KAIS 97 추계학술대회

실시간 객체지향 설계에서 슬라이싱맵의 데이터베이스 구축 및 활용에 관한 연구

박상백, 박만곤
부경대학교 전산정보학과

요약

실시간 시스템(realtime system)은 외부환경에서 보내진 요구사항(requirement)에 정해진 시간 내에 반응(action)해야만 한다. 시스템의 외부환경은 실세계(real world)의 물리적인 요소(physical element)가되며 이러한 물리적인 요소는 객체지향 설계(object-oriented design) 개념에서 객체(object)이다. 또한 시스템을 설계하는 경우에는 객체지향 방법론을 적용함으로서 보다 명확한 표현이 가능하게되며 실세계의 물리적인 각 요소가 부품단위(component units)로 동작하는 형태를 소프트웨어로 설계하는 것이 용이하게된다. 따라서 본 논문에서는 실시간 시스템설계에서 객체지향 방법을 사용하고, 원시(source) 소프트웨어의 분석(analysis)과 검증(verification) 및 흐름(flow)을 슬라이스 기법을 사용하여 표현함으로서 시스템의 전체적인 구성을 명확하게 표현하며 클래스(class) 단위로 슬라이스된 각부분으로 상속(inheritance)과 메시지 전달(message sending)의 흐름을 파악할 수가 있다. 그리고, 보다 효율적으로 시스템의 외부환경의 오동작 처리 및 시스템 흐름을 감시 추적(monitoring and tracing)할수 있도록 슬라이싱맵을 데이터베이스로 구축하여 활용하는 방법을 제시한다. 또한 슬라이싱맵 표현 방법에 있어서 기존의 표현을 보다 구체적이고 실제적인 표현기호를 제시한다.

1. 서론

실시간 시스템(realtime system)은 다수의 작업을 정해진 시간에 실행해야 하는 작업의 주기성을 가지고 있는 시스템이다. 그리고, 실시간시스템은 주어진시간 내에 비주기적인 요구사항에 대해서도 반응할 수 있어야 한다. 시스템을 처리하는 과정동안에 대기하고 또한 실행되어야 하며 주경로(critical path)를 따라 진행되도록 구축되어야 한다. 만약 필수작업, 즉 주경로상의 작업이 특별히 정해진 시간내에 처리되지 못하는 경우는 시스템의 전체적인 흐름이 변화되어야만 하며 이는 작업의 재분배 문제를 일으켜서 궁극적으로는 시스템의 결함이 되는 경우이다.
실시간 시스템이 필수적으로 갖추어야하는 특성은 다음과 같은 것들이 있다.

1. 적시성(timeliness)
2. 동적내부구조체(dynamic internal structure)
3. 반응성(reactiveness)
4. 병행성(concurrency)
5. 분산성(distribution)

시스템구축에 있어서 실시간 객체지향적 방법(object-oriented methods)은 객체(object)들의 상호통신 세트를 정의하여 구축된다. 그리고 실시간 시스템에서의 객체지향 시스템방법을 보면 Booch는 객체 및 클래스 (class)의 구조분석에 DFD(data-flow diagram)을 사용하였으며 객체인터페이스를 사용하여 Ada프로그램을 변환하였다. Booch는 분석과 설계를 분리하지 않았고 분석에서의 객체를 설계단계

에서 그대로 적용하며 시스템을 정적 모델(static model)과 동적 모델(dynamic model)로 나누어 설계하였다. 그러나 분석, 설계, 구현단계의 연관성이 부족하게 되었다.(Booch,1987) Shlaer와 Mellor는 분석을 중요시하며 데이터 모델링 단계를 중요시하였다. 또한 정보 모델(information model), 상태 모델 (state model), 프로세스 모델(process model)의 세단계로 구성하였다. 그러나, 시스템의 분해성은 우수하나 객체의 반복과 설계 단계 및 구현 단계의 상세함이 결여되었다.(Shlaer, 1985) Rumbough의 객체 모델링 기법 (OMT:object-modeling techniqes)은 그래픽 표기법이 우수하며 시스템의 분석, 설계, 구현단계의 일관성이 우수한 편이다. 그러나, OMT는 객체모델만 중요시하며 동적모델과 기능모델을 객체의 보완적인 요소로만 취급하고 있다(Rumbaugh,1991). ROOM(realtime object-oriented modeling)은 정형적인 설계모델(formal design model)로서 shlaer와 mellor의 설계방법에서 이론을 가져왔다. ROOM은 객체간의 메시지(message)를 이용하여 상호작용 엑터(actor)를 사용한 방법이다. 그러나, 정적 모델에서의 제어중심적인 방법으로서 동적인 모델을 처리하는 경우는 사용이 불편하다(Selic, 1994). 이외에도 여러가지 방법이 존재하지만 실시간 객체지향 시스템이 가져야 하는 실세계의 비동기적(asynchronous)이고 병행적(concurrent)이고 분산적(distributed)이며 객체간의 통신처리 환경에 정확하게 맞추는 것은 힘든일이다. 또한, 프로그램 슬라이싱(program slicing)은 디버깅(debugging), 코드 이해(code recognition), 프로그램 테스트(program test), 소프트웨어 메트릭스(software metrics), 재사용(reuse)등과 같은 어플리케이션에 이용할 수가 있다. 프로그램 슬라이싱에는 정적 슬라이싱(static slicing), 동적 슬라이싱(dynamic slicing), 객체지향 슬라이싱(object-oriented slicing)으로 나눌수가 있으며 정적 슬라이싱은 Weiser에의해서 처음으로 소개되었으며 Ottenstein은 프로그램 종속 그래프(PDG:program dependency graph) (Ottenstein, 1987)로 선형적 시간을 사용하는 내부 절차적인 슬라이스(intraprocedural slice)를 사용하였다. Horowitz는 시스템 종속 그래프(SDG: system dependency graph) 즉, PDG의 수퍼그래프로서 내부 절차적인 슬라이스를 구축하는 알고리즘을 소개하였다. 정적 슬라이싱은 주어진 변수의 값에 의해서 영향을 주는 모든문장들의 집합이다. 프로그램의 선언문(declaration), 할당문(assignment), 제어문(control statement) 등의 요소에 영향을 주는 것들이다. 또한, 정적 슬라이스는 어떤 노드(node)에서 변수가 도달가능한 노드들의 집합을 PDG를 운용하여 찾을 수가있다. 동적 슬라이스는 정적 슬라이스가 가지는 선언문, 할당문, 제어문에 영향을 받는 프로그램의 모든 문장 뿐만아니라 조건문(condition statement), 테스트사례(test-case), 실행히스토리(execution history)와 변수에 의해 영향을 받는 프로그램의 추적(tracing) 등에 사용가능하다(Hiralal Agrawal and J.R. Horgan, 1990). 또한 정적 슬라이스에서 사용되는 PDG가 가지는 문제인 동일변수의 여러문장에의 도달을 정의하지 못하는 단점을 실행 히스토리에서 노드를 분리시키는 그래프를 사용하여 해결한다. 이 그래프를 동적 종속 그래프(DDG:dynamic depecdency graph)라고한다. 그리고, 객체지향 슬라이스는 객체지향 프로그램의 중심이되는 클래스(class)와 객체(objects)의 흐름(flow)을 추적해 내는 것이라 할 수가 있다. 객체지향 프로그램에서의 문제는 다형성(polymorphism), 동적 바인딩(dynamic binding), 클래스 상속(class inheritance) 등을 표현하는 것이다. 이를 위해서 객체지향 슬라이스는 PDG(program dependency graph)를 기초로 하여 SDG(system dependency graph), CFG(control flow graph), CDG(class dependency graph), CHS(class hierachy subgraph), CDS(control dependency subgraph), DDS(data dependency subgraph) (Melloy, 1993) 등을 이용하여 객체지향 시스템을 표현한다. 본 논문의 목적은 이와같은 동적 슬라이스와 정적 슬라이스가 실시간 객체지향 설계에서 객체들을 표현하지 못하는 문제와 객체상호간의 관계를 표현하는데 한계를 나타내는 것을

해결하며 또한, 기존의 객체지향 슬라이스에서 객체모듈 흐름의 단순한 표현을 보완하고자 보다 구체화된 표현법을 제시하고, 제시된 표현법을 사용하기위해 프로그램을 모듈별로 분해하여 각모듈의 실행패턴 흐름을 매트릭스(matrix)로 구성하여 슬라이스맵을 데이터베이스로 구축하여 활용하는 방법을 제시하는 것을 목적으로 한다.

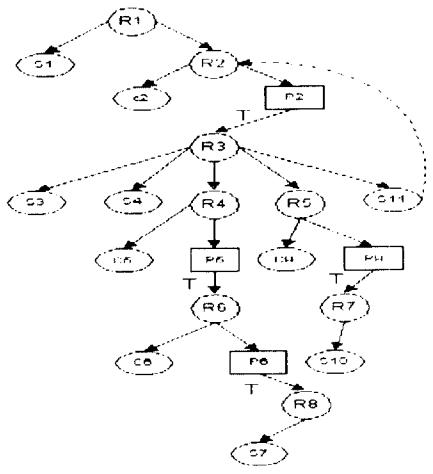
2. 프로그램 슬라이싱 표현

2.1 슬라이스 표기법

● 프로그램 종속 그래프
(PDG:program dependency graph)

단일 구조에서 제어(control)와 데이터(data)를 기호화한 프로그램의 그래픽적인 표기법이다. <그림 1>은 PDG의 예를 표현하며 타원(ellipse)은 문장(statement)을 사각(rectangle)은 제어문(condition)을 원(circle)은 제어 종속의 분기를 표현하며 실선(solid edges)은 데이터종속(data dependency)이고 점선(dashed edges)은 제어 종속(control dependency)을 나타내고 있다(Anand Krishnaswamy,1994).

```
S1:     n = 2;
P2,C2:  while(n <= 100)
        {
S3:         div = 2;
S4:         flag = 1;
P5,C5:  while(flag && div <= n/2)
        {
P6,C6:  if(n % div == 0)
S7:         flag = 0;
S8:         div++;
        }
P9,C9:  if(flag)
S10:        cout<<n;
S11:        n++;
        }
```



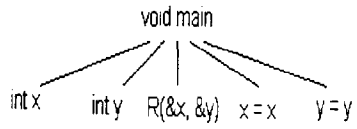
<그림 1> 프로그램 종속 그래프(PDG)

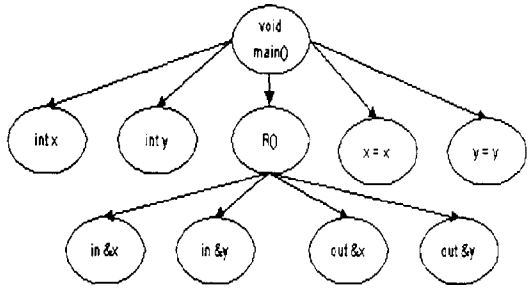
● 시스템 종속 그래프
(SDG:system dependency graph)

호출(call)을 위한 PDG와 클래스(class)를 위한 CDG(class dependency graph)로 구성된다. 또한 SDG는 프로그램의 파스트리(parse tree)형태이며 PDG의 directed multigraph이며 파라메타의 전달과 데이터와 제어의 내부절차적인 흐름(interprocedural flow)을 다룬다. <그림 2>는 SDG를 표현한것이며 각 표기법은 PDG와 동일하다(Horowitz S;Reps T and Binkley D, 1990).

```
1 : void main()
2 : {
3 : int x, y;
4 : R(&x, &y);
5 : x = x;
6 : y = y;
7 : }
8 :
9 : void R(int *x, *y)
10: {
11:     if(*y == 0)
12:       *x = *x + 1;
13:     else if(*y == 1) {
14:         *y = *y + *x;
15:         R(x, y);
16:         *x = *x + 1;
17:     }
18:     else {
```

```
19:          *x = *x - 1;
20:          *y = *y - 1;
21:          R(x, y);
22:      }
23: }
```



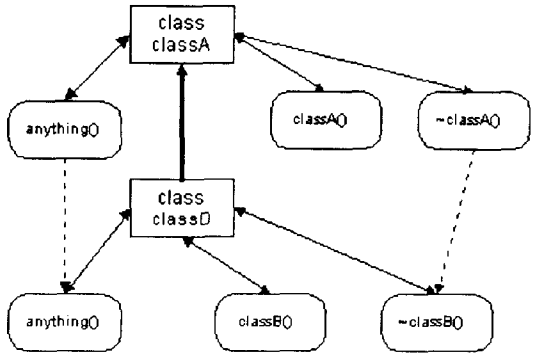


<그림 2> 시스템 종속 그래프(SDG)

● 클래스계층 서브그래프
(CHS:class hierachy subgraph)

클래스계층을 표현하는 기본적인 요소이며 상속성 관계(inheritance relationship), 매소드의 구성(composition of methods)을 표현한다. 그러나, 수퍼클래스와의 관계 표현은 하지않는다. 또한 상속성테스트와 유용한 정보를 포함한다. <그림 3>은 클래스계층 서브그래프의 표현이다.

```
class classA{
    private:
        . . . .
    public:
        classA();
        void anything();
    };
class classB : public classA{
    private:
        . . . .
    public:
        classB();
        ~classB();
        void anything();
}
```



<그림 3> 클래스계층 서브그래프(CHS)

2.2 슬라이싱 제안 표기법

실시간 객체지향 시스템(realtime object-oriented slicing)에서의 슬라이싱은 절차적인 슬라이싱(procedural slicing)보다 더욱 복잡하다. 그래서, 슬라이싱의 표기법 또한 복잡해지며 표기법에 사용되는 그래픽은 좀더 구체적이고 명확한 형태를 가지는 것이 시스템을 이해하기가 쉽고 프로그램의 슬라이싱을 표현하는 것이 쉽다. 다음의 <표 1>은 실시간 객체지향에서 사용하고자하는 제안 표기법이다.

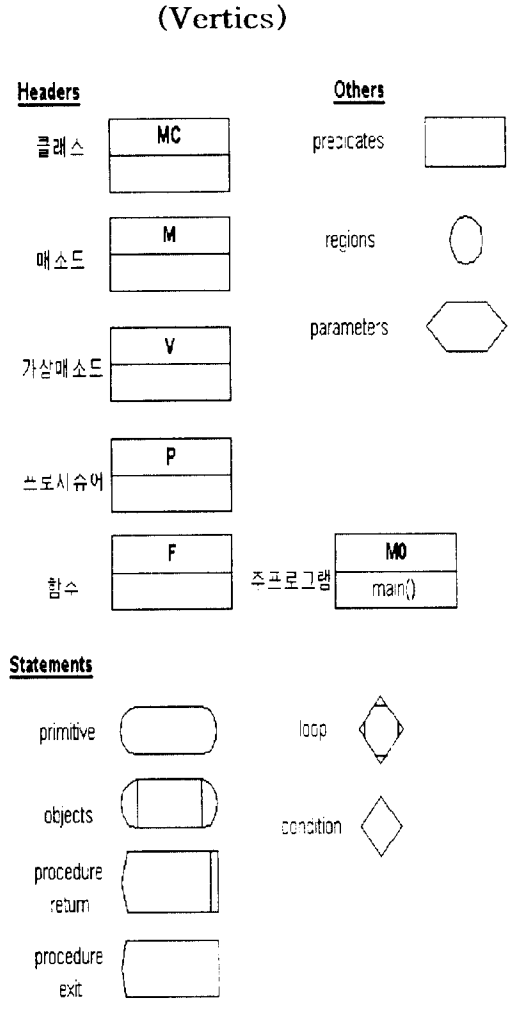
(Edges)

Control Edges

제어 종속 ------▶
데이타종속 ------▷
시간종속 -----◇○
호출 ——<
인스턴스 ——|<
다형성호출 ——|○
다형성선택 ——||○
연결 ——>

Membership Edges

상 속 ——▶▶
클래스멤버 ■——■
상속된메소드 ○——○



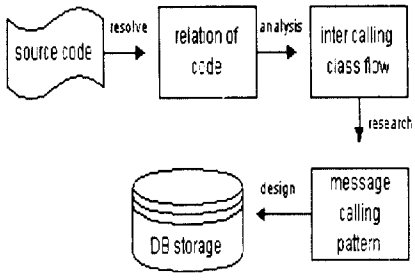
<표 1> 슬라이싱 제안 표기법

제안된 표기법은 기존의 객체지향 프로그램 슬라이싱에서 사용한 표기법을 명확하고 세분된 형태로 구성한 것으로서 하나이상의 의미를 내포했던 형태를 분리하였으며 이결과로 실시간 시스템에서 필요한 시간종속에 대한 표기를 추가함으로서 다음에서 구축하고자 하는 슬라이싱맵 구축시 좀더 구체적인 슬라이싱맵 형태가 형성될 것이다. 향후 이 표기법에 타임 스케쥴링에 대한 적절한 표기를 부가함으로서 슬라이싱맵 구축시 시간제약성도 포함시키고자 한다.

3. 슬라이싱맵 데이터베이스

3.1 슬라이싱맵 구성

슬라이싱맵 구성은 컴파일러 구성단계를 이용하여 구축될수있으며 특히 파스트리(parse tree)의 생성에 의해서 본 논문이 목적하는 슬라이싱맵 데이터베이스 구축이 가능한 것이다. 슬라이싱맵 구성의 단계를 전체 다섯 단계로 구성한다. 첫 단계는 원시 프로그램(source program)을 모듈별로 분해하는 것이며 이 작업에서는 메인 본체(main body), 클래스(class), 함수(function)별로 분해하는 과정이다. 두 번째 단계는 분해된 각 부분에 대한 분석단계로서 메인 본체의 메시지호출(message call), 클래스의 상속성(inheritance of class) 등을 분석하여 각 계층간의 관계를 구성한다. 세 번째 단계는 메시지호출에 의해서 호출된 클래스 자체의 흐름도를 구성한다. 네 번째 단계는 프로그램내에서 발생하는 모든메시지 호출에 대한 흐름의 패턴을 조합한다. 마지막 다섯째 단계는 슬라이싱맵 데이터베이스 구축(construction of database for slicing map)단계이다. 다음 <그림 4>은 슬라이싱맵구축의 다섯단계를 표현하는 것이다.



<그림 4> 슬라이싱맵 구축단계

3.2 슬라이싱맵 데이터베이스구축방법

3.2.1 소스 코드 분해

기존의 절차적 프로그램(procedural program)에서의 슬라이싱은 코드의 분해과정이 없이 작업을 진행하였지만 객체지향 프로그램의 슬라이싱의 메인 본체(main body), 클래스(class), 함수

(function)별로 코드를 일차적으로 분해한후 분해된 각 부분을 슬라이스 작업에 맞도록 변형하는 과정이 필요하다. 즉, 원시프로그램(source program)의 각 문장(statement)별로 라인부호를 부여하는 과정이다. 그리고, 변형된 각 모듈은 (2.2)에서 제안한 표기법을 사용하여 그래픽적으로 표현될 수가 있다. <표 2>는 C++원시프로그램이며 이를 분해한 후 라인 부호를 부여한 것을 <표 3>에 보여주고 있다.

```
#include <iostream.h>
const double PI = 3.14159265;
class circle{
    protected:
        double radius;
    public:
        circle()
        { radius = 0.0;}
        circle(double rad)
        { radius = rad; }
        double get_radius() const
        { return radius; }
        double get_area() const
        { return PI*radius*radius;}
    };
class cylinder:public circle{
    protected:
        double height;
    public:
        cylinder() : circle()
        { height = 0.0;}
        cylinder(double rad, double ht):
                                circle(rad)
        { height = ht; }
        double get_height() const
        { return height; }
        double get_height() const
        { return height*get_area(); }
    };
class hollow_cylinder:public cylinder{
    protected:
        double inner_radius;
    public:
        hollow_cylinder():cylinder()
        { inner_radius = 0.0; }
        hollow_cylinder(double outr,
            double ht, double_inr):cylinder(outr,
                                            ht)
            { inner_radius = inr; }
        double get_inner_radius() const
            { return inner_radius; }
        double get_volumn() const
            { return PI*(radius*radius-
              inner_radius*inner_radius)*height;}
    };

void main()
{
    circle cir(10.0);
    cylinder cyl(10.0, 15.0);
    hollow_cylinder hollow(10.0, 15.0, 5.0);
    cout<<"Radius of circle:"
            <<cir.get_radius()<<endl;
    cout<<"Radius of cylinder:"
            <<cyl.get_radius()<<endl;
    cout<<"Radius of hollow:"
            <<hollow.get_radius()<<endl;
    cout<<"Height of hollow:"
            <<hollow.get_height()<<endl;
    cout<<"Area of circle:"
            <<cir.get_area()<<endl;
    cout<<"Area of cylinder:"
            <<cyl.get_area()<<endl;
    cout<<"Volumn of cylinder:"
            <<cyl.get_volumn()<<endl;
    cout<<"Volumn of hollow:"
            <<hollow.get_radius()<<endl;
    cout<<"Height of cylinder:"
            <<cyl.get_radius()<<endl;
}
```

<표 2> 원시 프로그램

● 라인번호 부여 규칙

주 프로그램(main)은 M0으로 하고 클래스는 MC0, MC1, MC2, ..., 오브젝트는 OB0, OB1, OB2, ..., 함수는 F1, F2, F3, ... 로 표기하고 문장(statement)는 S1, S2, S3, ...으로 표기한다. 또한 문장중 제어문은 CO1, CO2, CO3, ...과 같이 표기하기로한다. 변수는 V1, V2, V3, ...으로 표기한다

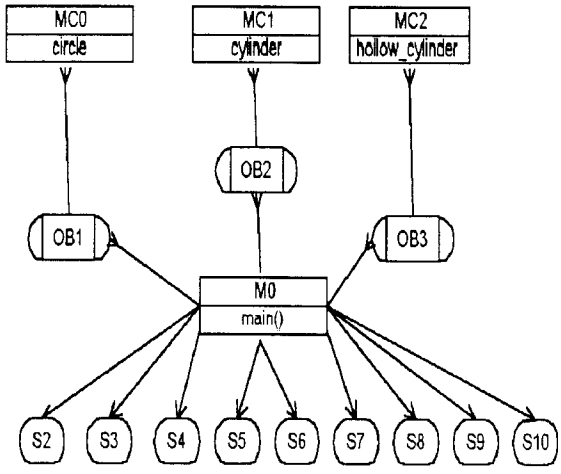
```
        #include <iostream.h>
V1 :        const double PI = 3.14159256;
MC0:
MC1:
MC2:
M0 :    void main()
        {
OB1 :    circle cir(10.0);
OB2 :    cylinder cyl(10.0, 15.0);
OB3 :    hollow_cylinder hollow(10.0, 15.0,
                                        5.0);
S2 :        cout<<"Radius of circle:"
                <<cir.get_radius()<<endl;
S3 :        cout<<"Radius of cylinder:"
                <<cyl.get_radius()<<endl;
S4 :        cout<<"Radius of hollow:"
              <<hollow.get_radius()<<endl;
S5 :        cout<<"Height of hollow:"
              <<hollow.get_height()<<endl;
S6 :        cout<<"Area of circle:"
                <<cir.get_area()<<endl;
S7 :        cout<<"Area of cylinder:"
                <<cyl.get_area()<<endl;
S8 :        cout<<"Volumn of cylinder:"
                <<cyl.get_volumn()<<endl;
S9 :        cout<<"Volumn of hollow:"
              <<hollow.get_radius()<<endl;
S10 :       cout<<"Height of cylinder:"
                <<cyl.get_radius()<<endl;
        }
MC0 : class circle{
        protected:
V1 :        double radius;
        public:
OB1 :       circle()
S1 :        { radius = 0.0;}
OB2 :       circle(double rad)
S2 :        { radius = rad; }
F1 :        double get_radius() const
S3 :        { return radius; }
F2 :        double get_area() const
S4 :        { return PI*radius*radius;}
        };
```

```
MC1:    class cylinder:public circle{
        protected:
V1 :        double height;
        public:
OB1 :       cylinder():circle()
S1 :        { height = 0.0;}
OB2 :       cylinder(double rad,
                    double ht:circle(rad)
S2 :        { height = ht; }
F1 :        double get_height() const
S3 :        { return height; }
F2 :        double get_height() const
S4 :        { return height*get_area(); }
        };
MC2:    class hollow_cylinder:public cylinder{
        protected:
V1 :        double inner_radius;
        public:
OB1 :       hollow_cylinder():cylinder()
S1 :        { inner_radius = 0.0; }
OB2 :       hollow_cylinder(double outr,
              double ht, double inr):
                        cylinder(outr, ht)
S2 :        { inner_radius = inr; }
F1 :        double get_inner_radius() const
S3 :        { return inner_radius; }
F2 :        double get_volumn() const
S4 :    { return PI*(radius*radius-
          inner_radius*inner_radius)*height;}
        };
```

<표 3> 분해된 원시코드

3.2.2 주프로그램과 클래스계층관계

이부분은 첫단계에서의 결과를 이용하여 메인프로그램과 클래스계층간의 관계를 표현하며 주프로그램의 각 문장과 클래스사이의 연관성, 즉 호출(call)을 그래픽적으로 표현하는 단계이다. 이 단계에서는 그래픽적인 표현에 의해서 전체 프로그램의 구조를 파악하는 단계이다. 표현에 이용되는 그래픽 요소들은 (2.2)에서 제안된 표기법을 이용하였다. 다음에 보여지는 <그림 5>는 <표 3>의 결과를 표현한 것이다.



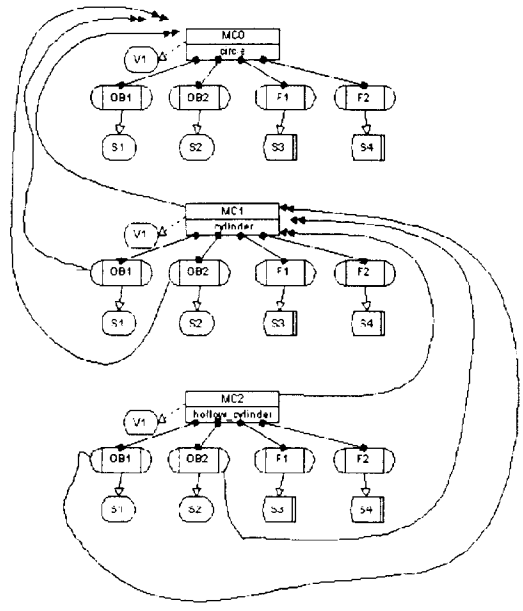
<그림 5> 객체클래스 종속 그래프(OCDG : Object Class Dependency Graph)

3.2.3 클래스내부 표현

분해된 각 모듈간의 전체적인 흐름을 더욱 세분화시키는 과정이며 또한 각 클래스 내부 구조에서의 흐름인 멤버함수간의 연관관계를 파악하고자하는 단계이다. 먼저 각 클래스 내의 흐름을 개별적으로 구하고 나서 클래스 내에서 다른 클래스의 상속성, 다형성 등에 연결성을 가지도록 하여 클래스사이의 연관성을 표현한다. 다음<그림 6>은 각 클래스의 내부 흐름을 표현하며 또한 클래스간의 상속성에 대한 표현이다.

3.2.4 실행패턴 테이블

앞의 세 단계에서 구해진 결과를 통합하고 각 메시지의 호출 흐름에 대한 여러가지 패턴을 구하는 단계이다. 그리고, 구해진 패턴을 데이터베이스화 하기위한 패턴테이블(pattern table)구축을 하는 단계이다. 이 과정에서 구해진 테이블이 슬라이싱맵 데이터베이스의 기초가 되는 것이다. 이것을 실행 히스토리(execution history)라고도 한다. 여기에서는 실행 패턴 테이블(execution pattern table)이라고 부를 것이다. 다음의 <표 4>는 모듈별 순서리스트와 실행 패턴 테이블의 형식과 예를 보여주고 있다.



<그림 6> 객체 클래스계층 서브그래프(CHS : Object Class Hierachy Subgraph)

(모듈별 순서리스트)

순서	모듈명	기호	상속순서	값
1	M0			*
2	M0	OB1		
3	M0	OB2		
4	M0	F1		
...	...	...	...	
14	MC0			*
15	MC0	OB1		
16	MC0	OB2		
...	...	...	...	
23	MC1		14	*
24	MC1	OB1	15	
25	MC1	OB2	16	
26	MC1	F1		
...	...	...	...	

▶ * : start

▶ 상속순서란의 번호는 모듈별 순서리스트의 연결순서 번호임

(실행패턴테이블)

순위	step1	step2	step3	step4	step5	...
1	1	2	16	20	*	...
2	3	25	29	*		...
3	4	34	38	*		...
4	5	17	21	*		...
5	6	23	17	21	*	...
6	7	32	17	21	*	...
...	...	...	...	...	...	...

▶ * :끝표시

▶ 숫자는 모듈별 순서리스트의 번호임

<표 4> 모듈별 순서리스트와
실행패턴 테이블

3.2.5. 슬라이싱맵 데이터베이스구성

이과정은 (3.2.4)에서 생성된 모듈별 순서리스트와 실행패턴 테이블을 데이터베이스화하는 단계로서 데이터베이스 구축방법은 모듈별 순서리스트는 원소의 저장순서가 논리적인 순서가 정해진형태가 아니며 단지 각 모듈에서의 문장(statement)순서를 표시하면 되므로 연결리스트로 구축하였다. 그리고 실행패턴 테이블의 데이터베이스 구축은 실행되는 순서가 논리적으로 정해진 형태가 되므로 실행순위에 따르는 스텝순서에 의한 배열(matrix)로 표현함으로서 실행패턴의 견고한 구축을 목적으로 한다. 다음의 <표 5>와 <그림 7>는 본 논문이 제시하는 모듈별 순서리스트와 실행패턴 테이블을 데이터베이스로 구축하는 알고리즘과 구성을 표현한 것이다.

```
struct modullist{
    int order;
    char modulname[10];
    char symbol[5];
    int inheritorder;
    int value[10];
};

struct node{
    struct modullist moddata;
    node* next;
};
```

```
class modulinklist{
    private:
        node* head;
        node* currentposition;
    public:
        modulinklist();
        ~modulinklist();
        ......
    };
```

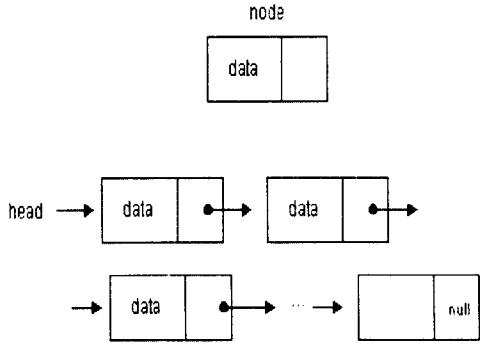
(모듈별 순서리스트)

```
struct execpattern{
    int ordernumber;
    int step[][10];
};

struct execnode{
    struct execpattern patdata;
    node* next, previous;
};

class execlist{
    private:
        node* head;
        node* currentposition;
    public:
        execlist();
        ~execlist();
        ......
};
```

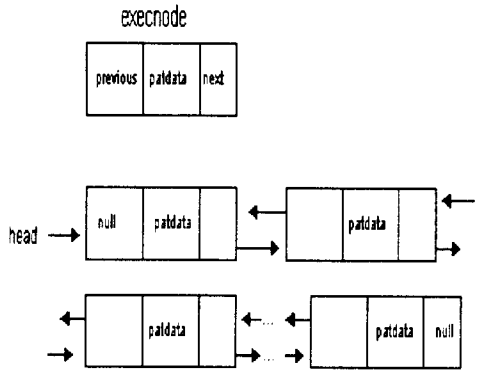
(실행패턴 테이블)

<표 5> 알고리즘

```
struct node{
    struct modullist moddata;
    node* next;
};
```



```
struct execnode{
    struct execpattern patdata;
    execnode* previous, next;
};
```



<그림 7> 데이터베이스 구성

4. 실시간 객체지향 시스템에서의 슬라이싱맵

실시간 시스템은 외부환경에서 전달된시간 제약 조건에 반응하여 동작되는 시스템이라 할수 있다. 또한 실시간 시스템은 전달되는 제약조건이 여러개 동시에 전달되는 처리인 다중처리 혹은 병행처리가 되는 경우가 많다. 그래서 실시간 시스템은 이러한 처리도 동시에 고려되어야만 한다. 본 논문이 제안하는 실시간 시스템에서 객체지향 방법론을 이용하여 시스템을 설계하고 슬라이싱맵을 데이터베이스화하여 시스템의 동작을 제어하고 시스템의 에러를 처리할 수 있을 것이다. 시스템의 외부적인 환경은 실세계의 반응이나 조건이므로 객체라할 수가 있다. 프로그램의 슬라이싱에서 기존의 표기들이 복합된 기능을 하나의 기호로 표기함으로서 명확한 표현이 이루어지지않은 것을 본 논문은 2.2절에서 세부적이고 단일기능을 표기할 수 있는 표기법을 제시한다. 다음의 예는 세척기의 동작을 보여준다. 이 프로그램은 실시간시스템 환경의 외부조건에 반응하는 것을 간단히 보여주며 실시간시스템은 종료조건이 주어지지않는 동안 무한히 반복하는 상태를 유지하는 경우가 많다. 이와같은 경우에 시스템콜과 인터럽트처리를 효율적으로 파악하기위해 슬라이싱은 유용하게 사용될수 가있다. <표 4>는 소스프로그램을 보여주며 <표 5>는 슬라이스 하기전의 소스프로그램을 분해한 것을 보여준다. <그림 8>은 분해된 소스프로그램을 슬라이싱한 객체클래스 종속 그래프를 보여준다. <표 5>는 프로그램의 모듈별 순서리스트와 실행패턴 테이블을 표현한다. 그리고, 모듈별순서리스트와 실행패턴 테이블의 데이터베이스 구성은 3.2.5절의 구성과 동일한 형태를 가진다.

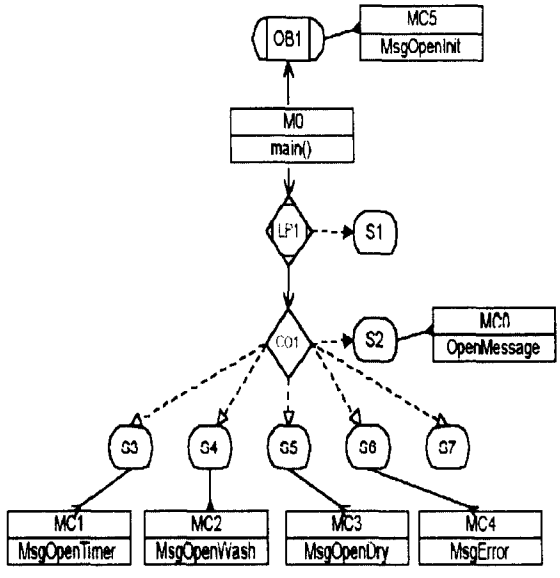
```
#include <iostream.h>
#include "timer.h"

class OpenMessage()
{
...
}
class MsgOpenTimer()
{
....
}
class MsgOpenWash()
{
...
}
class MsgOpenDry()
{
...
}

class MsgError()
{
...
}


class MsgOpenInit()
{
...
}

void main()
{
  int keyinit = 0;
  OpenMessage* msg;
```

```
    MsgOpenInit();

     while(1)
      {
       switch(msg.task){
         case MsgOpenTimer:
          MsgOpenTimer::opentimer(msg.
                                  task);break;
         case MsgOpenWash:
           MsgOpenWash::openwash(msg.
                                  task);break;
         case MsgOpenDry:
         MsgOpenDry::opendry(msg.task);
                                  break;
         case MsgError:
           MsgError::errorcheck();break;
         default: break;
         }
     }
}
```

<표 6> 원시코드

```
    #include <iostream.h>
    #include "timer.h"
    MC0:
    MC1:
    MC2:
    MC3:
    MC4:
    MC5:
    M0:  void main()
         {
    V1:      int keyinit = 0;
    V2:      OpenMessage* msg;
    OB1:     MsgOpenInit();

LP1,S1: while(1)
          {
CO1,S2: switch(msg.task){
    S3:      case MsgOpenTimer:
             MsgOpenTimer::opentimer(msg.
                                  task);break;
    S4:      case MsgOpenWash:
              MsgOpenWash::openwash(msg.
                                  task);break;
    S5:      case MsgOpenDry:
             MsgOpenDry::opendry(msg.task);
                                  break;
    S6:      case MsgError:
               MsgError::errorcheck();break;
    S7:      default: break;
             }
          }
     }
```

<표 7>분해된 원시코드



<그림 8> 객체클래스 종속 그래프(OCDG)

(실행패턴 테이블)

순위	step1	step2	step3	step4	step5	step6	끝
1	1	2	3	4	*		
2	5	6	*				
3	7	8	14				5
4	9	20					7
5	10	30					7
6	11	40					7
7	12	50					7

▶ * :끝표시
▶ 숫자는 모듈별 순서리스트의 번호임
▶ 끝 표시는 조건 반복에대한 것을 표시한다

(모듈별 순서리스트)

순서	모듈명	기호	상속순서	값
1	M0			*
2	M0	V1		
3	M0	V2		
4	M0	OB1		
5	M0	LP1		
6	M0	S1		
7	M0	CO1		
8	M0	S2	14	
9	M0	S3	20	
10	M0	S4	30	
11	M0	S5	40	
12	M0	S6	50	
13	M0	S7		
14	MC0			*
....	....			
20	MC1			*
....	....			
30	MC2			*
....	....			
40	MC3			*
....	....			
50	MC4			*
....	....			
60	MC5			*
....	.....			

▶ * :start

▶ 상속순서란의 번호는 연결순서번호임

<표 8> 모듈별 순서리스트와
실행패턴테이블

5. 결론

본 논문에서 제시한 실시간 시스템에서 객체지향 설계 방법을 이용한 슬라이싱기법의 사용은 실시간 소프트웨어의 골격을 이루는 클래스와 클래스를 호출하는 메시지의 흐름을 OCDG와 OCHS로서 명확히 구별할수 있고 본 논문이 제시한 표기법을 사용함으로서 클래스의 상속관계와 모듈간의 종속관계, 클래스 맴버 구별을 할 수가 있다. 그리고, 클래스 오브젝트의 흐름과 메시지의 흐름을 본 논문이 제시한 모듈별 순서리스트를 이용한 실행패턴 테이블을 이용하여 파악할 수가 있다. 제시된 순서리스트와 실행패턴 테이블을 데이터베이스로 구축하여 소스코드 레벨에서의 검증을 할 수가 있다. 구축된 데이터베이스에 시스템의 외부적인 환경변화를 실시간으로 파악하여 시스템 전체의 흐름을 중단시키는 치명적인 시스템 결함을 초래하지 않토록 하는 예외처리(exception process)를 추가함으로서 치명적인 오류를 방지할 수가 있다. 실시간 시스템은 다양한 형태로 나타날수가 있으나 이것은 하드웨어적인 형태인 경우가 대부분이며 소프트웨어적인 면에서 보면 실시간 시스템의 특성인 반응성, 병행성, 분산성, 통신 동기화 등의 처리를 원할히 하고 소프트웨어의 파괴시에는 데이터베이스에 구축된 모듈에 따라 복구가능하게 된다. 결론적으로 본 논문이 제시한 내용을 이용함으로서 실시간 시스템의 전체적인 신뢰성을 향상시킬수가 있다.

6. 향후연구

본 논문의 연구에서 슬라이싱 개념 적용이 실시간 시스템분야에도 적용이 가능하다고 생각되며 특히, 객체지향 방법과 슬라이싱 방법의 결합으로 인하여 실시간시스템에서 개발의 신뢰성을 저하시킨 요인으로 시스템의 병행성과 분산성 및 하드웨어부품의 객체구조로의 인식을 보다 명확히 가질수가 있다. 또한, 객체지향적인 개념으로 시스템을 설계하는 잇점인 객체간의 메시지통신을 명확히 확인 할 수가있다. 향후 본 논문이 제시한 내용을 기초로 하여 소스코드 레벨(source code level)에서의 슬라이싱맵의 활용을 오브젝트 코드(object code)레벨과 실행코드레벨(execution code level)까지 확대시켜 프로그램 실행중에도 시스템의 전반적인 사항을 추적할 수 있는 추적자 기능을 포함시켜야하며 또한, 슬라이싱맵에 실시간 시스템의 타임 스케쥴러를 포함하여 동적 실시간 객체지향 슬라이싱맵을 구축하여 외부환경의 변화에 동적으로 프로그램의 우선순위를 변경할수있는 멀티미디어 통신시스템에서의 슬라이싱맵 활용방법을 연구하여야 할 것이다.

참고문헌

[1] Anand Krishnaswamy, Program Slicing:An Application of Object-oriented Program Dependency Graphs, Dept. of Computer Science, Celmson University, 1994

[2] Bran Selic,Real-Time Object-Oriented Modeling, John Wiley&Sons, New York, 1994

[3] Bjarne Stroustrup, The C++ Programming Language, Addison-Welsley, Massachusetts, 1991

[4] Grady Booch,Software Component with ADA, Benjamin/Cummings, Menlo Park, 1987

[5] Hiralal Agrawal and J.R. Horgan, Dynamic Program Slicing, Proc. ACM SIGPLAN '90 Conf. Programming Language Design and Implementation, pp. 246-256, 1990

[6] Horowitz S;Reps T and Binkley D., Interprocedural slicing using dependence graphs, ACM Transaction on Programming Language and Systems, 1990

[7] James Rumbaugh, Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, 1991

[8] Loren D. Larsen and Mary Jean Harrold, Slicing Object-oriented Software, Proceeding of the 18th International Conference on Software Engineering, pp. 495-505, Berlin, March 1996

[9] Mary Jean Harrold, Brian Malloy and Gregg Rothermel, Efficient construction of program dependence graphs, ACM International Symposium on Software Testing and Analysis, 1993

[10] Ottenstein K. J and Ottenstein L. M., The program dependence graph in a software development environment, Techinical Report pp. 94-105, Clemson University, 1994

[11] Sally Shlaer, Object Lifecycles : Modeling the World in States, North Holland, New York, 1990