. Lang. Syst. Vol.9, No.3, pp.319-349, July 1987.

[3] Susan Horwitz, Thomas Reps, and David Binkley, "Interprocedural Slicing Using Dependence Graphs," ACM Trans. Progrm. Lang. Syst., Vol. 12, No.1, pp.26-60, Jan., 1990.

[4] Mary Jean Harrold and Brian Malloy, " A unified interprocedural program representation for a maintenance environment," IEEE Transactions on Software Engineering, Vol.19, No.6, pp.584-593, June 1993.

[5] Mary Jean Harrold, Brian Malloy and Gregg Rothermel, "Efficient construction of program dep-

endence graphs," ACM International Symposium on Software Testing and Analysis, Vol.18, No.3, pp.160-170, June 1993.
[6] Brian A. Malloy, John D. Mcgregor, Anand Krishnaswamy and Murali Medikonda, "An Extensible Progarm Representation for Object-Oriented Software," ACM SIGPLAN Notice., Vol.29, No.12, pp.38-47, December 1994.
[7] Loren D. Larsen and Mary Jean Harrold, "Slicing Object-Oriented Software," Proceedings of ICSE-18. pp. 495-505, March 1996.
[8] Loren D. Larsen and Mary Jean Harrold, "Slicing Object-Oriented Softw...," Technical Report 95-103, Department of Computer Science, Clemson University, March 1995.
[9] Mary Jean Harrold and Gregg Rothermel, "A Coherent Family of Analyzable Graph Representations for Object-Oriented Software," Technical Report OSU-CISRC-11/96-TR 60, Department of Computer and Information Science, Ohio State University, 1996.
[10] Panos E. Livadas and Adam Rosenstein, Slicing in the Presence of Pointer Variables," Technical Report, Computer and Information Sciences Department, University of Florida, Gainesville, FL, 1995.
[11] H. D. Pande and B. G. Ryder, "Static type determination for C++," Technical Report LCSR-TR-197, Rutgers University, February 1993.
[12] H. D. Pande and B. G. Ryder, "Static type determination and aliasing for C++," Technical Report LCSR-TR-250, Rutgers University, July 1995.

류 희 열

1990년 경상대학교 전산통계학과 졸업(이학사)
1992년 경상대학교 대학원 전자계산학과(석사)
1996년 8월 경상대학교 대학원 전자계산학과 박사과정 수료
1995년~현재 동의공업대학 사무자동화과 조교수
관심분야 : 객체지향 소프트웨어공학, 역공학, 유지보수

박 중 양

1982년 연세대학교 상경대학 응용통계학과 졸업(학사)
1984년 한국과학기술원 산업공학과 석사
1994년 한국과학기술원 산업공학과 박사
1983년~현재 경상대학교 통계학과 교수
관심분야 : 소프트웨어 신뢰도분석, 소프트웨어 품질관리

박 재 홍

1978년 충북대학교 수학과 졸업(이학사)
1980년 중앙대학교 대학원 전자계산학과(석사)
1988년 중앙대학교 대학원 전자계산학과(박사)
1983년~현재 경상대학교 컴퓨터과학과 교수
관심분야 : 소프트웨어공학,소프트웨어 재공학,유지보수