|  |
| --- |
| 碩士學位 請求論文  指導敎授 李殷碩 |
|  |
| 버그 수정 자동화 성능 향상을 위한 버그 수정 패턴 우선순위화 연구  pattern prioritization for automated program repair effectiveness |
|  |
| 成均館大學校 一般大學院  소프트웨어學科  金 柱 亨 |

|  |
| --- |
| 碩士學位 請求論文  指導敎授 李殷碩 |
|  |
| 버그 수정 자동화 성능 향상을 위한 버그 수정 패턴 우선순위화 연구  pattern prioritization for automated program repair effectiveness |
| 成均館大學校 一般大學院  소프트웨어學科  金 柱 亨 |
|  |

|  |
| --- |
| 碩士學位 請求論文  指導敎授 李殷碩 |
|  |
| 버그 수정 자동화 성능 향상을 위한 버그 수정 패턴 우선순위화 연구  pattern prioritization for automated program repair effectiveness |
| 이 論文을 工學 碩士學位請求論文으로 提出합니다.  2021 年 4 月 日 |
| 成均館大學校 一般大學院  소프트웨어學科  金 柱 亨 |
|  |

|  |  |
| --- | --- |
|  | |
|  | 이 論文을 金柱亨의 工學  碩士學位 論文으로 認定함. | | |  |
|  | | | | |
| 2021 年 6 月 日 | | | | |
|  | | | | |
|  |  | | 審査委員長 정재훈 |  |
|  |  | |  |  |
|  |  | | 審査委員 이종욱 |  |
|  |  | |  |  |
|  |  | | 審査委員 이은석 |  |

**목차**

**목차 Ⅰ**

**표 목차 Ⅱ**

**그림 목차 Ⅱ**

**제1장서론.......................................................................................................1**

**제2장 배경 및 관련 연구 4**

2-1 연구 배경 4 2-1-1 Fault Localization 4

2-1-2 Automated Program Repair(APR) 5

2-2 관련 연구 6

2-2-1 Search-based APR 6

2-2-2 Learning-based APR 6

2-2-3 Template-based APR 7

2-3 연구 동기 8

**제3장 연구 접근법** 10

3-1 기존 연구 10

3-2 제안된 접근법 14

3-4 예제 16

3-4-1 성능 향상 예제 16

3-4-2 성능 저하 예제 18

3-5 버그 수정 패턴 변화 19

**제4장 실험 평가** 25

4-1 실험 환경 25

4-2 실험 결과 26

4-3 연구 문제 29

**제5장 결론** 31

5-1 결론 31

5-2 미래 연구 31

참고문헌 33

Abstract 51

**표목차**

표 1 ............................................................................................. 12

표 2. ............................................................................................. 13

표 3 27

표 4 28

**그림목차**

그림 1 ....................................................................................... 1

그림 2. ....................................................................................... 5

그림 3 8

그림 4 10

그림 5 14

그림 6 16

그림 7 17

그림 8 18

**버그 수정 자동화 성능 향상을 위한 버그 수정 패턴 우선순위화 연구**

본 논문에서는 소스코드 버그들의 위치를 식별하고 식별된 버그 위치를 자동으로 수정하는 버그 수정 자동화 연구의 효율을 높이기 위해 Fix director를 제안한다. 최근 소프트웨어의 규모와 복잡성이 커짐에 따라 유지 보수에 대한 인적, 시간적, 물적 비용이 크게 증가되었다. 이 문제를 해결하기 위해 소스코드의 버그를 식별하고 식별된 위치를 수정하는 모든 과정들을 자동으로 해결해주는 연구가 진행되고 있고 그 중 가장 성능이 좋은 방식이 개발자들의 수정 방식들을 패턴으로 만들어 적용시키는 것이다. 하지만 최근 수정 패턴 방식들의 여러 연구들은 수정 패턴을 적용시킬 때 식별된 위치의 소스코드의 구조적 또는 의미적인 정보들을 활용하여 어떤 수정법을 선택할 것인지 정한다. 이에 본 논문에서는 식별된 위치와 함께 어떤 종류의 에러 때문에 버그가 발생하였는지에 대한 정보를 활용하여 어떤 수정 패턴들이 우선적으로 적용되어야 하는지에 대한 근거를 마련하고 적용하였다. 이 제안 방법을 통해 최신 자동 버그 수정화 연구에서 가장 성능이 좋았던 연구에서 보다 더 빠르게 적합한 후보 패치를 생성할 수 있었다.

**주제어 : software engineering, automated program repair, template based APR, pattern prioritization**

**제1장 서 론**

최근 소프트웨어의 규모와 복잡성이 증가됨에 따라 인적,물적,시간적 비용이 크게 증가 되고 있다[1]. 이에 따라 소프트웨어의 버그를 자동화 할 필요성이 대두되었고, 이를 Automated Program Repair(APR)이라고 한다. APR의 프로세스는 다음과 같다. 소스코드의 어느 위치에서 버그가 발생하였는지 식별하는 Fault Localization(FL)의 단계를 먼저 거친다. 그리고 식별된 버그의 위치를 활용하여 다양한 방식의 수정법을 통해서 후보 패치를 생성하고 생성된 패치들을 검증하여 옳은 패치를 찾는다.

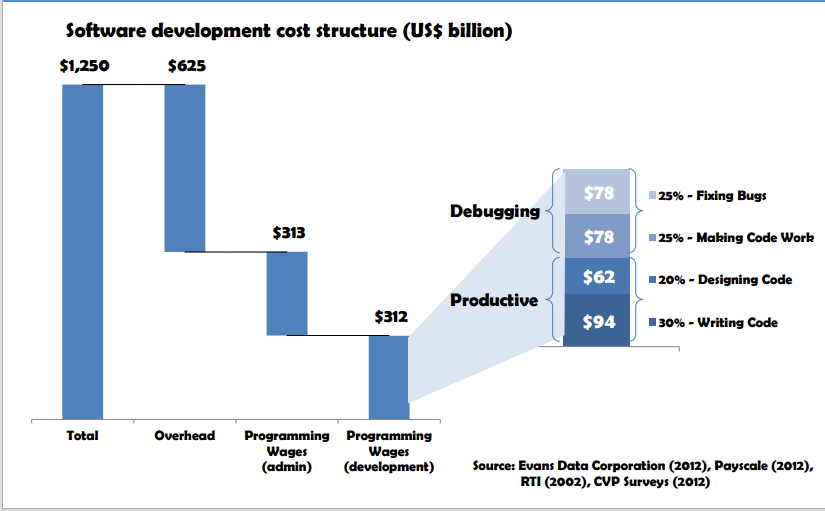


그림 1

이러한 관점에서 여러 APR기술들이 제안되고 있다. 대부분의 연구들은 search-based APR방식이다[2, 3, 4, 5, 6, 7, 8]. 이 기술들은 FL을 통해 식별된 버그가 발생된 소스코드들을 미리 정의한 mutation operator들을 적용시켜 후보 패치들을 생성하고 생성된 후보 패치들을 테스트 케이스를 활용하여 검증한다. 이 기발한 접근법을 활용하였음에도 불구하고, 여러 문제들이 존재한다. 첫번째는 이 APR기법이 옳은 패치를 생성하기 위해 찾아야하는 search space가 비약적으로 커져버리는 문제다. 두번째는 APR기법이 수많은 패치들을 생성하였음에도 불구하고 결국 그 후보 패치들에 옳은 패치가 존재하지 않는 문제다. 그렇게 되면 APR기술은 옳은 패치를 생성할 수 없다. 게다가 옳은 패치를 찾기 위해 search space를 크게 증가시켜도 옳은 패치를 찾을 수 있다는 확신이 없다. 오히려 시간이나 인적 비용만 증가시키는 결과를 만들 수도 있다. 이러한 문제를 해결하기 위해 templated-based APR[9, 10, 11, 12, 13, 14], machine learning-based APR[15, 16, 17, 18]과 같은 새로운 기법들이 제안되고 있다. 특히 template-based APR 기술들은 가장 효과적이며 뛰어난 성능을 보여주는 기법으로 평가되고 있다[19]. 이 접근법은 옳은 패치를 생성하기 위해 수정 패턴을 패치에 적용시킨다. 이 수정 패턴은 오픈 소스 레포지토리에서 사람에 의해 쓰여진 패치들의 정보를 추출하여 정의된다. 실제 사람에 의해 쓰여진 패치들에서 얻어진 소스코드들의 변화를 패턴화 하여 적용시키는 것이기 때문에, 우리는 더 효과적으로 옳은 패치를 찾고 이 옳은 패치를 찾기 위한 비용을 줄일 수 있다.

그런데, 이러한 노력에도 불구하고, APR 기법들은 아직도 투입한 비용에 비해 충분한 효과를 얻고 있지 못한다. 예를 들어서 이 template-based APR기법을 포함한 많은 기법들은 프로그램을 수정하는데 크게 세 단계를 거친다. 첫 번째 단계는 버그가 난 곳의 위치를 식별하는 Fault Localization(FL) 단계이다. 두번째 단계는 식별된 위치를 통해 여러가지 기법으로 후보 패치를 생성한다. 마지막 단계에서는 두번째 단계에서 생성된 후보 패치들을 주어진 모든 테스트 케이스들을 통과하는지 여부를 통해 옳은 패치를 찾게 된다. 이러한 과정에서 각 단계에서의 사용되는 정보들은 각각의 단계에서만 사용되고 버려지게 된다. 특히 실패 테스트 케이스들은 소스코드에 결함이 있는지 여부를 체크하는 것에 대부분 쓰이거나 또는 결함을 발생하는 위치를 식별하는 것에만 대부분 사용되며 결함을 수정하는 단계에서는 이 정보를 거의 사용되지 않는다. 반면에 실제 개발자가 소프트웨어의 버그를 수정할 때에는 단순히 테스트 케이스를 버그의 유무 확인이나 위치 식별로만 사용하지 않고 어떤 테스트 케이스가 실패하였는지 정보를 통해 어떤 버그가 발생하였는지에 대한 근거를 습득하고 그것을 바탕으로 버그 수정의 방향을 결정한다. 실제 개발자의 버그 수정 프로세스와 다른 APR기법의 이러한 정보의 단절은 프로그램을 수정하는데 있어 치명적인 손실이 될 수 있다.

본 논문에서는 위의 한계를 개선하기위해 실패 테스트 케이스의 정보를 이용하여 수정 방식의 근거를 마련한다. 실패 테스트 케이스를 활용하여, 어떤 결함이 일어났는지 추론하여 어떤 수정 패턴을 우선적으로 적용해야 하는지 결정한다. 이를 통해 버그 수정의 효율을 높이고 비용을 줄이려 한다.

**제 2장 배경 및 관련 연구**

**2-1 연구 배경**

2-1-1 Fault Localization(FL)

APR은 소스코드 버그를 FL기법을 통해 의심스러운 소스코드의 위치를 수집한다. 많은 연구자들은 spectrum-based, mutation-based, slice-based 기법과 같은 다양한 방식의 기법들을 제안해왔다. 이 기법들 중에서, Spectrum-based Fault Localization(SFL) 기법은 전체 FL 연구 중에서 35퍼센트를 차지하고 있는 인기있는 기법이다. 이 SFL은 의심스러운 코드들의 리스트를 순위화 하여 보여주며 이 코드들은 문장, 함수, 괄호, 클래스 단위들로 정의된다. SFL은 passing test와 failing test로 이루어진 test suite들을 이용해 측정된다. 각 statements들은 각각의 test cases들에 의해 실행되며 failing test와 passing test를 측정하여 이 수치를 계산하여 suspiciousness score를 측정하게 된다. 이 점수를 측정하는데 있어서 많은 SFL 기법들은 Tarantula, Jaccard와 같은 여러가지 방식의 유사도 계수를 통해 위치 식별 성능의 향상을 이루어 내고 있다. 이 유사도 계수를 구하는 방식들을 통해서 FL은 의심스러운 코드들을 순위화 하여 개발자들에게 제공되고 APR 연구에서는 이 리스트들을 활용하여 각각의 해당 기법들이 후보 패치를 진행하게 된다.

2-1-2 Automated Program Repair(APR)

APR 전체적인 프로세스 다음과 같다. FL 기법들을 통해 Suspiciousness score를 측정하게 되면 일련의 의심스러운 코드들의 리스트를 얻게 되고 이 리스트에 따른 순서에 따라 연구자들이 설계한 APR tools을 통해 후보 패치들을 생성한다. 생성된 후보 패치들은 주어진 test suite에 의해서 검증 절차를 밟게 된다. 이를 validation step이라고 하며 모든 test cases에 대하여 모두 통과한 패치를 (그럴듯한 패치)plausible patch로 정의한다. 실제 후보 패치가 모든 test cases들을 통과한 상태인 이 plausible patch는 개발자들의 마지막 검증이 필요하다. 주어진 test cases들에 통과하였다 하더라도 해당되는 test cases들에서만 올바르게 작동되는 상태일 수 있으며 이를 test cases에 overfitting되어 있다고 정의한다. 많은 APR연구들에서 이 plausible patch의 overfitting문제를 해결하기 위해 여러 방식을 제안되고 있다. 최종적으로 이 plausible patch가 사람에 의해서 옳게 패치가 된 것인지 검증이 완료된 것을 옳은 패치(correct patch)라고 정의한다. 사람의 개입없이 correct patch를 검증하려는 연구 역시 진행되고 있으며 correct patch의 비율을 높이는 연구 역시 활발히 진행되고 있다.

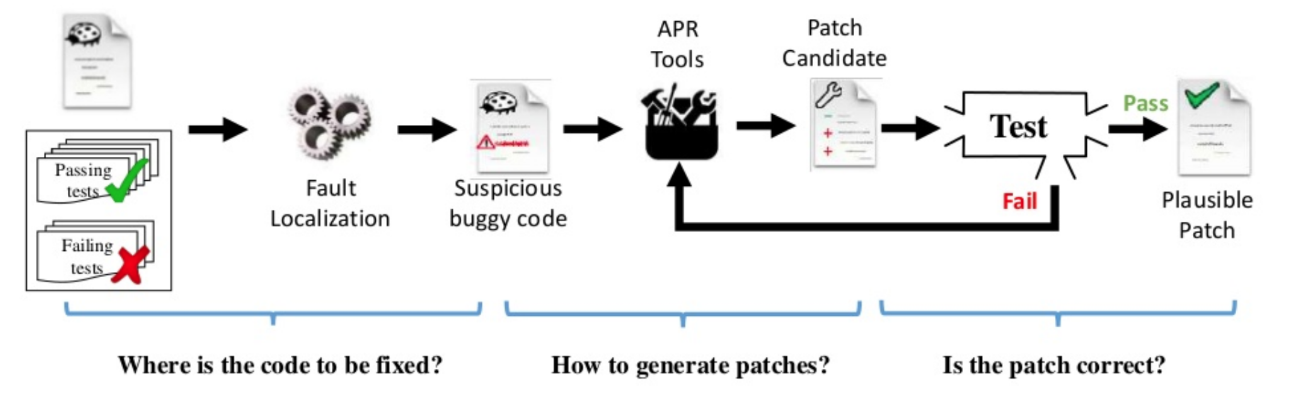


그림2

**2-2 관련 연구**

2-2-1 Search-based APR

기존의 Search-based APR기법은 FL을 통해 버그의 위치가 식별되면 미리 정의한 mutation operator들의 셋을 적용해서 후보 패치들을 임의적으로 생성한다. 생성된 후보 패치들을 주어진 테스트 케이스들을 모두 통과하는지 검증하는 과정을 통해 옳은 패치를 찾는다. Search-based APR기법은 특별히 추가적인 명세 없이 다양한 종류의 버그들에 대해 후보 패치들을 생성하였다. 하지만 이러한 효과에도 불구하고, 큰 두가지의 한계가 있다. 첫번째는 옳은 패치가 search space에 항상 존재하지 않는다는 것이고, 이것은 버그를 성공적으로 수정하지 못하도록 한다. 두번째 한계는 폭발적인 search space 증가 문제이다. 옳은 패치를 생성하기 위해서는 search space를 늘릴 필요가 있고, 이는 많은 비용이 소모되도록 한다. 결국 두 요소는 trade-off관계를 가지며 이 문제를 해결하기 위해 다른 방식의 APR기법들이 제안되었다.

2-2-2 Learning-based APR

기존의 search-based APR의 한계를 극복하기 위해 learning-based APR기술이 적용되었다. 그 중 가장 활발히 적용되는 learning -based APR 기법은 Neural machine translation(NMT)을 활용하는 기법이다. NMT기법은 신경망 구조를 사용해서 시퀀스를 입력 받아 특정 시퀀스들을 얻게 하는 심층 학습 접근 방식이다. NMT는 인코더와 디코더 구조를 통해서 버그가 발생한 코드와 옳게 패치 된 코드들을 training data와 target data로 각각 입력하여 학습시킨다. 이와 같은 방법을 통해 수동적으로 포착하기 힘든 소스코드들의 변화를 학습시킬 수 있어 이전의 기법들 보다 복잡한 형태의 버그들에 대해 효율적으로 수정할 수 있다. 또 기존의 기법들은 프로그래밍 언어가 달라짐에 따라 수정 방식이 달라지는 문제가 있어왔는데 NMT기법은 모델을 구성하게 되면 별 다른 비용 소모 없이 다양한 언어에 대해 적용시킬 수 있다.

2-2-3 Template-based APR

Template-based APR(Pattern-based APR이라고도 한다)은 사람에 의해 쓰여진 패치들로부터 사람이 버그 코드를 수정하는 방식들을 추출하여 패턴화 시켜 구성하여 이 수정 패턴들을 활용해 버그 코드에 적용시켜 자동으로 후보 패치들을 생성하는 기법이다. 기존의 template-based APR 기법들은 제안된 연구들에 따라서 다른 방식의 추출법과 수정 패턴 선택법 있다. Template-based APR 연구들 중 하나인 CAPGEN[14]에서처럼 실제 코드들에서 많이 쓰이는 수정 패턴들에 대해 높은 가중치를 부여하고 FL을 통해 버그를 포함하고 있는 위치의 소스코드들의 구조적인 유사도를 활용하여 수정 패턴을 선택하거나 또는 TBar[19]와 같은 연구에서는 식별된 위치에서 구성된 노드들의 종류에 따라 패턴을 선택하는 방식을 채택하여 패치를 생성한다. Template-based APR은 수정 패턴을 활용한 절차를 통해 기존의 search-based APR의 한계점들을 해소시켜주며 최근 가장 각광받고 있는 APR기법이다.

**2-3 연구 동기**

현재 APR기법중에서 가장 성능이 좋은 것으로 알려진 기법은 template-based APR기법이다. 기존의 search-based APR기법의 한계점들을 어느정도 극복하였고 실제 벤치마크 성능에서도 우수한 성능을 보여주고 있다. 많은 template-based APR기술들은 위에서 언급한 최신 기술들을 포함하여 자신들의 고유한 접근법을 활용해 수정 패턴을 수집하고 각각의 버그에 대해 적합한 패턴을 매칭 시킨다. 하지만 대다수의 template-based APR기법들에서는 수정패턴을 선택함에 있어서 대부분의 기법들이 왜 버그가 발생하였는지에 대한 고려가 없이 식별된 의심스러운 코드들의 위치 정보나 변수명이나 노드 타입과 같은 식별된 코드의 정보만을 활용한다. 실제 개발자들의 경우 코드상에서 버그가 발생함을 인식하게 되면, 어떤 곳에서 버그가 발생하였는지를 확인함과 동시에 어떤 이유 때문에 소스코드가 문제를 발생시키는지에 대한 이유를 살펴보고 이를 근거로 어떤 수정을 해주어야 하는지 결정하게 된다.

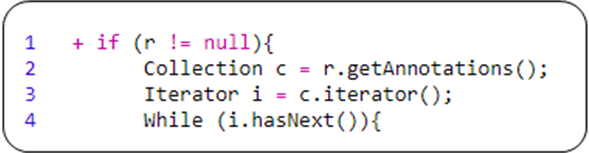


그림 3

그림3은 Defects4j 벤치마크에서의 Chart-4에 나타나는 버그이다[22]. FL을 통해서 Line2에서 버그가 발생함을 인식하게 되면, 대부분의 APR 어떤 문제점을 고쳐야 하는지 고려 없이 Line2의 구조적인 정보 또는 Line2를 포함한 주변 코드들의 구조정보들을 활용하여 수정 패턴들을 적용시킨다. 이러한 접근은 재원과 시간적인 비용에 대한 상당한 낭비가 될 수 있다. 하지만 테스트 케이스를 통해 NullPointerException 에러메시지가 발생되었고 이 정보를 활용하게 된다면 우리는 어떤 수정 행동을 취해야 하는지 근거를 얻을 수 있을 것이다. 이러한 과정은 프로그램 수정을 보다 더 용이하게 만들어 줄 것이다.

**제 3장 연구 접근법**

**3-1 기존 연구**

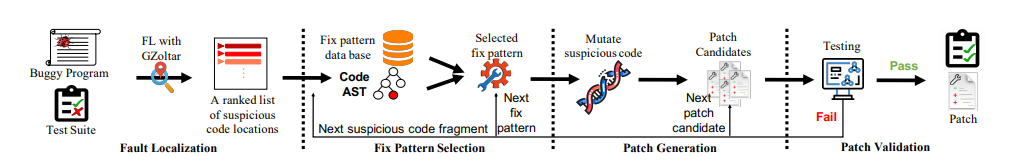


그림4

그림4는 현재 APR기법중 가장 성능이 좋다고 알려진 TBar의 전체 프로세스이다. TBar는 Template-based APR기법으로서 기존의 APR프로세스를 따르고 있다. FL을 통해 수집된 의심스러운 코드들의 리스트를 받아 코드들의 구조적인 정보를 통해서 수정 패턴을 선택하고 후보 패치들을 생성한 이후 주어진 테스트 케이스들을 통해서 검증 단계를 통해 최종적으로 옳은 패치를 산출하게 된다. TBar에서는 수정 패턴을 선택하기 위해서 코드를 AST구조로 구성하고 식별된 의심스러운 코드들의 자식 노드들을 탐색하며 매칭되는 수정 패턴을 고른다. 다음 표 1은 TBar에서 어떤 정보를 통해서 패턴을 고르는 것인지에 대한 정보이다. 버그를 포함한 코드의 위치가 식별되면 해당 코드의 자식 노드들을 순차적으로 순회하며 Bug Context에 해당되는 노드가 존재하면 해당 수정 패턴을 선택하게 된다. 예를 들어 식별된 코드가 infix-expression을 보유했다면 수정패턴 11을 선택하여 패치를 생성하는 방식이다. 식별된 코드는 한 개 이상의 자식 노드를 보유하기 때문에 여러 개 이상의 수정 패턴들이 존재한다. 해당 연구에서는 앞서 언급했듯이 버그를 포함한 코드의 위치가 식별되고 해당 위치의 구조적인 정보를 활용하기 때문에 어떤 수정 행동을 취해야 되는지에 대한 근거가 부족하다. 이로 인해 여러 경우에서는 불필요한 수정 행동을 취하게 된다.

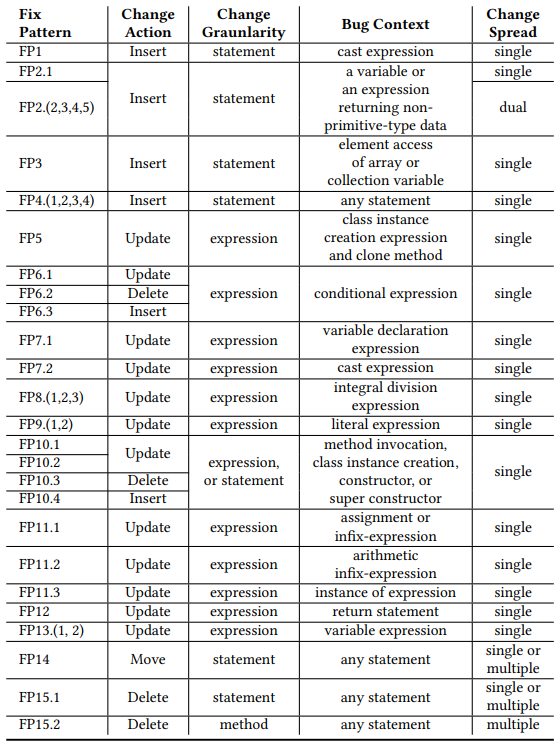


표 1

|  |  |
| --- | --- |
| 수정패턴번호 | 수정 패턴 종류 |
| FP 1 | Insert Cast Checker |
| FP 2 | Insert Null Pointer Checker |
| FP 3 | Insert Range Checker |
| FP 4 | Insert Missed Statement |
| FP 5 | Mutate Class Instance Creation |
| FP 6 | Mutate Conditional Expression |
| FP 7 | Mutate Data Type |
| FP 8 | Mutate Integer Division Operation |
| FP 9 | Mutate Literal Expression |
| FP 10 | Mutate Method Invocation Expression |
| FP 11 | Mutate Operator |
| FP 12 | Mutate Return Statement |
| FP 13 | Mutate Variable |
| FP 14 | Move Statement |
| FP 15 | Remove Buggy Statement |

표 2

해당 표는 TBar에서 정의한 15개의 수정 패턴이다. 각각의 수정 패턴은 버그를 포함하고 있는 코드의 자식 노드들의 구성에 따라 수정 패턴이 정해지게 된다.

**3-2 제안된 접근법**

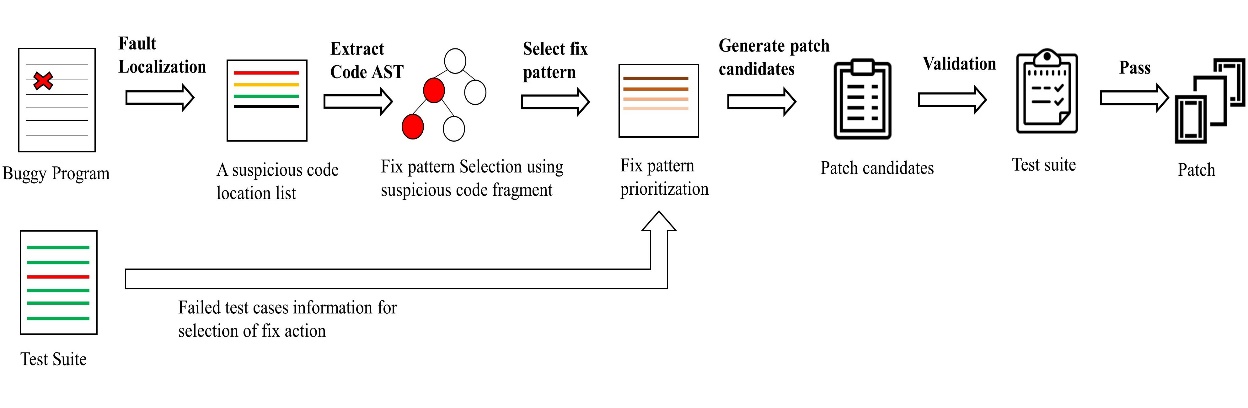


그림 5

그림 5는 우리의 접근법에 대한 전체적인 프로세스를 나타낸 것이다. 기존의 template-based APR기법에서 진행되는 버그 위치 식별, 패턴 매칭, 후보 패치 생성, 검증 단계를 거치게 된다. 우리 연구에서는 이 과정 중 패턴 우선순위화 단계를 추가적으로 적용한다. 앞서 언급했듯이 실제 개발자들은 단순히 버그가 발생한 위치에 해당되는 코드들의 구조적인 정보만 활용하는 것이 아니라 왜 버그가 발생하였는지에 대한 전체적인 고려를 포함하여 어떤 수정 행동을 취할 것인지 결정하는데 우리 연구에서도 이 프로세스를 거치기 위해 실패한 테스트 케이스들의 정보를 활용한다. 실패한 테스트 케이스의 에러 메시지를 통해서 어떤 타입의 버그가 발생하였는지에 대해 정보를 얻고 더 적합한 수정 패턴을 우선적으로 적용하여 후보 패치를 생성한다. 널 포인터 예외 처리에 관한 에러가 발생한 경우 널 포인터에 관련된 수정 패턴을 우선적으로 적용시키거나 또는 기대값의 데이터 타입이 다른 경우에 나타나는 에러메시지가 등장하였을 경우 데이터 타입을 바꿔주는 패턴을 우선적으로 적용시켜 후보 패치를 생성한다. 우리는 총 8개의 에러메시지 타입에 대하여 수정 패턴들을 우선순위화 하는 작업을 거쳤으며 현재 가장 성능이 좋은 것으로 알려진 TBar를 이용해 우선순위화 전과 후의 차이를 비교한다.

각각의 8 개의 에러 타입에 대한 우선적으로 적용 시킨 패치 패턴들은 다음과 같다.

1. AssertionFailedError : expected <정수> but was <정수> 해당 에러의 경우는 기대값과 다른 결과값이 나온 경우 이다. 이 경우에는 우선적으로 Variablereplacer, operatormutator, literalexpression의 패턴들을 적용하였다. 계산값이 달라지는 경우이기 때문에 수치의 변화에 줄 수 있는 수정 행동이 우선적으로 필요하다고 판단하였다.
2. NullPointerException 널포인터 예외 에러에 관해서는 널포인터 체커 수정 패턴이 우선적으로 취해지도록 설정하였다.
3. illegalAgrumentException 해당 에러에 관해서는 잘못된 인자값에 관한 에러이므로 variable replacer와 conditional expression의 패턴들을 우선적으로 적용시키도록 설정하였다.
4. ComparsionFailure expected:<fredag, week [53]> but was:<fredag, week [01]> 해당 에러는 기대값과 결과값이 다른 경우이므로 마찬가지로 variablereplacer, operatormutator, method invocation mutator 수정 패턴을 우선적으로 적용하였다.
5. ClasscastExxception 해당 에러메시지는 잘못된 객체 형에 관한 에러이기 때문에 Classcastchecker 수정 패턴을 우선적으로 적용 하였다.
6. Assertion FailedError : expected <0> but was<0> 해당 에러는 기대값과 결과값의 타입이 다른 경우에 발생한 에러이기 때문에 우선적으로 LiteralExpressionMutator 패턴을 적용하였다.
7. ArrayIndexOutOfBoundsException 해당 에러는 잘못된 인덱스에 관한 에러이므로 variablereplacer, operatormutator, conditional expression 패턴들을 우선적으로 적용시키도록 설정하였다.
8. IllegalFieldValueException 마찬가지로 잘못된 값을 호출한 경우에 발생되므로 variablereplacer를 우선적으로 적용시키도록 설정하였다.

**3-4 예제**

3-4-1 성능 향상 예제

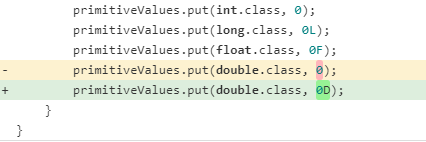


그림 6

해당 예제는 Defects4j의 Mockito프로젝트에서 발생된 버그이다. 기존의 TBar에서는 -로 표시되어 있는 문장이 버그로 식별된 이후 문장의 구조 정보를 활용하여 수정 패턴을 선택한다. 문장의 자식노드들의 타입을 통해 표1에 나오는 bug context에 매칭시켜 해당되는 수정패턴들을 순차적으로 적용시켜 후보 패치를 생성하게 된다. 기존의 TBar에서는 variableReplacer -> NullpointerChecker -> LiteralExpressionMutator 순으로 수정 패턴을 적용하게 되는데 실제 옳은 패치는 LiteralExpressionMutator 수정 패턴 적용을 통해서 생성된다. 우리의 접근법을 활용하게 되면 해당 버그에서 발생되는 ClassCastException에러를 통해서 데이터 형식의 문제인 것을 확인할 수 있게 되고 그에 따라 LiteralExpressionMutator 수정 패턴을 우선적으로 적용시켜 옳은 패치를 빠르게 생성하게 된다.

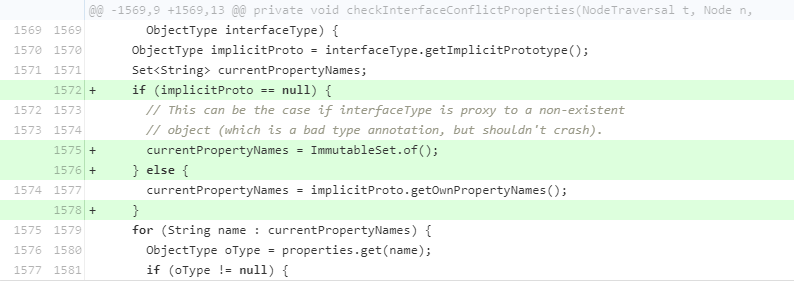


그림 7

그림 7은 Defects4j 벤치마크의 Closure 프로젝트에서 bugid2의 예시이다. 해당 버그는 line1574가 버그가 발생한 위치라고 식별되고 TBar에서는 마찬가지로 이 문장의 자식 노드들의 타입을 순차적으로 순회하여 표1에 해당되는 bug context에 맞는 수정 패턴들을 적용한다. 순서대로 VariableReplacer -> NullpointerChecker -> Method invocation Mutator 수정 패턴을 적용하게 된다. 해당 버그의 옳은 패치는 NullpointerChecker 수정 패턴에서 생성되며 이 패턴 역시 해당 버그에서 발생되는 NullpointerException 에러 메시지 정보를 통해서 우선적으로 적용될 수 있는 수정 패턴이다.

3-4-2 성능 저하 예제

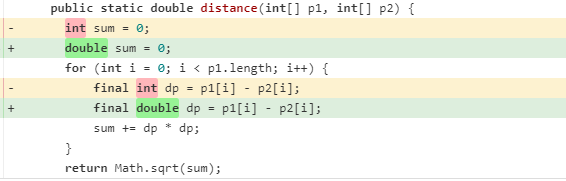


그림 8

해당 버그는 Defects4J의 Math 프로젝트 bugid 79번 예제이다. 해당 버그도 마찬가지로 버그를 포함하고 있다고 파악되는 문장이 식별되면 자식 노드들의 구성요소를 순회하여 맞는 수정 패턴을 채택한다. 그에 따라 DatatypeReplacer -> NullPointerchecker 순으로 패턴을 적용하게 되는데 이 경우는 DatatypeReplacer 수정 패턴으로 옳은 패치를 생성할 수 있었다. 우리가 제안한 에러메시지를 활용해서 수정 패턴을 우선순위화 하게 된다면 Nullpointerchecker 패턴을 우선적으로 적용하여 옳은 패치가 오히려 이전보다 늦게 생성되는 경우가 발생된다. 이는 DatatypeReplacer 수정 패턴으로도 해당 에러메시지를 처리할 수 있기 때문에 발생되는 문제점이다. 그렇기 때문에 단순 에러메시지 뿐만 아니라 버그발생에 관한 더 구체적인 정보의 필요성이 보였다.

**3-5 버그 수정 패턴 변화**

우리는 15개의 향상된 케이스와 2개의 성능 저하된 케이스를 측정하였다. 각 버그의 프로젝트명과 버그id 그리고 에러메시지 타입, 우선순위화 하기전과 후의 패턴 변화에 대한 구체적인 정보를 수집하고 이로 인해 옳은 패치를 생성하기 위해 생성하는 후보 패치수의 변화 또한 같이 구하였다.

Chart\_4

에러 메시지 : NullPointerException

수정 패턴 순서 ; Variable replacer -> Nullpointer Checker

옳은 패치 생성 패턴 : NullPointerChecker

우선순위화로 인한 패턴 랭킹 변화 : 2 -> 1

후보 패치 랭킹 변화 : 70 -> 1

Chart\_9

에러 메시지 : illegalArgumentException

수정 패턴 순서 ; Operator mutator -> conditional expression

옳은 패치 생성 패턴 : conditional expression

우선순위화로 인한 패턴 랭킹 변화 : 2 -> 1

후보 패치 랭킹 변화 : 11 -> 1

Chart\_24

에러 메시지 : illegalArgumentException

수정 패턴 순서 ; Datatype Replacer -> variable Replacer

옳은 패치 생성 패턴 : Variable Replacer

우선순위화로 인한 패턴 랭킹 변화 : 2 -> 1

후보 패치 랭킹 변화 : 7 ->1

Closure\_2

에러 메시지 : NullPointerException

수정 패턴 순서 ; Variable -> NullPointer Checker

옳은 패치 생성 패턴 : NullpointerChecker

우선순위화로 인한 패턴 랭킹 변화 : 2->1

후보 패치 랭킹 변화 : 37 -> 1

Closure\_70

에러 메시지 : AssertionFailedError Expected<2> but was<1>

수정 패턴 순서 ; method invocation mutator -> variable replacer -> nullpointer checker -> literal expression

옳은 패치 생성 패턴 : literalexpression

우선순위화로 인한 패턴 랭킹 변화 : 4 -> 3

후보 패치 랭킹 변화 : 71 -> 53

Closure\_117

에러 메시지 : ComparisonFailure: expected:<...p never defined on C[2]> but was:<...p never defined on C[3.c2\_]>

수정 패턴 순서 ; variable replacer -> method invocation mutator -> variable replacer -> statement mover

옳은 패치 생성 패턴 : statement mover

우선순위화로 인한 패턴 랭킹 변화 : 4 -> 5

Lang\_26

에러 메시지 : ComparisonFailure : expected:<fredag, week [53]> but was:<fredag, week [01]>

수정 패턴 순서 ; variable replacer -> nullpoiniter checker -> method invocation mutator

옳은 패치 생성 패턴 : method invocation mutator

우선순위화로 인한 패턴 랭킹 변화 : 3 -> 2

후보 패치 랭킹 변화 : 45 -> 38

Lang\_33

에러 메시지 : NullPointerException

수정 패턴 순서 ; Range checker -> nullpoionter checker

옳은 패치 생성 패턴 : nullpointer checker

우선순위화로 인한 패턴 랭킹 변화 : 2 -> 1

후보 패치 랭킹 변화 : 2 -> 1

Lang\_39

에러 메시지 : NullpointerException

수정 패턴 순서 ; datatype replacer -> nullpointer checker

옳은 패치 생성 패턴 : nullpointer checker

우선순위화로 인한 패턴 랭킹 변화 : 2 -> 1

후보 패치 랭킹 변화 : 7 -> 1

Lang\_47

에러 메시지 : NullpointerException

수정 패턴 순서 ; datatype replacer -> variable replacer -> nullpointer checker

옳은 패치 생성 패턴 : nullpointer checker

우선순위화로 인한 패턴 랭킹 변화 : 3 -> 1

후보 패치 랭킹 변화 : 33 -> 1

Math\_65

에러 메시지 : AssertionFailedError expected 0.004 but was 0.0019

수정 패턴 순서 ; variable replacer -> nullpointer checker -> ICASTdivcastoDouble -> operator mutator

옳은 패치 생성 패턴 : operator mutator

우선순위화로 인한 패턴 랭킹 변화 : 4 -> 2

후보 패치 랭킹 변화 : 60 -> 45

Math\_79

에러 메시지 : NullpointerException

수정 패턴 순서 ; datatype replacer -> nullpointer checker

옳은 패치 생성 패턴 : datatype replacer

우선순위화로 인한 패턴 랭킹 변화 : 1 -> 2

Math\_89

에러 메시지 : ClascastException

수정 패턴 순서 ; method invocation mutator -> variable replacer -> nullpointer checker -> classcast checker

옳은 패치 생성 패턴 : classcast checker

우선순위화로 인한 패턴 랭킹 변화 : 4 -> 1

후보 패치 랭킹 변화 : 14 -> 1

Mockito\_26

에러 메시지 : Assertion FailedErrorexpected <0> but was<0>

수정 패턴 순서 ; variable replacer -> nullpointer checker -> literal expression mutator

옳은 패치 생성 패턴 : literal expression mutator

우선순위화로 인한 패턴 랭킹 변화 : 3 -> 1

후보 패치 랭킹 변화 : 9 -> 1

Mockito\_38

에러 메시지 : NullPointerException

수정 패턴 순서 ; conditional expression mutator -> nullpointer checker

옳은 패치 생성 패턴 : nullpointer checker

우선순위화로 인한 패턴 랭킹 변화 : 2 -> 1

후보 패치 랭킹 변화 : 28 -> 8

**제 4장 실험 평가**

**4-1 실험 환경**

우리의 접근법을 평가하기 위해, 우리는 Defects4J 데이터셋을 평가 벤치마크로 사용하였다. 이 벤치마크는 APR분야에서 자바 프로그램을 대상으로 평가되는 최신 APR 데이터 셋이다. 총 6개의 프로젝트로 구성되어 있으며 395개의 버그가 있다. 제안된 접근법에 대해 FL과정에서 생기는 잡음을 제거하기 위해 우리는 버그가 존재하는 코드의 위치를 직접적으로 제공하였다. 많은 APR 기법에서 APR 기술에 대한 평가에 집중하기 FL기술을 통한 의심스러운 소스코드 리스트를 받기보다 완벽한 버그의 위치를 제공하는 방법을 택한다. 우리는 실험을 두가지 방식으로 진행을 하였다. 첫번째는 기존의 기법인 TBar를 그대로 구현하여 TBar가 옳은 패치를 생성하는 버그에 대상으로 몇 번째 선택한 패턴에서 옳은 패치를 생성하였는지 측정하는 것이다. 버그들마다 다수의 패턴들을 활용하여 후보 패치들을 생성하였으며 첫번째 선택된 패치에서 옳은 패치가 나온 경우는 1로 표기하고 5번째에서 옳은 패치가 생성된 경우에는 5로 표기하였다. 두번째 실험은 우리가 제안한 패턴 우선순위화 작업을 거친 후의 순위를 측정하였다. 각 버그에 대해 생성된 에러메시지를 활용하여 에러메시지에 더 밀접하다고 보여지는 패턴들을 우선적으로 적용하여 첫번째 실험과 마찬가지로 몇 번째 패턴에서 옳은 패치가 생성되는지 측정하였다.

**4-2 실험 결과**

| BugId | Pattern rank | |
| --- | --- | --- |
| without prioritization | with Prioritization |
| Chart\_1 | 1 | 1 |
| Chart\_4 | 2 | 1 |
| Chart\_8 | 1 | 1 |
| Chart\_9 | 2 | 1 |
| Chart\_11 | 1 | 1 |
| Chart\_12 | 5 | 5 |
| Chart\_19 | 1 | 1 |
| Chart\_20 | 1 | 1 |
| Chart\_24 | 2 | 1 |
| Chart\_26 | 2 | 2 |
| Closure\_2 | 2 | 1 |
| Closure\_4 | 1 | 1 |
| Closure\_10 | 1 | 1 |
| Closure\_11 | 1 | 1 |
| Closure\_13 | 3 | 3 |
| Closure\_38 | 1 | 1 |
| Closure\_40 | 3 | 3 |
| Closure\_46 | 1 | 1 |
| Closure\_62 | 1 | 1 |
| Closure\_70 | 4 | 3 |
| Closure\_73 | 1 | 1 |
| Closure\_102 | 4 | 4 |
| Closure\_115 | 5 | 5 |
| Closure\_117 | 4 | 5 |
| Lang\_6 | 2 | 2 |
| Lang\_10 | 4 | 4 |
| Lang\_24 | 1 | 1 |
| Lang\_26 | 3 | 2 |
| Lang\_33 | 2 | 1 |
| Lang\_39 | 2 | 1 |
| Lang\_47 | 3 | 1 |
| Lang\_51 | 3 | 3 |
| Lang\_57 | 2 | 2 |
| Lang\_59 | 1 | 1 |
| Math\_4 | 2 | 1 |
| Math\_5 | 1 | 1 |
| Math\_11 | 2 | 2 |
| Math\_57 | 1 | 1 |
| Math\_58 | 1 | 1 |
| Math\_65 | 4 | 3 |
| Math\_70 | 1 | 1 |
| Math\_75 | 1 | 1 |
| Math\_77 | 1 | 1 |
| Math\_79 | 1 | 2 |
| Math\_82 | 1 | 1 |
| Math\_85 | 1 | 1 |
| Math\_89 | 4 | 1 |
| Mockito\_26 | 3 | 1 |
| Mockito\_29 | 1 | 1 |
| Mockito\_38 | 2 | 1 |
| Time\_7 | 2 | 1 |
| Time\_19 | 1 | 1 |

표 3

표 3은 우선순위화 작업을 하지 않은 TBar에서 몇 번째 패턴에서 첫번째 옳은 패치가 등장하였는지 표기한 것과 우선순위화 작업을 거친 후 TBar에서 몇 번째 패턴에서 첫번째 옳은 패치를 생성하였는지 측정한 값이다. 한 패턴당 한 개에서 수십 여개 이상의 후보 패치들이 생성되었기 때문에 패턴 순위 한 개의 차이도 많은 후보 패치 생성 수 차이를 나타낼 수 있다. 파란색으로 표시한 버그는 우선순위화 작업을 거치면서 더 빠르게 적합한 패턴을 찾은 경우를 나타낸 것이고 빨간색으로 표시한 버그는 우선순위화 작업이 오히려 옳은 패치를 생성해낸 패턴을 찾는 것을 더 늦춘 경우를 나타낸 것이다. TBar가 395의 버그에 대해서 옳은 패치를 생성한 52개의 버그에 대해서 순위를 측정한 것이며 우리가 제안한 우선순위화 과정을 통해 15개의 버그에 대해서 기존의 방식보다 더 빠르게 옳은 패치를 생성한 패턴을 선택하였으며 2개의 버그에 대해서 기존의 방식보다 늦게 옳은 패치를 생성한 수정 패턴을 선택하는 결과를 얻게 되었다.

|  |  |  |
| --- | --- | --- |
| **BugId** | Patch candidate rank | |
| ***Without prioritization*** | ***With prioritization*** |
| Chart\_4 | 70 | 1 |
| Chart\_9 | 11 | 1 |
| Chart\_24 | 7 | 1 |
| Closure\_2 | 37 | 1 |
| Closure\_70 | 71 | 53 |
| Lang\_26 | 45 | 38 |
| Lang\_33 | 2 | 1 |
| Lang\_39 | 7 | 1 |
| Lang\_47 | 33 | 1 |
| Math\_4 | 16 | 13 |
| Math\_65 | 60 | 45 |
| Math\_89 | 14 | 1 |
| Mockito\_26 | 9 | 1 |
| Mockito\_38 | 28 | 8 |
| Time\_7 | 48 | 1 |

표 4

표 4는 우선순위화 과정을 통해 성능향상이 있었던 15개의 버그에 대해서 실제로 후보 패치 수에 있어서 얼마나 빠르게 옳은 패치를 찾을 수 있었는지에 대한 것을 나타낸 것이다. 실제로 적합한 수정 패턴을 1순위 더 먼저 찾은 경우 적게는 1개의 후보 패치에서 많게는 수십개의 불필요한 후보 패치들을 생성하지 않고 옳은 패치를 생성할 수 있었다.

**4-3 연구 문제**

4-3-1 실패 테스트 케이스들은 APR작업의 fix ingredients가 될 수 있나?

우리는 8가지 에러 메시지 타입에 대해 정보를 수집하였다. 각각의 에러들에 대해서 더 적합한 수정 패턴을 매칭 시켜주었고 이에 따라 패치 과정에서 우선순위화 되도록 환경을 구성하였다. 이러한 과정은 실제 개발자들이 수작업으로 버그를 고치는 프로세스와 유사하다. 어떤 종류의 에러인지에 대한 정보는 어떤 수정 행동을 취해야 하는지에 대한 근거가 될 수 있는 것이다.

4-3-2 실패 테스트 케이스 정보는 모든 버그 케이스들에 대해 활용될 수 있는가?

우리가 수집한 에러메시지들은 모든 버그들을 전부 다루지 못했다. 실제 코드들 에서는 많은 경우에 추가 적인 정보 없이 AssertionFailedError 메시지만 얻을 수 있었고 추가적인 정보 없이는 어떠한 버그의 종류인지 알 수 없었다. 이러한 경우에는 버그에 대한 정보가 부족하기 때문에 우선순위화 작업을 할 수 없었다. 또한 첫번째 등장하는 실패테스트 케이스에 대한 에러를 우선적으로 처리하기 때문에 여러 개의 에러메시지를 등장시키는 경우 실제 버그를 야기시키는 치명적인 에러메시지에 대한 우선적인 처리가 되지 않아 수정 패턴 향상을 만들어주지 못하는 경우가 발생되었다.

4-3-3 각 케이스에 대해 효율의 차이가 나는 이유는?

각 버그에서 발생되는 실패 테스트 케이스의 정보를 이용해 우선 작업되어야 할 수정 패턴들을 선택하게 되는데, 해당 에러를 고치기 위한 수정 패턴이 여러 개 존재하는 경우가 있다. 이런 경우 해당되는 수정 패턴들을 모두 탐색해야 하기 때문에 우선순위화 작업을 거친다 하더라도 큰 효율을 못 가질 수 있다. Closure\_70의 경우 해당되는 수정 패턴의 종류가 여러 개 발생하여 패턴 랭킹 상승이 크지 않았다. 또한 각각의 수정 패턴에서 발생되는 패치 수가 임의적으로 결정되기 때문에 일관적이지 못한 성능 향상을 보여준다. 예를 들어 Chart\_4의 경우 한 가지 패턴 랭킹 상승이 발생하였으나 선행되었던 패턴에서 발생되는 패치수가 69가지가 발생하였기 때문에 한 가지 패턴 랭킹 상승으로도 69개의 패치 랭킹 상승 효과를 볼 수 있었다. 반면 Closure\_70의 경우도 역시 1개의 패턴 랭킹 상승이 있었으나 Chart\_4에 비하여 패턴이 발생시켰던 패치 수가 적어 실제 패치 랭킹 변화는 작았다.

**제 5장 결론**

**5-1 결론**

Template-based APR 기술은 버그 프로그램을 고치기 위해 다양한 접근법으로 연구되고 있다. Template-based APR이 가장 효과적인 기법으로 알려져 있지만, 아직까지 전체 단계에 있어서 정보의 활용이 부족하고, 실제 개발자들이 수작업으로 버그를 고치는 것보다 효과가 부족하다. 대부분의 APR 기술들이 버그를 고치기 위해 버그가 왜 발생하였는지에 대해 고려 없이 버그의 위치에 대해서만 초점을 맞추고 있다. 이번 연구에서, 우리는 왜 버그가 발생하였는지에 대한 근거를 찾고 그 근거를 기반으로 어떤 수정 행동을 취해야 하는지 선택함으로써 우리의 접근법을 적용하기 전보다 더 효율적으로 옳은 패치 생성을 해냈다. 미래 APR연구에 있어서, 버그의 위치에 대한 정보뿐만 아니라 해당 버그가 왜 발생되는지에 대한 정보 역시 고려함으로써 옳은 패치 생성 효율을 높여야 할 것이다.

**5-2 미래 연구**

우리는 8개의 에러 메시지 타입에 대하여 우선적으로 적용되어야 할 수정 패턴들에 대해 정의하였다. 하지만 실제 코드에 등장하는 다양한 에러메시지들이 존재하기 때문에 추가적으로 더 많은 에러메시지에 대한 정의가 필요하다. 또한 몇몇의 경우에는 에러메시지 정보만으로 어떤 종류의 버그인지 판단하기 힘든 경우가 있었다. 그렇기 때문에 에러메시지 뿐만 아니라 어떤 버그인지 판단할 수 있는 추가적인 정보가 필요하다. 이번 연구에서는 실패한 테스트 케이스에 대해서만 정보를 취득하여 수정 패턴 우선순위화를 진행하였지만 추후 연구에서는 성공한 테스트 케이스들에 대한 정보를 같이 고려하여 어떤 수정 행동을 취해야 하는지에 대한 더 정확한 근거를 마련할 것이다

**참 고 문 헌**

**[1] T. Britton, L. Jeng, G. Carver, and P. Cheak, ‘‘Reversible debug-ging software - quantify the time and cost saved using reversible debuggers,’’**

**2013.**

**[2] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. IEEE Transactions on Software Engineering 38, 1 (2012), 54–72.**

**[3] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A largescale**

**experiment on the defects4j dataset. Empirical Software Engineering, pages**

**[4] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In Proceedings of the 31st International Conference on**

**Software Engineering. IEEE, 364–374**

**[5] Vidroha Debroy and WEric Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In 2010 Third International Conference on Software Testing, Verification and Validation. IEEE, 65--74.**

**[6] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. Empirical Software Engineering 20, 1 (2015), 176--205.**

**[7] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In ISSTA'2015. ACM, 24—36**

**[8] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pages 831–841. ACM, 2017.**

**[9] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In Proceedings of the 35th International Conference on Software Engineering. IEEE, 802–811.**

**[10] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. FixMiner: Mining relevant fix patterns for automated program repair Empirical Software Engineering volume 25, pages1980–2024(2020)**

**[11] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation. IEEE.**

**[12] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR : Fixing Semantic Bugs with Fix Patterns of Static**

**Analysis Violations. In Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering.IEEE.**

**[13] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. In Proceedings of the International Symposium on Search Based Software Engineering. Springer, 65–86.**

**[14] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In ICSE.**

**[15] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In Proceedings of the 31st AAAI Conference on Artificial Intelligence.**

**AAAI Press, 1345–1351.**

**[16] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In Proceedings**

**of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering. IEEE.**

**[17] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, pages 832–837, 2018.**

**[18] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi**

**Wei and Lin Tan. CoCoNuT: Combining Context-Aware Neural Translation Models using Ensemble for Program Repair. In ISSTA 2020**

**[19] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé.**

**Tbar: Revisiting template-based automated program repair. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, pages 31–42, New York,**

**NY, USA, 2019. ACM. ISBN 978-1-4503-6224-5. doi: 10.1145/3293882.3330577.**

**[20] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In Proceedings of the 2013 International Symposium on Software Testing and Analysis. ACM, 314–324.**

**[21] Rui Abreu, Arjan JC Van Gemund, and Peter Zoeteweij. 2007. On the accuracy of spectrum-based fault localization. In Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION. IEEE, 89–98.**

**[22] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the 23rd ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 437–440.**

**[23] Kui Liu, Koyuncu Anil, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. 2018. LSRepair: Live Search of Fix Ingredients for Automated Program Repair. In Proceedings of the 25th Asia-Pacific Software Engineering Conference. 658–662**

**[24] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR : Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering.IEEE.**

**[25] Kai Pan, Sunghun Kim, and E James Whitehead. 2009. Toward an understanding of bug fix patterns. Empirical Software Engineering 14, 3 (2009), 286–315.**

**[26] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2017. An empirical analysis of the influence of fault space on search based automated program repair. arXiv preprint arXiv:1707.05172 (2017).**

**[27] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair.**

**In Proceedings of the 40th International Conference on Software Engineering. ACM, 789–799.**

**[28] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, Marcelo A. Maia. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. SANER'18 (25th edition of IEEE**

**International Conference on Software Analysis, Evolution and Reengineering)**

# [29] David Landsberg, Hana Chockler, Daniel Kroening, Matt Lewis, Evaluation of Measures for Statistical Fault Localisation and an Optimising Scheme International Conference on Fundamental Approaches to Software Engineering 2015

# [30] Pavneet Singh Kochhar, Xin Xia, David Lo, Shanping Li, Practitioners' expectations on automated fault localization, ISSTA 2016: Proceedings of the 25th International Symposium on Software Testing and Analysis

# [31] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 each. In ICSE’2012. IEEE, 3–13.

# [32] M.A.Mullerburg, ‘‘The role of debugging within software engi- € neering environments,’’ ACM SIGSOFT Softw. Eng. Notes, vol. 8, no. 4, pp. 81–90, 1983

# [33] J. A. Jones and M. J. Harrold, ‘‘Empirical evaluation of the Tarantula automatic fault-localization technique,’’ in Proc. Int. Conf. Automated Softw. Eng., 2005, pp. 273–282,

# [34] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, ‘‘Spectrumbased multiple fault localization,’’ in Proc. Int. Conf. Automated Softw. Eng., 2009, pp. 88–99,

# [35] L. Mariani, F. Pastore, and M. Pezze, ‘‘Dynamic analysis for diagnosing integration faults,’’ IEEE Trans. Softw. Eng., vol. 37, no. 4, pp. 486–508, Jul./Aug. 2011,

# [36] D. Lorenzoli, L. Mariani, and M. Pezze, ‘‘Automatic generation of software behavioral models,’’ in Proc. Int. Conf. Softw. Eng., 2008, pp. 501–510,

# [37] I. Krka, Y. Brun, and N. Medvidovic, ‘‘Automatic mining of specifications from invocation traces and method invariants,’’ in Proc. Int. Symp. Found. Softw. Eng., 2014, pp. 178–189,

# [38] H. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, ‘‘Semfix: Program repair via semantic analysis,’’ in Proc. Int. Conf. Softw. Eng., 2013, pp. 772–781,

# [39] T. Ackling, B. Alexander, and I. Grunert, ‘‘Evolving patches for software repair,’’ in Proc. Annu. Conf. Genetic Evol. Comput., 2011, pp. 1427–1434,

# [40] Y. Tao, J. Kim, S. Kim, and C. Xu, ‘‘Automatically generated patches as debugging aids: a human study,’’ in Proc. Int. Symp. Found. Softw. Eng., 2014, pp. 64–74

# [41] A. Khalilian, A. Baraani-Dastjerdi, and B. Zamani, ‘‘On the evaluation of automatic program repair techniques and tools,’’ in Proc. Iranian Conf. Elect. Eng., 2016, pp. 61–66

# [42] M. Monperrus, ‘‘Automatic software repair: A bibliography,’’ Univ. Lille, Lille, France, Tech. Rep. hal-01206501, 2015.

# [43] D. Kelk, K. Jalbert, and J. S. Bradbury, ‘‘Automatically repairing concurrency bug with ARC,’’ in Proc. Int. Conf. Multicore Softw. Eng., Perform. Tools, 2013, pp. 73–84

# [44] C. Le Goues, S. Forrest, and W. Weimer, ‘‘Current challenges in automatic software repair,’’ Springer Softw. Quality J., vol. 21, no. 3, pp. 421–443, 2013

# [45] Y. Ke, K. Stolee, C. Le Goues, and Y. Brun, ‘‘Repairing programs with semantic code search,’’ in Proc. Int. Conf. Automated Softw. Eng., 2015, pp. 295–306

# [46] M. Stumptner and F. Wotawa, ‘‘A model-based approach to software debugging,’’ in Proc. Int. Workshop Principles Diagnosis, 1996, pp. 233–239

# [47] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, ‘‘Automatic program repair with evolutionary computation,’’ Commun. ACM, vol. 53, no. 5, pp. 109–116, 2010

# [48] E. T. Barr, M. Harman, P. McMinn, and M. Shahbaz, ‘‘The oracle problem in software testing: A survey,’’ IEEE Trans. Softw. Eng., vol. 41, no. 5, pp. 507–525, May 2015

# [49] Y. Zhang and A. Mesbah, ‘‘Assertions are strongly correlated with test suite effectiveness,’’ in Proc. Joint Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., 2015, pp. 214–224

# [50] H. Chang, L. Mariani, and M. Pezze, ‘‘Exception handlers for healing component-based systems,’’ ACM Trans. Softw. Eng. Methodology, vol. 22, no. 4, pp. 30:1–30:40, 2013

# [51] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, ‘‘Automated atomicity-violation fixing,’’ ACM SIGPLAN Notices, vol. 46, no. 6, pp. 389–400, 2011

# [52] P. Agarwal and A. Agrawal, ‘‘Fault-localization techniques for software systems: A literature review,’’ SIGSOFT Softw. Eng. Notes, vol. 39, no. 5, pp. 1–8, 2014

# [53] C. Le Goues, ‘‘Automatic program repair using genetic programming,’’ Ph.D. dissertation, Faculty School Eng. Appl. Sci., Univ. Virginia, Charlottesville, VA, USA, 2013.

# [54] Y. Qi, X. Mao, Y. Lei, and C. Wang, ‘‘Using automated program repair for evaluating the effectiveness of fault localization techniques,’’ in Proc. Int. Symp. Softw. Testing Anal., 2013, pp. 191–201

# [55] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, ‘‘Automated fixing of programs with contracts,’’ IEEE Trans. Softw. Eng., vol. 40, no. 5, pp. 427–449, May 2014

# [56] S. Mechtaev, J. Yi, and A. Roychoudhury, ‘‘Angelix: Scalable multiline program patch synthesis via symbolic analysis,’’ in Proc. Int. Conf. Softw. Eng., 2016, pp. 691–701

# [57] T. Ji, L. Chen, X. Mao, and X. Yi, ‘‘Automated program repair by using similar code containing fix ingredients,’’ in Proc. Annu. Comput. Softw. Appl. Conf., 2016, pp. 197–202

# [58] E. T. Barr, T. Brun, P. Devanbu, M. Harman, and F. Sarro, ‘‘The plastic surgery hypothesis,’’ in Proc. Int. Symp. Foundations Softw. Eng., 2014, pp. 306–317

# [59] Y. Qi, X. Mao, and Y. Lei, ‘‘Efficient automated program repair through fault-recorded testing prioritization,’’ in Proc. Int. Conf. Softw. Maintenance, 2013, pp. 180–189

# [60] Z. Qi, F. Long, S. Achour, and M. Rinard, ‘‘An analysis of patch plausibility and correctness for generate-and-validate patch generation systems,’’ in Proc. Int. Symp. Softw. Testing Anal., 2015, pp. 24–36

# [61] S. Tan and A. Roychoudhury, ‘‘Relifix: Automated repair of software regressions,’’ in Proc. Int. Conf. Softw. Eng., 2015, pp. 471– 482,

# [62] C. Liu, J. Yang, L. Tan, and M. Hafiz, ‘‘R2fix: Automatically generating bug fixes from bug reports,’’ in Proc. Int. Conf. Softw. Testing Verification Validation, 2013

# [63] M. Monperrus, ‘‘A critical review of ‘‘automatic patch generation learned from human-written patches’’: Essay on the problem statement and the evaluation of automatic software repair,’’ in Proc. Int. Conf. Softw. Eng., 2014, pp. 234–242

# [64] J. Xuan, et al., ‘‘Nopol: Automatic repair of conditional statement bugs in Java programs,’’ IEEE Trans. Softw. Eng., vol. 43, no. 1, pp. 34–55, Jan. 2017,

# [65] Y. Lin and S. Kulkarni, ‘‘Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability,’’ in Proc. Int. Symp. Softw. Testing Anal., 2014, pp. 237–247

# [66] Y. Pei, C. F. A, M. Nordio, and B. Meyer, ‘‘Automated program repair in an integrated development environment,’’ in Proc. Int. Conf. Softw. Eng., 2015, pp. 681–684

# [67] B. Elkarablieh, I. Garcia, Y. Suen, and S. Khurshid, ‘‘Assertionbased repair of complex data structures,’’ in Proc. Int. Conf. Autom. Softw. Eng., 2007, pp. 64–73

# [68] V. Dallmeier and T. Zimmermann, ‘‘Extraction of bug localization benchmarks from history,’’ in Proc. Int. Conf. Autom. Softw. Eng., 2007, pp. 433–436

# [69] E. Smith, E. Barr, C. Le Goues, and Y. Brun, ‘‘Is the cure worse than the disease? overfitting in automated program repair,’’ in Proc. Joint Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., 2015, pp. 532–543

# [70] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, ‘‘Does genetic programming work well on automated program repair?’’ in Proc. Int. Conf. Comput. Inform. Sci. 2013, pp. 1875–1878

# [71] H. Yokoyama, Y. Higo, K. Hotta, T. Ohta, K. Okano, and S. Kusumoto, ‘‘Toward improving ability to repair bugs automatically: a patch candidate location mechanism using code similarity,’’ in Proc. Symp. Appl. Comput., 2016, pp. 1364–1370

# [72] Y. Qi, X. Mao, Y. Lei, and C. Wang, ‘‘Using automated program repair for evaluating the effectiveness of fault localization techniques,’’ in Proc. Int. Symp. Softw. Testing Anal., 2013, pp. 191–201

# [73] X. D. Le, T. B. Le, and D. Lo, ‘‘Should fixing these failures be delegated to automated program repair?’’ in Proc. Int. Symp. Softw. Rel. Eng., 2015, pp. 427–437

**ABSTRACT**

**pattern prioritization for automated program repair effectiveness**

영문성명 : Ju Hyoung Kim

Department of Computer Science and Engineering

Sungkyunkwan University

As the scale of software increases, automated program repair (APR) is required as an essential technology to reduce bug-fixing manual effort. One of the most effective APR approach is the Template-based APR. Although this data-driven approach has made many improvements in fixing bugs, there is still a lack of information utilization at each stage of the program repair.

In this paper, we propose a fix pattern prioritization APR technique. We do not just use the test cases to determine whether the source file is fault or not, but as a material for why the bug occurred. With this information, we select a more appropriate fix action and apply it first to increase the efficiency of the automated program repair.

**Keywords : Software Engineering, Automated Program Repair, Fix pattern prioritization, Template-based APR**