개선된 동적 객체지향 종속 그래프

박순형, 박만곤
Christchurch College of Education, 부경대학교 컴퓨터멀티미디어 공학부

Improved Dynamic Object-Oriented Dependence Graph

Soon-Hyung Park, Man-Gon Park
Christchurch College of Education,
Division of Computers & Multimedia Engineering, PuKyong Nat'l University

요 약

동적 객체지향 프로그램 슬라이싱을 구현하기 위해서 프로그램 종속 그래프가 필요하다. 본 논문에서는 기존의 동적 객체지향 프로그램 종속 그래프 기법 보다 효율적인 동적 객체지향 프로그램 종속 그래프 기법을 제안하였다. 본 논문에서 제안한 기법이 기존에 비해 효율적임을 보이기 위해 그래프의 복잡도를 측정하여 비교하였다. 그리고 프로그램 슬라이스의 크기도 함께 측정하여 본 논문에서 제시한 기법이 효율적임을 증명하였다.

1. 서론

프로그램 슬라이싱(program slicing)은 프로그램 P에서 어떤 포인터 p에 있는 변수 var의 값에 직 간접적으로 영향을 끼치는 모든 명령문들을 찾는 과정으로서 주어진 어떤 프로그램 중에서 관심이 있는 변수에 대해 직접적 또는 간접적으로 관련된 명령문만을 모아놓은 또 다른 프로그램을 제공해 줌으로서 디버깅 등에 대한 효율성을 높여주는 기술이다[1-2].

동적 프로그램 슬라이싱(dynamic program slicing) 기법은 프로그램의 입력에 대해 실행 이력상의 어떤 실행위치 q에서 관심 변수에 관련된 원래의 프로그램과 그것의 결과가 일치하는 프로그램 실행의 부분을 산출하는 것으로, 동적 프로그램 슬라이스(dynamic program slices)는 프로그램의 입력 자료의 값에 의해 발생되는 프로그램의 실행 이력을 추적한 다음 실행 이력 상에서 기준 변수에 실제적으로 영향을 주는 모든 명령문들로 구성된다[3-4].

객체지향 프로그램 슬라이싱(object-oriented program slicing)은 객체지향 프로그램 종속 그래프에서 객체지향 프로그램의 중심이 되는 클래스와 객체의 흐름을 추적하여 객체지향 프로그램 속성들에 대한 슬라이스를 구하는 작업이다[5-6].

프로그램 슬라이싱 결과는 프로그램 종속 그래프(program dependence graph) 기법을 사용하여 산출할 수 있다[7]. 효율적인 프로그램 종속 그래프는 효율적으로 프로그램 슬라이싱을 할 수 있다. 그러므로 보다 효율적으로 슬라이싱을 표현할 수 있는 그래프에 대한 연구는 매우 가치가 있다고 할 수 있다.

본 논문에서는 기존의 동적 객체지향 프로그램 종속 그래프 (DOPDG: Dynamic Object-oriented Program Dependence Graph) 기법 보다 프로그램 슬라이싱을 더 효율적으로 표현할 수 있는 개선된 동적 객체지향 프로그램 종속 그래프 (IDOPDG: Improved Dynamic Object-oriented Program Dependence Graph 기법을 제안하였다.

본 논문에서는 다음과 같은 방법으로 구성된다. 먼저 2 장에서는 기존의 동적 객체지향 프로그램 종속 그래프 (DOPDG) 기법에 대한 관련 연구들에 대해 소개하였으며 3 장에는 본 논문에서 제안한 동적 객체지향 프로그램 슬라이싱을 위한 동적 객체지향 프로그램 종속 그래프 (IDOPDG)를 제안하였으며, 4 장에서는 IDOPDG 기법과 DOPDG 기법을 비교하기 위해 프로그램에 실제 적용시켜보았다. 그리고 5 장에서는 IDOPDG 기법과 DOPDG 기법을 비교한 다음 그 효율성에 대해 고찰하였다.

2. 기존의 동적 객체지향 프로그램 종속 그래프 기법

기존의 동적 객체지향 프로그램 종속 그래프(DOPDG) 기법은 슬라이스 결과를 산출하기 위해 Agrawal이 제안한 동적 프로그램 종속 그래프 기법을 사용한다. 즉, 해당 원시 프로그램의 제어 흐름과 자료 흐름에 대한 동적 분석에 기초하고 있다. 그리고 이것은 procedural 프로그램에 대한 동적 종속 그래프의 구축과 유사하다[8].

P를 객체지향 프로그램이라 하고, G = (V, A)를 P에 대한 DOPDG라 하자. 이때 V는 정점들의 집합이며, A는 간선들의 집합이다. 주어진 동적 슬라이스의 기준 (s, v, t, i)에 대한 G의 동적 슬라이스 DS_G는 G의 정점들의 subset이다. 객체지향 프로그램에 대한 슬라이싱 기준은 (s, v, t, i)이다. s는 프로그램의 명령문이고, v는 s에서 사용된 변수, t는 입력 i에 대한 실행 궤도이다. 만일, G에서 v'에서부터 v까지의 path가 존재한다면 $v' \in V$, $v' \in DS_G$ (s, v, t, i)에 대해 DS_G (s, v, t, i) $\subset$ V이다.

동적 슬라이스들을 산출하기 위해 주어진 실행 궤도에 대해 DOPDG를 구성하는 즉시 기준 노드에 대해 DOPDG를 운행하기 위해 depth-first 혹은 breadth-first 그래프 운행 알고리즘을 사용한다. 객체지향 프로그램의 DOPDG에 대한 동적 슬라이스는 DOPDG의 정점들의 subset이다.

3. 개선된 동적 객체지향 프로그램 종속 그래프 기법

개선된 동적 객체지향 프로그램 종속 그래프(IDOPDG) 기법은 기존의 DOPDG 기법에 비해 클래스간 인터페이스와 다형성 호출 등을 효율적으로 표현할 수 있는 동적 객체지향 프로그램 종속 그래프 기법이다. IDOPDG는 원시 프로그램에 대한 정적 정보와 주어진 입력 자료에 의한 실행이력을 기반으로 작성된다. IDOPDG를 작성하는 단계는 다음과 같다.

(1) 실행이력에 있는 노드에 대해 원시 프로그램의 정적 정보를 사용하여 아래에 있는 종속 그래프를 그린다.
   ① procedure 제어 종속 간선
   ② method 제어 종속 간선
   ③ while 문 제어 종속 간선
   ④ if 문 제어 종속 간선
   ⑤ call 문에 의한 inter-procedure 간선
   ⑥ return 제어 종속 간선
   ⑦ 다형성 종속 간선

(2) 실행이력에 있는 기준 노드로부터 자료 종속 간선을 구한 후 이 간선이 그래프에서 path로 존재하지 않으면 그래프에 추가한다.

(3) 실행이력에 있는 기준 노드로부터 제어 종속 간선을 구한 후 이 간선이 그래프에서 path로 존재하지 않으면 그래프에 추가한다. 그러나 한번만 실행되는 "if" 제어 노드는 슬라이스에서 제외되고, 해당 "if" 노드에만 종속되는 노드들은 슬라이스에서 삭제된다.

```
CE1 :  class Elevator {
       public:
E2:    Elevator(int 1_top_floor)
S3:     { current_floor = 1;
S4:       current_direction = UP;
S5:       top_floor = 1_top_floor; }
E6:    virtual ~Elevator() {}
E7:    void up()
S8:     { current_direction = UP; }
E9:    void down()
S10:     { current_direction = DOWN; }
E11:   int which_floor()
S12:    { return current_floor; }
E13:   Direction direction()
S14:    { return current_direction; }
E15:   virtual void go (int floor)
S16:    { if (current_direction == UP)
S17:       { while ((current_floor != floor) &&
                    (current_floor <= top_floor))
C18:         add(current_floor, 1); }
         else
S19:       { while ((current_floor != floor) &&
                    (current_floor > 0))
C20:           add(current_floor, -1); }
       }
       private:
E21:    add(int &a, const int& b)
S22:     { a = a + b; };
       protected:
        int current_floor;
        Direction current_direction;
        int top_floor;
       };
CE23:  class AlarmElevator : public Elevator {P
       public:
```

```
E24:    AlarmElevator(int top_floor):
S25:      Elevator(top_floor)
S26:      { alarm_on = 0; }
E27:    void set_alarm()
S28:      { alarm_on = 1; }
E29:    void reset_alarm()
S30:      { alarm_on = 0; }
E31:    void go(int floor)
S32:    { if (!alarm_on)
C33:        Elevator::go(floor)
        } ;
      protected:
        int alarm_on;
      } ;

E34:    main(int argc, char **argv) {
        Elevator *e_ptr;
S35:    if (argv[1])
S36:      e_ptr = new AlarmElevator(10);
        else
S37:      e_ptr = new Elevator(10);
C38:    e_ptr->go(3);
S39:      cout << "\n Currently on floor:" <<
                  e_ptr->which_floor()
               << "\n";
      }
```

프로그램 1. 예제 프로그램

4. 적용 사례

프로그램 1의 예제 프로그램에서 argv[1] = 3이고, 슬라이싱 기준이 (H, 39_{22}, which_floor)일 때, 동적 객체지향 슬라이스를 구하기 위해 기존의 DOPDG 기법과 제안한 IDOPDG 기법에 각각 적용시킨다.

4.1 기존의 동적 객체지향 프로그램 종속 그래프 기법

(H, 39_{22}, which_floor)를 슬라이싱 기준으로 했을 때 즉, 기준 노드 39번에 존재하는 변수 which_floor를 슬라이싱 기준 변수로 했을 때 DOPDG가 그림 1에 나타나 있다.

4.2 개선된 동적 객체지향 프로그램 종속 그래프 기법

(1) IDOPDG에서 procedure 제어 종속 간선은 (34→37), (34→38), (34→39), (39→11), (11→12)이다. 그리고 method 제어 종속 간선은 (2→3), (2→5), (15→17), (21→22) 이고, while 문 제어 종속 간선은 (17→18) 그리고 inter-procedure 제어 종속 간선은 (18→21), (37→2) 이고, return 제어 종속 간선은 (3→17), (5→17), (12→11), (22→18)이다. 그리고 다형성 종속 간선은 (38→15)이다.

(2) 추가자료 종속 간선은 (18→12) 이다.

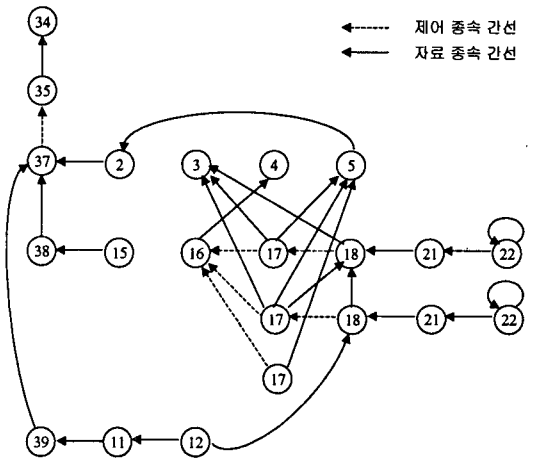

그림 1. 프로그램 1에서 C = (s39, which_floor)에 대한 DOPDG

프로그램 1의 예제 프로그램에 대한 IDOPDG가 그림 2에 나타나 있다.

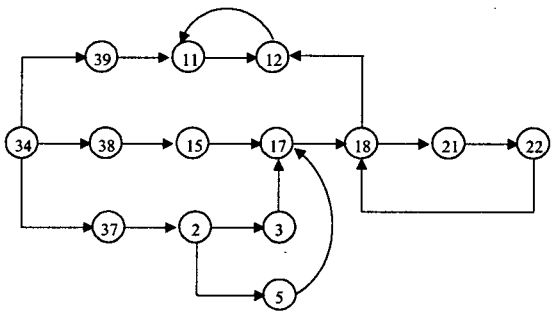

그림 2. 프로그램 1에서 C = (s39, which_floor)에 대한 IDOPDG

(H, 39_{22}, which_floor)를 슬라이싱 기준으로 했을 때 즉, 기준 노드 39번에 있는 슬라이싱 기준 변수 which_floor에 대한 동적 슬라이스를 구하기 위해 그림 2에 있는 IDOPDG를 역 운행한 결과 산출된 슬라이스 노드는 {2, 3, 5, 11, 12, 15, 17, 18, 21, 22, 34, 37, 38, 39}가 된다.

5. 효율성 비교 분석

기존의 DOPDG와 본 논문에서 제시한 IDOPDG의 슬라이스의 크기와 복잡도를 측정하면 다음과 같다. 복잡도는 노드와 간선들의 합이다.

5.1 효율성 비교

프로그램 1의 예제 프로그램에 대하여 명령문 39번의 which_floor()를 기준으로 했을 때, 기존의 DOPDG 와 본 논문에서 제시한 IDOPDG의 슬라이스의 크기와 복잡도를 비교하는 테이블이 표 1에 나타나 있다.

표 1. 복잡도를 비교하는 테이블

	슬라이스의 크기	복잡도
동적 객체지향 프로그램 종속 그래프 (DOPDG)	22	52
개선된 동적 객체지향 프로그램 종속 그래프 (IDOPDG)	14	31

5.2 효율성 분석

기존의 DOPDG 기법에서는 프로그램에서 반복 제어 횟수가 많은 반복문이 포함되면 그래프의 복잡도가 그 횟수에 비례해서 복잡해진다. 그러나 본 논문에서 제시한 IDOPDG 기법에서는 기존의 기법에 비해 그래프의 복잡도가 작아짐을 알 수 있다. 특히 반복문과 같은 제어문의 표현에 효율적이다. 그리고 IDOPDG 기법을 적용하면 기존의 DOPDG 기법에 비해 프로그램 슬라이스의 크기가 작아짐을 알 수 있다. 그리고 기존의 DOPDG 기법이 종속 그래프에서 depth-first 혹은 breadth-first 방식의 탐색 방법을 사용하였기 때문에 그래프가 복잡해지면 탐색이 복잡해질 수 있으나 IDOPDG 기법은 back tracking 방법으로 종속 그래프를 탐색함으로써 그래프의 복잡성과는 상관없이 탐색을 손쉽게 할 수 있다.

6. 결론

정적 슬라이스는 주어진 기준변수에 영향을 끼치는 모든 노드이고, 동적 슬라이스는 주어진 프로그램 입력에 대해 발생된 변수 값에 실제적으로 영향을 주는 명령문들로 구성되어 있다. 그러므로 어떤 시험 사례를 통해 프로그램을 분석하는 디버깅 분야에서는 동적 슬라이싱이 정적 슬라이싱 보다 더 유용하게 사용될 수 있다. 본 논문에서는 개선된 동적 객체지향 프로그램 종속 그래프(IDOPDG) 기법을 제안하였다. 기존의 동적 객체지향 프로그램 종속 그래프(DOPDG) 기법과 본 논문에서 제시한 IDOPDG 기법을 사용하여 프로그램 1의 예제 프로그램에 적용한 결과 IDOPDG의 복잡도가 31이고, 기존의 DOPDG의 복잡도는 52임을 알 수 있었다. 그리고 본 논문에서 제시한 IDOPDG 기법을 적용하여 슬라이스를 산출한 결과 14개의 슬라이스를 산출하였다. 이것은 기존의 DOPDG 기법을 적용한 결과 산출된 22개의 슬라이스보다도 작기 때문에 본 논문에서 제안한 IDOPDG 기법이 기존의 DOPDG 기법에 비해 효율적인 기법임을 알 수 있었다.

[참고문헌]

[1] Bodan Korel, "Computation on Dynamic Program Slices for Unstructured Programs", IEEE Trans. on Software Engineering, vol. 23, No. 1, pp.17-34, January 1997

[2] Margaret Ann Francel, Spencer Rugaber, "The value of slicing while debugging.", Science of Computer Programming, Volume 40, Number 2-3, pp.151-169, July 2001

[3] B.Korel and J. Laski, "Dynamic Slicing in Computer Programs", The Journal of System and Software Engineering, vol.23, No.1, pp.17-34, 1997

[4] B.Korel and J. Laski, "Dynamic Program slicing", Information Proceeding Letters, vol.29, No.3, pp.155-163, 1998

[5] Loren D. Larsen and Mary Jean Harrold, "Slicing Object-Oriented Software.", Technicdal Report 95-103, Department of Computer Science, Clemson University, March 1995

[6] Zhengqiang Chen, Xu Baowen, "Slicing Object-Oriented Java Programs.", SIGPLAN Notices, Volume 36, 2001, Volume 36, Number 4, pp.33-40, April 2001

[7] 박순형, 박만곤, "Dynamic Slicing using Dynamic Spendence Graph.", 한국멀티미디어학회 논문지, 제5권 제3호, pp.331-341, 2002. 6

[8] J. Zhao, "Dynamic Slicing of Object-Oriented Programs," Technical-Report SE-98-119, pp.17-23, Information Processing Society of Japan (IPSJ), May 1998