KIPS Tr. Software and Data Eng.
Vol.3, No.2 pp.65~72 pISSN: 2287-5905

http://dx.doi.org/10.3745/KTSDE.2014.3.2.65

A Technique to Detect Change-Coupled Files Using the Similarity of Change Types and Commit Time

Jung il Kim† · Eun joo Lee††

ABSTRACT

Change coupling is a measure to show how strongly change-related two entities are. When two source files have been frequently changed together, they are regarded as change-coupled files and they will probably be changed together in the near future. In the previous studies, the change coupling between two files is defined with the number of common changed time, that is, common commit time of the files. However, the frequency-based technique has limitations because of 'tangled changes', which frequently happens in the development environments with version control systems. The tangled change means that several code hunks have been changed at the same time, though they have no relation with each other. In this paper, the change types of the code hunks are also used to define change coupling, in addition to the common commit time of target files. First, the frequency vector based on change types are defined with the extracted change types, and then, the similarity of change patterns are calculated using the cosine similarity measure. We conducted experiments on open source project Eclipse JDT and CDT for case studies. The result shows that the applicability of the proposed method, compared to the previous studies.

Keywords : Change Coupling, Source Code Repository Mining, Change Type Similarity

변경 유형의 유사도 및 커밋 시간을 이용한 파일 변경 결합도

김 정 일† · 이 은 주††

요 약

변경 결합도는 두 요소들 사이의 향후 변경 연관성을 알려준다. 만약, 소스 파일들이 자주 함께 변경된다면, 그 소스 파일들의 변경 결합도는 높다고 볼 수 있으며, 나중에 다시 함께 변경될 확률이 높다. 일반적으로 소스 파일들 사이의 변경 결합도는 공통 변경 횟수에 기반하여 정의되었다. 그런데 연관성이 낮은 변경들이 일괄적으로 함께 커밋되는 경우, 즉 뒤얽힌 변경(tangled change)과 같은 경우들이 빈번히 발생한다. 따라서 함께 변경된 횟수만으로 소스 파일의 변경 결합도를 결정하는 것은 한계가 있다. 본 논문에서는 기존의 방법을 보완하기 위해, 소스 파일의 변경 시간뿐 아니라 소스 코드 변경 유형의 유사성을 함께 고려하는 것을 제안하였다. 이를 위하여, 우선 추출된 변경 유형 정보를 이용하여 변경 유형 빈도 벡터를 정의하고, 다음에 코사인 유사도 측정을 통해서 각 소스 파일 버전에서 적용된 코드 변경 유사성을 계산한다. 이후 Eclipse 프로젝트인 JDT와 CDT에 대한 사례 연구를 통해 제안된 방법의 효용성을 보였다.

키워드 : 변경 결합도, 소스 코드 저장소 마이닝, 변경 유형 유사도

1. 서 론

소프트웨어 진화양상을 분석하는 것은, 시스템 유지보수 작업에 도움을 준다. 소프트웨어 시스템은 코드 리팩토링, 오류 수정, 코드 최적화 또는 시스템 설계 변경 등과 같은 다양한 이유로 계속해서 변경된다 [1]. 이러한 변경들은 SVN, CVS 또는 Git 같은 소프트웨어 버전 관리 시스템(version control system)에 저장되며, 저장된 변경 정보들은 진화 분석을 통해서 향후 수행되어야 되는 변경 또는 오류를 예측하는 데 이용된다.

소스 파일 변경 결합도 (Change coupling)는 각 소스 파일들이 변경으로 인해 서로 얼마나 연관이 있는지 나타낼 수 있는 척도이다. 예를 들어, 과거에 계속해서 함께 변경되

※ 이 논문은 2013년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(2012R1A1A3011005).
† 비 회 원 : 경북대학교 컴퓨터학부 박사과정
†† 종신회원 : 경북대학교 컴퓨터학부 부교수
논문접수 : 2013년 12월 26일
수 정 일 : 1차 2014년 1월 21일
심사완료 : 2014년 1월 21일
* Corresponding Author : Eun joo Lee(ejlee@knu.ac.kr)

는 소스 파일들은 향후 또 다시 함께 변경될 수 있다 [2]. 기존 연구에서는, 변경 결합도의 결합률은 함께 변경된 횟수로 결정되며, 횟수가 많을수록 향후에 다시 함께 변경될 확률이 높기 때문에 결합률이 높다고 정의되었다 [3, 4]. 이러한 종류의 변경 결합도는 소프트웨어 버전 관리 시스템의 커밋 로그 (Commit log)를 통해서 분석할 수 있다.

그런데, 연관성이 낮은 변경들이 일괄적으로 함께 커밋되는 경우가 종종 발생한다. 이러한 변경들을 뒤얽힌 변경 (tangled code change)이라 일컫는데 [5], 뒤얽힌 변경은 주로 변경에 대한 커밋이 지연되어 일괄적으로 커밋될 때 나타난다. 뒤얽힌 변경을 포함하는 커밋 기록은 소스 코드 저장소 기반의 변경 결합도 분석을 어렵게 만들면서 분석 결과의 품질을 떨어뜨리는 주요 원인이 된다. 그리고 이어서 변경 시간만을 고려하면, 해당 파일들이 어떻게 변경되었는지 알 수 없기 때문에 분석 정확도가 떨어질 수 있다. 즉, 소스 파일들이 전혀 다른 목적으로 변경된 경우에도 변경 결합 연관관계가 높다고 판단될 수 있기 때문이다. 따라서, 소스 파일들 사이의 변경 결합도를 결정할 때 세밀한 코드 변경 정보 (Fine-grained source code change)를 고려할 필요가 있다.

본 논문에서는 비교 대상이 되는 파일들의 변경된 코드 조각들에 대해, 그 변경 유형 정보를 이용하여 소스 파일들의 변경 연관성을 이전 방식보다 효과적으로 알아볼 수 있는 방법을 제안한다. 예를 들어, 그림 1과 같이 3개의 소스 파일들이 전체 5번의 커밋 순간에서 3번째 커밋 순간 (t_3)에서 함께 변경되었고, 이때 파일 f_a와 f_b에는 동일하게 3개의 새로운 스테이트먼트 코드가 추가 및 삭제되었고, f_c에는 특정 메소드에 새로운 매개변수가 2개 추가되었다고 가정할 경우, t_3에서 각 파일에 발생된 변경 유형은 그림 2와 같다. 이전 방식처럼 커밋 시간만을 이용한다면 f_a, f_b, f_c는 변경 연관되었다고 볼 수 있으나, 변경 유형까지 활용하면, f_c는 후보에서 제외되고 f_a와 f_b가 더 연관성이 높다고 판단된다. 즉, 같은 커밋에서 변경되었다 하더라도, f_c처럼 다른 파일과 연관성이 없는 파일이 함께 변경될 수가 있는데, 본 연구에서 제안한 방식은 이러한 경우를 걸러줄 수 있다.

본 논문에서는 동일한 커밋에 함께 변경된 소스 파일들에 대해, 변경된 코드 조각의 변경 유형을 추출하였다. 변경 유형은 Fluri 등이 정의한 35가지 변경 유형 [6]을 기반으로 하되, [6]에서 정의된 유형에서 빠져있는 import 변경 및 예외 처리 부분까지 추가하였다. 자바 기반의 소프트웨어 개발에서 Import 변경은 스테이트먼트 변경 다음으로 가장 빈번하게 발생될 수 있으며, 또한 예외 처리 관련 코드 변경에 끼치는 영향이 크기 때문에 정확한 코드 변경 유형을 분석을 위해서 import 변경을 고려할 필요가 있다. 실험을 위해 Fluri 등이 개발한 변경 유형 추출 도구인 ChangeDistiller [6]을 수정, 확장하여 이용하였다. 이후, 추출한 변경 유형들을 변경 유형 빈도 벡터로 구성하고, 변경 유형 빈도 벡터들에 대해서 코사인 유사도를 측정하여 변경 결합도를 예측하는데 이용하였다. 실험에서는 Eclipse프로젝트인 JDT, CDT를 대상으로 하여, 기존의 변경 횟수만 고려한 경우 및 시간 기반 결합도 측정 기법 [12]과 본 연구에서 제안하는 기법을 비교하여, 제안된 방법의 우수성을 보였다.

본 논문의 구성은 다음과 같다. 2장에서 관련 연구들에 대해서 소개하고, 3장에서는 제안된 방법을 기술한다. 4장에서 사례연구를 보이고, 5장에서 결론을 맺는다.

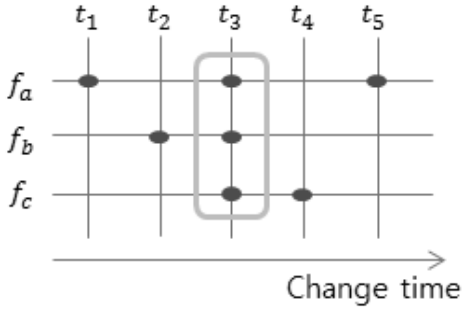

Fig. 1. An example of commit history for source files

Change Type	Frequency		
	f_a	f_b	f_c
STATEMENT_INS	3	3	0
STATEMENT_DEL	3	3	0
.		.	
.		.	
.		.	
PARAMETER_INS	0	0	2
.		.	
.		.	
.		.	

Fig. 2. Change types applied to each source file at t_3

2. 관련 연구

2.1 변경 결합도

소프트웨어 버전 관리 시스템은 소스 파일을 포함하여 소프트웨어를 구성하는 여러 종류의 요소들 사이의 변경에 대한 결합 정도를 측정하는 데 많이 사용되어 왔다 [7]. 진화 분석에 대한 앞선 연구들에서는 소프트웨어 버전 관리 시스템에서 관리하는 커밋 이력을 통해서 소프트웨어 모듈 또는 파일들 사이의 변경에 대한 결합도를 식별하고, 식별 결과를 기준으로 소프트웨어 결합 검출 및 변경 예측을 도울 수 있다는 것을 보였다.

Gall 등은 규모가 큰 소프트웨어 시스템 내부에 존재하는 모듈들의 버전 변경 이력을 관찰하여 함께 변경되는 모듈들은 변경에 대한 결합도가 높다는 것을 보았다 [1, 3, 8]. 하지만, 대체로 많은 양의 변경 정보가 필요하기 때문에 소규

모의 프로젝트에는 적용하기 힘든 점이 있다. Kagdi 등은 Gall 등이 제안하는 것과 유사한 방법으로 소스 파일들뿐만 아니라 프로젝트에 포함된 모든 파일에 대한 추적 관계(Traceability link)를 복구하는 방법을 제안했다. 파일들 사이의 추적 관계는 변경 순서가 필요한 파일들을 식별하는 데 유용한 정보가 된다 [9]. Fluri 등은 CVS의 리비전 번호를 기반으로 리비전 벡터(Revision vector)를 만들고, 리비전 벡터와 해당 리비전에서 적용된 구조적 변경 정보를 비교하여 소스 파일들 사이의 결합도를 결정하는 방법을 제안했다 [10]. [10]에서 각 소스 파일 리비전에서 구조적 변경 유무를 추출하기 위해서 Eclipse 에서 소스 파일 비교를 위해서 제공되는 플러그인을 이용했다. Robbes 등은 변경 시간을 기반으로 하는 변경 결합도 측정 방법을 제안했다 [11, 12]. Robbes 등이 제안한 방법은 두 변경 요소들 사이의 변경 시간의 간격이 짧을 수록 강한 결합도를 가지며, 비교적 적은 데이터만으로 성공적으로 변경 결합도를 결정할 수 있는 장점이 있다. 하지만, 변경 요소들 사이의 변경 결합도를 결정할 때 적절한 변경 기간을 세밀하게 조정하지 않으면 쉽게 잘못된 결합도 값을 결과로 얻을 수 있다. Steff 등은 소스 코드 저장소의 전체 변경 이력 정보를 이용하여 커밋 그래프를 생성하고, 커밋 그래프를 기반으로 연속적인 변경 이력들의 관계를 식별하는 것을 제안했다 [4]. [4]에서 제시하는 방법은 커밋 이력간에 결합도 찾는데 적절하지만, 소스 파일들 사이의 결합도를 찾는 데 한계가 있다.

2.2 소프트웨어 진화 분석

소프트웨어 버전 관리 시스템에서 제공되는 여러 가지 변경 정보들을 마이닝하여 변경 패턴을 찾아내는 것이 것이 가능하다. 변경 패턴은 특정 변경 요소들의 변경 정보를 나타내며, 변경 결합도와 마찬가지로, 소프트웨어 진화 분석, 예를 들어 향후 발생될 수 있는 변경을 예측하는 데 유용한 정보가 된다.

Zimmermann 등은 FPM (Frequent Patterns Mining [13])을 기반으로 하는 변경 예측 도구 ROSE를 개발했다. ROSE는 FPM을 이용하여 학습한 결과를 통해서 변경 패턴을 도출하고, 도출된 변경 패턴을 기준으로 사용자가 변경을 적용할 때 높은 확률로 함께 변경되어야 될 후보 코드를 알려준다 [2]. Zimmermann 등이 제안하는 방법은 코드 요소들 사이의 변경 패턴을 검출하는 데 적합하지만, 소스 파일 단위의 변경 패턴을 검출하는 데 한계가 있다. Ying 등은 Zimmermann 등과 유사하게 데이터 마이닝 기법 중 하나인 연관 분석 (Association Rule Mining) 을 이용하여 변경 패턴을 추출하는 방법을 제안했다 [14]. Antonio 등은 패턴 인식을 위해서 사용되는 DTW(Dynamic Time Warping)을 이용하여 함께 진화되는 파일들을 결정하는 방법을 제안했다 [15, 16]. Antonio 등이 제안하는 방법에서 DTW는 CVS의 변경 정보들만을 의존하기 때문에 조금의 데이터 노이즈 없이 잘 정리된 CVS 저장소를 필요로 한다. Xing 등은 클래스의 변경 릴리즈 기간 동안에 발생되는 변경 오퍼레이션을 기반으로 클래스의 진화 정보를 분석하는 방법을 제안했다 [17]. [17]에서 이용되는 DiffUML은 클래스의 진화 정보를 클래스 수준에서 추출하는 것이 가능하며, 우수한 진화 정보의 시각화를 제공할 수 있다. 하지만, 코드 수준이 진화 정보를 제공하는 데 한계가 있다. Livshits 등은 높은 연관관계를 가지는 메소드 호출 패턴을 추출하는 도구인 DynaMine 개발했다 [18]. DynaMine은 자바 소스 파일의 각 리비전에서 발생된 메소드 호출 문장 변경을 자동으로 검출한 후 마이닝 작업을 통해서 호출 문장 변경 패턴을 찾아낸다. 추출된 호출 문장 변경 패턴은 메소드, 클래스간의 변경을 예측하는 데 도움을 줄 수 있지만, 세밀한 코드 변경 및 파일 단위의 변경 예측하는 데 한계가 있다.

3. 변경 유형을 고려한 변경 결합도 정의

두 소스 파일이 자주 함께 변경되었을 경우, 그 소스 파일들은 변경 결합도가 높다고 볼 수 있다 [19]. 일반적으로 두 소스 파일 사이의 변경에 대한 결합 정도는 얼마나 자주 함께 변경되었는가를 기준으로 결정될 수 있다. 함께 변경된 횟수는 소프트웨어 버전 관리 시스템에서 관리하는 각 소스 파일들의 커밋된 시간을 통해서 쉽게 얻을 수 있다. 즉, 두 소스 파일이 특정 개발 기간 동안 3번 동일한 시간에 함께 변경되었다면 그들 사이의 공통 변경 빈도수는 3이다. 하지만, 서론에서도 잠시 언급했듯이 단순히 소스 파일들이 동일한 시간에 함께 변경되었다는 이유만으로는 그들이 관련성이 높다고 판단할 수 없는데, 그 이유는 관련성이 높은 파일들이 반드시 함께 커밋 된다는 보장이 없기 때문이다 [10]. 따라서, 단순히 함께 변경된 횟수를 기반으로 변경 결합도를 결정할 경우 잘못된 변경 결합 관계를 얻는 결과를 초래할 수 있다.

반면, 유사한 소스 코드 변경이 동시에 여러 소스 파일들에 적용되었을 경우, 이들은 향후 함께 변경될 확률이 높다. 예를 들어, 적용된 소스 코드 변경에 오류가 검출되어 이를 수정해야 될 경우, 동일한 코드 변경이 적용된 모든 소스 파일들은 동일하게 변경되어야 한다. 그러므로, 소스 파일들 사이의 변경 결합도를 판단할 때, 얼마나 자주 함께 변경되었는가와 더불어 어떠한 소스 코드 변경이 적용되었는지를 고려할 필요가 있다. 이러한 문제를 고려하기 위해서 소스 코드 변경에 대한 유사성을 구분하는 방법이 필요하다. Fluri 등은 여러 종류의 소스 코드 변경들을 일반화 할 수 있는 변경 유형 (Change type)을 정의했다. 하나의 변경 유형은 소스 파일에 실제로 적용된 소스 코드 변경을 나타낸다. 그러므로, 소스 코드 변경 유사성은 변경 유형 유사성을 비교하여 구분하는 것이 가능하다.

본 논문에서는 공통 변경 시간과 변경 유형 유사성을 이용한 변경 결합도 결정 방법을 제안한다. 이어지는 3.1에서는 Fluri 등이 정의한 변경 유형의 정의와 한계점 그리고 본 논문에서 새롭게 정의하는 변경 유형과 변경 유형을 추출하는 방법에 대해서 설명하고, 그 다음 이어지는 3.2절에서는

본 논문에서 정의한 변경 유형 유사도 측정과 변경 결합도에 대해서 설명한다.

3.1 변경 유형

Fluri 등은 35가지 변경 유형을 클래스 내부 및 메소드 내부 변경들로 나누고, 각 변경 유형의 변경 중요도를 정의했다 [20]. 변경 유형은 코드 수준에서 발생되는 변경 패턴 분석을 위해서도 사용될 수 있다 [21, 22]. 대표적으로 스테이트먼트 (Statement) 변경이 가장 빈번하게 발생되는 소스 코드 변경인데, 예를 들어, r-1, r 버전 사이에서 특정 메소드 내부에 새로운 스테이트먼트가 하나가 추가되었을 경우 해당 코드 변경은 STATEMENT INSERT로 구분하며, 스테이트먼트 변경은 메소드 내부 다른 코드에 영향을 미칠 가능성이 있지만, 클래스 전체 기능을 수정하지는 않기 때문에 이 변경에 대한 중요도를 보통 (Medium)으로 정의하고 있다. 그들은 변경의 중요성이 단순히 코드 라인의 추가/삭제 수에 비례하여 판단될 수 없다는 것을 전체 변경 중요도 값과 비교한 결과를 통해서 보였다.

Fluri 등은 두 소스 파일 버전 사이에서 적용된 소스 코드 변경들에 대해서 그들이 정의한 변경 유형으로 추출할 수 있는 Change distiller [6]를 개발했다. Change distiller는 소스 파일의 두 버전을 비교하여 클래스 내부 및 메소드 내부에 적용된 소스 코드 변경에 대한 변경 유형들을 추출한다. 위에서 언급했듯이 대표적으로 메소드 내부에서 가장 빈번하게 발생되는 변경 유형으로는 스테이트먼트 변경들, 예를 들어 STATEMENT INSERT, DELETE, UPDATE 등이 있다.

그런데 Fluri 등은 import 변경을 자신들의 변경 유형에 포함하지 않았다. import 변경 (Import change)은 클래스 외부에 적용되는 소스 코드 변경으로, 기존의 소스 코드에 새로운 import 코드를 추가하거나, 기존에 존재하는 import 코드를 제거 또는 수정할 수 있으므로 스테이트먼트 변경과 같이 3가지 형태의 변경 유형 (IMPORT INSERT, IMPORT DELETE, IMPORT UPDATE)으로 정의할 수 있다.

다음 표 1은 두 오픈 소스 프로젝트인 JDT (Eclipse Java Development tools)[1]와 CDT (C/C++ Development Tooling)[2]에서 발생된 import 변경 유형 발생 패턴을 보여준다. 표 1에서 import 변경이 발생된 경우와 그렇지 않은 경우들에 대해서 평균 변경 유형 발생 비율(Avg. CT)을 비교해보면 import 변경이 발생되었을 경우 더 많은 변경 유형이 발생되는 것을 확인할 수 있다. 즉, import 변경이 발생될 때 추가적으로 다른 코드 변경이 빈번하게 발생될 수 있다는 것을 나타낸다 [23]. 이는 곧 import 변경이 소스 코드 변경에 미치는 영향이 크다는 것을 의미한다. 또한 일반적으로 import 변경은 스테이트먼트 변경 다음으로 자바 소스 파일에서 빈번하게 적용될 수 있다. 따라서, 본 논문에서는 import 변경을 추가로 정의하여 포함시켰다. 그리고, 자바에서는 모든 예외는 클래스로 정의되기 때문에 예외 처리에 관련된 코드 변경들 또한 import 변경과 밀접한 관계가 있다. 예를 들어, 새로운 예외 처리를 위해서는 해당 예외와 관련된 import 코드를 반드시 추가하여야 한다. 반면, Fluri 등은 예외 처리에 관련된 코드 변경에 대한 변경 유형을 따로 정의하지 않고, 단순히 스테이트먼트 변경 유형으로 분류하고 있다. 따라서, 예외 처리와 관련하여 기존에 정의된 변경 유형을 가지고는 스테이트먼트 변경과 예외 처리 변경을 구분하는 것이 어렵다. 따라서, 본 논문에서는 단순히 스테이트먼트 변경 유형으로 분류되었던 예외처리 관련된 변경들을 TRY-CATCH, CATCH, THROWS, FINALLY와 같이 개별적으로 구분하여 분류하였다. 표 2에서는 import 변경과 더불어서 예외 처리와 관련된 변경 유형들과 각각의 변경 중요도를 보여준다. import 변경과 동일하게 4개의 예외 처리 관련 변경들은 추가 (INS), 삭제 (DEL) 그리고 수정 (UPD)으로 구분되고, TRAY-CATCH, CATCH 그리고 THROWS 변경은 클래스 전체 기능을 변경하지 않지만, 메

Table 1. Analysis result of IMPORT change

Repositories		JDT Core	JDT Debug	JDT.UI	CDT
Change history without IMPORT change	#.CH	35070	6545	24591	26991
	#.CT	408558	39968	140532	264512
	Avg.CT	11.64	6.1	5.71	9.8
Change history with IMPORT change	#.CH	7560	4256	20679	19110
	#.CT	167173	59662	319619	340963
	Avg.CT	22.11	14.01	15.45	17.84

Table 2. Added change types & significant Level

	Change types	Significant level
1	TRY-CATCH INS	Medium
2	TRY-CATCH DEL	Medium
3	TRY-CATCH UPD	Medium
4	CATCH INS	Medium
5	CATCH DEL	Medium
6	CATCH UPD	Medium
7	THROWS INS	Medium
8	THROWS DEL	Medium
9	THROWS UPD	Medium
10	FINALLY INS	Low
11	FINALLY DEL	Low
12	FINALLY UPD	Low
13	IMPORT INS	Low
14	IMPORT DEL	Medium
15	IMPORT UPD	Medium

1) http://projects.eclipse.org/projects/eclipse.jdt
2) http://projects.eclipse.org/projects/tools.cdt

소드 내부의 다른 코드에 변경 영향을 미칠 수 있기 때문에 보통 수준의 중요도를 가지는 반면 FINALLY 변경은 클래스 전체 기능 및 메소드 내부의 다른 코드에 아무런 영향을 미치지 않기 때문에 낮은 수준의 중요도를 가진다.

3.2 변경 유형 빈도 벡터 유사도 평가

3.1절에서 소개한 변경 유형은 회당 변경의 빈도수에 기반한 벡터로 변환 가능하며, 이렇게 변환된 벡터를 변경 유형 빈도 벡터라고 정의한다. 즉, 각 소스 파일의 특정 버전에서 발생된 여러 가지 코드 변경들은 변경 유형들로 추출할 수 있으며, 추출된 변경 유형들은 아래 (1)에서 정의하는 변경 유형 빈도 벡터로 나타낼 수 있다. 단일 변경 유형 빈도 벡터를 구성하는 전체 요소의 길이는 3.1절에서 소개된 전체 변경 유형의 종류의 수가 되며, 각 요소의 값은 이전 버전과 이후 버전, 즉 단일 버전 사이에서 해당 변경 유형이 발생된 횟수를 나타낸다.

정의 1 : 변경 유형 빈도 벡터

$$\overrightarrow{V_{a,r}} = < CT_1, CT_2, \ldots, CT_m > \quad (1)$$

여기서,

CT_i : 변경 유형 i가 발생된 빈도수

m : 변경 유형 빈도 벡터의 길이

$V_{a,r}$ 는 소스 파일 a의 리비전 r에 대해, 그 이전 리비전 (r-1)과 비교하여 발생된 모든 변경 유형들을 포함한다. 대부분의 변경 유형은 추가, 삭제, 수정 같이 3부분으로 분류되고, 코멘트 변경 같이 소스 코드에 아무런 영향을 미치지 않는 변경 유형은 제외하여 변경 유형 빈도 벡터의 길이 m은 56이 된다. 변경 유형이 수정되거나, 추가 및 삭제된다 하더라도 수식 (1)의 벡터 엔트리에만 관련이 있으므로, 수식 자체는 변경되지 않는다.

일반적으로 두 벡터의 유사성은 아래 식 (2)와 같이 벡터의 각 요소의 거리를 구하는 코사인 유사도 측정 [24]을 통해서 값으로 나타낼 수 있다.

$$CVesSim(f_{a,r}, f_{b,s}) = cossim(\overrightarrow{V_{a,r}}, \overrightarrow{V_{b,s}}) = \frac{\overrightarrow{V_{a,r}} \cdot \overrightarrow{V_{b,s}}}{\|\overrightarrow{V_{a,r}}\| \cdot \|\overrightarrow{V_{b,s}}\|} \quad (2)$$

여기서,

$f_{a,r}$: 소스 파일 a의 r번째 리비전

각 변경 유형 빈도 벡터는 특정 소스 파일의 특정 버전 사이에서 발생된 모든 변경 유형을 나타내기 때문에 위 식 (2)를 통해서 두 소스 파일의 특정 버전들 사이에서 발생된 코드 변경 유사성을 계산할 수 있다. 즉 파일 a의 "r" 버전에서 발생된 코드 변경과 파일 b의 "s" 버전에서 발생된 코드 변경이 완전히 동일하다면 1, 완전히 다르다면 0을 결과로 나타낸다.

두 소스 파일의 변경 결합도는 두 소스 파일이 얼마나 유사하게 자주 함께 변경되었는가로 나타낼 수 있다. 예를 들어, 두 소스 파일이 다섯 번 함께 변경되었고, 매 변경 거의 유사한 변경 유형이 적용되었다면 그 소스 파일들은 변경 결합도가 높다고 볼 수 있다. 따라서, 두 소스 파일의 변경 결합도를 다음 식 (3)과 같이 정의한다.

정의 2 : 두 소스 파일의 변경 결합도

$$ChCoup(f_a, f_b) = \sum_{<r_i s_j> \in CCom(f_a, f_b)} CVecSim(f_{ar}, f_{bs}) \quad (3)$$

여기서,

CCOM(fa,fb)={<ri,sj> | fa,fb가 함께 커밋되었을 때의 리비전 쌍, $r_i \in R$, $s_j \in S$, }

R: f_a의 리비전 집합, S: f_b의 리비전 집합

만약 두 소스 파일 f_a와 f_b가 각각의 전체 리비전 R, S 중에 5번 함께 변경되었다면, 집합 CCOM(f_a,f_b)는 5번의 리비전 정보로 구성되고, 각 리비전 'r'과 's'에서 발생된 모든 변경 유형의 유사성, 즉 변경 유형 빈도 벡터의 유사도를 기준으로 변경 결합도를 결정할 수 있다. 위 식 (3)은 해당 소스 파일들 사이의 공통 변경 횟수 (공통으로 변경된 빈도수)에 따라서 최대값이 달라지기 때문에 공통 변경 횟수를 기준으로 임계치를 결정하고, 높은 결합, 낮은 결합을 구분할 필요가 있다.

4. 사례 연구

4.1 실험 환경

소스 코드 저장소 마이닝 (Mining software repository)에 관한 여러 앞선 연구에서는 대부분 오픈 소스 프로젝트를 대상으로 다양한 실험을 수행해 왔으며 이클립스 프로젝트는 자주 이용된 타겟 시스템 중의 하나이다 [25]. 본 논문에서도 유명한 오픈 소스 프로젝트인 이클립스 JDT와 CDT의 Git 소스 코드 저장소를 실험 데이터로 선택했다. 다음 <표 3>은 각 소스 코드 저장소의 프로젝트 기간, 소스 파일의 수 그리고 전체 커밋의 수를 나타낸다.

CDT와는 달리 JDT는 코어, 디버그 그리고 유저 인터페이스에 대한 소스 파일들을 각각 구분하여 관리하고 있으며, JDT의 변경 히스토리는 2001년에서 2013년이며, CDT는 2002년부터 2013년까지 존재했다.

Table 3. Data of target projects

Repository	Period of change history	#file	#commit
JDT.Core	2001.6.5~2013.3.22	1782	42630
JDT.Debug	2001.5.18~2013.3.18	1241	10801
JDT.UI	2001.5.2~2013.3.22	3324	45269
CDT	2002.6.26~2013.5.8	5841	46101

4.2 평가 방법

실험은 크게 두 가지로 진행된다. 먼저 제안하는 방법과 공통 변경 횟수만을 고려한 변경 결합도 측정 방법으로 변경 예측을 수행한 후 그 결과들을 비교한다. 그리고 [12]에서의 비교 대상이었던 시간 기반의 변경 결합도 측정 방법(Time-based Coupling : TC)과 제안하는 방법으로 수행한 변경 예측 결과를 비교한다. [12]에서는 시간 기반의 변경 결합도가 변경 횟수 기반 결합도보다 더 나은 성능을 가진다는 것을 보이고 있다.

다음 식 (4)를 통해서 각 예측 모델의 예측 결과를 측정한다.

$$Prediction\ Result = \frac{|P \cap E|}{|P|} \times 100\% \quad (4)$$

여기서,

P : 예측 결과 집합

E : 평가 집합

예측 결과 집합 (P)는 각 예측 모델이 학습데이터를 기반으로 예측한 변경 결합도가 높은 파일들을 나타낸다. 평가 집합(E)는 실제로 변경 결합도가 높은 파일들, 즉 함께 변경된 파일들의 집합이다.

4.3 공통 변경 빈도 기반 결합도 측정 방법과 비교

본 절에서는 공통 변경 횟수만을 고려한 경우와 제안하는 방법으로 예측을 수행하는 경우를 통해서 얻은 결과를 비교한다. 실험에서는 우선 아래 그림 3과 같이 각 실험 프로젝트의 변경 히스토리를 두 부분으로 나누어서 한 부분을 예측을 위한 학습 데이터 (Reference Set)로써 사용하고, 나머지 부분을 예측 결과를 평가하기 위한 평가 데이터 (Evaluation Set)로 사용한다.

변경 예측은 먼저 학습 데이터에서 자주 함께 변경되는 소스 파일들을 찾는 절차와 결정된 소스 파일들이 평가 데이터에서 정말 함께 변경되었는지 확인하는 절차로 수행된다. 제안하는 방법의 경우, 함께 변경되는 소스 파일들을 찾은 후 3.2절의 식 (3)을 기반으로 실제 유사한 코드 변경이 발생된 소스 파일들을 필터링하는 절차가 추가된다. 정의된 변경 결합도에서 변경 결합성 판단을 위한 임계치 값은 실험을 통하여 5로 설정하였다.

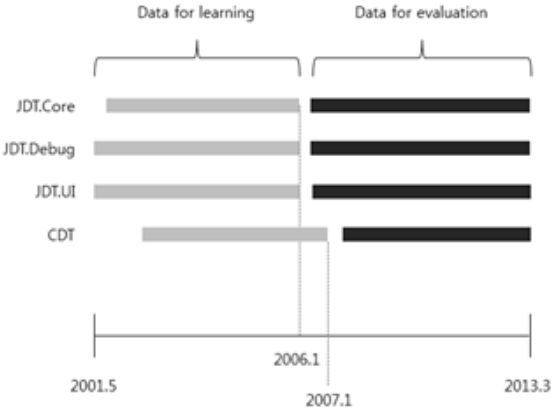

Fig. 3 The period of learning & evaluation data sets in each project

아래 표 4는 예측 결과를 보여준다. 각 프로젝트에서 수행된 예측 결과에서 제안하는 방법은 49%, 기존의 방법은 39%로 제안하는 방법이 평균적으로 10% 높은 변경 예측 결과를 가지는 것을 확인하였다.

Table 4. Comparative experimental results

Prediction Model	Proposed model (commit time+change types)	Existing model (commit time)
Repository	Prediction result	
JDT_DEBUG	57%	48%
JDT_CORE	46%	37%
JDT_UI	58%	46%
CDT	35%	24%
Average	49%	39%

4.4 시간 기반 결합도 측정 방법 (TC)과 비교

본 절에서는 시간 기반 변경 결합도인 TC와 비교한다. [12]에서는 TC를 이용하여 변경 예측을 수행할 때 실험 데이터를 3세션(3 Session)으로 나누고, 각 세션별로 변경 예측을 수행한 결과를 분석하였다. 본 논문에서도 [12]에서는 수행한 방식과 유사하게 실험을 수행하였다. 각 실험 프로젝트의 데이터 셋을 날짜로 구분하여 아래 표 5처럼 3세션으로 나누고, 각 세션에 대해서 제안하는 방법과 TC를 이용하여 얻은 변경 예측 결과를 비교한다.

Table 5. Date of Sessions for dataset

	Session1	Session2	Session3
JDT.CORE	2002.6~2005.5	2005.6~2009.5	2009.6~2013.5
JDT.DEBUG	2001.6~2005.5	2005.6~2009.5	2009.6~2013.3
JDT.UI	2001.5~2005.4	2005.5~2009.4	2009.5~2013.3
CDT	2001.5~2005.4	2005.5~2009.4	2009.5~2013.3

Table 6. Prediction results of CC and TC

		Session1	Session2	Session3	Avg
JDT.CORE	CC	33%	21%	40%	31%
	TC	37%	46%	33%	39%
JDT.DEBUG	CC	50%	57%	24%	44%
	TC	47%	27%	10%	28%
JDT.UI	CC	71%	43%	16%	43%
	TC	56%	26%	14%	32%
CDT	CC	42%	31%	40%	38%
	TC	23%	22%	25%	24%

표 6은 변경 예측을 수행한 결과를 보여준다. ‘JDT.CORE’에 대한 데이터 셋을 제외하고, 평균적으로 9~16% 정도 높은 결과를 얻는 것을 확인하였다. 반면, ‘JDT.CORE’에 대한 데이터 셋에 대해서는 상대적으로 평균 8%정도 낮은 예측 결과를 가지는 것을 확인할 수 있었다. 전체 평균으로 보면, 제안한 방법이 39%의 정확도를 보이며, 시간 기반 결합도는 약 31%의 정확도를 보이고 있다.

4.5 실험 결과 분석

횟수 기반 방법과의 비교(4.3절)는 제안하는 방법이 공통 변경 횟수보다 상대적으로 평균 10% 정도 높은 결과를 얻는 것을 확인할 수 있었다. 횟수 기반 방법 및 시간 기반 측정 방법과 비교한 결과, 본 연구에서 제안한 방법이 상대적으로 나은 결과를 보이는 것을 확인할 수 있었다. 기존 연구들에서 사용하는 평가 메트릭이 본 연구에서 사용하는 정확도 (식 (4))와 차이가 있고, 대상 데이터와 실험 방식도 다름을 감안하였을 때, 49%의 정확도 및 10% 향상도 의미가 있다고 볼 수 있다. 본 결과를 통하여, 단순히 변경 횟수만을 고려하여 변경 결합도가 높은 소스 파일들을 결정할 경우, 실제로 변경에 대한 연관성이 없는 소스 파일들을 잘못 식별할 수 있는 문제가 나타날 수 있다는 것을 보여준다. 이는, 제안하는 방법이 함께 변경된 횟수뿐만 아니라 세부적인 코드 변경을 추가적으로 고려하여 파일들 사이의 변경 결합도를 결정하기 때문에 나타난 결과로 해석할 수 있다. 따라서, 기본적으로 함께 변경된 횟수를 기반으로 우선 변경 결합도가 높은 소스 파일들을 구분한 후 제안하는 방법과 같은 추가적으로 결합 관계를 파악할 수 있는 요소들을 통해서 필터링 작업을 수행한다면 더욱 정확하게 변경 결합 연관성이 높은 소스 파일 검출이 가능하다는 것을 알 수 있다. 시간 기반 결합도와 비교한 4.4절의 예측 결과에서는 대체적으로 제안하는 방법이 더 좋은 예측 결과를 나타내었다. 그러나 두 방법은 결합도 정의 시 고려하는 요소가 차이가 있으므로 두 방법을 적절히 활용할 수 있다면 더 나은 결과를 얻을 것으로 예상된다.

본 연구에서 제안하는 변경 결합도는 타 연구와 유사하게 CVS, SVN 또는 Git 같은 소프트웨어 버전 관리 시스템으로 얻을 수 있는 변경에 대한 데이터들을 기반으로 소스 코드 변경 정보를 이용하여 변경 결합도를 결정한다. 또한 이용하고 있는 변경 유형은 대체로 객체 지향 언어들의 특징을 반영하고 있다. 따라서 본 연구는 버전 관리 시스템으로 관리되는 객체지향 소프트웨어들에 적용성이 있다.

5. 결 론

일반적으로 두 소스 파일이 자주 함께 변경된 경우, 이들은 변경 결합도가 높다고 볼 수 있다. 변경 결합도는 향후에 발생될 변경을 예측하는 데 중요한 요소가 되며, 일반적으로 소스 파일들 사이의 변경 결합도는 주로 해당 소스 파일들이 함께 변경된 횟수를 기반으로 결정되었다. 본 논문에서는 변경 횟수뿐 아니라 소스 코드 변경 유사성을 함께 고려하여 소스 파일들 사이의 변경 결합도를 결정하는 방법을 제안했다. 기존 [10]에 정의된 변경 유형 이외에 임포트 변경 및 예외 처리 관련 변경 등 모두 15개의 변경 유형을 추가로 정의하였고, 이를 활용하여 변경 유형 빈도 벡터를 정의하고, 두 소스 코드 변경 유사성을 코사인 유사도 측정을 통해서 결정하는 방법을 제시했다. 사례 연구에서는 제안하는 방법으로 수행한 변경 예측 결과와 공통 변경 빈도 기반과 시간 기반 결합도 측정 방법을 이용한 변경 예측 수행 결과를 비교 분석하였다. 공통 변경 빈도 기반 변경 결합도의 경우 제안하는 방법이 평균적으로 10% 정도 높은 예측 결과를 보이는 것을 확인할 수 있었고, 시간 기반 결합도의 경우 하나의 실험 프로젝트를 제외하고, 평균적으로 9~16% 정도 높은 예측 결과를 제안하는 방법이 보이는 것을 확인할 수 있었다.

향후 연구로 실험 데이터를 확대하고 다른 예측 모델을 설계하여 적용해 볼 것이다. 그리고 변경 유형 유사도를 통해 필터링 된 파일들, 즉 변경 유형 유사도는 높지 않으나 실제 변경 결합도가 높은 파일들을 찾을 수 있도록 확장할 것이다.

참 고 문 헌

[1] H. C. Gall, K. Hajek and M. Jazayeri, “Detection of logical coupling based on product release history.” In Proceedings of the IEEE Software Maintenance, 1998, pp.190-198.

[2] T. Zimmermann, P. Weilbgerber, S. Diehl and A. Zeller, “Mining version histories to guide software changes.” IEEE Transactions on Software Engineering, Vol.31, No.6, 2005, pp.429-445.

[3] H. Gall, M. Jazayeri and J. Krajewski, “CVS release history data for detecting logical couplings.” In Proceedings of the 6th International Workshop on Software Evolution, 2003, pp.13-23.

[4] M. Steff and B. Russo, “Co-evolution of logical couplings and commits for defect estimation.” In Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, 2012, p.213, 216.

[5] H. Kim and A. Zeller, “The impact of tangled code changes.” In Proceedings of the 10th International Workshop on Mining Software Repositories. 2013, pp.121-130.

[6] B. Fluri, M. Wursch, M. Pinzger and H. C. Gall. “Change distilling: Tree differencing for fine-grained source code change extraction.” IEEE Transactions on Software Engineering, Vol.33. No.11, 2007, pp.725-743.

[7] M. Fischer, M. Pinzger and H. Gall, “Populating a release history database from version control and bug tracking systems.” In Proceedings of IEEE International Conference on Software Maintenance, 2003, pp.23-32.

[8] H. Gall, M. Jazayeri, R. Klosch and G. Trausmuth, “Software evolution observations based on product release history.” In Proceedings of IEEE Software Maintenance, 1997, pp.160-166.

[9] H. Kagdi, J. I. Maletic and B. Sharif, “Mining software repositories for traceability links.” In Proceedings of 15th IEEE International Conference on Program Comprehension, 2007, p.145, 154.

[10] B. Fluri, H. C. Gall, and M. Pinzger, “Fine-grained analysis of change couplings.” In Proceedings of the 15th IEEE International Workshop on Source Code Analysis and Manipulation, 2005, pp.66-74.

[11] R. Robbes and M Lanza, “A change-based approach to software evolution.” Electronic Notes in Theoretical Computer Science, Vol.166, 2007, pp.93-109.

[12] R. Robbes, D. Pollet and M. Lanza, “Logical coupling based on fine-grained change information.” In Proceeding of 15th Working Conference on Reverse Engineering, 2008, pp.42-46.

[13] J. Han, J. Pei and Y. Yin, “Mining frequent patterns without candidate generation.” In Proceeding of the 2000 ACM SIGMOD international conference on Management of data, 2000, pp.1-12.

[14] A.T.T. Ying, G.C. Murphy, and R. Ng, and M.C. Chu-Carroll, “Predicting source code changes by mining change history.” IEEE Transactions on Software Engineering, Vol.30, No.9, pp.574-586, 2004.

[15] G. Antoniol, V. F. Rollo and G. Venturi, “Detecting groups of co-changing files in CVS repositories.” In Proceedings of the 8th International Workshop on Principles of Software Evolution, 2005, pp.23-32.

[16] S. Bouktif, Y. Gueheneuc and G. Antoniol, “Extracting change-patterns from cvs repositories.” In Proceedings of the 13th Working Conference on Reverse Engineering, 2006, pp.221-230.

[17] Z. Xing, and S. Eleni, “Understanding class evolution in object-oriented software.” In Proceedings of the 2th IEEE International Workshop on Program Comprehension, 2004, pp.34-43.

[18] B. Livshits and T. Zimmermann, “DynaMine: finding common error patterns by mining software revision histories.” In Proceedings of the 10th European software engineering conference on Software Engineering, 2005, pp.296-305.

[19] D. Jr, R. Michael, “Predicting software change coupling.” Ph.D. dissertation, University of Drexel, 2008.

[20] B. Fluri and H. C. Gall, “Classifying change types for qualifying change couplings.” In Proceedings of the 14th IEEE International Conference on Program Comprehension, 2006, pp.35-45.

[21] B. Fluri, G. Emanuel and H. C. Gall, “Discovering patterns of change types.” In Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008, pp.463-466.

[22] H. Gall, B. Fluri and M. Pinzger, “Change analysis with evolizer and changedistiller.” IEEE Software Vol.26, No.1, pp.26-33, 2009.

[23] J. Kim and E. Lee, “The Effect of IMPORT Change in Software Change History” In Proceedings of the 29th Symposium on Applied Computing, 2014. pp..(accepted)

[24] P. N. Tan, M. Steinbach and V. Kumar, “Introduction to Data Mining.” 1st ed., Addison Wesley, 2006.

[25] http://zorba.knu.ac.kr/research/MSR_Survey/overall_table.html

김 정 일

e-mail : 2009307043@knu.ac.kr

2011년 경북대학교 전자전기컴퓨터학부 (석사)

2011년~현 재 경북대학교 컴퓨터학부 박사과정

관심분야 : Automated bug fixing, Mining software repository

이 은 주

e-mail : ejlee@knu.ac.kr

1997년 서울대학교 계산통계학과(학사)

1999년 서울대학교 전산과학과(석사)

2005년 서울대학교 전기컴퓨터공학부(박사)

2005년 3월~2005년 10월 서울대학교 공과대학 BK박사후 연구원

2005년 11월~2006년 2월 삼성종합기술원 전문연구원

2006년 3월~현 재 경북대학교 컴퓨터학부 전임강사, 조교수, 현재 부교수

관심분야 : Mining software repository, Change pattern, software metric, software evolution