소프트웨어의 유지보수를 위한 PSDG기반 의미분할모형의 설계

여 호 영† · 이 기 오†† · 류 성 열†††

요 약

소프트웨어의 품질을 향상시키며, 기존코드의 결함식별을 용이하게 하는 방법으로 프로그램의 후상태 종속성 분석을 통한 프로그램 분할 및 유지보수 지원 기법을 제안한다. 결함을 식별하고 분석하기에 이해도가 중요시 되는 교정유지보수를 위해서, 기존 코드의 분석 및 세그먼트화를 후상태 종속성모형(PSDG)을 이용하여 정적분할과 동적분할 및 의미분할의 장점을 살린 코드분할로 수행한다. 분할의 원리는 기존코드의 상태 종속성을 추적하여 그래프로 모형화한 후, 조각화(Clustering)와 강조분할(Highlighting)을 통해서 프로그램을 분할한다. PSDG 모형화의 결과로 비효율적인 프로그램 결함코드(Deadcode)의 식별 및 제거가 가능하며, 관련 프로그램 문장들을 일반화할 수 있고, 상태전이도 모형과의 확장연계로 분석 및 설계의 문서로 이용될 수 있다.

A design of the PSDG based semantic slicing model for software maintenance

Ho-Young Yeo† · Kee-O Lee†† · Sung-Yul Rhew†††

ABSTRACT

This paper suggests a technique for program segmentation and maintenance using PSDG(Post-State Dependency Graph) that improves the quality of a software by identifying and detecting defects in already fixed source code. A program segmentation is performed by utilizing source code analysis which combines the measures of static, dynamic and semantic slicing when we need understandability of defect in programs for corrective maintanence. It provides users with a segmental principle to split a program by tracing state dependency of a source code with the graph, and clustering and highlighting. Through a modeling of the PSDG, elimination of ineffective program deadcode and generalization of related program segments are possible. Additionally, it can be correlated with other design models as STD(State Transition Diagram), also be used as design documents.

1. 서 론

소프트웨어의 유지보수는 개발단계에서 만들어진 소프트웨어를 사용자에게 인도한 다음에 이루어지는 활동으로써, 주로 기 개발된 소프트웨어의 결점이나 오류를 교정하고, 주변 환경의 변화에 따른 적응 및 확장과 결함예방 활동들을 일컫는다. 유지보수를 위한 세부방법들을 분류해보면, 원시코드를 이해하기 쉽게 재구성하거나, 원시코드를 이용하여 설계 문서와 분석관련 문서의 회복 및 표준화 문서로 재 작성하는 재공학(Reengineering) 방법이 있으며[1], 개발 및 유지보수에 관련된 모든 문서를 데이터베이스화하여 관련 정보를 효

† 종신회원 : 한국 기술사 정보처리부문회 이사
†† 정 회 원 : 대전전문대학 전자계산과 전임강사
††† 정 회 원 : 숭실대학교 정보과학대학원 원장

논문접수 : 1997년 12월 31일, 심사완료 : 1998년 6월 1일

용적으로 검색하고 관리 유지하기 위하여 통합 자료저장소의 구축 및 관리방법이 있고, 유지보수에 관련된 모듈을 라이브러리 화하고 재 사용할 수 있도록 하여 모듈 단위로 유지보수 할 수 있도록 하기 위한 재사용 라이브러리의 구축방법 등으로 분류할 수 있다[16].

이러한 연구의 방법들을 분석해보면, 결국 기존코드나 분석 및 설계관련 문서들의 이해용이성과 정형화가 유지보수의 중요한 인자임을 알 수 있다. 또한 대부분의 유지보수에 관한 연구 및 설문에 관한 자료를 보더라도 원시코드의 유지보수 시점에 겪는 문제점으로써, 요구사항과 일치하지 않는 비기능성(Malfunction)과 이해불능(Misunderstanding) 및 비 표준화가 지적되고 있으며, 이러한 문제의 해결방안으로 기존문서나 코드의 정형화가 유지보수의 매우 중요한 인자임을 알 수 있다[5]. 따라서 본 연구는 기존 프로그램의 유지보수를 위한 코드의 정형화 및 이해용이성을 위한 프로그램 분할기법의 제안을 통하여 코드의 교정이나 확장 그리고 예측을 위한 지침을 제공하는 활동들을 수행하여 유지보수 비용의 절감과 품질향상을 목적으로 한다.

연구내용을 살펴보면, 유지보수 결함에 관한 정의 및 결함극복을 위한 접근방법으로 논리적인 해결방법 및 자동화 또는 경험적인 방법들에 관한 고찰을 수행하고, 이중에서 분할기법의 필요성과 분할을 위한 정형화의 필요성을 바탕으로 여러 분할기법들을 비교분석한 후 이로부터 후상태 종속성 그래프(PSDG)를 바탕으로 한 프로그램 분할기법을 설계하고, 이의 적용 및 실험 기법을 제안한다. 이러한 연구는 프로그램 분할이라는 것이 그동안 경험적인 부분과 자동화의 한계성을 연계하지 못하던 문제점을 인식하고, 이로부터 이해용이성과 정형화의 형식을 갖춘 분할기법을 소개함으로 유지보수의 향상과 기존코드의 재사용성을 극대화 시키기 위한 기초연구로써 유익하며, 이를 확장하여 개발비의 70%를 차지하고 있는 유지보수의 어려움과 고비용을 줄여나갈 수 있는 계기로 삼을 수 있다. 따라서 이 연구를 통하여 향후에는 분할 알고리즘의 보완과 자동화를 위한 CASE적인 접근이 필요하리라 본다.

2. 이해력증진 유지보수

2.1 유지보수의 역공학적 접근

소프트웨어의 결함은 원시코드의 비기능성이나 표준화되지 않은 기능성의 접목이나 확장을 통한 이해불능성 그리고 논리적인 자체결함등으로 구분할 수 있다. 이러한 결함등을 식별하기 위한 방법으로 원시코드를 구문분석(Parsing)하거나 경험적인 방법에 따라 여러 문서를 유지보수자의 환경과 특성에 따라 이해하는것이 대부분이었다. 이러한 이해에 소요되는 비용과 노력을 살펴보면 경험적인 이해와 자동화에 의한 이해가 연계되어야 만이 적절한 효과를 거둘 수 있음을 알 수 있는데, Stanley에 따르면, 유지보수의 전 과정중 요구사항을 파악하고 이해하는데 50-90%가 소비되며, 이해의 방법도 프로그램과 프로그램의 요구사항을 이해하는 방법과, 원시코드와 관련해서 추출된 분석 및 설계 정보에 관련된 문서를 이해하는 방법과, 원시코드로 부터 정적 및 동적 분석을 수행하여 초기화되지 않거나 실행하지 않은 변수를 검색해 내는 실행분석 및 추적을 통한 이해의 방법이 있다[12].

이러한 접근은 역공학적인 접근으로 인식될 수 있는데, 역공학은 이미 존재하고 있는 소프트웨어에 대한 물리적인 특성으로부터 자동화 기법에 의한 논리적 또는 개념적 정의를 유도해 내는 과정이며, 정립된 개념이나 논리적 단위는 새로운 시스템으로의 전환이나 확장에 대한 지침을 마련해 주며, 재사용의 기초자료가 된다.

2.2 이해력증진 접근방법

2.2.1 경험적인 접근방법

교정 유지보수를 위한 경험적인 접근방법으로는 원시코드에 대한 경험적인 인식과 문서에 대한 이해로 나눌 수 있다[1]. 원시코드에 의한 인식은 원시프로그램의 줄단위(Line by Line)인식과 주석문을 통한 인식, 그리고, 변수의 의미예측이나, 실행결과로부터의 프로그램 역추적과 같은 원시적인 방법이 이용될 수 있다[12]. 또한 문서에 의한 이해는 원시코드에 대한 문서로써 간주될 수 있는 제어흐름도나 자료흐름도 또는 상태전이도나 관련문서를 종합하여 유추할 수 있다. 하지만 이러한 접근은 문서를 인식하는 사람의 경험정도나 기술의 정도에 의존하는 부분이기 때문에 이해의 객관성 평가를 할 수 없다.

2.2.2 논리 흐름분석방법

논리적인 흐름분석은 구조의 흐름과 제어의 흐름 그리고 기능의 흐름과 같은 3부분으로 나누어서 분석할

수 있으며, 대개는 분석과 설계의 산출물이 논리흐름분석의 대상이 된다. 즉, 경험적인 접근방법의 문서에 의한 접근이 대상문서를 그대로 인식하는 것이라면, 논리흐름분석은 이런 활동을 포함하여, 이로부터 논리적인 검증과 변경에 대한 일관성 및 완전성을 분석하는 방법으로 정의할 수 있다. 대개의 경우 이러한 논리적인 검증은 부분적인 자동화도구에 의존하고 있다[9].

2.2.3 자동화 접근방법

자동화 접근방법은 CASE와 같은 자동화도구를 통해서 경험적인 분석이나 논리적인 분석을 가능케 하는 분석활동이라 할 수 있다. 따라서 이러한 접근은 비단 제어구조나 자료구조와 같은 한정적인 부분에만 적용되는 개념이 아닌 유지보수를 위한 자동화접근이면 모두 해당한다. 이를테면, 요구사항의 이해를 위한 정형화접근을 위하여 PSL/PSA의 이용이나 정형화언어(Formal language)의 사용에서부터, 프로그램 분할과 같은 고도의 유지보수 이해활동등이 이에 포함된다. 이중에서 자동화 접근방법의 한 부분인 프로그램 분할에 대하여 살펴본다.

2.3 프로그램 분할기법

2.3.1 정적분할

정적분할(Static slicing)이란 프로그램의 구조를 실행이전에 예측가능한 형태로 분할하는 기법으로 현재 많이 쓰이고 있는 기법들을 정확성 측면에서 비교분석해 보면 다음과 같다[3]. Weisers의 알고리즘은 소스코드의 각 라인을 하나의 단위로 고려하였다. 이것은 한 라인에 하나이상의 문장을 포함하고 있으면 부정확한 분할의 결과를 산출하게 된다. 하지만 정보흐름관계성 분석이나 프로그램 종속성그래프(PDG: Program Dependency Graph)에 바탕을 둔 알고리즘들은 이러한 문제를 제공하지 않는다. 이러한 분할 알고리즘들의 특징은 (s, V)의 쌍으로 구성 되는데, 여기에서 s는 문장이고 V는 변수들의 집합이다. 반면에, PDG에 바탕을 둔 분할방법의 특징은 (s, Vars(s))쌍에 효율적으로 일치한다는 것인데, 여기서 s는 문장이고 Vars(s)는 문장 s에 사용되거나 정의된 모든 변수들의 집합이다. 그러나, PDG바탕의 분할방법은 모호한 변수(V)에 대한 기준을 가지고 분할을 수행할 수 있는데, 여기서의 모호함이란 의미없는 변수일지라도 문장내에 존재하는 변수일 경우 모두 그래프로 표현되어짐으로써, 이해를 어렵게 한다는 것이다. 또한, 이러한 모호함은 그래프내에 사용되지 않는 논리의 흐름이나 문장을 계속 유지하게 함으로써 유지보수시 재사용의 판단을 어렵게 한다.

D = dataflow equations,
F = functional/denotational semantics,
I = information-flow relations,
G = reachability in a dependence graph.
bS = structured, bA = arbitrary.
cS = scalar variables, cA = arrays/records,
cP = pointers.

표 1에서 Weiser[17]의 상호절차정적분할(IPSS)알고리즘은 상호절차 요약정보가 모든 입력 파라미터 집합과 모든 출력 파라미터 집합 사이의 프로시듀어 호출 횟수에 의존하는 기법이므로 자료흐름과 구조화된 제어흐름을 지원할 때 효과적이다. Bergeretti와 Carre의 알고리즘은 완전한 상호절차정적분할을 수행 못하는데, 그것은 오직 주프로그램 만이 분할될 수 있기 때문

〈표 1〉 정적분할 비교표

비교항목 / 분할기법	Computation Method	Interprocedural Solution	Control Flow	Data Types	Interprocess Communication
Weiser	D	Yes	bS	cS	No
Lyle	D	No	bA	cS, cA	No
Hausler	F	No	bS	cS	No
Horwitz	G	Yes	bS	cS	No
Agrawal	G	No	bS	cS, cA, cP	No
Cheng	G	No	bS	cS	Yes

이다. 다만, 재귀(Recursive) 프로그램을 처리하는 능력도 가지고 있지 않다. Bergeretti-Carre slice는 입력과 출력 파라미터 사이에 정확한 종속성 존재시나 호출구문 사례에 대해서만 분할이 가능하다.

프로그램에 복합변수와 포인터가 사용되었을 때의 분할을 살펴보면, Lyle과 Agrawal은 배열이 나타나는 정적분할에 대하여 효과적이었으며, 상호병행성이 추가된 병행 프로그램의 정적분할에 대한 접근법은 Cheng에 의해 제안되었다.

2.3.2 동적분할

동적분할(Dynamic Slicing)은 프로그램의 실행결과를 토대로 프로그램의 구조를 분할하는 기법으로 그 기법들을 비교하면 다음과 같다.

aD = dynamic flow concepts,
aI = dynamic dependence relations,
aG = reachability in a dependence graph.
bS = scalar variable.
aA = arrays/records,
aP = pointers.

korel과 Laski의 알고리즘에 의해 수행된 분할은 Agrawal과 Horgan, Gopal에 의해 수행된 알고리즘보다 비효율적으로, 이는 Korel과 Laski의 분할 후 적용의 제약성에 기인한다. 즉, 분할된 조각은 수행가능해야 한다는 것이다. Agrawal과 Horgan, Gopal의 연구에 의하면, 프로그램을 중단시키는 분할은 프로그램을 분기시키는 요소들로 구성되어진다. 상호절차적인 동적분할 알고리즘에 기반을 둔 종속성 그래프는 Agrawal, DeMillo, Spafford 그리고 Kamkar등이 제안했다.

복합변수와 포인터가 있는 프로그램의 동적분할 알고리즘은 Korel, Laski, Agrawal, DeMillo, Spafford에 의해 제안되었다.

2.3.3 의미분할

의미분할(semantic slicing)은 정적분할이나 동적분할과는 달리, 프로그램의 분할단위를 보다 합리적인 모듈이나 기능의 단위로 분할하기 위하여 경험과 의미적 해석에 바탕을 둔 분할 기법이다[2]. 여러 관련연구가 선행되어 졌으나, 가장 최근의 연구로 Dijkstra의 취약부분 선조건(Weakest Pre-condition) 알고리즘[14]을 적용한 분할기법이 소개 되었다. 이는 프로그램을 분할하는 기법으로, 주어진 프로그램 S와 프로그램 종료를 위해 만족해야할 조건 R에 대한 S의 취약부분 선조건을 wp.S.R이라 할 때, 이는 조건R에 의해 정의된 여러상태들중 하나의 실행만족을 보장하기위한 최소 제약조건으로 정의할 수 있다. 이때, 프로그램 분할을 위한 사후조건은 변수 x 와 연관된 상태의 의미적 분할 덩어리를 획득하기 위하여 X=m의 자유변수 형태로 기술되어지며, 모든 문장으로부터 계산된 wp와 후조건사이의 변화가 없는 문장들은 제거되어진다. 즉, 관심있는 변수의 상태에 영향을 미치지 않는 문장들의 제거를 통해서 프로그램의 분할이 이루어 지기 때문에, 분할과정이 매우 단순하고 짧게 소요된다[2].

이러한 종류의 의미분할은 프로그램의 상태를 이해함으로써 관련분석이 이루어 지기 때문에, 큰 규모의 프로그램을 분할하는데 매우 효과적이다. 또한 프로그램 변경의 효과로 결함을 검색해 내거나, 다른 프로그

〈표 2〉 동적분할 비교표[4]

비교항목 / 분할기법	Computation Method	Executable	Interprocedural Solution	Data Types	Interprocess Communication
Korel, Laski	aD	Yes	No	bS, bA, bP	No
Korel, Ferguson	aD	Yes	No	bS, bA	Yes
Gopal	aI	No	No	bS	No
Agrawal, Horgan	aG	No	No	bS	No
Kamkar	aG	No	Yes	bS	No
Deusterwald et al	aG	Yes	No	bS, bA, bP	Yes
Cheng	aG	No	No	bS	Yes

램들 사이의 상호접속을 분석해 내는데 효과적이며, 컴파일러가 모든프로그램을 다 분석할 필요가 없기 때문에 매우 경제적이다.

하지만, Dijkstra 의 취약부분 선조건 알고리즘의 분할원리가 Unity라고 하는 한정된 정형화 언어를 이용해야 하는 상향식 프로그램 변수추적 기법이므로, 검증을 위한 기법으로는 유용하나 유지보수라는 일반적인 적용사례에서 이해용이성을 제공하기에는 문제점이 있음이 지적[9]되고 있다. 따라서, 의미분할과 Dijkstra의 알고리즘을 보완한 후상태 종속성 그래프를 이용한 확장모형을 제안한다.

3. PSDG 확장모형화

3.1 관련연구

후상태 종속성 그래프(Post-State Dependency Graph)를 이용한 확장모형화는 의미분할을 응용한 기법으로써 선조건에 대한 후기상태를 종속성 관계로 표현한 그래프에 형식화와 강조기능을 첨가한 확장모형이다. 이와 관련한 연구인 Petius software[15], Wisconsin's Program Slicing Project[9]들의 특징은 특정 정형화 언어의 이용이나 PDG와 같은 기법을 이용하고 있다는 점이다. 따라서, 이에 확장된 유지보수 모형을 적용하여 이해용이성과 정형성의 효과를 거두고자 한다.

3.2 후상태 종속성 확장모형화

후상태 종속성 확장모형화의 기본적인 골격은 기존의 PDG가 프로그램의 문단 구조로부터 하향식 문상종속성을 표현함으로써, 효율적이지 못한 프로그램의 경우 PDG자체가 복잡할 뿐만아니라 이를 단순화 시키는 그래프 축소기법(Graph reduction)기법의 적용이 매우 어려웠다. 이러한 문제점을 인식하고, 프로그램 실행시의 최초 단일문장의 초기 상태에서부터 마지막 상태에 이르는 경로를 후조건 검색 알고리즘에 의하여 그래프로 표현한다. 표현된 그래프의 각 노드는 문장내의 변수들로써, 실행시에 영향을 미치는 다음문장의 변수에 종속적인 관계를 가지고 있으므로, 이에 그래프로의 표현이 가능하다. 이는 소프트웨어 공학의 상태전이도나 Petri-net과 같은 도구를 이용하여 표현가능하며, 프로그램 구조의 정적인 구조뿐만 아니라 동적인 활동성을 추적할 수 있다. 이렇게 완성된 그래프는 사용자의 경험이나 알고리즘의 지침에 의하여 부분적인 조각화(Clustering) 및 이진 그래프의 분할 및 중요한 부분의 강조 그리고 잘못된 모듈을 표시할 수 있는 가능을 제공한다[4].

이 모형의 주된설계는 정적분할의 PDG와 의미분할의 하나인 Dijkstra의 알고리즘을 표본으로 삼고 이루어 졌으며, 원시코드의 이해용이성과 확장성 그리고 프로그램의 재사용성을 위한 설계회복의 관점에서 작성되었다. 주요기법으로는 조각화, 추상화, 강조등이다.

3.2.1 모형화 구조

모형의 구조는 PSDG 확장모형의 구조적 특성과 다른 도구들과의 유사성을 분석한다. 우선, 이 확장모형의 특성은 최초의 초기문장을 트리의 루트로 하여, 종속적인 관련이 없는 후상태들을 자식(Child)으로 만들어 나가는 수평적 확장을 수행하다가, 종속적인 관계가 있는 후상태들을 깊이우선의 하위노드로 연결해 나가는 방식을 채택하고 있다. 이 방식에 의하면, 단지 프로그램의 구조성 뿐만아니라 실행성을 추적할 수 있으므로, 사용자로 하여금 가시적인 효과와 이해를 바탕으로한 그래프의 생성이 가능하다는 특징을 가지고 있다. 또한, 상태전이도와 같은 모형화 도구와 연계해서 표현할 수 있으므로, 프로그램의 자동화 가능성 및 도달성(Reachability)과 같은 모형점검의 기능을 확장할 수 있다. 그리고, 잘못된 프로그램이나 재사용불능(Non-reusable), 연속재사용(Serially reus- able)과 같은 프로그램 재사용성을 분석해 낼 수 있는 장점이 있다.

3.2.2 모형화 알고리즘

Algorithm_PSDG

단계 1. Parse the program source

단계 2. from the initial statement or state, find the root.

단계 3. decide the condition of a next statement whether it has horizontal or vertical path (horizontal : has not correspondency, vertical : has dependency), until it meets the end.

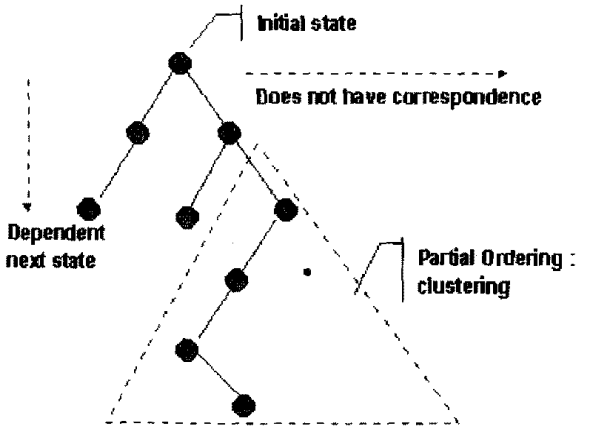
단계 4. If it finds a set of nodes which do not affect to any other nodes, clustering just be happened and hightlighting also.

단계 5. 'IF', 'CASE', and iteration statements

are treated as a newly generalized node.
단계 6. creates a PSDG and generates a report of the program(module reusability, errors, and others)

알고리즘의 단계 1은 프로그램의 인식단계로 구문분석 단계이다. 단계 2에서는 프로그램의 초기문장을 초기상태로 설정하여 트리구조를 만든 후에 루트에 위치시킨다. 단계 3은 루트의 자식노드를 만들어 나가는 과정인데, 분석된 구문을 바탕으로 다음문장이 이전의 루트문장과 상관관계가 있는지를 조사하여 있으면 수직적으로, 없으면 수평적으로 노드를 형성해 나간다.
그런 후에, 단계 4에서 더 이상 영향받지 않는 문장이 발생하면 지금까지의 노드들을 조각화하여 분리시키고, 현재노드와 이전에 기록된 자료를 비교하여 상관이 없을 때 까지 역추적한 후 그 시점의 수평위치에 형제노드로 위치시킨다. 단계 5는 노드의 운행중 조건문이나 분기문에 대한 처리로 이러한 문장을 만나면 수평의 위치에 형제노드로써 인식함을 의미한다. 그리고 마지막으로 단계 6은 분석된 자료를 바탕으로 모듈의 재 사용성이나 결함코드(Deadcode)에 대한 보고를 하여 유지보수시의 재사용에 대한 지침을 제공한다.

위의 알고리즘에 대한 구현의 방법은 적용환경이 분산환경인지 단일환경(Stand-alone)인지에 따라 방법을 달리할 수 있는데, 분산환경에선 JAVA를 단일환경에서는 Visual Basic으로 작성한다.



(그림 1) PSDG 구조도
(Fig. 1)

3.2.3 PSDG적용

PSDG는 그러므로 최초의 상태에서 출발하여, 바로 다음의 프로그램 문장이 후조건으로 성립할 수 있는지 없는지를 판별하여 종속적으로 관련 있으면 하위노드에 그렇지 않으면 수평적인 좌우노드에 위치시킨다. 하지만 이러한 노드의 배열은 구조화 되어있지 않거나 프로그램의 구조가 매우 비효율적일 경우 출력가지(Fan-out)가 너무 커진다. 그러나, 이러한 현상은 조각화를 통해서 관련 없는 몇몇의 노드들을 하나의 독립노드로 일반화시키는 작업을 통해서 PSDG의 단순화는 가능하다. 이때, 관련 없는 노드들을 분석하여 프로그램에 직접 연관을 미치지 않는 문장들을 삭제하거나 변경함으로써 유지보수의 이해용이성을 극대화 할 수 있다. 이러한 활동을 통해서 프로그램의 논리구조는 단순화되며, 전체 그래프는 자연스럽게 하향식의 그래프로써, 왼쪽에서 오른쪽으로 위에서 아래로의 순차구조를 보이는 부분순서화(Partial Ordering) 그래프가 작성되어 진다.

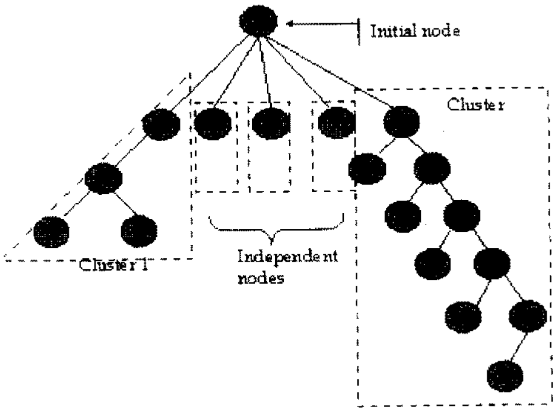
```
1  double km_per_mi = 2.54 * 12 * 5280 * 0.00001
2  double mi_per_km = 1/km_per_mi;
3  cout << " " << km_per_mi
          << " is kilometers per mile \n ";
4  cout <<" " <<mi_per_km
          << " is miles per kilometers\n\n";
5  double km, mi;
          // To control number of columns
6  int i = 0;
7  cout.setf(ios::showpoint);
8  for (km = 0; km (= 10; km +.1){
9       mi = km * mi_per_km;
10      i++;
11      if( i == 1){
12   cout << "   " c( setprecision(1)<< setw(4)<< km;
      }
13    cout << " " << setprecision(2)<< setw(4)<< mi;
14  if (i == 5){
15        i = 0;
16        cout << endl;
        }
    }
17  cout (( endl;
    return 0;
```

(그림 2) km를 Mile로 변환하는 c++프로그램
(Fig. 2)

또한, 이렇게 단순화된 PSDG는 Graph이론에 의해 이진트리나 감축 가능한 형태의 그래프로 변형 가능하다. 이러한 특성을 비춰볼 때, 프로그램의 유지보수시 이해의 용이성은 물론, 프로그램의 효율성을 평가할 수 있는 접근방법으로 평가할 수 있다.

그림 2는 Km를 마일로 계산하는 절차중심의 C++ 기능성 모듈(Functional module)에 대한 프로그램 원시코드로, 각각의 문장 앞에는 문장번호가 부여되어 있다. 이들 문장들을 PSDG에 적용하면 변수의 후상태 종속성을 중심으로 트리형태의 그래프를 만들어 나가기

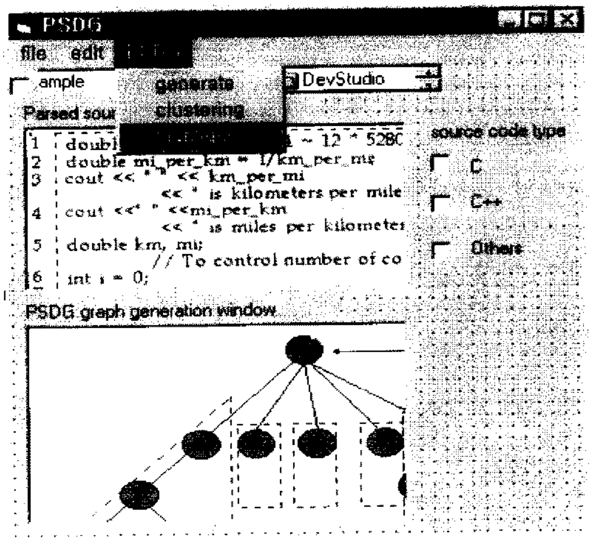
때문에, 그림 3과 같은 구조를 형성하게 된다. PSDG의 적용은 프로그램 실행의 문장, 변수간 상관관계를 쉽게 추적하여 볼 수 있으며, 유지보수자로 하여금 조각화나 강조(Highlighting) 와 같은 기법을 적용하여 프로그램의 이해용이성과 일반화가 가능하다.



(그림 3) PSDG 적용 그래프
(Fig. 3)

그림 3은 PSDG를 적용한 후의 그래프로 다중트리의 형태로 표현되고 있다. 이때 반복되는 종속성관계의 부분트리인 숲(Forest)을 조각화하여 보다 단순한 심볼로 표현해 버리면, 설계관점에서의 추상화와 이해의 용이성을 제공하게 된다.

그림 4는 PSDG를 구현하기 위한 화면설계 부분으로 Visual Basic으로 처리되었다.



(그림 4) PSDG 적용화면
(Fig. 4)

PSDG의 비교를 위해 이용되는 속성(Properties)을 살펴보면 다음과 같다. 논리코드의 의미분석 적용 여부, 소프트웨어 생명주기 초기단계 적용 여부, 프로그램 통합여부, 정확성 증명 여부, 프로그램 이해 용이성 여부, 프로그램의 실행가능성 여부, 프로그램 일반화 용이성 여부, 자동화 가능성 여부, 유지보수 용이성 여부, 지원 도구 여부등을 비교한다. 여기서 Y는 있음을 의미하고, N은 없음을 의미한다. 그리고 N/A는 없거나 모호한 경우를 말한다.

이상의 비교표에서 보았듯이, PSDG는 기존의 정적, 동적분할 기법에 비해 소프트웨어의 이해와 유지보수라는 관리측면에서 이해용이성을 제공하며, 이를 확장하여 프로그램의 완전성이나 자동화 및 도구지원 기법등과 연계한 연구가 가능하다.

4. PSDG 비교

〈표 3〉 PSDG 와 분할기법과의 비교표

비교항목	Static Slicing	Dynamic Slicing	PSDG
Semantic Logics	N	N	Y
Software Lifecycle's Each Phase	N	N	Y/A
Program Synthesis	N	Y	Y
Proofs of correctness	N/A	N/A	Y
Program Understanding	N/A	N/A	Y
Program Completeness	N	N	Y/A
Generalization	N	N	Y
Automation	N/A	N/A	Y/A
Maintenance	Y	Y	Y
Tools Assistance	N	N	Y/N

5. 결론 및 향후 연구과제

PSDG는 프로그램의 후상태를 추적하여 도형화한 이해용이성 모형으로, 프로그램 원시코드의 모형화를 통하여 유지보수시의 이해도 증진을 향상하였다. 특히, 요약화 및 추상화 그리고 강조의 기능들은 프로그램 구조의 이해 및 결함식별, 재사용의 지침을 제공하였다. 이는 대부분의 실제회복이 거시적(Macro)인 측면의 모듈구조도나 클래스와 같은 자료집합을 모형화하는데 비해, 기능성 중심의 모듈내부를 의미기반의 후상태 종속성에 기반을 둔 적당한 단위들로 분할하고 이를 모형화함으로써 재사용의 효율을 증대하였다.

향후 연구로는 이 모형의 확장을 통한 자동화 및 자동화 도구들과의 연계 그리고 정확성 검증등의 기법이 연구되어야 한다.

참 고 문 헌

[1] Coleman, D.; Ash, D.; Loather, B.; Oman, P.: Using Metrics to Evaluate Software System Maintainability, pp. 44-49, IEEE Computer, August 1994.

[2] Joseph J. Comuzzi, "Program slicing using weakest precondition", Peritus Software Services Technical Report, Software Practice & Experience, to appear, 1995.

[3] Centrum voor Wiskunde en Informatica, Report: A survey of program slicing techniques, computer science/department of software technology, CS-R9438, 1994.

[4] Daniel Jackson and Eugene J. Rollins, "Chopping: a generalization of slicing", CMU-CS-94-169, 1994.

[5] Domenico Natale, "On the Impact of Metrics' Application in a Large Scale Software Maintenance Environment", IEEE 1991.

[6] Foster, J.: Program Lifetime: A Vital Statistic for Maintenance, pp. 98-103, Proceedings of the Conference on Software Maintenance 1991, Sorrento, Italien, October 15-17.

[7] Gallagher, K.B.and Lyle, J.R., "Using Program Slicing in Software Maintenance.", pp. 751-761, IEEE Transactions on Software Engineering.

[8] Harrison, W.; Cook, C.: Insights on Improving the Maintenance Process Through Software Measurement, pp. 37-44, Proceedings of the International Conference on Software Maintenance, San Diego, November 26-29, 1990.

[9] Hart, J.M., "Experience with Logical Code Analysis in Software Maintenance," Peritus Software Services Technical Report, Software Practice & Experience, to appear, 1995.

[10] Jean-Pierre Queille, "The impact analysis task in software maintenance: a model and a case study", IEEE 1994.

[11] Kafura, D.; Reddy, G.R.: The Use of Software Complexity Metrics in Software Maintenance, pp. 335-343, IEEE Transactions on Software Engineering, 13(1987)3.

[12] Kari Juul Wedde, "A case study of a maintenance support system", IEEE 1995.

[13] Oman, P.; Hagemeister, J.: Metrics for Assessing a Software System's Maintainability, pp. 337-344, Proceedings of the Conference on Software Maintenance, Orlando, Nov. 9-12 1992.

[14] A. Pizzarello, "A New Method for Location of Software Defects", 1AQuIS 1993, Venice, Italy 1993.

[15] A. Pizzarello, "Formal Techniques For Understanding Programs", Peritus Software Services Technical Report, Software Practice & Experience, to appear, 1995.

[16] Harry M. Sneed, "Planning the reengineering of regacy systems", IEEE 1994.

[17] Weiser, "Program Slicing", IEEE Trans. on Software Engineering, VOL. SE-10 No 4, 1984.

여 호 영

1977년 2월 고려대학교 토목환경공학과 (공학사)

1990년 2월 숭실대학교 정보과학대학원 전자계산학과(공학석사)

1993년 2월 숭실대학교 대학원 전자계산학과 박사수료

1991년 5월~현재 감리법인 (주)GIS 대표이사

관심분야 : 소프트웨어 재사용, CASE

이 기 오

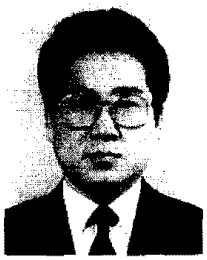
1989년 2월 순천향대학교 전자계산학과 (공학사)

1994년 2월 숭실대학교 대학원 전자계산학과 (공학석사)

1997년 8월 숭실대학교 대학원 전자계산학과 박사수료

1996년 3월~현재 대전전문대학 전자계산과 전임강사

관심분야 : 분산객체컴퓨팅, 리엔지니어링



류 성 열

1997년 2월 아주대학교 컴퓨터공학부(공학박사)

1997년 3월~1998년 2월 George Mason University 교환교수

1981년 3월 - 현재 숭실대학교 컴퓨터 학부 교수

1998년 3월~현재 숭실대학교 정보과학대학원 원장

관심분야 : 리엔지니어링, 분산객체컴퓨팅, 소프트웨어 재사용