객체지향 프로그램 슬라이싱을 위한 개선된 시스템 종속성 그래프에 대한 연구

류희열[1], 김은정
동의공업대학 컴퓨터정보계열, 부산외국어대학교 컴퓨터전자공학부
hyryu[1]@dit.ac.kr, ejkim@taejo.pufs.ac.kr

A study on the Enhanced System Dependence Graph for slicing of object-oriented Program

Hee-Yeol Ryu[0] Eun-Jung Kim
Dept. of Computer Information, Dongeui Institute of Technology
Dept. of Computer Engineering, Pusan University of Foreign Studies

요 약

객체지향 프로그램 슬라이싱을 위한 Loren D Larsen and Marry Jean Harrold가 제안하는 방법은 절차적 프로그램 슬라이싱을 위한 시스템 종속성 그래프 표현방법에 객체지향 패러다임을 표현할 수 있도록 확장하며 2단계 마킹 알고리즘을 적용하여 슬라이스를 계산한다. 시스템 종속성 그래프를 이용한 슬라이싱 방법은 클래스 멤버 변수와 전역변수 및 인스턴스 변수에 대하여 각 메소드 호출 및 진입정점에 actual_in, actual_out, formal_in, formal_out 정점들이 추가되어 복잡도가 증가한다. 본 논문에서는 이들 변수를 클래스 정점의 멤버간선으로 연결하여 각 메소드의 문장에서 사용하면 진출간선, 정의하면 진입간선으로 연결하여 정점과 간선들의 개수를 최소화할 수 있도록 시스템 종속성 그래프를 개선하였다. 제안하는 시스템 종속성 그래프는 그래프 복잡도의 최소화와 2단계 알고리즘에 의한 정확한 슬라이스 계산이 장점이다. C++ 예제 프로그램을 적용하여 그래프 복잡도의 감소와 정확한 슬라이스 계산을 기존의 방법과 비교하여 개선됨을 확인하였다.

1. 서 론

프로그램 슬라이싱은 자료흐름과 제어흐름 분석을 이용하여 프로그램 행위를 관심있는 특정한 부분으로 제한하는 기법으로 테스팅, 디버깅, 코드이해, 유지보수등에 유용하게 이용된다. 문장번호와 관심있는 변수의 쌍으로 구성된 슬라이싱 기준이 주어지면 자료흐름과 제어흐름 분석을 이용하여 자동적으로 슬라이스가 계산된다. 슬라이스는 원시 프로그램과 동일한 행위와 결과를 생성하고 실행가능해야 한다는 특징이 있다.[1] 절차적 프로그램에 대한 슬라이싱은 소스코드를 프로그램 종속성 그래프(Program Dependence Graph ; PDG)로 변환하여 Susan Horwitz, Thomas Reps, David Binkley가 제안하는 2패스 마킹 알고리즘을 적용하는 방법이 가장 정확한 슬라이스를 계산할 수 있다고 알려져 있다.[2] 또한 프로시저간의 슬라이싱을 위해 PDG를 확장한 시스템 종속성 그래프(System Dependence Graph ; SDG)를 이용하였다.[2]

최근 객체지향 프로그램의 테스팅과 유지보수를 위한 슬라이싱에 대한 연구가 진행되면서 클래스, 파생클래스, 상속, 다형성등 객체지향 개념을 표현할 수 있는 다양한 표현방법이 연구되었다. Loren D. Larsen and Marry Jean Harrold가 SDG에 객체지향 개념을 표현할 수 있도록 확장하여 새로운 SDG를 제안하였으며 여기에 2단계 마킹 알고리즘을 적용하여 객체지향 프로그램 슬라이스를 계산하였다.[3] 이 방법은 클래스, 파생클래스, 상속, 다형성등의 객체지향 개념을 단일 그래프로 표현할 수 있으며 기존의 슬라이싱 알고리즘을 적용하여 정확한 슬라이스를 계산할 수 있는 것이 장점이다. 그러나 멤버변수, 전역변수, 인스턴스변수들에 대하여 클래스의 각 메소드 진입정점에 *formal_in, formal_out, actual_in, actual_out* 정점들을 추가하여 호출문맥을 표현하기 때문에 상당히 많은 정점이 표현되고 각 정점들간의 간선은 그래프의 복잡도를 높이고 슬라이싱 알고리즘의 실행시간을 증가시키는 문제점이 있다.

본 논문에서는 클래스 멤버변수와 인스턴스변수를 클래스 진입정점의 멤버간선으로 연결하고 각 메소드의 문장에서 이들 변수를 사용하면 그 문장정점과의 진출간선으로 연결하고, 이들 변수를 정의하면 그 문장정점과의 진입간선으로 연결하여 호출문맥에 따른 정점과 간선의 수를 최소화하는 ESDG(Enhanced SDG)를 제안한다. 또한 C++로 작성된 예제 프로그램을 적용하여 제안하는 SDG로 표현하고 2패스 마킹 알고리즘을 적용하여 그래프의 복잡도 감소와 정확한 슬라이스 계산을 기존의 방

법과 비교하여 개선됨을 확인하였다.

2. ESDG

2.1 클래스 종속성 그래프

단일 클래스를 표현하는 클래스 종속성 그래프는 클래스의 멤버변수와 메소드들에 대해서 클래스 멤버간선으로 연결한다. 각 메소드는 PDG로 표현되며 클래스 멤버변수가 각 메소드의 문장정점에서 참조되면 멤버변수에서 문장정점으로 자료 종속성 간선을 연결한다. 또한 각 메소드의 문장정점에서 멤버변수가 정의되면 문장정점에서 멤버변수로의 자료 종속성 간선으로 연결한다. C++의 return문장에 대한 표현은 return 문장에서 호출한 정점으로 자료 종속성 간선을 연결한다. 파생 클래스는 파생 클래스의 메소드와 중복되지 않으면 클래스 진입정점에서 기저 클래스의 메소드 진입정점으로의 클래스 멤버간선으로 연결한다. 다형성 메소드는 포인트와 같이 참조된 객체의 자료형이 컴파일 시간에 결정되지 않기 때문에 호출가능한 모든 메소드를 호출간선으로 연결한다. 다형성 호출의 모든 가능한 목적지를 줄이는 것은 NP-hard로 알려져 있다.[2]

2.2 호출문맥 표현

메소드 호출이 발생하면 형식매개변수와 실매개변수 사이의 대응관계를 표현하기 위해서 actual_in, actual_out, formal_in, formal_out 정점을 이용하여 표현하며 각 정점들을 메소드 진입정점에서 parameter_in, parameter_out 간선으로 연결하여 매개변수 전달을 명시적으로 나타내는 방법이다. (actual_in, formal_in), (formal_out, actual_out)으로 자료 종속성 간선으로 연결되고 또한 actual_in 정점에서 대응되는 actual_out 정점으로의 이행적 종속성 관계가 존재하면 actual_in --> actual_out 이행적 종속성 간선으로 연결한다.

3. 실험 및 결과

Loren D. Larsen and Mary Jean Harrold의 방법에서 사용한 C++로 작성된 Elevator 예제 프로그램을 이용하여 그래프의 표현과 정확한 슬라이스를 계산하는 것을 비교 평가한다.

표 1 C++ 예제 프로그램

1	class Elevator {
	private:
2	Elevator (int 1 top floor) {
3	current floor = 1;
4	current direction = UP;
5	top floor = 1 (top floor);
6	Elevator() {};
7	virtual void up() {
8	current direction = UP;};
9	virtual void down() {
10	current direction = DOWN;};
11	int which floor() {
12	return current floor;};
13	Direction diretion() {
14	return current direction;};
15	virtual void go (int floor) {
16	if(current direction == UP) {
17	while(current floor != floor) && (current floor <= top floor))
18	add(current floor, 1);}
19	else {
20	while(current floor != floor) && (current floor > 0))
	}
21	private : add(int &a, const int &b) {
22	a = a + b;};
	protected :
	int current floor;
	Direction current direction;
	int top floor;
	};
23	AlarmElevator(int top floor) : public Elevator {
	public :
24	AlarmElevator(int top floor) :
25	Elevator(top floor) {
26	alarm on = 0;};
27	void set alarm() {
28	alarm on = 0;};
29	void reset alarm() {
30	alarm on = 0;};
31	void go(int floor) {
32	if(!alarm on)
33	Elevator::go(floor);
	};
	protected :
	int alarm on;
	};
34	main (int argc, char **argv)
	{
	*Elevator *e ptr;*
35	if (argv[1])
36	e ptr = new AlarmElevator(10);
	else
37	e ptr = new Elevator(10);
38	e ptr -> go(3);
39	cout << "\n Currently on floor: " << e ptr -> which floor() << "\n";
	}
60	Producer p1 = new Producer (e, 1);
61	Consumer c1 = new Consumer (c, 1);
62	p1.start();
63	c1.start();
64	}
65	}

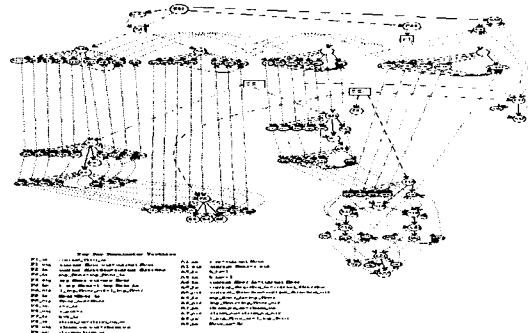

그림 1 SDG 표현

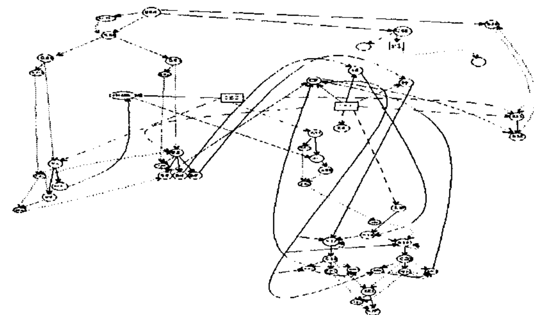

그림 2 ESDG 표현방법

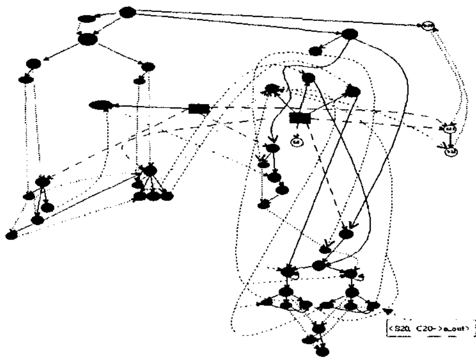

그림 3 슬라이싱된 ESDG

표 2 슬라이스 코드<s20, s20->a_out>

1	class Elevator {
2	private :
	Elevator (int 1 top floor) {
3	current floor = 1;
4	current direction = UP;
5	top floor = 1 top floor;}
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	virtual void go (int floor) {
16	if(current direction = UP) {
17	while((current floor != floor) && (current floor <= top floor))
18	add(current floor, 1);}
19	else {
20	while((current floor != floor) && (current floor > 0))
	}
	private
21	add(int &a, const int &b) {
22	a = a + b;};
	protected :
	int current floor;
	Direction current direction;
	int top floor;
	};
23	AlarmElevator(int top floor) : public Elevator {
	public
24	AlarmElevator(int top floor) :
25	Elevator(top floor) {
26	alarm on = 0;};
27	
28	
29	
30	
31	void go(int floor) {
32	if(!alarm on)
33	Elevator::go(floor);
	};
	protected
	int alarm on;
	};
34	main (int argc, char **argv)
	{
	Elevator *e ptr;
35	if (argv[1])
36	e ptr = new AlarmElevator(10);
	else
37	e ptr = new Elevator(10);
38	e ptr -> go(5);
39	
	}

4. 분 석

본 논문에 제안하는 ESDG 구성을 위한 메모리 공간 크기 분석을 Loren D Larsen and Mary Jean Harrold가 사용한 방법을 사용한다. 이 방법은 그래프의 크기를 정점개수에 대한 상한경계값으로 표현한다. <표 3>은 ESDG의 크기에 영향을 주는 요소들을 나타낸다.

표 3 ESDG의 크기에 영향을 주는 요소

항목	정 의
V	단일메소드 내에서 문장과 술부 정점의 최대수
E	단일메소드 내에서 간선들의 최대수
P	메소드 및 함수의 형식매개변수의 최대수
G	시스템 내에서 전역변수의 개수
I	클래스에서 인스턴스 변수의 최대수
C	단일메소드 내에서 호출지역의 최대수
T	상속 트리의 깊이
M	시스템 내에서 메소드의 개수

SDGParamVertices(m) = (P + G + I)
SDGSize(m)= O(V+C*(1+T*(2*SDGParamVertices(m)))
+ 2 *SDGParamVertices(m))
Size(SDG) = O(SDGSize(m) * M)

ESDGParmVertices(m) = P
ESDGSize(m) = O(V + C*(1 + T*(2 * P)) + 2 *P)
Size(ESDG) = O(ESDG(m) * M)

슬라이싱을 위한 2단계 마킹 알고리즘을 적용하면 순회비용은 그래프의 크기에 선형비례 한다. 따라서 본 논문에서 제안하는 ESDG가 기존의 SDG보다 정점의 개수가 적기 때문에 슬라이싱 알고리즘의 순회비용도 감소됨을 알 수 있다.

5. 결 론

본 논문에서는 객체지향 프로그램 슬라이싱을 위한 ESDG를 제안하였다. 클래스 멤버변수, 전역변수, 인스턴스변수에 대한 parameter_in, parameter_out 정점의 개수를 최소화함으로써 그래프의 복잡도와 간선의 수를 줄였으며 C++ 예제 프로그램에 적용하여 비교하였으며 2단계 마킹 알고리즘을 적용하여 정확한 슬라이스가 계산됨을 확인하였다. 향후 연구과제로는 제안하는 ESDG를 이용한 슬라이싱 툴의 개발과 객체지향 동적 슬라이싱을 들 수 있다.

참고 문헌

[1] Weiser, M. "Program Slicing." IEEE Transactions on Software Engineering 10,4(1884),pp.352-357.1984

[2] Suasn Horwits and Tomas Reps, and David Binkly,"Interprocedural Slicing Using Dependence Graph." ACM Transaction Programming Languages and Systems,Vol.12,No.1, January 1990,Pages 26-60.

[3] Loren D. Larsen and Marry Jean Harrold,"Slicing Object-Oriented Software." Proceedings of ICSE-18. pp.495-505, March 1996.