절차지향 프로그램으로부터 객체의 지속성을 결정하기 위한 방법론

(A Methodology to Determine Persistence of Objects from Procedural Program)

최 정 란 † 이 문 근 ††
(Jeong-Ran Choi) (Moon-Kun Lee)

요 약 본 논문은 절차지향 소프트웨어를 객체지향 소프트웨어로 재공학하는 과정에서 객체들의 안전한 지속성에 대한 결정 방법을 제안한다. 본 논문에서는 지속성 결정을 위해 다섯 단계의 과정을 제시한다: 정적 정보, 투영, 반영, 인스턴스, 정제 단계. 각 단계를 통해 객체의 정확한 생성과 소멸 시점을 추출하고, 정제 과정을 거침으로써 객체의 메시지 전달과 생성/소멸 과정에서 안전성과 일관성을 유지할 수 있도록 한다.
키워드 : 객체 생성, 소멸, 재공학

Abstract This paper presents a methodology to determine *safe persistence* of objects from C code during reengineering process. The methodology consists of five steps: *the static information methodology, reflection, instantiation, and the refinement.* The steps assist to a reengineer to decide appropriate construction and destruction points of an object during its life cycle. Further the steps guarantee safe and consistent interactions among objects.
Key words : Object construction, destruction, reengineering

1. 서 론

최근의 파라다임은 알고리즘의 분해, 구조적 설계와 절차적 구현에서 객체지향적 설계와 프로그래밍으로 이동하고 있다. 이러한 발전 변화에 따라 기존의 구조가 가지고 있는 오랜 프로그램을 재구성하기 위한 필요성이 증가하고 있다[1]. 또한 소프트웨어를 재공학할 때 유지·보수 및 재사용 측면에서 기존의 방식들보다 유리한 객체지향 파라다임을 적용하여 기존의 시스템으로 재개발한다면, 소프트웨어 생산성을 향상시킬 수 있고, 소프트웨어의 유지·보수 비용을 절감할 수 있으며, 시스템에 새로운 요구를 수용할 수 있게 되는 등 많은 장점을 가지게 된다[2].

이러한 재공학 방법은 <그림 1>과 같은 절차로 진행되는 것이 바람직하다[3]. 객체 추출 단계에서는 전역변수, 사용자 자료형, 함수와 파라미터를 기반으로 관련성 정도를 기준으로 클러스터링한다. 클래스 추출 단계에서는 클러스터링된 객체 후보들의 공통적인 특성을 추출하여 클래스를 추출한다. 클래스 추출 단계의 결과는 클래스들간에 대등한 평면 관계를 이루고 있다. 상속 관계 추출 단계는 전 단계에서 추출된 평면화된 클래스들의 공통적 특성을 추출하여 *part-of* 관계나 *is-a* 관계를 추출하여 계층 구조를 만든다. 지속성 단계에서는 절차 지향 프로그램에는 존재하지 않는 객체 지향 특성들과 동적인 부분을 추가한다. 이러한 것들은 클래스의 생성자, 소멸자, 동적 메모리 할당/해제 등이 있다. 동일성 검증 단계에서는 생성된 객체지향 프로그램이 정상적으로 작동하는지와 원래의 절차지향 프로그램과 변환된 객체지향 프로그램이 의미상으로 동등한지에 대한 검사를 한다. 코드 생성 단계에서는 전단계에서 생성된 뼈대 코드를 기반으로 실행 가능한 객체 지향 프로그램을 생성한다.

본 논문에서는 <그림 1>의 절차 중 지속성 결정 단계에 대하여 기술한다. 사용한 지속성 결정 방법은 다섯 단

· 본 연구는 한국과학재단 특정기초연구(1999-2-303-003-3) 지원으로 수행되었음.
† 비 회 원 : 전북대학교 전산통계학과
jlchai@cs.chonbuk.ac.kr
†† 정 회 원 : 전북대학교 전자정보공학부 교수
mklee@cs.chonbuk.ac.kr
논문접수 : 2001년 1월 27일
심사완료 : 2002년 1월 10일

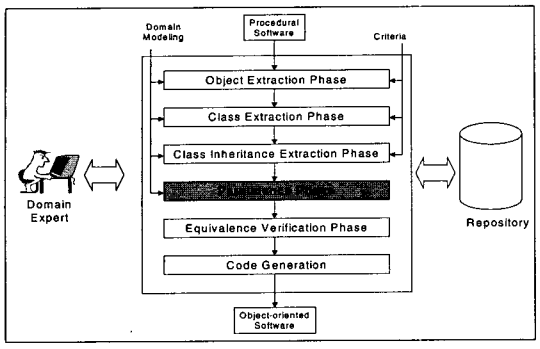


그림 1 절차중심 SW의 객체중심 SW로의 재공학 절차 흐름도

계에 기초를 두고 있다: 1) 정적정보(*static information*) 추출 단계는 프로그램의 정적정보를 추출하고, 2)투영(*projection*) 단계는 정적정보 결과를 상속성 추출 결과 생성된 클래스들과의 투영을 통하여 서로 관련성이 큰 함수, 자료형, 변수를 그룹화하며, 3)반영(*refleciton*) 단계는 2단계의 투영 결과 생성된 그룹을 캡슐화하여 객체로 표현, 객체의 시작/종료 시점을 추출하고, 4) 인스턴스(*instantiation*) 단계는 반영 결과를 기반으로 객체의 생성과 소멸 시점을 추출하고, 5)정제(*refinement*) 단계에서는 인스턴스 과정을 통하여 추출된 생성/소멸 시점에 대해 안전성과 일관성을 유지하기 위해 정제하는 과정이다.

프로그래머는 알맞은 속성(변수)과 오퍼레이션(메소드), 그리고 이들간의 관계성을 갖은 적절한 객체(클래스)를 미리 결정해야 하기 때문에 재사용 가능한 객체지향 소프트웨어를 설계하는 것은 어렵다[4]. 이에 본 논문에서는 소스코드를 기반으로 하여 정확한 시점에서 객체의 생성과 소멸 시점을 결정하는 방법론을 제안한다.

객체지향 프로그래밍에서 객체를 생성하는 시점을 결정하는 방법은 크게 두 가지가 있다. 그 첫 번째는 모든 객체가 초기에 생성되어 프로그램이 종료 시 소멸하는 방법과 다른 하나는 필요시 생성되었다가 더 이상 쓰이는 곳이 없을 때 소멸되는 방법이다. 초기 객체 생성은 모든 필요한 객체가 이미 존재하고 있다는 점에서 안전성이 보장되지만 불필요한 메모리 낭비로 인하여 과부하를 초래하게 된다. 두 번째 방법은 객체가 필요할 때 동적으로 할당되었다가 소멸되므로 써, 과부하는 줄일 수 있지만 정확한 시점에서 객체의 생성/소멸 시기를 유지함으로써 안전성과 일관성을 보장해 주어야 한다. 본 논문에서는 두 번째 방법을 사용하여 상속성 추출결과 생성된 클래스들이 실제 시스템에서 효율적인 객체 생성과 소멸 과정을 거침으로써 시스템의 부하를 최소화 할 수 있도록 하였다. 객체들의 효율적 생성과 소멸 시점을 파악함으로써 생성되지 않은 객체간의 메시지 교환 문제를 없앨 수 있어서 전체 시스템의 안전성, 일관성, 정확성을 높일 수 있다.

본 논문은 기존의 재공학 연구와 비교하여 다음과 같은 장점을 가진다.

1) 객체의 생성/소멸 시점을 결정하기 위해 소스 코드를 기반으로 생성된 클래스들과 분석된 절차지향 프로그램의 정적정보를 사용함으로써 동적정보를 사용시 발생할 수 있는 복잡성을 간소화한다.

2) 본 논문에서 제안한 방법론의 마지막 단계에 객체의 생성/소멸을 제어하는 정제 과정을 추가하여 객체의 안전하고 일관성 있는 생성과 소멸시점을 결정한다.

3) 본 논문에서 제안한 방법론을 통해 실험 및 분석 결과 생성된 객체의 수를 볼 때 프로그램의 크기와는 관계가 비교적 독립적이다.

본 논문의 구성은 다음과 같다. 2절에서는 관련 연구를 기술하며, 3절에서는 본 연구의 기본이 되는 그래프 모델에 대해 정의하고, 4절에서는 지속성을 결정하기 위한 방법론을 다섯 단계를 기술하며, 5절에서는 본 논문의 방법론에 대한 실험 및 분석 결과를 기술하며, 6절에서는 결론 및 향후 연구에 대해 기술하였다.

2. 관련 연구

2.1 UML(Unified Modeling Language)

시각적 모델링은 실세계 시스템의 프로세스와 그래픽한 표현과의 사상관계를 나타낸다. 모델링에서 개발자는 복잡한 시스템을 효율적으로 개발하기 위해 청사진을 추상화해야하고 정확한 표기법에 따라 모델을 설계하고 모델이 시스템의 요구사항을 만족하는지 검증하여 결국 세부사항을 추가하여 모델에 맞추어 구현하게 된다[5]. 이러한 대표적 시각적 모델링이 UML이다.

UML은 소프트웨어 시스템을 명세하고, 시각화하며, 구축하고, 문서화하기 위해 만들어진 시각적 모델링 언어이다. 이러한 모델링 언어는 순공학 과정에서 객체지향 시스템을 구축하기에 용이하다[5].

UML은 정적 모델(*static model*), 동적 모델(*behavioral model*), 사용 모델(*usage model*), 구조적 모델(*architectural model*)로 구분되어 있다. 동적 모델 중 객체간의 상호작용을 나타내는 인터액션 다이어그램(*interaction diagram*)이 있다. 인터액션 다이어그램은 객체간의 시간을 중심으로 메시지 교환을 나타내는 순

서도(*sequence diagram*), 객체가 서로 어떠한 연관을 가지고 있는지 보여주는 협력도(*collaboration diagram*)가 있다. 순서도는 객체가 그들의 생명주기 동안의 메시지 교환을 보여준다. 순서도에서 객체 생명주기는 특정 시간에 객체가 어떠한 역할을 수행하고 있음을 나타낸다. 즉, UML의 순서도는 본 논문에서 제안한 시간 도표 모델과 행위적 측면에서 상당한 유사점을 가지고 있다. 그러나 UML의 순서도는 순공학적인 측면에서 객체지향 시스템을 구축하기 위한 모델이라면 본 논문에서 제시하고 있는 모델은 코드를 기반으로 하여 순수한 절차지향 프로그램에서 객체지향 프로그램으로 변환하는 과정을 위한 과정이라 할 수 있다.

2.2 SDL(Specification and Description Language)

*SDL*은 시스템의 동작을 계층적으로 표현하고 설계하기 위한 명세 언어이다. *SDL*의 표현 방식으로는 흐름도와 유사하게 그래픽 다이어그램으로 전체 시스템을 표현하는 *SDL/GR*(*Graphical Representation*)과 프로그래밍 언어와 유사한 *SDL/PR*(*textual Phrase Representation*)이 있다. 특히, 본 논문에서 제안할 시간 도표 모델과 유사한 *SDL/GR*은 시스템의 구조를 분명히 보여주고 제어의 흐름과 자료의 흐름을 시각화시키는 도구로서, 여기에서 제공하는 순서도(*Sequence Chart*)는 시스템내의 신호 흐름을 시간의 개념을 갖는 그래픽 형태로 표현하여 사용자의 이해를 쉽게 한다[6,7].

2.3 Automated Method-Extraction Refactoring by Using Block-Based Slicing

K. Maruyama[4]는 프로그램 슬라이싱(*program slicing*)을 사용하여 객체지향 프로그램의 메소드를 자동으로 *refactoring*하는 방법론을 제안하였다. 식별할 수 있는 동작의 변화 없이 메소드를 재구축하기 위해 전체 프로그램으로부터 코드의 조각을 추출하는 것이 아니라 프로그램의 연속적인 기본-블록(*basic-block*)으로 구성된 영역(*region*)으로부터 추출하는 블록-기반 슬라이싱(*block-based slicing*)을 사용하였다. *Refactoring* 도구를 사용하여 추출된 코드를 포함하는 새로운 메소드를 구축하는 방법, 남은 코드와 실행 시 반드시 필요한 문장들을 포함하는 소스 코드의 메소드를 재구성(*re-form*)하는 방법을 구현하였다. *K. Maruyama*가 제안한 방법론은 다음과 같은 장점을 가진다: (1)프로그래머가 존재하는 메소드의 코드 내에서 유일하게 특정한 변수와 관련이 있는 추출된 메소드 후보들을 얻을 수 있다. 또한 그 후보들 사이에서 적당한 후보를 선택한다; (2) 방법론은 항상 메소드들의 식별할 수 있는 동작을 변화없이 유지할 수 있다. 결론적으로 본 방법론은 프로그래머가 프로세스가 에러를 가지고 있고 시간 소모적인 수동의 *refactoring*을 피하도록 할 수 있다.

본 논문에서 제안한 블록-기반 슬라이싱을 정의하기 위해 제어 흐름 그래프(*control flow graph, CFG*)를 사용하였다. 또한 클래스들의 코드를 표현하기 위해 클래스 의존 그래프(*class dependence graph, C1DG*)[8]를 사용하였다. *C1DG*안에서 각각의 메소드는 프로그램 의존 그래프(*program dependence graph, PDG*)로 나타나며, PDG는 여러 노드와 에지의 집합으로 이루어져 있다. 노드는 할당(*assignment*)과 조건 술어(*condition predicate*)가 있는 문장을 표시한다. 에지는 제어와 데이타의 의존성을 나타낸다. 제어 의존 에지는 문장의 실행 시 의존하는 제어 조건을 나타내고 있고, 데이타 의존 에지는 루프가 존재하거나 또는 루프에 독립적인 문장사이의 데이타의 흐름을 나타내고 있다. *C1DG*는 또한 인스턴스 변수와 메소스 호출에 대한 파라미터의 입력과 출력에 대한 특별한 노드를 가지고 있으며 이들간의 관계를 나타내는 *call* 에지(호출 노드와 호출된 노드)와 *member* 에지(클래스와 메소드)가 있다.

K. Maruyama는 *refactoring*을 원하는 메소드에 해당하는 영역을 *CFG*를 통해 *basic block*들로 표현하고 *CFG*를 기반으로 한 *C1DG*는 기본적으로 클래스 멤버간의 메소드 호출과 파라미터 바인딩, 데이타의 의존성, 제어 의존성 등을 그래프를 통해 보이고 있으며 메소드 내부의 변수들의 상호작용(즉, *read/write*)은 테이블로써 보이고 있다. 또한 클래스들간의 메시지 교환을 독립적으로 표현하고 있다. 이러한 *C1DG*를 통해 특정 노드들을 블록화하고 *refactoring* 가능한 곳을 추출하게 된다.

*C1DG*는 본 논문에서 제안한 시간 도표 모델과 메소드, 인스턴스 변수, 또는 클래스들간의 메시지 교환이라는 측면에서는 의미적으로는 동일한 듯 보이나 시간 도표 모델은 메시지 교환의 일련의 행위들을 시간을 기반으로 하여 하나의 도표로 표현하여 효율성을 더하였다. 또한 *C1DG* 객체지향 프로그램의 재사용성을 위한 방법론이라면 본 논문에서 제시하고 있는 모델은 코드를 기반으로 하여 순수한 절차지향 프로그램에서 객체지향 프로그램으로 변환하기 위한 과정이라 할 수 있다.

2.4 Reusable Component Interconnection Patterns for Distributed Software Architectures

H.Gomaa[9]는 클라이언트/서버 시스템에서 재사용 가능한 컴포넌트의 상호연결도의 설계를 연구하였다. 특히 *H.Gomaa*는 클라이언트와 서버 컴포넌트가 서로 통신하는 방법을 정의하고 캡슐화하는 컴포넌트 상호연결 패턴(*component interconnection pattern*) 설계를 기술

하였다. 이러한 패턴은 새로운 분산 응용 설계자가 적당한 컴포넌트 상호작용 패턴을 선택하고 재사용할 수 있도록 한다. 또한 이러한 패턴을 포함하는 분산 설계의 수행결과를 분석 가능하게 한다.

컴포넌트 상호연결 패턴은 *UML*을 사용하여 나타내며 크게 정적 모델링과 동적 모델링으로 나뉜다. 정적 모델링은 여러 컴포넌트들과 이들을 연결하는 *connector*들을 구별하기 위해 *UML*의 클래스 다이어그램의 *stereotype*을 사용한다. 컴포넌트를 연결하는 *connector*는 하나의 클래스로 표현되며 상호작용의 세부사항은 나타내지 않는다. 컴포넌트 상호연결 패턴의 다른 하나는 동적 모델링이다. 동적 모델링은 *UML*의 협력도를 사용하여 컴포넌트와 *connector* 객체사이의 동적인 상호작용을 설명한다. 메시지는 협력도에 번호를 메겨 발생의 순서에 따라 보여진다. 이러한 동적 모델링은 동기적 통신과 비동기적 통신으로 나뉘어 달리 설명된다. 동적 모델의 개발은 컴포넌트와 *connector*사이의 상호작용의 순서를 자세히 묘사한다.

*H.Gomaa*는 컴포넌트의 재사용성을 위해 상호연결을 위한 특정 패턴을 제안하였고 그 안에서 *UML*의 협력도를 사용하였다. 본 논문에서 제시한 시간 도표 모델과 비교하여 메시지 상호작용을 시간의 순서에 따라 보여주었다는 측면에서 유사하지만 개발 도메인 상에서 차이를 보인다.

3. FTV 그래프[10]

FTV(*Function-Type-Variable*) 그래프(G^{FTV})는 절차 중심 소프트웨어로부터 객체와 클래스를 추출하기 위하여, 소프트웨어를 시각적으로 표현한 그래프인 VPR (*Visual Program Representation*) 그래프(G^{VPR})[10]로부터 함수(N_{spec}), 자료형(N_{type})과 전역변수(N_{var})에 관한 정보만을 추출한 것이다. FTV 그래프의 노드는 함수, 자료형, 전역 변수이고, 이들 노들의 관계성, 에지(*edge*)가 존재하고 이러한 에지에는 방향성이 있으며 호출/사용 횟수에 따른 가중치를 가지고 있다. 관계성은 함수와 함수의 관계를 나타내는 에지(E_{call}), 자료형의 관계를 나타내는 에지(E_{type}), 변수의 사용 관계를 나타내는 에지(E_{memory})에 관한 정보를 추출한다.

그래프 G^{FTV}의 형태는 다음과 같이 4-쌍의 튜플로 정의된다.

$$G^{FTV} = \langle id,\ name,\ ,N_{FTV},\ E_{FTV}\rangle$$

여기에서, id는 G^{FTV}의 식별자를, name은 G^{FTV}의 이름을, N_{FTV}는 G^{FTV}의 노드를 E_{FTV}는 G^{FTV}의 에지를 의미한다. N_{FTV}는 N_{spec}, N_{type}, N_{var}(단, 지역 변수는 제외)로 구성되며 E_{FTV}는 N_{FTV}들간의 관계성을 나타내는 E_{call}, E_{type}, E_{memory}로 구성된다

4. Persistence Phase

지속성을 결정하기 위해 본 논문에서는 5 단계로 나누어 기술한다. 각 과정을 살펴보면 다음과 같다: 1) 정적정보 단계는 프로그램의 정적정보를 추출하고, 2)투영 단계는 정적정보 추출 결과를 상속성 추출 결과 생성된 클래스들과의 투영을 통하여 서로 관련성이 큰 함수, 변수 자료형을 그룹화하며, 3)반영 단계는 2단계의 투영 결과 생성된 그룹을 함축하여 객체로 표현, 객체의 시작/종료 시점을 추출하고, 4) 인스턴스 단계는 반영 결과를 기반으로 객체의 생성과 소멸 시점을 추출하고, 5)정제 단계에서는 인스턴스 단계의 과정을 통하여 추출된 생성/소멸 시점에 대해 안전성과 일관성을 유지하기 위해 정제하는 과정이다.

본 논문에서는 지속성 결정의 5단계를 DFS(*Depth First Search*) 프로그램 예제를 통하여 설명한다. 예제는 가고자 하는 장소를 입력하면 재귀적으로 가장 최단거리의 경로를 찾아 주는 프로그램이다. <그림 2>는 DFS를 사용하여 추출된 FTV 그래프이다.

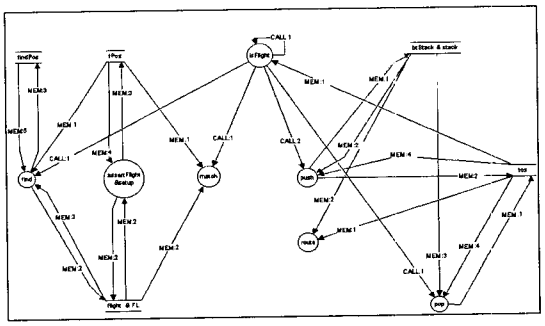


그림 2 DFS 프로그램에 대한 G^{FTV}

4.1 정적정보 추출 단계(Static Information Extraction Step)

지속성을 결정하는 5단계 중 첫 번째 단계인 정적정보 추출은 절차지향 프로그램에서 함수, 자료형, 변수들의 메시지 교환을 상대적 시간에 기반하여 프로그램의 정적정보를 추출하는 과정이다. 결과적으로, 생성된 그래프인 *FTV-time* 그래프(G^{time})는 객체 추출과정에서 사용되었던 *FTV* 그래프(G^{FTV})를 확장한 그래프이다. G^{FTV}에서 노드로 표현되었던 함수, 변수, 자료형이

G^{time}에서는 노드간의 상호작용이 시간의 변화에 따라 여러 노드와 에지로 나타나게 되어 이들의 주기를 가시적으로 나타내고 있다. 그래프 G^{time}의 형태는 다음과 같이 3-쌍의 튜플로 정의된다:

$$G^{time} = \langle N^{time}, E^{time}, S^{time} \rangle$$

여기에서, N^{time}는 G^{time}의 노드, E^{time}은 G^{time}의 에지, S^{time}은 G^{time}의 에지의 순서를 나타낸다.

4.1.1 노드

FTV-time 그래프에서 발생하는 노드는 3가지로 그 종류와 의미는 다음과 같다.

표 1 노드의 종류와 의미

종 류	의 미
$f^k_{i,j}$	함수
$t_{i,j}$	자료형
$v_{i,j}$	변수

각 노드의 인자의 의미를 살펴보면 i는 함수, 자료형, 변수를 식별하는 노드 번호이다. j는 함수, 자료형 변수들 간의 메시지 교환 시점으로 i번째 노드의 메시지 교환을 발생순서에 따라 나타낸다. k는 함수의 인스턴스 번호로 특정 함수의 호출 횟수를 나타낸다. 이를 수식으로 표현하면 다음과 같다:

$$N^{time} = N^{time}_{func} \cup N^{time}_{type} \cup N^{time}_{var}$$

N^{time}의 세부적 노드 N^{time}_{func}, N^{time}_{type}, N^{time}_{var}를 수식으로 표현하면 다음과 같다:

$$N^{time}_{func} = \{ f^k_{i,j} \mid 1 \le i \le \#func,\ 1 \le j \le \#ip,\ 1 \le k \le \#iof \}$$
$$N^{time}_{type} = \{ t_{i,j} \mid 1 \le i \le \#type,\ 1 \le j \le \#ip \}$$
$$N^{time}_{var} = \{ v_{i,j} \mid 1 \le i \le \#var,\ 1 \le j \le \#ip \}$$

여기에서, *#func*은 함수의 개수이고, *#ip*는 각 노드의 메시지 교환의 개수이며, *#iof*는 함수의 호출 횟수를 나타낸다. *#type*은 자료형의 개수이고, *#var*은 변수의 개수이다. $f^k_{i,j}$의 k는 함수의 인스턴스의 수를 말하며, 하나의 함수가 여러 번 호출될 수 있음을 의미한다. 예를 들어 *DFS*에서 다음과 같이 *push* 함수는 두 번 호출되어 사용됨을 볼 수 있다. 함수가 처음 호출($push^{1}_{4,1}$)되고, 반환($push^{1}_{4,5}$)되었다가, 프로그램이 진행 중 다시 호출($push^{2}_{4,1}$)이 발생되어, 반환($push^{2}_{4,5}$)됨을 볼 수 있다.

$$\begin{pmatrix} push^1_{4,1} & push^2_{4,1} \\ \vdots & \vdots \\ push^1_{4,5} & push^2_{4,5} \end{pmatrix}$$

1.4.2 관계성

G^{time}에서 함수, 자료형, 변수들간에 발생할 수 있는 관계성은 9가지이다. 그중 실제 프로그램을 실행 시 발생할 수 있는 관계성은 5가지이다. 이들간의 관계성은 <그림 3>과 같다.

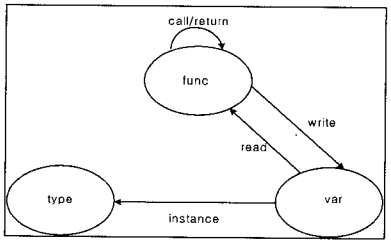


그림 3 함수, 변수, 자료형의 관계성

표 2 관계의 종류

→	F	T	V
F	$\langle f^k_{i,1}, f^n_{l,1} \rangle$ 또는 $\langle f^k_{i,y}, f^n_{l,y} \rangle$	×	$\langle f^k_{i,j}, v_{l,m} \rangle$
T	×	×	×
V	$\langle v_{l,m}, f^k_{i,j} \rangle$	$\langle v_{i,j}, t_{k,l} \rangle$	×

에지의 종류는 그 사용한 곳에 따라 종류와 의미가 다르고 방향성이 존재한다. 함수, 변수, 자료형의 주기를 나타내는 수직방향의 3개 에지와 이들간의 메시지 교환을 보여주는 수평방향의 5개의 에지가 있다. <표 1>은 에지의 종류와 의미, 에지의 방향을 설명하고 있다. *1*은 노드의 시작, *y*는 노드의 끝을 나타낸다.

표 3 에지의 종류와 의미

종 류	기 호	의 미	방향
$\langle f^k_{i,j}, f^k_{i,l} \rangle$	e_{func}	함수와 함수를 연결	수직
$\langle t_{i,j}, t_{i,k} \rangle$	e_{type}	자료형과 자료형을 연결	수직
$\langle v_{i,j}, v_{i,k} \rangle$	e_{var}	변수와 변수를 연결	수직
$e_{func} \vee e_{type} \vee e_{var}$	$e_{persistence}$	위 3가지를 표현	수직
$\langle f^k_{i,1}, f^n_{l,1} \rangle$	e_{call}	함수 호출시작(*call*)	수평
$\langle f^k_{i,y}, f^n_{l,y} \rangle$	e_{return}	함수 호출 종료(*return*)	수평
$\langle v_{l,m}, f^k_{i,j} \rangle$	e_{read}	함수에서 변수를 판독(*read*)	수평
$\langle f^k_{i,j}, v_{l,m} \rangle$	e_{write}	함수에서 변수를 기록(*write*)	수평
$\langle v_{i,j}, t_{k,l} \rangle$	e_{inst}	자료형 변수(*instance*)	수평

$e_{persistence}$는 함수, 자료형 변수의 주기를 모두 표현한 것으로 나타나며 그 방향은 수직이다.

$E_{persistence}=\{e_{persistence} \mid e_{persistence}\in(e_{func}\vee e_{type}\vee e_{var})\}$

e_{call}, e_{return}, e_{read}, e_{write}, e_{inst}는 기존 함수, 자료형, 변수간에 존재하는 관계성, 즉 메시지 교환을 나타낸다. G^{time}의 에지(E^{time})를 수식으로 표현하면 다음과 같다:

$$E^{time} = E^{time}_{persistence} \cup E^{time}_{call} \cup E^{time}_{return} \cup E^{time}_{read} \cup E^{time}_{write} \cup E^{time}_{instance}$$

단, $E_{persistence}=\{e_{persistence} \mid e_{persistence}\in(e_{func}\vee e_{type}\vee e_{var})\}$.

각각 에지들을 수식으로 표현하면 다음과 같다:

$$E^{time}_{func} = \{\ e_{func} = \langle f^{k}_{i,j}\ ,\ f^{k}_{i,j+1}\rangle \mid f^{k}_{i,j}\ ,\ f^{k}_{i,j+1}\in N^{time}_{func},\ 1\le i\le \#func,\ 1\le j< \#ip,\ 1\le k\le \#cof\ \}$$

$$E^{time}_{type} = \{\ e_{type} = \langle t_{i,j}\ ,\ t_{i,j+1}\rangle \mid t_{i,j},\quad t_{i,j+1}\in N^{time}_{type},\ 1\le i\le \#type,\ 1\le j< \#ip\}$$

$$E^{time}_{var} = \{\ e_{var} = \langle v_{i,j}\ ,\ v_{i,j+1}\rangle \mid v_{i,j}\ ,\ v_{i,j+1}\in N^{time}_{var},\ 1\le i\le \#var,\ 1\le j< \#ip\ \}$$

$$E^{time}_{call} = \{\ e_{call} \mid e_{call}\in \langle f^{k}_{i,j}\ ,\ f^{l}_{m,1}\rangle,\ f^{k}_{i,j}\in N^{time}_{func},\ f^{l}_{m,1}\in N^{time}_{func},\ i\neq m\}$$

$$E^{time}_{return} = \{e_{return} \mid e_{return}\in\langle f^{k}_{i,x},\ f^{l}_{m,n}\rangle,\ f^{k}_{i,x}\in N^{time}_{func},\ f^{l}_{m,n}\in N^{time}_{func},\ i\neq m\}$$

$$E^{time}_{read} = \{e_{read} \mid e_{read}\in\langle v_{i,j},\ f^{k}_{m,n}\rangle,\ v_{i,j}\in N^{time}_{var}, f^{k}_{m,n}\in N^{time}_{func}\}$$

$$E^{time}_{write} = \{e_{write} \mid e_{write}\in\langle f^{k}_{m,n},\ v_{i,j}\rangle,\ v_{i,j}\in N^{time}_{var},\ f^{k}_{m,n}\in N^{time}_{func}\}$$

$$E^{time}_{inst} = \{e_{inst} \mid e_{inst}\in\langle v_{i,j},\ t_{m,n},\rangle,\ v_{i,j}\in N^{time}_{var},\ t_{m,n}\in N^{time}_{type}\}$$

여기에서, #cof는 함수의 호출 수이고, E^{time}_{call}에서 $f^{l}_{m,1}$의 첨자가 1은 호출되는 함수의 시작을 의미한다. E^{time}_{return}에서 $f^{k}_{i,x}$의 첨자 x는 f^{k}_{i}의 마지막 메시지 교환 시점을 나타낸다.

4.1.3 순서

FTV-time 그래프에서 순서는 메시지 교환이 발생하는 에지의 순서와 일치한다. 순서를 표현함으로써 에지들간의 상대적 위치를 시간의 흐름에 따라 가시적으로 볼 수 있도록 하였다. 이를 수식으로 표현하면 다음과 같다:

$$S^{time} = \langle\ \ldots\ e_i <_t e_j \ldots\ \rangle$$

단, $e_i,\ e_j \in E^{time}$.

여기에서 $<_t$는 상대적 시간 순서를 의미하여 e_i가 e_j보다 시간적으로 순서가 앞선다.

4.1.4 예제 프로그램에 대한 FTV-time 그래프

본 논문에서 사용한 예제 프로그램을 적용한 정적정보 추출 결과 생성된 G^{time}는 다음과 같다.

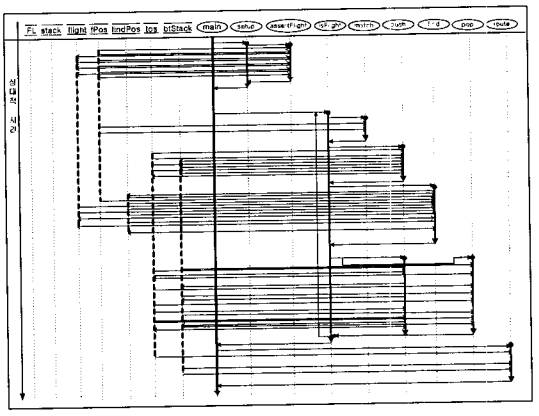


그림 4 DFS 프로그램에 대한 G^{time}

그림에서 직선의 화살표가 아래로 향하고 있는 것은 프로그램의 실행방향을 나타낸다. 수직방향의 굵은 점선은 변수와 관련된 것으로 함수들과의 메시지 교환이 발생되는 시기를 상대적 시간의 흐름에 의존하여 나타내고 있다. 굵은 수직방향의 화살표는 함수가 사용되는 시기를 표현한 것으로 함수와 함수간, 변수와 함수간의 메시지 교환의 순서를 나타내어 함수의 주기를 보인다. 또한 화살표 처음의 굵은 점은 함수가 처음 메시지 교환을 시작하고 있는 시점을 표현한다. 수평방향의 에지를 살펴보면 각각이 방향성이 존재함을 알 수 있다. 함수에서 변수로의 에지는 함수가 변수의 값을 수정(*write*)했을 때를 나타내며 반대의 경우는 함수가 변수를 사용(*read*)하는 경우이다. <표 1>에서 설명했듯이 이러한 수평방향의 에지는 5종류로 표현된다.

<그림 4>에서 보는 것처럼 특정 변수와 함수간에 밀접한 관련성이 있음을 알 수 있다. 예를 들어, 변수 *fPos*는 함수 *assertFlight*와 밀접한 관계성이 존재함을 그래프를 통해 알 수 있다. 또한 G^{FTV}에서 노드로 표현되었던 함수, 변수, 자료형들이 여러 에지와 노드가 추가되어 그들의 생성주기를 가시적으로 볼 수 있도록 한다. 예를 들어 *setup* 함수를 보면 프로그램 초기에만 사용되고 더 이상 사용되지 않음을 알 수 있다.

4.2 투영 단계(Projection Step)

투영 단계는 정적정보 추출 단계에서 추출된 결과를

상속성 추출결과 생성된 클래스와의 매핑 과정을 통해 관련성이 큰 함수, 자료형, 변수를 그룹화하는 과정이다. 절차지향 형태를 이루고 있는 정적정보를 객체지향 형태로 그룹화하여 객체 형태를 이룬다. 즉, 함수, 변수, 자료형을 *main* 함수를 기준으로 재배치한다. 정적정보 추출 결과 생성된 G^{time}에 클래스를 투영한 결과, 투영 *FTV-time* 그래프($G^{time}_{projection}$)가 추출되고 다음과 같이 3-쌍의 튜플로 정의된다:

$$G^{time}_{projection} = \langle N^{time}_{projection}, E^{time}_{projection}, S^{time}_{projection} \rangle$$

여기에서, $N^{time}_{projection}$은 $G^{time}_{projection}$의 노드, $E^{time}_{projection}$는 $G^{time}_{projection}$의 에지, $S^{time}_{projection}$는 $G^{time}_{projection}$에서의 순서를 의미한다. $G^{time}_{projection}$는 투영 결과 여러 개의 그룹으로 묶여지며, 그룹화된 $G^{time}_{projection:i}$는 소스코드에서 클래스를 투영한 객체*(CPO: CP-object* 또는 *a classed projected object in source code)*라고 한다. $G^{time}_{projection:i}$는 G^{time}에 포함된다. $G^{time}_{projection}$는 G^{time}에 포함되며, 이들 *CPO*들은 서로 결합된 부분이 존재하지 않는다. 이를 수식으로 표현하면 다음과 같다:

$$G^{time}_{projection} = \{G^{time}_{projection:i} \mid G^{time}_{projection:i} \subseteq G^{time},\ G^{time}_{projection:i} \cap G^{time}_{projection:j} = \Phi,\ i \neq j,\ 1 \leq i \leq \#obj\}$$

단, $G^{time}_{projection:i} \subseteq G^{time}$ $(1 \leq i \leq n)$ *n: CPO*의 개수

4.2.1 노드

투영된 결과 발생하는 노드는 그룹화된 $G^{time}_{projection:i}$의 집합으로 표현되며 N^{time}과 같은 형태를 취하고 있다. 이를 수식으로 표현하면 다음과 같다:

$$N^{time}_{projection} = \{N^{time}_{projection:i} \mid N^{time}_{projection:i} \subseteq N^{time},\ N^{time}_{projection:i} \cap N^{time}_{projection:j} = \Phi,\ i \neq j,\ 1 \leq i \leq \#obj\}$$

단, $N^{time}_{projection:i} \subseteq N^{time}$ $(1 \leq i \leq n)$ *n: CPO*의 개수

여기에서, $N^{time}_{projection:i}$은 $G^{time}_{projection:i}$의 노드이다.

4.2.2 관계성

$G^{time}_{projection}$의 에지의 집합은 G^{time}의 것과 같은 종류로 발생되고 함수, 변수, 자료형의 그룹화로 인해 G^{time}의 에지와 포함관계를 가지고 있다. 투영결과 생성된 에지의 집합을 수식으로 표현하면 다음과 같다:

$$E^{time}_{projection} = E^{time}_{persistence} \cup E^{time}_{read} \cup E^{time}_{write} \cup E^{time}_{call} \cup E^{time}_{return} \cup E^{time}_{inst} = \{E^{time}_{projection:i} \mid E^{time}_{projection:i} \subseteq E^{time}_{projection},\ 1 \leq i \leq \#cpo\} \cup \{e \mid e \in (G^{time}_{projection:i},\ G^{time}_{projection:j}),\ i \neq j\}$$

여기에서, $E^{time}_{projection:i}$은 $G^{time}_{projection:i}$의 에지이다.

4.2.3 순서

$S^{time}_{projection}$는 G^{time}와 마찬가지의 순서를 가지고 있으며 $S^{time}_{projection}$가 S^{time}에 포함된다. $G^{time}_{projection}$의 순서를 순서쌍으로 표현하면 다음과 같다:

$$S^{time}_{projection} = \langle \ldots e_i <_t e_j \ldots \rangle$$

단, $e_i,\ e_j \in E^{time}_{projection}$

$S^{time}_{projection}$에서 $<_t$는 e_i가 e_j보다 시간적으로 순서가 우위임을 나타낸다.

4.2.4 예제 프로그램에 대한 투영 FTV-Time 그래프

<그림 5>은 투영 결과 생성된 DFS의 투영 FTV-time 그래프이다. 노드는 함수, 변수, 자료형의 그룹으로 표현되고 이 그룹간의 관련 에지들만 존재한다. CPO_1과 CPO_2은 함수, 변수, 자료형을 그룹화한 $G^{time}_{projection:i}$를 의미한다.

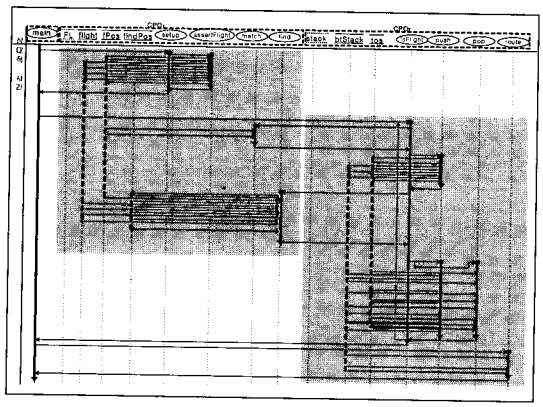


그림 5 DFS 프로그램에 대한 $G^{time}_{projection}$

그림에서 에지는 <그림 5>의 G^{time}과 같은 형태이다. $G^{time}_{projection}$에서 그래프 의미는 절차지향의 프로그램 형태에서의 메시지 교환을 객체지향의 메시지 교환 형태로 표현한 것이다. 객체 내부의 메시지 교환보다는 객체들 간의 메시지 교환에 중점을 둔 형태를 추출하기 위한 그룹화 과정이라 할 수 있다.

<그림 5>에서 보는 것처럼 관련성이 밀접한 함수, 변수, 자료형이 그룹으로 묶이는 것을 볼 수 있다. 예를 들어, CPO_1은 자료형 *FL*과 변수 *flight, fPos, findPos*, 그리고 함수 *setup, assertFlight, match, find*가 밀접한 관련성이 있어 하나의 그룹으로 형성됨을 알 수 있다.

4.3 반영 단계(Reflection Step)

지속성을 결정하는 목적 중의 하나가 객체의 정확한

메시지 교환의 시작/종료 시기를 추출하는 것이다. 그러므로 내부의 메시지 교환은 이차적인 문제이다. 지속성 결정의 반영 단계는 2단계 결과 생성된 *CPO* 내부의 메시지 교환을 함축하여 표현하고 *CPO* 사이의 메시지 교환에 초점을 둔다. 즉, 투영 과정에서 그룹화 되었던 *CPO* 내부 메시지는 랩핑하는 형태로 *CPO*들간의 내부 메시지 교환을 은닉한다. 반영 결과 생성된 그래프를 반영 time 그래프($G^{time}_{reflection}$)라하고 다음과 같이 3-쌍의 튜플로 정의된다:

$$G^{time}_{reflection} = \langle\ N^{time}_{reflection}\ ,\ E^{time}_{reflection}\ ,\ S^{time}_{reflection}\ \rangle$$

$G^{time}_{reflection}$는 내부 메시지교환은 이제 보이지 않으며 하나의 직선으로 표현된다. 그리고 *CPO* 외부의 메시지 교환에 초점을 둔다. *CPO*간의 외부 메시지 교환에서 *CPO*의 시작/종료 메시지 교환을 추출한다.

4.3.1 노드

$N^{time}_{reflection}$는 *CPO* 사이의 메시지 교환 노드가 추출된다. 이를 수식으로 표현하면 다음과 같다:

$$N^{time}_{reflection} = \{ n_{i,k} \mid 0 \le i \le \#cpo,\ 1 \le k \le \#ip \}$$

*#cpo*는 *CPO*의 개수이고, *#ip*는 메시지 교환 시점 중 *CPO*들 사이의 메시지 교환 개수이다.

4.3.2 관계성

노드와 노드를 연결해 주는 에지($E^{time}_{reflection}$)는 *CPO* 간에 메시지 교환을 나타낸다. 즉, $G^{time}_{projection}$에서 *CPO*의 외부 메시지 교환에 초점을 두었다. 이를 수식으로 표현하면 다음과 같다:

$$E^{time}_{reflection} = \{ e \mid e \in (N^{time}_{projection.i},\ N^{time}_{projection.j}), i \neq j \}$$
$$= \{ e \mid e \in (G^{time}_{projction.i},\ G^{time}_{projection.j}), i \neq j \}$$

4.3.3 순서

$S^{time}_{reflection}$는 에지의 튜플로 이루어져 있으며 이들간에는 시간적 순서가 존재한다. 이를 수식으로 표현하면 다음과 같다:

$$S^{time}_{reflection} = \langle\ \ldots\ e_i <_t e_j \ldots\ \rangle$$
$$\text{단},\ e_i,\ e_j \in E^{time}_{reflection}$$

$S^{time}_{reflection}$에서 $<_t$는 e_i가 e_j보다 시간적으로 순서가 앞선다는 것을 의미한다.

4.3.4 예제 프로그램에 대한 반영 FTV-Time 그래프

<그림 6>는 DFS 예제를 적용한 반영 FTV-time 그래프이다.

위 그림에서는 제어 역할을 하는 *main*을 중심으로 객체의 형태를 가진 *CPO*들간의 메시지 교환을 나타내고 있다. *main*은 프로그램의 시작이며 종료를 나타내므로 *main*에서의 수직 에지는 그것을 표현하고 있다. $G^{time}_{projection}$에서 그룹화된 *CPO*가 하나의 굵은 수직 에지로 표현되고 각각 양쪽 끝에 굵은 점은 이들의 시작과 종료를 나타낸다. 수평방향의 에지는 *CPO*간의 메시지 교환 시점을 표현한 것이다. 수평 에지 중에 굵은 화살표로 나타나는 것은 *CPO*간의 최초 최후의 메시지 교환 시점이다.

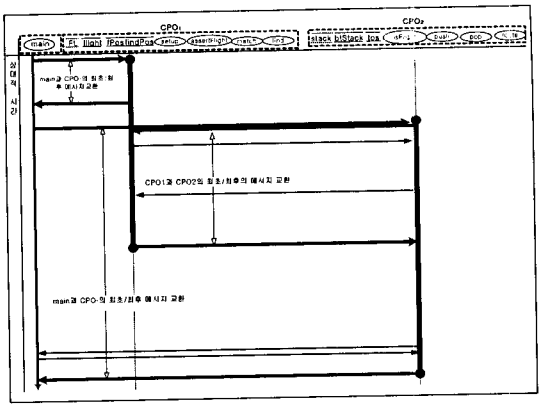


그림 6 DFS 프로그램에 대한 $G^{time}_{reflection}$

<그림 6>를 보면 *CPO*가 어느 시기에 메시지 교환이 시작되고 종료되는 지를 가시적으로 볼 수 있다. 예를 들어, CPO_1은 *main*의 초기에 시작하여 중간까지 사용되고, 다시 CPO_2와 마지막으로 메시지 교환이 이루어지고 종료됨을 알 수 있다.

4.4 인스턴스 단계(Instantiation Step)

지속성 결정 네 번째는 인스턴스 단계이다. 반영 단계 통해 객체의 시작시점과 종료 시점을 알게 되었다. 그러나 객체들간의 메시지 교환이 이루어지기 전에 먼저 객체의 생성이 이루어져야 하고, 메시지 교환이 종료된 후에는 객체가 소멸됨으로써 시스템이 효율적으로 메모리를 관리할 수 있도록 해야 한다. 인스턴스 단계에서는 객체의 메시지 시작 시점 전과 종료 시점 후 객체의 생성/소멸에 관한 관련성을 추가하는 작업이 이루어지게 된다. 일련의 작업 결과 인스턴스 FTV-time 그래프 ($G^{time}_{instantiation}$)가 생성되고 다음과 같이 3-쌍의 튜플로 정의된다:

$$G^{time}_{instantiation} = \langle\ N^{time}_{instantiation},\ E^{time}_{instantiation},\ S^{time}_{instantition}\ \rangle$$

4.4.1 노드

$G^{time}_{instantiation}$의 노드($N^{time}_{instantiation}$)는 $N^{time}_{reflection}$와 생성과 소멸에 관련한 노드를 새롭게 추가하는 과정으로 이루어진다. 이를 수식으로 표현하면 다음과 같다:

$$N^{time}_{instantiation} = N^{time}_{reflection} \cup \{n_{i,0},\ n_{i,z}\}$$

단, $n_{i,0}$는 $N^{time}_{reflection:\,i,1}$와 연결될 생성 노드, $n_{i,z}$는 $N^{time}_{reflection:\,i,y}$와 연결될 소멸 노드.

4.4.2 관계성

$G^{time}_{instantiation}$의 에지($E^{time}_{instantiation}$)는 $E^{time}_{reflection}$와 객체들 간의 생성노드를 연결한 에지와 소멸노드를 연결한 에지를 첨가한다. 새롭게 추가된 에지의 종류와 의미는 다음과 같다.

표 4 생성/소멸에 관련한 에지의 종류와 의미

종 류	기 호	의 미	방 향
$(n_{i,0},\ n_{i,1})$	$e_{constructor\_v}$	객체의 생성	수직
$\langle n_{0,i},\ n_{j,0}\rangle$	$e_{constructor\_h}$	객체의 생성	수평
$(n_{i,y}, n_{i,z})$	$e_{destructor\_v}$	객체의 소멸	수직
$\langle n_{0,i},\ n_{j,z}\rangle$	$e_{destructor\_h}$	객체의 소멸	수평

$G^{time}_{instantiation}$의 에지($E^{time}_{instantiation}$)를 수식으로 표현하면 다음과 같다:

$$E^{time}_{instantiation} = E^{time}_{reflection} \cup e_{constructor\_v} \cup e_{destructior\_v} \cup e_{constructor\_h} \cup e_{destructior\_h}$$

단, $e_{constructor\_v} \in (n_{i,0},\ n_{i,1})$, $e_{destructor\_v} \in (n_{i,y}, n_{i,z})$ $e_{constructor\_h} \in \langle n_{0,i},\ n_{j,0}\rangle$, $e_{destructor\_h} \in \langle n_{0,i},\ n_{j,z}\rangle$, (n_0는 *main* 함수)

생성/소멸에 관한 메시지 전달은 *CPO*들이 마지막 메시지 교환이 시작되고 끝났음을 인지하고 *main*에서 각각의 생성/소멸되는 *CPO*들에게 메시지를 보내준다.

4.4.3 순서

$G^{time}_{instantiation}$의 순서는 $E^{time}_{instantiation}$의 발생 순서와 일치한다. 이를 수식으로 표현하면 다음과 같다:

$S^{time}_{instantition} = \langle\ \ldots\ e_i <_t e_j \ldots\ \rangle$,

단, $e_i,\ e_j \in E^{time}_{instantiation}$

$S^{time}_{instantiation}$에서 $<_t$는 e_i가 e_j보다 시간적으로 순서가 앞선다는 것을 의미한다.

4.4.4 예제 프로그램에 대한 인스턴스 FTV-Time 그래프

<그림 7>는 DFS 예제를 적용한 인스턴스 FTV-time 그래프이다.

<그림 7>의 수직 에지는 $G^{time}_{reflection}$과 *CPO*의 시작/종료 시점의 앞뒤에 새롭게 생성/소멸에 관련된 점선으로 표현된 수직 에지와 노드가 첨가된다. 수평 에지는 $G^{time}_{reflection}$의 수평 에지와 생성/소멸을 나타내는 에지이

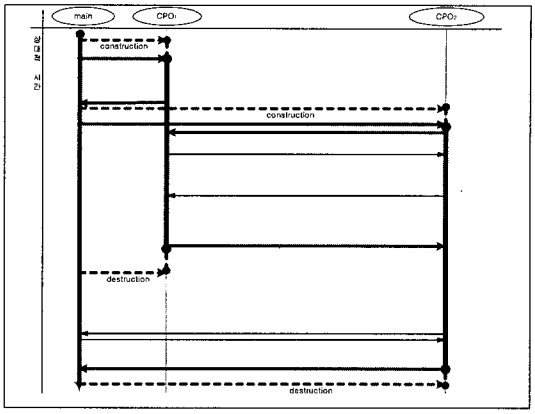


그림 7 DFS 프로그램에 대한 $G^{time}_{instantiation}$

며 방향은 제어역할을 하는 *main*에서 각각의 *CPO*에게 메시지를 보내는 형태를 이루고 있으며 점선으로 나타난다.

<그림 7>에서 보는 것처럼 객체의 생성과 소멸시기를 추출한다. 예를 들어, CPO_1은 *main*의 초기에 사용되므로 초기에 생성이 이루어지고, 다시 CPO_2와 마지막으로 메시지 교환이 이루어지고 종료되기 때문에 그 시점에서 소멸됨을 볼 수 있다.

4.5. 정제 단계(Refinement Step)

정적정보 추출을 통해 프로그램의 메시지 교환을 시간의 흐름에 따라 가시적으로 볼 수 있게 하였고, 이 정적정보에 클래스를 투영하여 절차지향 프로그램을 객체의 형태로 그룹화 하였다. *CPO* 내부의 메시지 교환을 은닉하여 *CPO*의 시작/종료 시점을 추출하고, 이를 바탕으로 생성/소멸 시점을 추출하였다. 객체의 정확한 생성과 소멸 시점을 안다는 것은 객체의 불필요한 생성과 소멸을 감소시켜 효율적으로 메모리를 사용할 수 있어 과부하를 최소화 할 수 있는 장점이 있다. 그래서 정확한 객체의 생성과 소멸을 위해서는 이들간의 안전성과 일관성이 보장되어야 한다.

이를 위해 정제 단계에서는 제어 메소드를 추가하여 객체간의 정확한 메시지 교환을 보장한다. 객체간의 메시지 교환이 이루어지기 위해 객체의 생성/소멸 시점 전에 제어 객체에게 객체의 생성/소멸을 요구하는 요구(*request*)메시지를 보내고 객체가 정확히 생성/소멸되었다는 응답(*acknowledgement*)메시지를 받은 후 객체들 간의 메시지 교환이 이루어지도록 하고 있다.

정제 결과 생성된 그래프를 정제 FTV-time 그래프($G^{time}_{refinement}$)라하고 다음과 같이 3-쌍의 튜플로 정의된다:

$$G^{time}_{refinement} = \langle N^{time}_{refinement}, E^{time}_{refinement}, S^{time}_{refinement} \rangle$$

$G^{time}_{refinement}$는 객체가 생성과 소멸되기 전에 먼저 요구하고 그에 대한 응답의 제어 기능을 하는 메시지 교환에 관한 관련성이 새롭게 추가된다.

4.5.1 노드

$N^{time}_{refinement}$은 $G^{time}_{instantiation}$의 노드와 객체 생성/소멸에 관련한 제어 노드의 집합이다. 이를 수식으로 표현하면 다음과 같다:

$$N^{time}_{refinement} = N^{time}_{instantiation} \cup \{ n^{request}_{i,j}, n^{ack}_{i,j} \mid j: number\ of\ node \}$$

$N^{time}_{refinement}$에서 $N^{time}_{instantiation}$은 $N^{time}_{refinement}$에 포함된다. $n^{request}_{i,j}$, $n^{ack}_{i,j}$는 객체가 생성/소멸 전에 제어 객체에게 보내고 받는 제어 노드이다. j는 객체가 생성/소멸할 때 노드 번호이다.

4.5.2 관계성

$E^{time}_{refinement}$은 $G^{time}_{instantiation}$의 에지와 제어와 관련한 에지($e_{control}$)를 추가하게 된다. 새롭게 추가된 에지의 의미와 종류는 다음과 같다.

표 5 추가된 에지의 종류와 의미

종 류	기 호	의 미	방향
($n^{request}_{i,j}$, $n^{request}_{k,l}$)과 ($n^{ack}_{i,j}$, $n^{ack}_{k,l}$)	$e_{control}$	객체의 생성/소멸 제어	수평

$E^{time}_{refinement}$를 수식으로 표현하면 다음과 같다:

$$E^{time}_{refinement} = E^{time}_{instantiation} \cup e_{control}$$

$$단, e_{control} \in \{ (n^{request}_{i,j}, n^{request}_{k,l}), (n^{ack}_{i,j}, n^{ack}_{k,l}) \}$$

4.5.3 순서

$G^{time}_{refinement}$의 순서는 $E^{time}_{refinement}$의 발생 순서와 일치한다. 이를 수식으로 표현하면 다음과 같다.

$$S^{time}_{refinement} = \langle \ldots e_i <_t e_j \ldots \rangle$$

$$단, \ e_i, \ e_j \in E^{time}_{refinement}$$

4.5.4 예제 프로그램에 대한 정제 FTV-Time 그래프

<그림 8>는 DFS 예제를 통해 생성된 정제 FTV-time 그래프이다.

위 그림에서 수직 에지는 $G^{time}_{instantiation}$에서 생성된 수직 에지와 같은 형태를 이루고 있고, 생성/소멸의 제어를 위한 요구/응답에 관련한 노드가 새롭게 추가된다. 수평 에지의 경우 $G^{time}_{instantiation}$의 수평에지와 생성/소멸의 요구/응답에 관련한 수평 에지가 점선의 방향성을

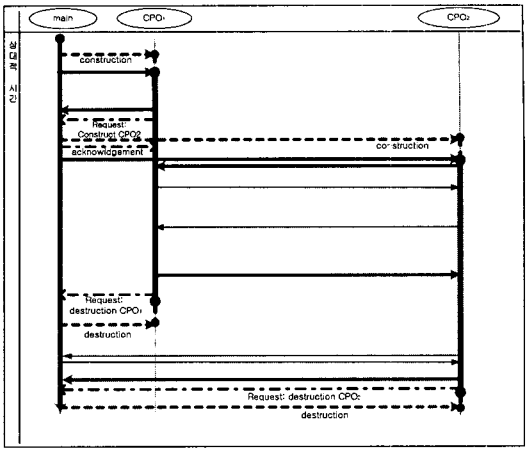


그림 8 DFS 프로그램에 대한 $G^{time}_{refinement}$

가진 에지로 표현된다. 이 에지의 요구 방향은 *CPO*의 생성을 요구하는 쪽에서 *main*에게 보내지고, 응답은 *main*이 *CPO*에게 보내는 형태로 이루어지고 있음을 알 수 있다. 그리고 수평 에지에서 인스턴스에서 생략되었던 *CPO* 사이의 메시지 교환이 있는 에지가 새롭게 추가되었음을 볼 수 있다.

<그림 8>에서 보는 것처럼 CPO_1과 CPO_2의 생성과 소멸시점을 추출하여 이들이 안전하게 생성/소멸되고 유지될 수 있도록 하기 위해 CPO_2가 CPO_1과 메시지 교환이 시작되기 전에 *main* 함수에게 생성을 요구하는 메시지를 보내게 되고, 생성된 후 안전하게 생성되었다는 응답 메시지를 받은 후 메시지 교환을 시작하게 된다. CPO_1의 경우 마지막 메시지 교환이 이루어진 후 더 이상이 메시지 교환이 없을 때 *main* 함수에게 CPO_1의 소멸을 요구하는 메시지를 보낸 후 소멸됨을 볼 수 있다.

5. 실험 및 분석

<표 4>은 본 논문에서 제안한 지속성 결정 단계의 효율성과 실행성을 보이기 위해 임의로 선택한 절차지향 소프트웨어(*Procedural Software, PSW*) 즉, C 코드를 적용한 결과이다.

<표 4>의 실험에서는 다음의 결과가 나타난다: 1) 초기시점에서 반영 단계까지는 프로그램의 함수 및 데이타 수에 변화가 보이지 않지만, 인스턴스 단계에서는 객체의 사용을 위해 새롭게 생성/소멸 함수가 추가되고, 2) 객체간의 메시지 교환의 안전성을 위해 정제단계에서 요구/응답에 관한 함수를 추가하며, 3) 코드의 크기가 크다고 해서 클래스의 수가 증가하는 것은 아니며

표 6 실험 결과

PSW		Dfs.c	bintree.c	Hier.c	Chory.c	Os.c	Add-Manger.c	compiler.c	graphic.c
Size(Line of Code)		195	210	300	1065	1374	434	424	344
총 노드의 수		147	202	254	817	1279	448	411	250
자료형의 수		2	5	4	4	1	2	0	0
전역변수의 수		5	7	4	13	2	8	9	2
함수의 수		9	9	15	40	14	18	14	9
클래스의 수		3~7	3~7	4~8	7~17	1~3	3~9	1~9	1
객체의 수		1~5	1~7	1~5	1~11	1~3	1~7	1~9	1
선택된 클래스		4	3	6	9	2	4	6	2
정적정보단계 투영단계 반영단계	데이타 수	7	12	8	17	3	10	9	2
	함수의 수	9	9	15	40	14	18	14	9
인스턴스단계	증가한 함수의 수	4	2	4	2	2	6	12	4
정제단계	증가한 함수의 수	4	0	4	0	0	4	2	0
생성(create) 함수의 수		2	1	2	1	1	3	6	2
소멸(delete) 함수의 수		2	1	2	1	1	3	6	2
요구(request) 함수의 수		3	0	3	0	0	2	1	0
응답(ack) 함수의 수		1	0	1	0	0	2	1	0

객체의 생성/소멸에 대한 효율성에 큰 영향을 주고 있지 않음을 알 수 있다. *Chory.c*의 경우 클래스의 수가 8개임에도 불구하고 경우에 따른 클래스의 선택적 객체 생성/소멸이라는 프로그램에 특성에 따라 객체의 생성/소멸이 적음을 알 수 있다. <도표 1>은 *PSW*를 [3]의 재공학 방법에 따라 클래스를 생성하고 실제 실행단계에서 생성된 객체의 생성과 소멸 함수의 수이다. 도표에서 보여지는 것처럼 프로그램의 크기가 크다고 해서 객체의 수가 비례적으로 증가하는 것이 아님을 알 수 있다. 즉, 프로그램의 크기는 클래스, 또는 객체의 생성에 직접적인 영향을 주지 않는다.

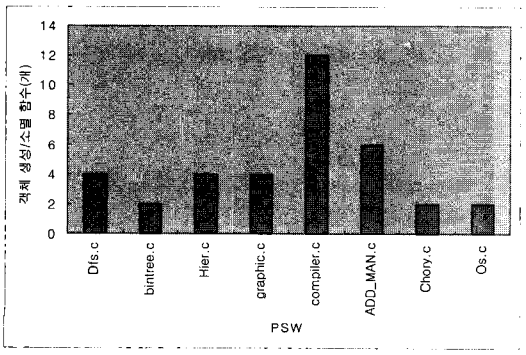


도표 1 PSW의 크기(LOC)와 객체의 생성/소멸 관계

6. 결론 및 향후 연구

소프트웨어는 제작된 후 지속적으로 유지·보수를 해야한다. 개발된 시스템들은 사용자의 새로운 요구, 새로운 기술의 개발, 새로운 환경에 맞도록 수정 보완되어야 하며 이는 막대한 비용을 필요로 한다. 그렇기 때문에 시스템을 재공학할 때 소프트웨어의 유지·보수 및 재사용적인 측면에서, 시스템을 기존의 방식들보다 유리한 객체지향 파라다임으로 재개발한다면 소프트웨어 생산성을 향상시킬 수 있다.

본 논문에서는 재공학 과정 중 네 번째 단계인 안전한 지속성 결정에 대해 기술하였다. 상속성 추출 과정을 통하여 추출된 클래스들이 정확한 시기에 객체로 생성되고 소멸되는 과정은 객체지향 프로그램에서는 매우 중요하다. 만약 객체가 프로그램 초기부터 종료시까지 존재한다면 불필요한 메모리 낭비와 과부하를 초래하게 된다. 본 논문에서는 객체의 정확한 생성과 소멸 시점을 제시하였고 정제과정을 통해 정확한 메시지 교환을 추출하였다.

본 논문에서는 지속성 결정을 위한 방법론으로 5단계를 기술하였다. 정적정보에 시간 개념을 첨가하고 클래스와의 투영과정을 통해 객체의 형태로 그룹화하였으며, 반영 단계에서는 객체의 시작과 종료시점을 추출하였다. 이를 바탕으로 인스턴스 단계에서는 객체의 생성/소멸

시점을 추출하였고, 마지막으로 정제 과정을 통하여 객체 생성의 안정성과 일관성을 보장하였다.

향후 연구로는 생성된 객체지향 프로그램이 정상적으로 작동하는지와 원래의 절차지향 프로그램과 변환된 객체지향 프로그램이 의미상으로 동등한지를 대한 동일성 검사를 한다.

참 고 문 헌

[1] Sagar Pidaparthi, Grzegorz Cysewski, *"Case Study in Migration to Object-Oriented System Structure Using Design Transformation Methods,"* The Proceeding of CSMR '97, IEEE press. Berlin, Germany, 1997.3.

[2] Robert S. Arnold, *"Software Reengineering,"* IEEE Computer Society Press, 1994.

[3] Moon-Kun Lee, Sung-Og Park, *"A Methodology to Extract Objects from Procedural Software,"* The proceedings of COMPSAC2000, pp.557-566, IEEE press, Taipei, Taiwan, 2000.10.

[4] K. Maruyama, *"Automated Method-Extraction Refactoring by Using Block-Based Slicing,"* The Proceeding of SSR'01, ACM/SIGSOFT. Toronto, Ontario, Cananda, 2001. 5.

[5] www.omg.org, *"OMG Unified Modeling Language Specification,"* ver1.3, 1999.6.

[6] 최완, *"SDL 환경,"* 한국전자통신연구소, 1994.1.

[7] R.Dssouli, G.V. Bochmann, Y. Lahav, *"SDL '99,"* Elsevier, 1999.

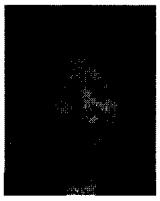
[8] G. Rothermel and M.J.Harrold, *"Selecting regression tests for object-oriented software,"* In Proceeding of Int'l Conf. Software Maintenance (ICSM), pp.14-25, Aug. 1994.

[9] H.Gomma, D.A.Menascé, M.E.Shin, George Mason University, USA, *"Reusable Component Interconnec tion Patterns for Distributed Software Architectures,"* The Proceeding of SSR'01, ACM/SIGSOFT. Toronto, Ontario, Cananda, 2001. 5.

[10] 박성옥, 노경주, 이문근, *"최적합 객체 선정을 위한 다중 객체군 추출,"* 한국 정보 과학회 논문집(B), 제26권, 제12호, pp. 1468-1481, 1999.

[11] 박성옥, 최정란, 이문근, *"절차지향 SW를 객체지향 SW로 재공학하기 위한 클래스와 상속성 추출에 관한 연구,"* 2000 한국 소프트웨어공학 학술대회 논문집, pp. 51-60, 2000.02.

[12] L. Bass, P.Clements, and Rich Kazman, *"Software Architecture in Practice,"* Addison-Wesley, 1998.



최 정 란

1999년 전북대학교 컴퓨터과학과 졸업 (이학사). 1999년 ~ 현재 전북대학교 전산통계학과 석사 과정 중. 관심분야는 소프트웨어 재·역공학, 실시간 시스템, 운영체제, 컴파일러 등

이 문 근

1989년 The Pennsylvania State University, Computer Science 학과 졸업 (이학사). 1992년, The University of Pennsylvania, Computer and Information Science 학과 졸업 (이공학석사). 1995년 The University of Pennsylvania, Computer and Information Science 학과 졸업 (이공학박사). 1992년 5월 ~ 1996년 1월 미국 Computer Command and Control Company, Computer Scientist로 근무. 1996년 4월 ~ 1998년 3월 전북대학교 컴퓨터과학과 전임강사. 1998년 4월 ~ 1999년 2월 전북대학교 컴퓨터과학과 조교수. 1999년 3월 ~ 현재, 전북대학교 전자정보 공학부 조교수. 관심분야는 소프트웨어 재·역공학, 실시간 시스템, 운영체제, 형식언어, 병렬함수언어, 컴파일러 등.