Git 히스토리를 이용한 코드리뷰 도구 구현*

오창욱[1], 정현오[1], 박현석[1], 손명희[1], 박혁주[1], 강동옥[2], 이용규[1]
[1]동국대학교 컴퓨터공학과-서울
[2]파수닷컴
e-mail : cndyqjqtk@naver.com

Implementation of a Code Review Tool Using Git History

Chang Uk Oh[1], Hyeon Oh Jung[1], Hyun suk Park[1], Myeong Hee Son[1]
Hyeok Ju Park[1], Dong Ok Kang[2], Yong Kyu Lee[1]
[1]Department of Computer Science and Engineering, Dongguk University - Seoul
[2]Fasoo.com

요 약

코드리뷰는 소프트웨어의 신뢰성을 향상시키며, 개발 기간을 단축시킨다. 기존의 코드리뷰 도구들은 문법적인 오류는 검출하지만, 논리적인 오류를 찾아내지 못하는 한계가 있다. 본 논문에서는 Git 히스토리를 이용하여 코드 간의 연관성 그래프를 만들고, 이를 이용하여 논리적인 오류를 찾아내는 도구를 구현하였다. 코드상의 논리적 오류를 검출하여 프로그램 개발을 용이하게 하고, 내부에 잠재되어 있는 결함을 예방할 수 있다.

1. 서 론

최근 코드리뷰에 대한 연구가 활발히 진행되고 있으며 Gerrit, Collaborator등의 도구가 널리 사용되고 있다[1][2]. 코드리뷰는 소프트웨어 품질과 중요한 연관성을 가지고 있다[1]. 따라서 적절한 코드리뷰는 소프트웨어 품질을 높이고 결함을 줄이는데 중요한 역할을 한다[3]. 기존의 코드리뷰 도구는 코드상에 존재하는 프로그래밍 언어의 문법적인 오류를 찾아서 표기한다. 그러나, 프로그래머의 실수나 오타로 인해 발생하는 논리적인 오류는 잘 찾지 못한다는 한계가 있다[4].

일반적으로 프로그램을 개발하는 과정에서 연관성이 높은 코드는 유사한 형식으로 구성되고, 동시에 수정되는 경우가 많다. 즉, 코드가 동시에 수정이 될 때마다 연관성 수치를 누적한다면 논리적인 의미에서 연관성이 있는 코드의 쌍은 높은 수치의 연관성 값을 가지게 된다. 연관성이 높은 코드의 쌍 중 한 쪽의 코드에서 수정이 발생한다면 연관성이 있는 다른 코드도 함께 수정이 이루어져야할 확률이 높다. 이러한 특징을 이용하여 코드 상에 존재하는 논리적 오류를 찾고, 사용자에게 알려줄 수 있다.

본 연구에서는 Git을 이용하여 분석 대상이 되는 소스코드에 대한 Git 히스토리를 수집하고, 코드 간의 연관성 수치를 누적하는 방법을 제안한다. 이를 기반으로 사용자가 소스코드를 작성 및 커밋(Commit)을 할 때 프로그램의 논리적인 오류를 지적할 수 있다.

* 본 연구는 과학기술정보통신부 및 정보통신기술진흥센터의 SW중심대학지원사업의 연구결과로 수행되었음(2016-0-00017)

2. 관련 연구

2.1 코드리뷰 도구

코드리뷰 도구는 오픈 소스 및 상용 소프트웨어 모두에서 많이 사용된다. 코드리뷰의 이점으로는 코드 검토에 따른 소프트웨어 품질 향상과 그에 따른 결함을 감소시키는 장점이 있다[1].

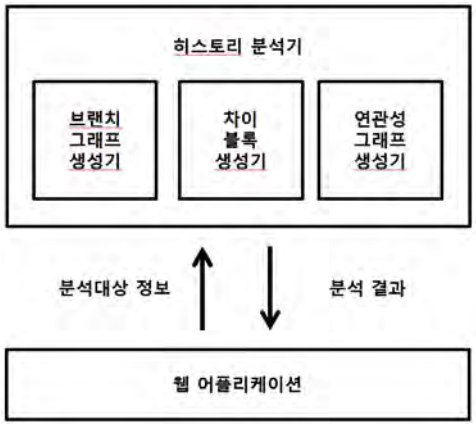
상용 코드리뷰 도구인 Gerrit, Collaborator 등의 코드리뷰기는 자동으로 수정할 코드를 알려주지 않으며, 일부 시각화와 함께 수정해야하는 코드를 지적하는 코드리뷰기에서도 문법 오류나 오타만을 탐지하며, 논리적인 오류는 탐지해 내지 못하고 있다[5].

2.2 Git 히스토리

Git은 대표적인 형상관리 도구로 파일의 변화를 누적해서 기록하며, 각 버전의 코드를 쉽게 관리할 수 있는 도구이다. 사용자가 Git의 커밋 명령어를 통해 코드의 내용을 저장하면, 이전 코드와 비교를 통해 수정된 내용과 일자, 그리고 커밋을 한 사용자 등의 정보가 Git 히스토리 로그에 기록된다. 그리고, 브랜치(Branch)를 생성하여 원래의 개발 과정과 독립된 작업 영역을 만들고 Git 명령어를 통해서 다시 병합할 수 있다. 병합 이후, 브랜치 생성 이전의 프로젝트의 브랜치에 수정된 사항들이 모두 반영된다. 브랜치의 생성 및 병합 역시 커밋의 단위로 이루어지며 Git 히스토리 로그에 저장된다.

본 논문에서 구현한 시스템은 Git 히스토리 로그를 사용하여 코드 간의 연관성을 누적하여 저장한다. 이를 기반으로 논리적인 오류가 존재할 시 수정을 권고한다.

3. 코드리뷰 도구 시스템 구성



(그림 1) 시스템 구성

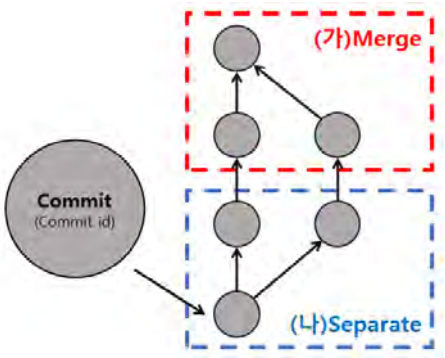
본 논문에서 제안하는 코드리뷰 도구의 시스템 구성은 (그림 1)과 같으며, 웹 애플리케이션과 히스토리 분석기로 구성되어있다. 웹 애플리케이션은 분석 대상 프로젝트의 Git 히스토리 파일을 업로드 하고, 분석한 결과를 보여주는 기능을 수행한다.

히스토리 분석기는 브랜치 그래프(Branch Graph) 생성기, 차이 블록(Diff Block) 생성기, 그리고 연관성 그래프(Correlation Graph) 생성기로 구성되어 있다. 웹 어플리케이션으로부터 Git 히스토리를 입력으로 받으면 그 결과로 각 시점에 대한 연관성 그래프를 만들고, 이를 이용하여 JSON 형식으로 작성된 최종 결과물을 웹 어플리케이션에 전달한다. 브랜치 그래프 생성기는 사용자가 Git 히스토리 정보를 이용하여 커밋 id로 구성된 브랜치 그래프를 생성한다. 이후 브랜치 그래프를 탐색하면서 차이 블록 생성기와 연관성 그래프 생성기가 작동한다. 차이 블록 생성기에서는 각 분석 시점이 되는 커밋 id 값을 입력으로 소스코드의 변화 정보를 받고, 이를 재구성하여 차이 블럭을 만든다. 이와 함께 이전 커밋 시점의 연관성 그래프를 조정한다. 차이 블록 생성기에 의해서 생성된 이전 시점의 연관성 그래프와 차이 블록은 연관성 그래프 생성기에 전달된다. 연관성 그래프 생성기는 전달받은 차이 블록을 이용하여 코드 블록(Code Block)를 생성한다. 그리고 생성된 코드 블록을 기준으로 이전 시점의 연관성 그래프를 업데이트하며, 분석 대상이 시점의 연관성 그래프를 완성한다.

4. 히스토리 분석기 모듈별 처리 절차

4.1 브랜치 그래프 생성기

브랜치 그래프 생성기는 Git 히스토리에 누적된 로그 데이터를 이용한다. 이를 통해 커밋 id로 구성된 브랜치 그래프를 생성한다. 생성된 브랜치 그래프는 방향성 그래프로 Git에서의 로그 순서와 일치하며, 히스토리 분석기 메인에서 사용된다. 이후 브랜치 그래프를 탐색하는 형태로 각 커밋 시점의 분석을 스케줄링 하는데 사용된다.



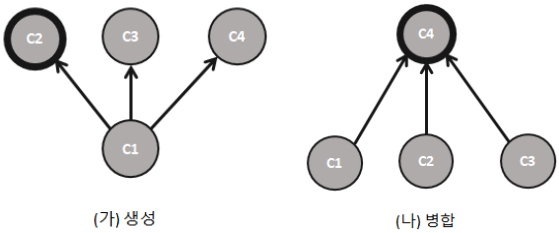
(그림 2) 브랜치 그래프

Git을 이용하면서 브랜치는 생성 혹은 병합된다. (그림 2)와 같이 생성되는 경우와 병합되는 경우는 각각에 대한 추가적인 처리가 필요하다. 특히 (가)의 병합되는 경우, 이전 시점의 모든 커밋에 대해서 연관성 그래프가 있어야만 한다. 따라서 브랜치 그래프를 위상 정렬하여 그래프를 탐색하는 것만으로 각 커밋의 처리 순서를 스케줄링 할 수 있도록 하였다.

4.2 차이 블록 생성기

차이 블록 생성기에서는 브랜치 그래프의 각 커밋 id를 입력으로 받는다. 그리고 해당 시점 이전의 커밋 id와 입력받은 커밋 id를 이용하여 Git으로부터 차이 정보를 받고 이를 이용하여 차이 블록을 생성한다. 차이 블록은 파일명과 삭제된 라인 정보, 그리고 추가된 라인 정보로 구성된다.

분석 시점의 커밋과 이전 커밋이 1:1의 관계라면 차이정보를 차이 블록으로 재구성 후 연관성 그래프 생성기에 전달하면 된다. 하지만, 브랜치가 생성 혹은 합병되는 경우에는 위와 같은 1:1 관계가 성립되지 않는다. 따라서 이러한 경우에는 별도로 다른 처리를 수행해야 된다.



(그림 3) 브랜치의 생성과 병합

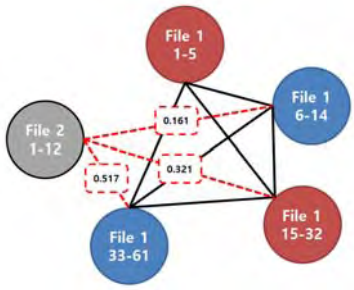
(가)와 같이 브랜치가 생성되는 경우에는 이전 시점의 연관성 그래프(C1)를 브랜치의 개수만큼 가중치를 분할한다. 이후 이전 시점과의 차이 정보를 가공하여 차이 블록을 만든다.

(나)의 경우 브랜치가 병합되는 경우에는 이전 커밋 시점이 여러 개가 된다. 따라서 이전 시점의 모든 연관성 그래프와 분석 시점(C4)과 이전 시점(C1, C2, C3) 간의 차이 정보를 생성하고 차이 블록을 만든다.

4.3 연관성 그래프 생성기

연관성 그래프 생성기는 차이 블록 생성기로부터 차이 블록과 이전 시점의 연관성 그래프를 입력으로 받는다. 만일 이전 시점이 없다면 차이 블록만 입력 받는다.

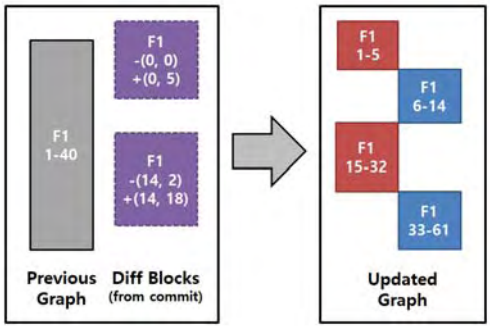
차이 블록은 파일명과 코드의 시작과 끝 라인 번호로 구성된 코드 블록을 생성하는데 이용된다. 생성된 코드 블록을 기준으로 이전 시점의 연관성 그래프를 업데이트 하고, 업데이트 된 이전 시점의 연관성 그래프를 모두 통합하여 연관성 그래프를 생성한다.



(그림 4) 연관성 그래프

연관성 그래프는 각각의 코드 블록간의 연관성을 그래프로 표현한다. 연관성 그래프는 코드 블록을 점으로 하고, 각 코드 블록 간의 연관성을 가중치의 형태로 표현한 그래프이다.

코드 블록을 만드는 과정은 다음과 같다. 입력받은 차이 블록을 코드 블록으로 변경한다. 연관성 수치를 누적하기 위해서는 이전 그래프에서의 특정 코드 블록이 분석 시점에서는 어디에 위치하는지 추적할 필요가 있다. 이는 차이 블록의 추가 및 제거된 라인 정보를 이용하여 다른 시점의 소스코드 라인을 동기화함으로써 해결하였다. 코드 블록과 코드 블록 간의 간선은 계속 되는 커밋을 통해서 수정 및 삭제된다.



(그림 5) 코드 블록 분할

코드 블록의 수정은 분석 시점의 차이 블록에 의해 생성된 코드 블록을 기준으로 두어 시행된다. (그림 5)와 같이 기존의 연관성 그래프가 특정 파일(F1)에 대해서 1-40까지의 코드 블록을 가지고 있다. 분석 시점의 커밋에 의해서 생긴 차이 블록은 좌측 박스 안의 점선으로 되어있는 부분이며 0번째로부터 5개의 라인 추가와 14번째로부터 2개의 라인제거 및 18개의 라인 추가라는 데이터를 가지고 있다.

연관성 그래프 생성기는 차이 블록을 이용하여 우측 박스의 (1-5)와 (15-32) 코드 블록을 생성한다. 이를 기준으로 하여 이전의 그래프의 코드 블록 정보를 수정한다. 기존의 연관성 그래프의 코드 블록이었던 (1-40)은 라인 정보 동기화 및 코드 블록에 의해 (6-14), (33-61) 코드 블록으로 분할한다.

코드 블록에 대한 업데이트 후, 코드 블록 간의 연관성을 의미하는 가중치를 계산한다. 분할 및 생성된 코드 블록이 이전 시점에서의 코드 블록에 대한 라인의 비율에 따라서 가중치를 분할한다.

(그림 6) 코드 블록 간 가중치 분할

분석 시점의 커밋으로 인해 최초로 생긴 코드 블록 간의 연관성은 가중치가 1인 간선으로 코드 블록을 연결한다. (그림 6)과 같이 이전 그래프에서 간선으로 연결되어 있는 경우 코드의 라인 비율로 가중치를 분할한다. 라인의 수의 추가 혹은 삭제 없이 차이 블록에 의해 생성된 코드 블록이 (3-7)이라면, 코드 블록 업데이트 알고리즘에 의해 (1-2), (3-7), (8-10) 코드 블록으로 수정된다.

코드 블록 업데이트를 하면서 새롭게 수정된 코드 블록에 대해 이전 연관성 그래프에 존재하던 코드 블록의 라인 비율을 분석한다. 이후 식 (1)을 통해 코드 블록 쌍에 대한 가중치를 최종 결정한다.

$$W_{new} = CB_1 * CB_2 * W_{old} \qquad \text{식 (1)}$$

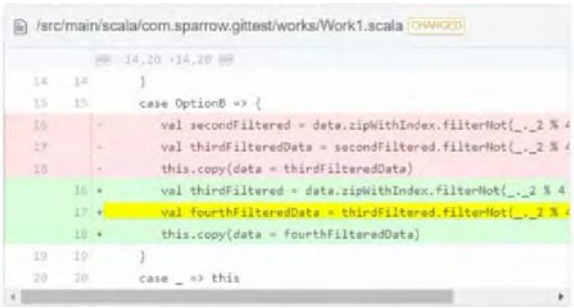
W_{new} : 계산된 가중치
W_{old} : 이전 그래프에서의 가중치
CB_1 : 이전 그래프에서의 코드 블록 A의 라인 비율
CB_2 : 이전 그래프에서의 코드 블록 B의 라인 비율

그 결과 (그림 6)의 우측과 같이 연관성 그래프의 가중치가 수정된다. 가중치 수정 후 분석 시점의 이전 시점들에 대한 연관성 그래프를 병합한다. 그러나 브랜치의 합병의 경우, 차이 블록 정보의 대부분이 브랜치의 합병을 위한 코드 수정 내용이므로 코드간의 연관성이 적다.

따라서 차이 블록에 의한 코드 블록의 수정과 연관성 수치의 수정은 이루어지지만, 새로 추가된 코드 블록 쌍에 대해 1의 가중치를 가진 간선으로 연결하지 않는다. 그 결과 간선이 존재하지 않은 코드 블록은 최종 연관성 그래프를 출력할 시 제거되며, 브랜치 합병을 위한 코드 간의 연관성을 무시하게 된다.

5. 구현 결과

웹 어플리케이션에서는 연관성 그래프의 결과를 기반으로 다음과 같은 결과를 보여준다. 코드리뷰기는 사용자가 업로드 한 .git 폴더를 일련의 과정을 통해 수정사항을 분석한다.



(그림 7) 파일별 코드 수정사항

코드 분석 후, 분석 시점의 파일별 수정사항을 (그림 7)과 같이 출력한다. 사용자는 웹 어플리케이션에 표시된 수정사항 중 분석할 대상 코드를 선택한다.

(그림 8) 연관 코드 리스트

분석 대상 코드가 선택되면 (그림 8)과 같이 연관성이 높은 코드 리스트를 표시한다. 각 코드의 수정 여부는 강조 색으로 구별한다. 현재 커밋에서 사용자가 수정한 사항은 점선 강조, 수정하지 않은 사항은 실선 강조로 표시하였다. 이를 통해 사용자는 분석 시점에 연관성이 높지만, 동시에 수정되지 않은 코드를 확인할 수 있다.

6. 결 론

본 논문에서, 우리는 Git 히스토리를 이용해 코드 간의 연관성을 분석하는 코드리뷰도구와 코드 연관도 분석 알고리즘을 제안하였다.

제안한 리뷰도구는 개발자에게 논리적 연관성이 높음에도 불구하고 수정되지 않은 부분에 의한 논리적 오류를 예방할 수 있으며, 개발 기간을 축소할 수 있음과 동시에 개발되는 소프트웨어의 신뢰성을 향상 시킬 수 있다는 이점을 제공한다.

그러나 소규모 프로젝트에서는 Git 히스토리의 규모가 작으며 따라서 최종적으로 생성되는 연관성 그래프의 신뢰성이 충분히 높지 않다는 문제가 있다. 이러한 문제를 해결하기 위해, 향후 프로젝트 규모에 종속되지 않는 연관도 측정 알고리즘의 고안이 필요할 것으로 사료된다.

참 고 문 헌

[1] McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E., “The Impact of Code Review Coverage and Code Review Participation on Software Quality: A case study of the qt, vtk, and itk projects”, In Proceedings of the 11th Working Conference on Mining Software Repositories ACM, pp. 192-201, 2014.

[2] Rachamadugu, Raghavendra, Perraju Bendapudi, Manoj Jain., “Integrated Code Review Tool″, U.S. Patent Application No. 11/754,231, 2008.

[3] McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E., “An Empirical Study of the Impact of Modern Code Review Practices on Software Quality”, Empirical Software Engineering, 21(5), pp. 2146-2189, 2016.

[4] Bacchelli, A., & Bird, C., “Expectations, Outcomes, and Challenges of Modern Code Review”, In Proceedings of the 2013 international conference on software engineering IEEE Press, pp. 712-721, 2013.

[5] Axelsson, S., Baca, D., Feldt, R., Sidlauskas, D., & Kacan, D., “Detecting Defects with an Interactive Code Review Tool based on Visualisation and Machine Learning”, In the 21st International Conference on Software Engineering and Knowledge Engineering, 2009.