EMPS : 의미를 보존하는 효율적인 소프트웨어 병합

김 지 선[†] · 윤 청[††]

요 약

분기 및 병합은 대규모 소프트웨어 개발에 있어 병렬개발을 지원할 수 있는 소프트웨어 형상관리의 대표적인 기법이다. 상업적인 용도로 쓰이는 기존의 소프트웨어 병합은 문자적 병합에만 의존하고 의미를 고려하지 않아 병합결과에서 의미적 오류를 발생시킬 수 있다. 현재까지 의미적 병합충돌 탐지 및 병합에 대한 연구가 이루어지고 있으나 의미적 병합충돌 및 병합에 대해서만 개별적으로 이루어지고 있으므로, 문자적 병합과정에서 의미적 충돌을 탐지하고 해결할 수 있는 기법에 대한 연구가 필요하다.

본 논문에서는 프로그램 원본에서 분기된 두개의 프로그램들을 병합하는데 있어서 그들 각각의 원본에 대한 각 라인별 수정, 삭제, 삽입 오퍼레이션들을 정의하여 병합과정에서 발생하는 문자적 충돌 및 실행행위에 관련된 의미적 충돌을 탐지하고 해결하여 병합하는 문자적 병합기법과 의미적 병합기법을 결합한 하이브리드 병합기법을 제시하였다.

키워드 : 소프트웨어 병합, 소프트웨어 형상관리

EMPS : An Efficient Software Merging Technique for Preserving Semantics

Jisun Kim[†] · Cheong Youn[††]

ABSTRACT

Branching and merging have been being the outstanding methods for SCM in terms of supporting parallel developments. Since well-known commercial merging tools based on textual merging have not detecting semantics conflicts, they can cause semantic errors in the result of merging. Although a lot of researches for detecting semantic conflict and merging up to recently, these researches have been doing individually. Therefore, it is necessary for a research detecting semantic conflict on textual merging and solving it.

In this paper, we propose a new method for merging which preserve semantics on textual merging. The method merging two revisions from a source program is as follows : 1) defining changing operations, which include Update, Delete, and Insert operation, per line on two revisions corresponding to the line in source program, 2) detecting textual conflicts and semantic conflict in terms of executional behaviors, 3) solving these conflicts before merging. So, the proposed method can be regarded as a hybrid method that combines a method of textual merging and a behavioral semantic merging.

Key Words : Software Merging, Software Configuration Management

1. 서 론

대규모 소프트웨어 개발은 소프트웨어 엔지니어링 연구 분야에서 많은 어려운 문제점을 제시하고 있다. 소프트웨어 진화는 릴리즈(release)뿐만이 아니라 릴리즈들 사이에서도 병렬개발에 대한 문제를 야기하고 있다. 또한 소프트웨어의 크기 증가는 병렬개발정도를 심화시키고 있으며 개발자간에 상호작용과 상호의존성을 극대화시키고 있다. 그리고 다차원적인 시스템 구조는 개발단위를 독립적인 작업 단위들로 정돈하여 분류할 수 없도록 되어있어 다수 개발자들 사이에서의 지식전달에 있어 문제점을 발생시킨다. 따라서 복잡하고 대규모의 소프트웨어 시스템을 개발하고 진화시키는데 있어서 어떻게 병렬변화의 양상(state)을 어떻게 체계적으로 관리하느냐는 가장 기초적이고 중요한 문제라고 할 수 있다[1, 2].

소프트웨어 형상관리 (Software Configuration Management)는 소프트웨어 개발 및 유지보수과정에서 발생하는 각종 결과물들에 대한 계획, 개발, 운용 등을 종합하여 시스템 형상을 만들고 이에 대한 변경을 체계적으로 관리, 제어하기 위한 활동으로써 대규모 소프트웨어 개발에 있어서 병렬변화 양상(Parallel Change States)을 제어하고 관리할 수 있도록 하고 있다[2]. 소프트웨어 형상관리는 소프트웨어 제품에 대한 변화를 통제하기 위해 관리측면에서 형상식별, 형상제어, 형상상태보고, 형상감사 및 검토 등을 지원하고 있다. 또한 개발자 측면에서는 개발자들 사이에서 파일 변

† 준 회 원 : 국방과학연구소 기술원
†† 정 회 원 : 충남대학교 컴퓨터과학과 정교수
논문접수 : 2005년 2월 15일, 심사완료 : 2005년 9월 1일

화를 통합하기 위한 파일버전 식별 및 소프트웨어 빌드(Build), 릴리즈 활동 등을 지원하고 있다[3].

분기(Branching)와 병합(Merging)은 여러 명의 개발자들의 독립적인 작업 환경을 제공하는 형상관리의 개발측면에서 지원을 위한 대표적인 기법이다. 공통의 형상항목을 두 가지 이상의 독립된 흐름으로 나누는 것을 분기라 하며 이렇게 분기된 형상 항목의 변경을 통합하는 것을 병합이라 한다. 분기 및 병합기법을 사용하면 다른 개발자가 작업하고 있는 형상항목을 작업이 끝날 때까지 기다릴 필요 없이 동시에 작업할 수 있도록 하는 것에 의해 병렬개발을 지원하므로 결과적으로 개발시간을 단축할 수 있다[4].

소프트웨어 병합은 병합될 소프트웨어들 사이에 상호 연결된 다양한 요소들을 포함하고 있으며 구성요소들의 문법 및 의미 모두에 의존적이기 때문에 시간이 오래 걸리고 복잡하며 잠재적인 에러가 내포된 작업이다. 특히 대부분의 기존 접근기법들은 융통성이나 표현력이 부족하기 때문에 일반적으로 상업적으로 널리 사용되는 병합도구는 문자적(Textual) 병합기법에 기초한다. 문자적 병합기법은 프로그램 뿐 아니라 문서 등과 같은 서로 다른 종류의 소프트웨어 산출물들을 병합하기 위해 사용될 수 있고, 정확성 및 효율성면에서 뛰어나지만 병합과정에서 문법 및 의미를 고려하지 않고 모든 것을 텍스트로 간주하기 때문에 병합결과에 문법적, 의미적 충돌이 발생할 수 있다.

이런 문제점들을 해결하기 위해 문법적, 의미적 병합기법에 대한 연구들이 진행되었지만 독립적인 연구로 수행되어 문법적 병합기법은 여전히 의미적 오류가 잠재되어 있고, 의미적 병합기법은 실행행위와 관계없는 문자적 변화를 포함하여 병합하지 못한다는 문제점들을 가지고 있다. 따라서 이들 문제점들을 상호보완하여 해결할 수 있는 향상된 기법에 대한 연구가 필요하다.

본 논문에서는 상업적으로 널리 쓰이는 효율적인 문자적 병합기법을 토대로 라인별 수정, 삽입, 삭제 오퍼레이션을 정의 및 사용하여 병합과정에서 발생하는 문자적 충돌 및 실행행위에 관련된 의미적 충돌을 탐지하고 해결하여 병합하는 문자적 병합기법과 의미적 병합기법을 결합한 하이브리드 병합기법인 EMPS기법(An Efficient Software Merging Technique for Preserving Semantics)을 제안한다.

본 논문의 구성은 다음과 같다. 제 2장에서는 여러 가지 병합기법 및 병합충돌 해결기법에 대한 선행된 관련연구를 살펴보고, 제 3장에서는 EMPS기법에 대해 제시한 후 제 4장에서 의미적 병합결과에 대한 타당성 검증하고 제 5장에서 타 기법들과 비교 평가한다. 마지막으로 제 6장에서는 결론을 통한 향후 연구방향을 모색한다.

2. 관련 연구

기존의 상업적인 툴 및 연구를 통한 프로토타입에 적용된 기법들은 비교 대상의 범위에 따라 크게 2-Way 방식의 병합기법과 3-Way 방식의 병합기법으로 나눌 수 있다. 그리고 이들 두 가지 방식들을 토대로 소프트웨어 산출물을 어떻게 표현하여 병합하느냐에 따라 문자적, 문법적, 의미적, 그리고 오퍼레이션 기반 병합기법 등이 있다[5].

2.1 2-Way 방식의 병합기법과 3-Way 방식의 병합기법

2-Way 방식의 병합기법은 상태기반 병합이라고도 불리며, 병합하려는 두 버전에 대해 공통 조상에 대한 정보에 의존하지 않고 병합하는 방식으로서 단순히 두 버전들 사이의 차이점만 비교하기 때문에 대응되는 부분이 서로 다를 경우에는 차이점이 어떤 이유에서 발생되었는지 알 수 없다[5].

3-Way 방식의 병합기법은 변화기반 병합이라고도 불리며, 병합하려는 두 버전에 대해 공통 조상에 대한 정보를 이용하여 병합하는 방식이다. 이 방식은 (그림 1)과 같이 두 개의 버전에서 공통 조상에 대한 각각의 차이점을 공통조상에 동시 반영시켜 병합하는 방식이다. 병합된 결과는 두 버전에서 일어난 모든 수정과 삽입 및 삭제를 포함하므로 2-Way 병합기법에 비해 강력하다[5].

(a) 2-Way 방식 (b) 3-Way 방식

(그림 1) 2-Way 방식과 3-Way 방식

2.2 문자적 병합기법

문자적 병합기법은 소프트웨어 산출물을 그 의미나 문법에 관계없이 단순히 텍스트 파일 또는 이진 파일로 취급하여 병합하는 방식으로써, 프로그램 뿐 아니라 문서와 데이터 파일 및 기타 텍스트 개체의 병합도 가능하다. 병합 시 서로 다른 파일 간의 비교단위는 문자 또는 문자열, 블록, 라인 등이 될 수 있으며 현재 라인(Line) 기반 방식이 가장 많이 사용되고 있다. 라인 기반 문자적 병합을 통해 병합된 파일내용은 정확성 및 효율성면에서 뛰어나지만 문법적 오류 및 의미적(실행행위) 오류를 발생시킬 수 있으므로 안전하지 않다[6].

2.3 문법적 병합기법

문법적 병합기법은 소프트웨어 산출물 중 프로그램을 Parse Tree, Abstract Syntax Tree, Attributed Graph 등으로 표현하고, 프로그램 병합 시 프로그램의 문법적 요소(token)만을 비교하고 병합하는 방식이다. 문자적 병합과 달리 문법적 요소만을 비교하고 병합하므로 문법에 어긋나지 않도록 병합할 수 있으며, 주석문 및 공백, tab 등과 같은 부분들은 무시한다. 일반적으로 사용되는 라인단위 문자적 비교 기법을 사용할 때보다 문법적 요소에 대해서 더 정밀한 비교가 가능하다[7].

2.4 의미적 병합기법

문법적 병합기법의 경우, 프로그램에 대한 병합 시 문법적 구조를 유지할 수 있으나 여전히 실행행위를 모두 보존하면서 병합되는지에 대해 보장할 수 없다. 실행행위에 대한 병합을 위해서는 Denotational Semantics, Program Dependence Graph, Program Slicing 등과 같은 복잡한 수학적 형식을 사용한다. 특히, 그래프 기반의 병합기법은 그래프 표현을 통해 변수 정의와 사용에 대한 명확한 링크를 유지한다. 그래서 실행행위에 관련된 의미적 병합충돌이 없는 경우에 대해 실행행위를 모두 보존하면서 병합된 결과를 얻을 수 있다. 대표적인 의미적 병합기법으로는 HPR알고리즘[8]과 Semantic Diff[9]등이 있다.

2.4.1 HPR알고리즘

Horitz 등은 제한된 간단한 프로그램 요소를 가지고 PDG (Program Dependence Graph)을 기반으로 실행행위에 관련된 의미적 병합충돌을 탐지하고 충돌이 발생하지 않는 경우 병합하는 HPR알고리즘(HOROWITZ S, PRINS J, and REPS R, "Integrating Non-Interfering Versions of programs")[8]을 최초로 소개했다.

HPR알고리즘은 원본 프로그램 Base와 그것에 대한 수정본 A와 B가 있을 때 Base, A, B를 각각 PDG로 변환하고, PDG(Base)와 PDG(A)사이 및 PDG(Base)와 PDG(B)사이에서 각각 변화된 부분과 PDG(Base)와 PDG(A), PDG(B) 모두에서 보존된 부분에 대한 후방 프로그램 슬라이스를 구하여 각각 PDG(A), PDG(B)와 PDG(Base)의 서브그래프로 나타내고, 그것들을 결합(Union)하여 새로운 PDG G_M를 만들어 낸 다음 의미적 병합 충돌을 찾고 충돌이 발견되지 않는 경우 G_M을 프로그램 코드로 변환함으로써 병합된 결과를 얻는 방식으로 이루어진다.

HPR알고리즘은 그래프 기반으로 병합이 이루어지므로 제한된 가정 하에서 실행행위에 관련된 실행행위에 대한 병합충돌 없이 병합할 수 있는 안전한 기법을 제공한다. 하지만 HPR알고리즘은 실행행위와 관련된 프로그램 요소만을 그래프화하여 병합함으로써 프로그램이 아닌 일반문서 및 실행행위와 관련되지 않은 프로그램 요소에 대해서는 병합을 지원하지 못하며, 실행행위와 관련된 프로그램 요소에서도 변수의 정의 및 사용과 관계없는 변화에 대해서는 병합된 결과에 반영하지 못하고 있다. 또한, 실행행위 병합충돌이 발생한 경우에 대한 해결방안도 제시되지 않고 있다. 따라서 HPR알고리즘과 같은 의미적 병합기법만으로 병합대상의 원문을 보존하면서 병합하기는 어렵다.

2.4.2 Semantic Diff

의미변화를 탐지하는 유용한 툴로 알려진 Semantic Diff[9]는 HPR알고리즘과는 달리 PDG 대신 Dependence Relation을 이용하여 두 버전 사이의 의미변화를 찾아낸다. HPR알고리즘이 변수 renaming과 같은 의미에 관계없는 변화에 대해서도 병합 충돌로 간주하는 반면, Semantic Diff는 변수의 input과 output만을 비교하고 문법적인 비교는 수행하지 않으므로 변수 renaming과 같은 변화에 영향을 받지 않고 실질적인 의미적 변화만을 탐지한다. Semantic Diff는 의미적인 변화에 대해 다음과 같이 표현한다.

old version .

```
void add (int x) { /* old version */
  if (x != HI) {TOT = TOT + x;
  else TOT = TOT + DEF;}
```

old version의 의존관계 쌍=>

(TOT,TOT), (TOT,x), (TOT,DEF), (TOT,HI)
(x,x), (DEF,DEF), (HI,HI)

new version

```
void add (int x) {/* new version */
  if (x = HI) {TOT = TOT + DEF;
  else TOT = TOT + x;}
```

new version의 의존관계 쌍=>

(TOT,TOT), (TOT,DEF), (TOT,HI)
(x,HI), (DEF,DEF), (HI,HI)

```
new version removes dependences: x on x, TOT on x
new version adds dependences: x on HI
```

결과물 : old version과 new version사이의 의미변화

위의 예에서 x on HI는 변수 x를 정의함에 있어 변수 HI를 사용하는 것을 뜻하며, 결국 새로운 버전은 HI에 대한 x의 의존성을 추가했고, x에 대한 x의 의존성 및 x에 대한 TOT의 의존성을 삭제했다는 사실을 나타낸다.

Semantic Diff를 사용해서 Base에서 A, B 각각이 만들어지면서 일어난 실행행위 변화를 탐지하고 그것들을 서로 비교할 수 있다. 하지만, 그 결과를 가지고 A와 B의 병합 시 실행행위 병합충돌을 탐지하는 방법이 제시되어 있지 않으며, 병합하는 방법도 제시되어 있지 않다.

2.5 오퍼레이션 기반 병합기법

오퍼레이션 기반 병합기법은 3-Way방식의 병합기법을 토대로 원본 프로그램 Base와 그것에 대한 수정본 A와 B가 있을 때 A 및 B에서 Base에 대해 발생된 변화들을 명확한 오퍼레이션으로 모델링하여 표현하고 그것들을 병합 시 활용하는 방식이다. 이 기법은 단순히 오퍼레이션들을 비교하는 것으로 병합충돌을 탐지할 수 있고 그것을 원본에 적용하여 병합할 수 있는 단순하면서도 효과적인 결과를 얻을 수 있는 병합기법이다.

예를 들면, 원본 프로그램 Base에서 변화된 프로그램 A와 B가 있을 때, 프로그램 A는 프로그램 Base의 Math안에서 프로시져 fac을 호출하는 부분을 추가하는 오퍼레이션 AddInvocation(Math,fac)을 가지고, 프로그램 B는 프로그램 Base에서 fac을 프로시져가 아닌 함수로 바꾸는 오퍼레이션

ProcTofunc(fac)을 가진다면 오퍼레이션을 비교하는 것만으로 의미적 충돌이 발생함을 알 수 있다. 하지만 오퍼레이션 기반 병합기법은 오퍼레이션의 설계방식에 따라 긴 Command History를 가지거나 기록되는 단위가 아주 정밀할 경우, 비효율적일 수도 있다[5].

2.6 병합충돌 해결

병합과정에서 일단 병합충돌이 탐지되면 이러한 문제를 해결해야 한다. 해결방법에는 충돌의 종류 및 정확도 수준에 따라 시간이 요구되는 수동 처리에서부터 개발자 상호작용 처리, 완전히 자동화된 병합충돌 해결도구 등을 이용하는 방법들이 있을 수 있다. 또한, 병합충돌의 종류에 따라 서로 다른 병합충돌 해결 전략이 필요하다. 예를 들면 2개의 버전을 병합할 경우, 각각에서 변화된 것들을 특정 순서대로 반영함으로써 충돌없이 병합되는 경우도 있다. 특히 renaming이 적용된 프로그램(Pr)과 modifications이 적용된 프로그램(Pm)을 병합할 경우에는 Pr내에서 rename된 요소를 참조하는 Pm내의 모든 요소가 병합에 반영된 후 Pr내의 rename된 부분이 반영되어야만 충돌을 피할 수 있다. 한편 동일 프로그램 요소에 서로 다른 modification이 적용된 2개의 프로그램을 병합할 경우에는 둘 중의 한 프로그램에 적용된 modification을 무시함으로써 충돌 없이 병합하는 기법도 있다[5].

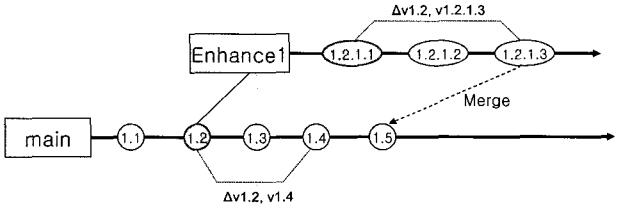
대부분의 병합충돌은 자동화 기법에 의해서 해결 가능하지만 병합 시 어떤 선택을 하느냐가 결과에 큰 영향을 미친다. 그러므로 선택 기준을 사용자가 직접 판단할 수 있도록 상호작용기능이 자동화 기법에 적용된다면 더 나은 병합 결과를 얻을 수 있다.

3-Way 방식의 병합기법에서 주로 사용되는 충돌 해결 전략으로는 사용자에 의해 튜닝 될 수 있는 병합 매트릭스를 사용하는 기법이 있다. 이 병합 매트릭스를 이용한 기법에는 consolidation과 reconciliation 두 가지가 있는데, consolidation은 두개의 프로그램 버전 내에서 수행된 변화가 동일 프로그램 요소에서 일어났을 경우 어느 쪽의 변화내용을 선택할 것인지를 사용자의 상호작용을 통해 결정하는 기법이다. 반면에 Reconciliation은 두 프로그램 버전 내에서 발생한 변화를 사용자의 상호작용에 의해서 결합하는 기법이다[7].

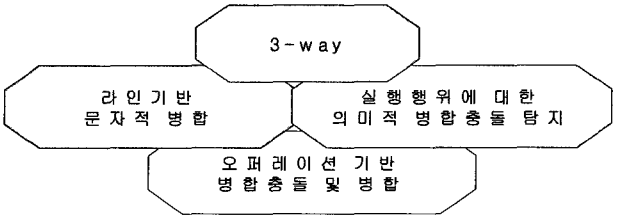
3. 제안하는 병합 기법 (EMPS기법)

분기 및 병합은 어떤 버전이 개발라인의 어느 시점에서 분기되고 주된 개발라인과 분기된 라인에서 각각 새로운 버전이 개발된 후 주된 개발라인으로 병합하는 형식으로 이루어진다(그림 2).

EMPS기법은 분기 및 병합모델에서 분기 시 원본이 보존되는 특성을 이용하여 효율적인 3-Way 병합기법을 채택하고 있다. 본 논문에서는 분기시점의 최초버전을 Base로, 병합시점에서의 병합대상에 대한 버전을 각각 A, B로 지칭하여 설명하고 있다.



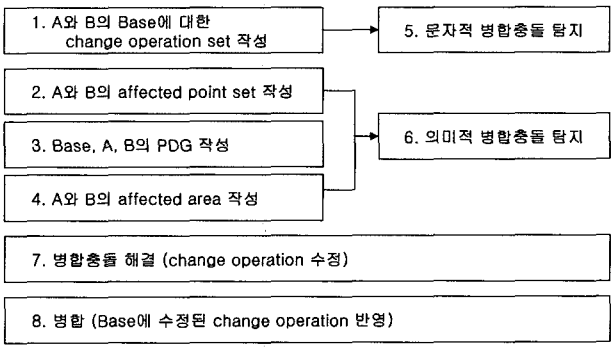
(그림 2) 분기 및 병합 모델



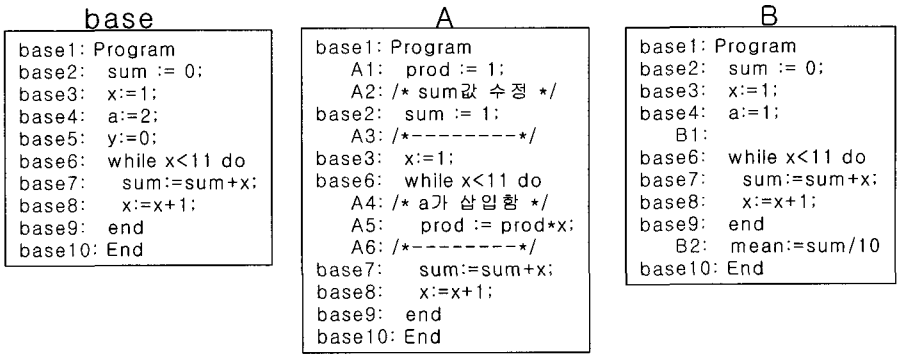
(그림 3) EMPS기법의 특징

EMPS기법은 일반문서 및 실행행위와 관련없는 문자도 모두 병합하기 위해 라인기반 문자적 병합기법을 기본으로 채택하고, 부가적으로 문자적 병합과정에서 효율적인 의미적(실행행위) 병합충돌을 탐지 및 해결방안을 제시하기 위해 프로그램 요소들에서 변수의 정의 및 사용에 대한 링크를 표현하는 PDG와 병합대상 A 및 B에서 Base에 대해 발생된 변화들을 프로그램의 해당 라인에 대한 오퍼레이션으로 모델링 한 것을 사용하여 문자적 병합충돌 및 의미적 병합충돌을 탐지한다. 또한, 병합충돌을 유발한 오퍼레이션과 충돌해결에 대한 가이드라인을 토대로 사용자와 상호작용을 통해 수정된 오퍼레이션을 Base에 반영시켜 병합충돌이 해결된 병합결과를 얻을 수 있도록 했다. 부가적으로, 컴파일러 등에 의해 탐지될 수 있는 문법적 충돌에 대해서는 고려하지 않았다.

EMPS기법은 (그림 4)와 같이 8단계의 과정을 가진다. (1) Base에 대한 A와 B 각각의 변화내용을 이용하여 A와 B를 병합하기 위해 Base에 대한 A와 B 각각의 변화내용을 각각 라인별 수정, 삭제, 삽입 오퍼레이션으로 작성하여 COS(Change Operation Set)를 만들고, (2) 의미(실행행위)에 변화를 주는 COS의 부분집합 APS(Affected Point Set)



(그림 4) EMPS기법의 개요



(그림 5) 태그를 포함한 파일 내용

를 구한다. (3) Base, A, B에 대해 실행행위에 관련된 라인을 가지고 PDG를 만들고, (4) APS와 PDG를 이용하여 Base에 대한 A, B각각의 변화의도에 영향을 줄 수 있는 영역인 AA(Affected Area)를 구한다. (5) COS에서 같은 라인에 대한 오퍼레이션 존재유무를 조사하여 문자적 병합충돌을 탐지하고, (6) APS와 AA를 이용하여 의미적(실행행위) 병합충돌을 탐지한다. (7) 병합충돌을 유발한 오퍼레이션에 대해 충돌위치 및 범위와 병합충돌 해결에 대한 가이드라인을 가지고 해당 오퍼레이션을 수정하여 병합충돌을 해결한다. (8) 수정된 오퍼레이션을 Base에 적용하여 병합된 결과를 산출한다.

3.1 가 정

EMPS기법은 병합하는 대상이 원본에 대해 무엇이 변화되었는지 구별하고, 병합충돌 발생시 그 위치를 표시할 수 있도록 하기위해 프로그램의 각 라인에 그것을 신규로 작성한 버전 및 라인번호를 나타내는 고유한 태그(Tag)를 지정하여 저장할 수 있도록 하는 에디터가 지원된다고 가정한다.

(그림 5)는 에디터에서 작성되어 저장된 프로그램 Base, 프로그램 A, 그리고 프로그램 B의 내용이며 각각에서 사용된 태그는 별칭으로 명명되었다. 프로그램 Base내의 태그 Base2가 A에서 Base2와 같이 Base의 특정 라인에 대해 수정된 경우에는 태그 번호가 바뀌지 않음을 알 수 있다. 또한 Base의 어떤 특정 라인의 내용을 다른 위치로 옮길 경우에는 에디터에서 Base의 라인 및 태그를 삭제하고 새로운 라인을 생성되는 방식으로 처리된다.

EMPS기법은 라인단위 문자적 병합기법을 토대로 의미적 병합충돌을 탐지하기 위해서는 한 라인에 하나 이하의 선언문 및 제어문이 존재한다고 가정하며 의미적 병합충돌 탐지 범위는 한 프로시져 내의 배열 및 포인터변수를 제외한 단순한 변수와 선언문 및 제어문으로 제한한다.

3.2 병합 충돌 탐지 및 병합과정

3.2.1 COS(Change Operation Set) 작성

EMPS기법에서 COS는 Base로부터 A와 B가 만들어지기 위해 Base의 각 라인에 대해 일어난 오퍼레이션(O_{tag})들의 집합이며 오퍼레이션의 구조는 다음과 같이 정의한다.

$COS = \{O_{tag}\}$

O_{tag} = **태그(tag : 라인식별자)+오퍼레이션 종류+위치+변경 후 내용+변경 전 내용**

오퍼레이션의 종류는 라인별 수정, 삭제, 삽입 오퍼레이션으로 나눈다. A를 위한 Base의 COS는 $\Delta S_{Base,A}$ 로, B를 위한 Base의 COS는 $\Delta S_{Base,B}$ 로 표시한다.

COS를 만들기 위한 절차를 간략하게 설명하면 다음과 같다.

step 1 : BASE와 A, BASE와 B를 각 라인별로 비교한다.
step 2 : 태그 값이 같을 경우에는 내용을 비교하여 같으면 변화가 없는 경우이므로 다음 라인으로 진행한다.
step 3 : 태그 값이 같으나 내용이 다를 경우에는 일반적으로 수정 오퍼레이션이 작성되고 변경 전 내용과 변경 후 내용을 함께 저장한다.
step 4 : 만약, 그 내용변화가 선언문이고, 왼쪽 변수 값이 서로 다르다면 삭제되어 다시 생성된 것으로 간주하고 삭제 및 삽입 operation을 생성한다.
step 5 : 삽입 operation을 제외한 모든 operation은 위치정보에 자신과 같은 태그 값을 저장한다.
step 6 : 태그가 다를 경우 (A또는 B에서 생성된 태그일 경우)
- 삽입 오퍼레이션이 작성되고 삽입위치는 Base버전의 내용으로부터 대신 삽입된 해당라인의 태그를 저장한다.
- 삽입위치가 Base버전의 마지막이 될 경우에는 '-1'을 저장한다.
- 태그 Base1과 태그 Base2사이에 삽입이 되었을 경우에는 삽입위치가 태그 Base2가 된다.
- A에서 Base의 해당라인이 존재하지 않을 경우는 삭제 오퍼레이션을 작성한다.

step 7 : step 1로 이동한다.

상기 절차를 통해 (그림 5)에 대해 만든 $\Delta S_{Base,A}$은 (그림 6)과 같다.

```
{ O_A1=(A1,삽입,base2,"prod:=1;",),
  O_A2=(A2,삽입,base2," /* sum값 수정 */",),
  O_BASE2=(base2,수정,base2,"sum:=1;", "sum:=0;"),
  O_A3=(A3,삽입,base3," /*----------*/",),
  O_BASE4=(base4,삭제,base4,,"a:=2;"),
  O_BASE5=(base5,삭제,base5,,"y:=0;") }
  O_A4=(A4,삽입,base7," /* a가 삽입함 */",),
  O_A5=(A5,삽입,base7," prod := prod*x;",),
  O_A6=(A6,삽입,base7," /*---------*/",)}
```

(그림 6) A를 위한 Base의 COS($\Delta S_{Base,A}$)

$\Delta S_{BASE,A}$ ={
O_{A1}=(A1,삽입,base2,"prod:=1;",),
O_{A2}=(A2,삽입,base2," /* sum값 수정 */",),
O_{BASE2}=(base2,수정,base2,"sum:=1;", "sum:=0;"),
O_{A3}=(A3,삽입,base3," /*---------*/",),
O_{BASE4}=(base4,삭제,base4,,"a:=2;"),
O_{BASE5}=(base5,삭제,base5,,"y:=0;") }
O_{A4}=(A4,삽입,base7," /* a가 삽입함 */",),
O_{A5}=(A5,삽입,base7," prod := prod*x;",),
O_{A6}=(A6,삽입,base7," /*---------*/",)}

$\Delta S_{BASE,A}$' ={
O_{A1}=(A1,삽입,base2,"prod:=1;,),
O_{BASE2}=(base2,수정,base2,"sum:=1;", "sum:=0;"),
O_{BASE4}=(base4,삭제,base4,,"a:=2;),
O_{BASE5}=(base5,삭제,base5,,"y:=0;"),
O_{A5}=(A5,삽입,base7," prod := prod*x;",) }

$\Delta S_{BASE,B}$ ={
O_{b1}=(B1,삽입,base5," ",),
O_{BASE4}=(base4,수정,base4,"a:=1;","a:=2;"),
O_{BASE5}=(base5,삭제,base5,,"y:=0;"),
O_{b2}=(B2,삽입,base10," mean:=sum/10;",)}

$\Delta S_{BASE,B}$' ={
O_{BASE4}=(base4,수정,base4,"a:=1;","a:=2;")
O_{BASE5}=(base5,삭제,base5,,"y:=0;"),
O_{b2}=(B2,삽입,base10," mean:=sum/10;",)}

(그림 7) A와 B의 APS ($\Delta S_{Base,A}$", $\Delta S_{Base,B}$")

3.2.2 APS(Affected Point Set) 작성

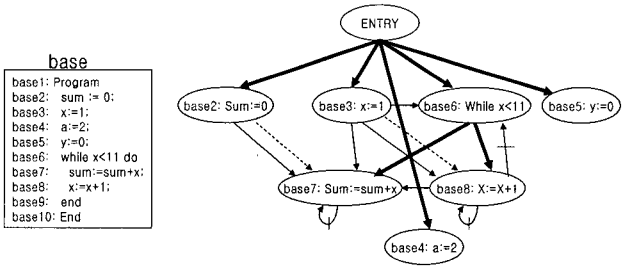
EMPS기법에서는 COS중 프로그램 실행행위에 영향을 주는 오퍼레이션들을 모아 놓은 것을 APS(Affected Point Set)로 지칭한다. 프로그램의 실행행위에 영향을 주는 오퍼레이션들은 COS의 각 오퍼레이션 내부에 포함되어 있는 변경 전 내용과 변경 후 내용에 대한 정보를 확인함으로써 구분되어 진다. 일반적으로 오퍼레이션 내부의 변경 전 및 변경 후의 내용이 명령문이나 제어문일 경우에는 실행행위에 영향을 주는 오퍼레이션이 된다. 그러나 수정 오퍼레이션일 경우, 변경 전과 후의 라인내용이 명령문이나 제어문이더라도 "x : =1"을 "x : =1 /*주석*/"로 바꾸거나 "x : =1"로 바꾸는 등과 같이 실행행위가 변화되지 않는 경우는 실행행위에 변화를 주는 오퍼레이션 선택에서 제외된다. Base에서 수정된 프로그램 A, B 각각의 Base에 대한 APS은 COS의 부분집합으로써 $\Delta S_{Base,A}'$, $\Delta S_{Base,B}'$로 명시되어 (그림 7)과 같이 표현한다.

3.2.3 PDG (Program Dependence Graph) 작성

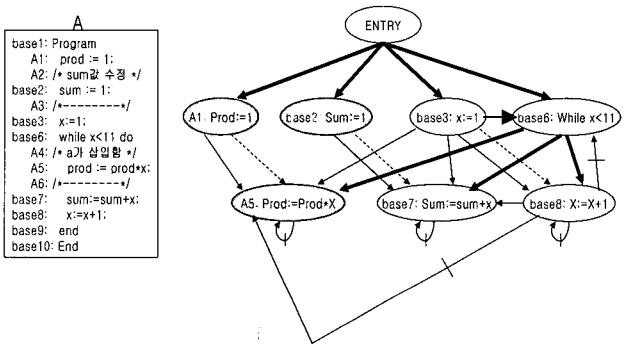
PDG는 HPR알고리즘에서 의미적 병합을 위해 사용된 방향성이 있는 그래프로써 프로그램의 각 구성요소 간의 제어 및 데이터 의존성을 표현한다. PDG를 구성하는 점들은 의미와 관계된 프로그램 요소(선언문 및 제어문)을 나타내며, 점들을 연결시키는 화살표(Edge)들은 프로그램 요소에서 나타나는 변수에 대한 정의 및 사용의 관계(flow edge)를 나타내거나, 같은 변수에 대한 정의 순서(Def-order edge)를 나타내거나, 제어문과 제어에 의해 실행되는 다른 프로그램 요소간의 관계(control edge)를 나타낸다. 따라서 프로그램 내에서 어느 한 요소를 중심으로 그것의 실행행위에 영향을 주거나(후방 프로그램 슬라이스) 그것의 실행행위에 의해 영향을 받는(전방 프로그램 슬라이스) 다른 프로그램 요소들은 PDG의 서브그래프로써 표현될 수 있다. 예를 들면, (그림 8)에서 Base8번 태그를 가진 점이 나타내는 프로그램 요소에 영향을 주는 요소는 Base3와 Base6이 되며, Base8이 영향을 미치는 프로그램 요소 Base7이다.

EMPS기법에서는 BASE, A, B각각에 대해 각 라인에서 선언문과 제어문이 있는 라인만을 선택하여 점으로 만들고 그것으로 구성된 PDG로 작성하고 각 라인을 구별하는 태그를 포함시킨다. Base 및 A에 대한 PDG는 (그림 8), (그림 9)과 같다.

의미적 병합기법인 HPR알고리즘은 실행행위에 대한 병합



(그림 8) 프로그램 Base에 대한 PDG



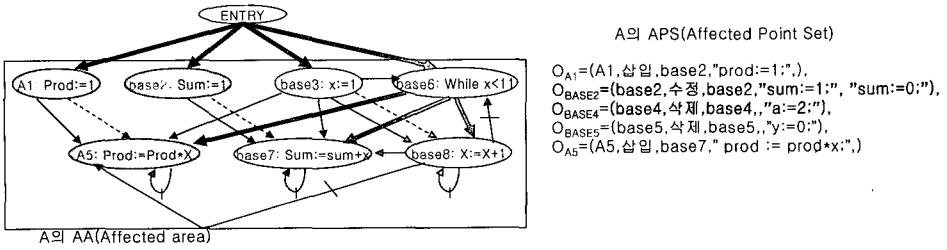
(그림 9) 프로그램 A에 대한 PDG

충돌을 PDG를 이용하여 탐지하고 병합충돌이 없을 경우 PDG의 서브그래프를 결합하는 방식으로 병합하지만 EMPS기법에서는 문자적 병합기법을 기본으로 하므로 의미적(실행행위) 충돌발생여부를 확인하기 위해서만 PDG를 이용한다.

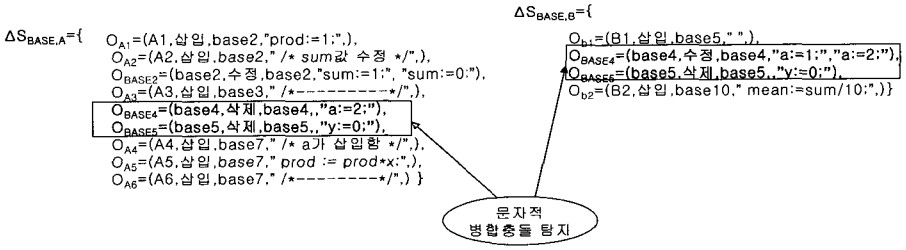
3.2.4 AA(Affected Area) 작성

AA는 APS가 영향을 미치거나 영향을 받을 수 있는 영역 즉, 실행행위에 대한 변화의도에 영향을 받는 범위를 PDG를 이용하여 찾은 영역이다. APS은 실행행위에 관련된 선언문 및 제어문에 대한 오퍼레이션들을 의미하므로 그것에 영향을 미치거나 영향을 받는 영역을 찾으려면 선언문이나 제어문에 쓰인 변수의 정의와 사용의 관계를 찾아야 한다.

따라서 PDG내에서 APS가 나타내는 해당 점을 중심으로 전방 프로그램 슬라이스를 찾아 APS에 의해 영향을 받는 다른 프로그램 요소를 찾는다. 이렇게 찾은 전방 프로그램 슬라이스와 APS 모두는 실행행위에 대한 변화의도를 표현



(그림 10) A의 AA(Affected area)



(그림 11) 문자적 병합충돌 탐지

하므로 그것에 대한 후방 프로그램 슬라이스를 찾아 변화의도에 영향을 미치는 PDG내의 점도 찾아낸다. 이렇게 찾은 APS을 중심으로한 전방 프로그램 슬라이스 및 후방 프로그램 슬라이스를 포함하여 AA라고 지칭한다.

A의 APS에서 수정 및 삽입 오퍼레이션은 PDG(A)의 점에 대응되며, 삭제에 관한 것은 PDG(Base)에서 찾을 수 있다. 따라서 수정 및 삽입 오퍼레이션에 대해서는 PDG(A)내에서 그것을 중심으로 전방 프로그램 슬라이스 및 후방 프로그램 슬라이스를 구하여 찾는다. 그러나 삭제 오퍼레이션에 대응하는 점은 PDG(A)에서 찾을 수 없으므로 PDG(Base)에서 찾아 그것의 outgoing flow edge에 연결된 점들 중 PDG(A)내의 점에 일치하는 최초의 점을 찾고, PDG(A)내에서 그 점을 중심으로 전방 프로그램 슬라이스 및 후방 프로그램 슬라이스를 구한다.

A의 AA는 PDG(A) 내에서 해당 점이 속하는 영역으로 (그림 10)과 같이 나타낼 수 있다. A와 APS에서 삭제 오퍼레이션(O_{Base4},O_{Base5})에 대응되는 점을 (그림 8) PDG(Base)내에서 찾았을 때 그것의 전방 프로그램 슬라이스에 대응되는 점을 PDG(A)에서 찾고( ), 수정 및 삽입 오퍼레이션(O_{A1}, O_{Base2}, O_{A5})에 대응되는 점을 PDG(A)에서 찾는다(A1, Base2, A5). 그리고 이것의 전방 프로그램 슬라이스 및 후방 프로그램 슬라이스를 찾으면 A의 AA는 {A1, A5, Base2, Base3, Base6, Base7, Base8}를 포함하는 (그림 10)과 같은 PDG(A)내의 영역이 됨을 알 수 있다.

3.2.5 문자적 병합충돌 탐지

문자적 병합충돌은 수정된 버전 A와 B가 원본 Base의 같은 라인에 대한 COS를 가질 경우 발생한다. (그림 5)의 Base, A, B에 대해 A와 B의 병합 시 (그림 11)과 같이 서로의 COS에 대한 비교를 통해 문자적 병합충돌을 탐지할 수 있다.

3.2.6 의미적 병합충돌 탐지

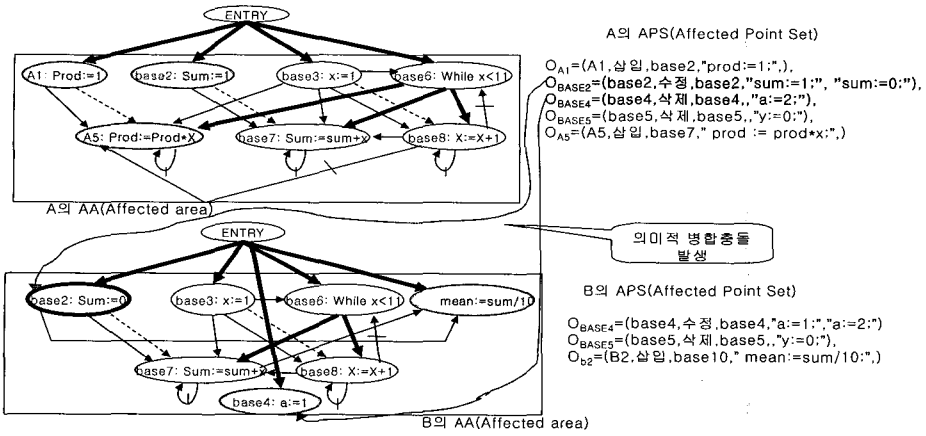
의미적 병합충돌은 기저 버전 Base에 대해 수정된 버전 A와 B가 있을 때, B의 실행행위에 대한 변화의도에 영향을 미칠 수 있는 영역(AA)에 A의 의미와 관련된 COS 즉, APS가 영향을 주거나 그 반대의 경우 발생한다.

예를 들어 (그림 5)의 경우, (그림 12)와 같이 A에서 Base2에 대한 실행행위와 관련된 수정이 일어났고 Base2가 B의 AA에 있으므로 의미적 병합충돌이 발생했음을 알 수 있다.

의미적 병합충돌에 대한 탐지과정은 다음과 같이 세 가지 단계로 나뉜다.

첫째, A의 APS에 있는 오퍼레이션에 해당된 프로그램 요소가 B의 AA에 나타나는지 조사한다. 이런 경우, A와 B의 병합 충돌은 3가지 이유로 발생한다. 1) B에서는 그것에 대해 다른 방법으로 수정을 하거나, 2) 그것이 변화되기 전의 상태를 사용하는 프로그램 요소를 삽입하거나 수정한 경우, 또는 3) 삽입되거나 수정된 프로그램 요소가 영향을 미치는 다른 프로그램 요소를 A가 수정 및 삭제한 경우이다. (그림 12)의 예에서 O_{Base2}는 2)의 경우에 해당되어 A의 O_{Base2}에 대응되는 프로그램 요소인 Base2가 B의 AA에 나타나므로 의미적 병합충돌이 일어남을 알 수 있고, O_{Base4}에 관련된 프로그램 요소 Base4가 B의 AA에 나타나므로 3)의 경우에 해당되므로 병합충돌이 발생함을 알 수 있다. 또한, B의 수정 및 삭제 오퍼레이션과 관련된 Base4, Base5가 A의 AA에 나타나지 않으므로 첫 번째 단계에서는 그것과 관련된 병합충돌이 탐지되지 않는다.

둘째, A가 새로운 프로그램 요소를 삽입하거나 기존의 프로그램 요소를 수정하여 그것을 참조하는 다른 프로그램 요소의 실행행위에 간접적으로 영향을 미치게 되므로 병합 충돌이 발생한다. 이런 경우에는 병합대상 A에서 삽입되거나 수정된 프로그램 요소가 영향을 미치는 다른 기존의 프로그램 요소를



(그림 12) 의미적(실행행위) 병합충돌 탐지

찾기 위해 PDG(A)에서 그것의 전방 프로그램 슬라이스에 속한 점을 구하고, 그것이 B의 AA에 있는지 검색하여 일치하는 프로그램 요소를 찾으므로써 병합충돌을 탐지한다.

(그림 12)의 예에서는 A의 삽입 오퍼레이션인 O_{A1}, O_{A5}와 관련된 A1과 A5에 대한 PDG(A)내의 전방 프로그램 슬라이스에서 A1과 A5 이외의 점이 존재하지 않으므로 병합충돌이 탐지되지 않으며, 수정 오퍼레이션인 Base2는 그것을 중심으로 찾은 전방 프로그램 슬라이스 내에 Base7이 있고 Base7이 B의 AA에 나타나므로 병합충돌이 발생했음을 탐지할 수 있다. 그러나 Base2는 첫째 단계에서 이미 병합충돌이 탐지되었으므로 더 이상 병합충돌 탐지에 사용하지 않도록 한다. 또한, B의 수정 및 삽입 오퍼레이션과 관련된 Base4, Base5에서 PDG(B)내의 전방 프로그램 슬라이스를 찾으면 A에서 삽입된 요소를 제외한 점을 찾을 수 없으므로 병합충돌이 탐지되지 않는다.

셋째, A에서 어떤 변수에 값을 할당하는 프로그램 요소를 삽입했을 때, B에서는 그 변수에 대한 기존 값을 사용하는 프로그램 요소를 삽입하거나 수정한 경우이므로 병합충돌이 발생한다. 이런 경우에는 병합대상 A에서 삽입된 프로그램 요소에 있는 변수에 대한 정의 순서를 나타내는 Incoming Def-order Edge(점선모양)에 연결된 프로그램 요소(Da)를 PDG(A)에서 찾는다. 만약, B의 AA에서 Da의 Outgoing Flow Edge에 연결된 프로그램 요소가 있을 경우, A에서 삽입된 요소와 B에서 Da를 참조하는 요소(Da1)들을 병합된 결과 프로그램 내에서의 위치를 비교하여 삽입된 요소가가 Da보다 앞에 위치하게 된다면 충돌이 발생한다.

(그림 12)의 예에서 A의 삽입 오퍼레이션과 관련된 프로그램 요소 A2, A5에서 서로 간에 연결된 Def-order Edge이외에는 다른 프로그램요소와 연결된 Def-order Edge가 존재하지 않으므로 병합충돌이 발생하지 않는다. 또한, B의 삽입 오퍼레이션과 관련된 프로그램 요소 B2와 연결된 Def-order Edge가 존재하지 않으므로 병합충돌이 발생하지 않는다.

3.2.7 병합 충돌 해결

EMPS기법은 병합충돌을 발생시킨 오퍼레이션에 대해 사용자와의 상호작용을 통해 수정 또는 삭제하거나 새로운 오퍼레이션을 삽입함으로써 병합충돌을 해결한다.

A와 B의 병합 시 발생한 병합충돌을 해결하기 위해 사용자에게 도움이 되는 5가지 가이드라인은 다음과 같다.

첫째, 동일 라인에 대해 A, B 모두에 같은 내용을 삽입 또는 수정, 삭제한 경우에는 병합 시 각 해당 오퍼레이션을 1개씩만 적용하도록 한쪽 오퍼레이션을 삭제함으로써 해결가능 하다.

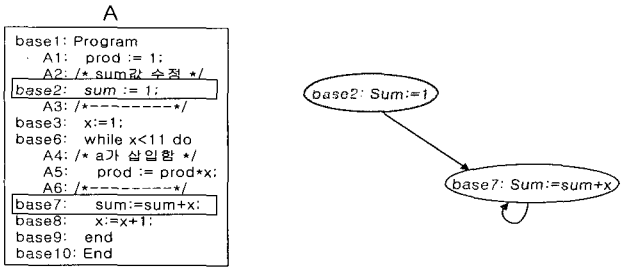
둘째, A에서는 수정을, B에서는 삭제를 했을 경우에는 그 내용이 의미(실행행위)에 영향을 미치지 않을 때에는 1) 삭제 오퍼레이션을 지우고 수정 오퍼레이션을 선택하거나 2) 두 오퍼레이션들을 모두 삭제하고 새로운 수정 오퍼레이션을 삽입함으로써 해결할 수 있다.

셋째, A와 B 모두에 서로 다른 내용을 삽입한 경우도 충돌이 일어났다고 볼 수 있으나 의미(실행행위)에 영향을 미치지 않을 경우는 순서에 관계없이 모두 수용한다. 만약 의미(실행행위)에 영향을 미치는 경우는 같은 변수에 대해 한쪽은 정의를 다른 한쪽은 그 변수를 사용했을 때에 사용에 대한 요소를 병합 시 먼저 위치시키도록 한다. 그리고 병합하려는 두 버전에서 같은 변수에 대해 사용하는 프로그램 요소를 삽입하거나, 서로 다른 변수에 대한 정의를 하는 프로그램 요소를 삽입했을 경우 서로간의 변화된 의미에 영향을 주지 않으면 의미적 병합충돌이 발생하지 않으므로 모두 수용한다.

넷째, A와 B 모두에서 프로그램 항목을 다른 내용으로 수정한 경우, 의미에 관계없는 수정이라면 사용자가 관계된 오퍼레이션들을 모두 삭제하고 모든 내용을 반영한 새로운 수정 오퍼레이션을 작성한다.

다섯째, 의미적(실행행위) 병합충돌이 발생한 경우, 관련 프로그램 요소가 정의문일 때 정의하는 변수에 대해 적절히 다른 이름으로 바꾸고(renaming), 그것을 참조하는 다른 프로그램 요소도 그에 따라 변수이름을 바꾼 형태로 만들어 해당위치에 추가로 삽입하는 오퍼레이션을 만든다.

사용자가 병합 충돌이 발생한 오퍼레이션과 그것의 AA에서 해당부분을 확인하여 COS를 수정하고, 더 이상의 병합충돌이 없다면 그것을 자동으로 Base에 적용시킬 수 있다. 의미적 병합충돌을 발생시키는 오퍼레이션에 대해서는 사용자에게



(그림 13) 의미적 병합충돌 관련 프로그램 요소

AA내의 병합충돌이 발생한 부분의 전방 프로그램 슬라이스에 속하는 프로그램 구성요소를 (그림 13)와 같이 프로그램 코드로 시각적으로 제시하여 의미적 병합충돌을 추가로 일으키지 않는 범위 내에서 해결방안을 강구할 수 있도록 한다.

3.2.8 병합

병합충돌 해결에 대한 4가지 경우를 가이드라인을 토대로 모든 병합충돌이 해결된 COS와 서로 다른 내용을 같은 위치로 삽입한 것에 대한 오퍼레이션의 적용 순서 정보를 기저버전 Base에 반영하면 (그림 14)과 같이 병합된 결과를 얻어낼 수 있다. (그림 14)의 예에서 문자적 병합충돌이 일어난 O_{Base4}, O_{Base5}는 앞에서 제시한 가이드라인에 따라 O_{Base5}는 A, B 모두에서 삭제 오퍼레이션이므로 둘 중 하나만 남겨두고, O_{Base4}는 A에서는 수정 오퍼레이션이고 B에서는 삭제 오퍼레이션이므로 가이드라인에 따라 실행행위에 관련된 오퍼레이션이므로 사용자가 실행행위에 대한 판단을 하고 어느 한쪽을 없애주어야 한다. 이 경우, A, B모두에서 O_{Base4}는 프로그램 실행 후 변수 a에 대한 최종 결과 값에만 영향을 미치고 다른 프로그램 요소가 변수 a값을 참조하지는 않으므로 삭제 오퍼레이션을 삭제하고 수정 오퍼레이션만을 선택하는 방법으로 해결될 수 있다.

(그림 14)의 예에서 의미적 병합충돌을 발생시킨 A의 오퍼레이션인 O_{Base2}에 대한 병합충돌 해결방법은 다섯 번째 가이드라인을 따른다. O_{base2}에서 정의하고 있는 변수값 sum에 대해 sum_1로 이름을 바꾸고 sum값을 참조하는 A내의 다른 프로그램 요소(그림 13)에 대해서도 sum을 sum_1을 바꾼 형태의 프로그램 요소를 (그림 14)과 같이 해당위치에 삽입하는 오퍼레이션을 추가함으로써 B의 프로그램 실행행위에 영향을 주지 않고 A의 실행의도를 보존할 수 있다.

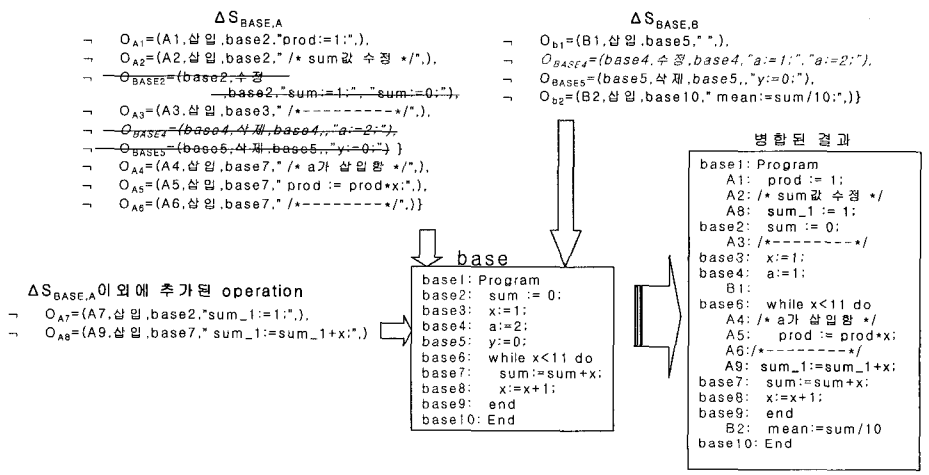
4. 의미적(실행행위) 병합결과에 대한 타당성 검증

의미적(실행행위) 병합충돌은 두 병합대상의 실행행위가 병합 후에 병합된 결과에서 보존되지 못했을 경우 발생된다. 이때, 두 병합대상 서로의 실행행위에 영향을 미치는 부분에 대해 서로 영향을 미치지 못하도록 병합충돌을 일으킨 쪽을 수정하는 것에 의해 의미적(실행행위) 병합충돌을 해결할 수 있다. 이때, 수정된 결과가 수정 전의 그것과 실행행위가 일치하는 형태로 수정되어야 한다.

정의 : 임의의 state σ에서 프로그램 P와 Q가 초기화될 때 둘 다 분기하거나, 둘 다 모든 변수 값에 대해 같은 최종 값을 가지고 끝날 때 "프로그램 P와 Q가 *Strongly equivalent*하다."

HPR알고리즘에서는 임의 두 프로그램을 PDG형태로 변환했을 때, 같은 형태의 PDG가 된다면 두 프로그램이 Strongly equivalent하다고 하고 있으며, 두 프로그램의 PDG가 같다면, 위의 정의에 따라 임의 상태에서 두 프로그램이 초기화되었을 경우 프로그램 안의 각 변수는 같은 최종 값을 가지고 끝난다고 정의하고 있다[10]. 따라서 두 프로그램의 실행행위가 동일하다고 간주하려면 모든 변수의 값에 대해 동일한 최종 값을 가져야 함을 알 수 있다.

(그림 14)의 예에서 의미적 병합충돌을 해결하기 위해 EMPS기법의 다섯 번째 가이드라인을 이용하여 의미적 병합충돌을 해결했다. 그 결과로 COS의 수정에 의해 실행행위가 변하지 않아 2차적으로 발생할 수 있는 의미적 오류를 방지하고 있다.



(그림 14) 병합충돌 해결 및 병합

(그림 14)의 예에서는 병합충돌을 유발한 프로그램 A에 대해 충돌을 해결하면서 수정된 결과의 실행행위가 수정전의 그것과 일치할 수 있도록 하기위해, A의 의미적 병합충돌 발생 오퍼레이션 O_{Base2}에 관련된 프로그램 요소에서 정의되는 변수 sum에 대해 병합대상 프로그램(B)에 없는 변수명인 sum_1로 변환하고 sum을 참조하는 다른 프로그램 요소들에 대해서도 sum을 sum_1로 변환하여 새로운 프로그램 요소로 만들어 삽입하는 오퍼레이션을 추가하는 방식을 사용했다. 실제로 이렇게 하는 방식은 실행행위 면에서 sum이 포함된 A안의 모든 프로그램 요소에 대해 sum을 sum_1로 변수이름만 바꾼 효과를 나타낸다. 이렇게 함으로써 B에 대한 실행행위가 보존되고 수정전후의 A에서의 실행행위도 보존될 수 있다.

A에서의 변수집합(Var_set(A))={prod, sum, x}
by (그림 5)
B에서의 변수집합(Var_set(B))={sum, x, a, mean}
by (그림 5)
A'에서의 변수집합(Var_set(A'))={prod, sum_1, x}
by (그림 14) 및 다섯 번째 가이드라인
병합된 결과에서의 변수집합(Var_set(M))={prod, sum_1, sum, x, a, mean} by (그림 14)

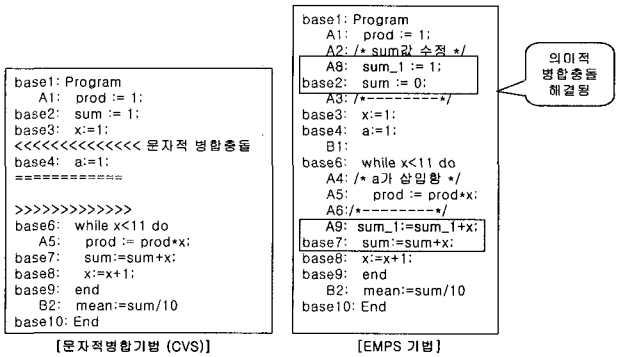
(그림 5)의 병합대상 A, B에서 각각 프로그램에서 사용되는 변수목록 및 병합충돌 해결을 위해 A를 수정하여 만든 A′ 그리고, 병합된 결과에서의 변수 목록은 위와 같다. 실행행위에 대한 정의에 따라 어떤 상태에서 병합된 결과에서의 각 변수의 최종 값과 A′와 B각각에서 각 변수에 대한 최종값이 같다면 A와 B의 실행행위가 병합된 결과에 보존됨을 확인할 수 있고, A와 A′ 각각의 변수에 대한 최종 값이 같다면 A를 수정하여 만든 A′가 A의 실행행위를 보존하고 있다고 확인할 수 있다.

임의 상태에서 변수에 대한 최종 값이 같은지 비교하기 위해서는 앞에서 설명한 바와 같이 PDG를 이용하여 해당 변수의 최종 값을 정의하는 프로그램 요소를 중심으로 그것이 참조하는 다른 프로그램 요소 즉 후방 프로그램 슬라이스를 비교하면 된다. 따라서 Var_set(M)내의 모든 변수에 대해 PDG(M)에서 각 변수의 최종값을 정의하는 점의 후방 프로그램 슬라이스를 구하고, 각 변수가 속하는 A′ 및 B에서도 각각의 PDG에서 해당 변수의 최종 값을 정의하는 점에 대한 후방 프로그램 슬라이스를 구하여 일치하는지 비교하는 것으로 병합결과에 대한 타당성을 검증할 수 있다.

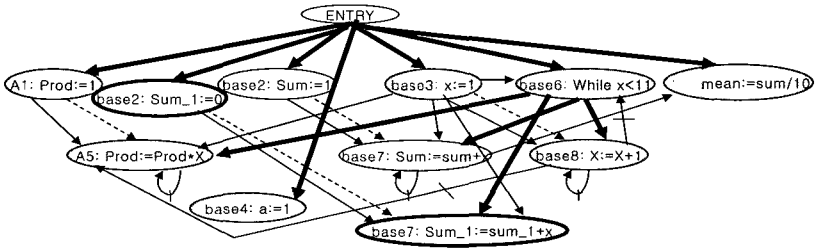
(그림 14)과 같이 병합충돌을 해결할 경우 병합된 결과 프로그램 코드를 PDG로 변환하여, PDG(M)에서 Var_set(M)내의 각 변수의 최종 값에 대한 정의문을 중심으로 한 후방 프로그램 슬라이스와 Var_set(A′)내의 각 변수의 최종 값을 중심으로 PDG(A′)에서 후방 프로그램 슬라이스를 구한 것이 일치하고, Var_set(B)내의 각 변수의 최종 값을 중심으로 PDG(B)에서 후방 프로그램 슬라이스를 구한 것과도 일치되므로 병합된 결과에서 각 A'와 B의 실행행위가 보존되었음을 알 수 있다. 또한 A′는 A의 sum변수가 sum_1변수로 단순히 renaming된 것이므로 sum과 sum_1을 포함한 A, A′내의 모든 변수의 최종 값이 같고, 그 이외의 변수들에 대해서도 최종 값이 같으므로, A와 B의 실행행위가 병합된 결과에 보존되고 있음이 확인된다. 따라서 (그림 14)에서 병합충돌이 해결된 결과가 타당함을 알 수 있다.

5. 타 기법과 비교평가

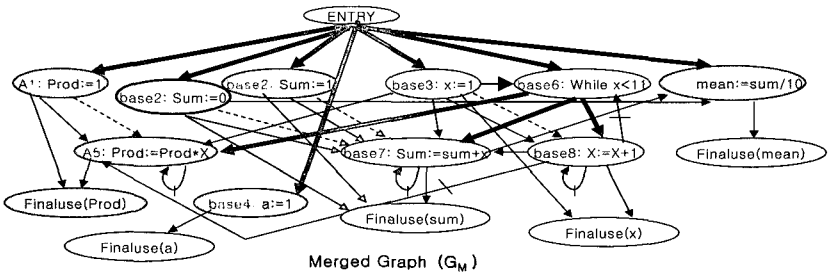
CVS[10]와 같은 기존의 상업적 버전관리 도구에서 지원되는 병합 도구에서는 3-Way방식의 라인기반 문자적 병합기법을 사용하고 있다. EMPS기법에서도 두 병합대상 버전(A, B)이 원본(Base)에 대한 변화를 모두 수용하는 3-Way 방식의 라인기반 문자적 병합기법을 사용하고 있고, 두 버전에서 같은 라인에 대해 문자적 변화가 있었을 경우에는 병합충돌이 발생한다고 본다. 이렇게 병합충돌이 발생한 경우에는 (그림 16)과 같이 라인별로 문자적인 병합충돌 위치를 제시하고 충돌위치에 대한 사용자의 수정을 통해 병합충



(그림 16) 문자적 병합기법(CVS)과 EMPS기법의 병합결과 비교



(그림 15) (그림 14)에서의 병합된 결과에 대한 PDG(M)



(그림 17) HPR알고리즘에 의한 A와 B의 병합결과 PDG

〈표 1〉 병합기법 비교평가

비교기준	EMPS기법	CVS[10]	HPR알고리즘[8]	Semantic Diff[9]
병합시 비교대상	3-way	3-way	3-way	2-way
충돌탐지 및 병합대상	문자/의미(실행행위)	문자	의미(실행행위)	의미(실행행위)
충돌탐지 단위	라인 및 선언문,제어문	라인	선언문, 제어문	변수들의 input output relation
의미적 충돌탐지 범위	한 프로시져 내의 선언문 및 제어문	탐지 않됨	한 프로시져 내의 선언문 및 제어문	한 프로시져 내의 선언문 및 제어문
병합충돌 해결방안	병합충돌 위치 및 충돌을 유발한 변경 내용과 해결을 위한 가이드라인제시	병합결과 내에서 문자적 충돌 위치 제시	해결방안이 제시되지 않음	input,output의 관계가 변경된 변수명 제시
병합자동화 정도	충돌해결 : 상호작용 처리 병합 : 자동처리	충돌해결 : 상호작용 처리 병합 : 자동처리	병합 : 자동처리	병합 : 수동

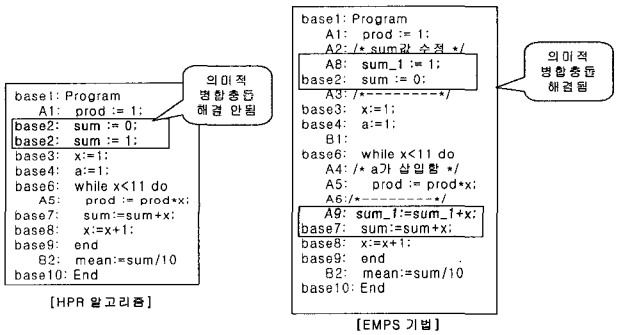
돌 해결 및 병합을 한다. CVS와 같은 문자적 병합기법은 두 버전의 모든 문자를 포함하여 병합했지만 병합된 결과에서 의미적(실행행위) 병합충돌이 발생되었다. 그러나 EMPS 기법은 (그림 16)에서와 같이 A와 B에 있는 문자를 모두 병합하면서 의미적(실행행위) 병합충돌이 발생한 부분에 대해 병합충돌을 해결한 병합결과를 만들어 낸다.

의미적(실행행위) 병합 알고리즘인 기존의 HPR알고리즘과 EMPS기법은 (그림 5)에서의 Base와 A와 B에 대해 다음과 같이 다른 병합 결과를 만들어 낸다.

HPR알고리즘에서는 (그림 5)에서의 Base와 수정된 버전 A와 B 각각에 대해 A와 B 각각의 Base에 대한 변화된 부분을 나타내는 후방 프로그램 슬라이스와 Base에 대해 보존된 부분에 대한 후방 프로그램 슬라이스를 구하고, 다시 그것들을 union하여 (그림 17)와 같은 병합된 PDG를 얻는다. 이때, (G_A / $AP_{A,\ Base}$)와 (G_B / $AP_{B,\ Base}$)가 G_M 내에서의 (G_M / $AP_{A,\ Base}$)와 (G_M / $AP_{B,\ Base}$)가 일치하지 않으므로 병합된 그래프 G_M에서 의미적 충돌이 탐지된다[6]. HPR알고리즘은 충돌이 일어난 위치 및 변화원인을 탐색하는 기법을 제시하지 않기 때문에 더 이상의 병합은 진행될 수 없다.

HPR알고리즘으로 의미적 충돌이 탐지된 상태의 PDG(그림 17)를 프로그램 코드로 변환하면 (그림 18)의 왼쪽 그림과 같다.

HPR알고리즘이 실행행위와 관련된 프로그램 요소만 PDG의 점으로 표현하고 PDG의 결합을 이용하여 병합에 이용하기 때문에 그 이외의 요소는 병합된 결과에 반영되지 못하는 반면, EMPS기법은 문자적 병합기법을 기반으로 하기 때문에 프로그램내의 병합충돌이 발생하지 않은 모든 내용에 대해서 병합된 결과에 보존된다. 하지만, HPR알고리즘은 (그림 18)에서처럼 A2줄의 /*sum값 수정*/ 및 A3와 A6줄의 /*---------*/과 같이 실행행위와 관계없는 문자에



(그림 18) EMPS기법과 HPR알고리즘 결과 비교

대해서는 병합된 결과에 보존하지 못하고 있다. 또한, 의미적 충돌이 일어난 경우에는 HPR알고리즘에서는 병합충돌 해결이 되지 않은 상태에서 의미적 병합충돌이 잠재된 상태에서 병합되나, EMPS기법에서는 (그림 14)과 같이 해결과정을 거쳐 병합충돌이 해결된 결과(그림 18)를 만들어 낸다.

타기법과 본 논문에서 제안하는 EMPS기법과의 비교평가는 <표 1>과 같다.

6. 결 론

대규모 소프트웨어 개발 및 유지보수 과정에서 분기 및 병합을 이용한 병렬개발은 개발시간을 단축시킬 수 있으며 이를 지원하기 위한 형상관리 도구가 필요하다. 분기 및 병합을 지원하는 기존의 상업적인 형상관리 도구는 문자적 병합기법만을 따르고 있어 문자적인 요소는 보존하지만 병합된 결과가 병합되는 대상이 가지는 의미를 보존하면서 병합되었는지 보장할 수 없어 병합된 결과를 신뢰하기 어렵다.

또한, 일부 연구 프로토타입으로 제시된 HPR알고리즘과 같은 의미적 병합기법은 의미를 보존하며 병합하지만 병합대상이 프로그램이 아닌 일반 문서이거나, 프로그램 내에서도 의미와 관련 없는 주석 등의 요소에 대해서는 병합을 지원하지 못하며 의미적 병합충돌이 발생한 경우에 대해서도 해결책을 제시하지 않고 있다.

이런 문제점을 보완하기 위해 본 논문에서는 효율성 및 정확성면에서 뛰어나다고 알려진 3-way 라인기반 문자적 병합기법을 기본으로 하고 의미적 병합충돌 탐지 및 병합을 위해 오퍼레이션 기반 병합기법을 이용하여 의미적 병합기법을 결합한 하이브리드 병합기법인 EMPS기법을 제시했다.

EMPS기법은 3-way 라인기반 문자적 병합기법을 기본으로 하고, 병합대상이 그것의 원본에 대해 라인별로 수정된 내용을 라인별 수정, 삭제, 삽입 오퍼레이션(COS)으로 모델링하여 그것을 이용하여 문자적 병합충돌을 탐지했다. 또한, 의미적(실행행위) 병합충돌에 대한 탐지를 위해 프로그램 변수의 정의와 사용관계를 도식화한 PDG를 이용하여 COS를 중심으로 실행행위 병합충돌을 탐지하고 문자적, 의미적 병합충돌이 탐지된 오퍼레이션을 수정하여 병합대상의 원본에 적용함으로써 병합충돌 해결 및 병합했다.

EMPS기법은 이와 같이 오퍼레이션을 이용함으로써 기존의 상업적 병합도구가 효율적이기는 하나 문자적 병합만 가능하고 실행행위에 대한 병합충돌을 탐지 및 해결할 수 없다는 단점과 기존의 의미적 병합기법인 HPR알고리즘이 실행행위에 관련된 요소만으로 나타낼 수 있는 PDG자체를 의미적 병합충돌 뿐 아니라 병합에 이용하기 때문에 의미(실행행위)에 관련된 프로그램 요소만 병합가능하다는 단점을 보완하여 문자적 병합기법과 의미적(실행행위) 병합기법의 결합을 가능하게 했다. 따라서 EMPS기법을 이용하여 일반문서의 병합 뿐 아니라 실행행위와 관련 없는 주석과 같은 요소를 포함하는 프로그램에 대해서도 실행행위를 보존하면서 병합할 수 있다.

또한, 의미적(실행행위)에 대한 병합충돌 발생시 기존의 HPR알고리즘이 병합충돌이 일어나지 않을 경우만 병합이 가능한 데 비해 EMPS기법은 제한된 가정 하에서 의미적(실행행위) 병합충돌이 탐지되었을 경우, 의미적 병합충돌 위치 및 범위와 병합충돌을 야기한 변경내용 및 해결방안에 대한 가이드라인을 제시함으로써 병합충돌이 발생했을 때 이를 해결하여 병합할 수 있다.

EMPS기법은 기존의 HPR알고리즘과 같이 하나의 프로시져 내에 있는 선언문 및 제어문에 대해 의미적 병합충돌을 탐지하며, 의미에 관계없는 위치변경 및 변수이름 변경을 제한하는 개발환경에서 유용하게 적용될 수 있다. 향후, 연구과제로 제안된 기법이 상업적으로도 유용하게 사용될 수 있도록 하기 위해 첫째, 의미적 병합충돌에 대한 탐지가 하나의 프로시져 안에서 뿐 아니라 여러 프로시져에 걸쳐 이루어질 수 있도록 하는 기법[10]을 EMPS에 적용할 수 있는 방법을 연구하고, 둘째, 병합대상의 어느 한쪽이 원본에 대해 의미에 관계없는 위치변경 및 변수이름 변경이 이루어진 경우 병합충돌로 간주되지 않고 변경내용을 수용할 수 있도록 보강되어야 할 것이다. 셋째, 병합충돌 해결을 위해 제시한 5가지 가이드라인외에 병합충돌을 해결하기 위한 있는 새로운 가이드라인에 대한 추가적인 연구가 계속되어야 할 것이다.

참 고 문 헌

[1] D. E. Perry, H. P. Siy, and L. G. Votta, "Parallel Changes in Large Scale Software Development : An Observational Case Study," Proc. Int'l Conf. Software Eng., pp.251-260, 1998.

[2] D.E. Perry,H.P.Siy, and L.G. Votta, "Parallel Changes in Large Int'l Conf.Software Eng.[ICSE '98], pp.251-260,1998.

[3] C. Walrad and D. Strom, "The Importance of Branching Models in SCM," IEEE 2002, pp.31-38, September, 2002.

[4] 임신화, "효율적인 병렬 개발 지원을 위한 분기 및 병합 기법 연구," 충남대학교 석사학위논문, 2004년 2월.

[5] Tom Mens, "A State-of-the-Art Survey on Software Merging," IEEE Transactions, Vol.28, No.5, pp.449-461 May, 2002.

[6] D. Binkely, S. Horwitz, And T. Reps, "Program Integration for Languages with Procedure Calls," ACM Trans. Software Eng. and Methodology, Vol.4, No.1, pp.3-35, 1995.

[7] J.E.Grass, "Cdiff : A Syntax Directed Diff for C++ Programs," Proc. The Advanced Computing Systems Professional and Technical Association [USENIX] Conf. C++, pp.181-193, 1992.

[8] HOROWITZ S, PRINS J, and REPS R, "Integrating Non-Interfering Versions of programs," ACM Transactions on. Program Languages and System, Vol.11, No.3, pp. 345-387, July, 1989.

[9] D.Jackson and D.A.Ladd, "Semantic Diff: A Tool for Summarizing Effects of Modifications," Int'l conf. Software Maintenance, 1994.

[10] P. Glezen, "Branching With Eclipse and CVS", http://www.eclipse.org/.

[11] J.P. Munson and P. Dewan, "A Flexible Object Merging Framework," proc. ACM Conf. Computer Supported Collabrative Work, pp.231-352, 1994.

김 지 선

e-mail : sunny2255@bcline.com
1997년 충남대학교 컴퓨터과학과(학사)
2005년 충남대학교 컴퓨터과학과(석사)
1997년~현재 국방과학연구소 기술원
관심분야 : 형상관리, 컴포넌트 개발, 객체지향 디자인 등

윤 청

e-mail : cyoun@cs.cnu.ac.kr
1979년 서울대학교 물리학과(학사)
1983년 일리노이 주립대학교 Computer Science(석사)
1988년 Northwestern대학교 Computer Science(박사)
1985년 웨인 주립대학교 전임강사
1987년 Northwestern대학교 전임강사
1993년 Bell Communication Research 선임연구원
1993년~현재 충남대학교 컴퓨터과학과 정교수
관심분야 : V&V, 형상관리, 객체지향 디자인 등