■ 2013년도 학생논문 경진대회 수상작

FEAST: 테스트 케이스의 결함 실행확률을 이용한 향상된 결함 위치추정 기법

(FEAST: An Enhanced Fault Localization Technique using Probability of Test Cases Executing Faults)

문 석 현 † 김 윤 호 † 김 문 주 ††
(Seokhyeon Moon) (Yunho Kim) (Moonzoo Kim)

요 약 프로그램 오류의 원인을 찾는 과정인 결함 위치추정(fault localization)은 프로그래머가 직접 실패한 실행을 추적하면서 결함으로 의심되는 코드를 찾아야 하기 때문에, 프로그램 디버깅 과정 중 가장 많은 노력과 시간을 요구한다. 따라서, 결함 위치추정에 사용되는 비용을 줄이기 위해서 많은 기법들이 제안되었고, 그 중 커버리지 기반 결함 위치추정 기법(CBFL: Coverage Based Fault Localization)은 프로그램 커버리지를 이용하여 결함으로 의심되는 코드들에 우선순위를 부여함으로써 결함의 위치를 추정한다. 하지만 CBFL기법에서 사용되는 테스트 케이스 중 결함을 실행했음에도 오류를 발생시키지 않는 테스트 케이스인 Coincidentally Correct Test case(CCT)의 수가 많을 경우, CBFL 기법의 정확성이 크게 떨어지는 문제점이 있다. 본 논문에서는 해당 문제의 해결을 위해서, 결함 가중치가 부여된 테스트 케이스에 기반한 결함 위치추정 기법 FEAST(Fault-wEight bASed localization Technique)을 제안한다. FEAST는 대상 프로그램을 실행한 각 테스트 케이스가 결함을 실행했을 확률에 따라 각 테스트 케이스에 가중치를 부여함으로써 CBFL기법의 정확성을 향상 시킨다. 본 논문에서 제안한 FEAST기법의 정확성을 측정하기 위해 SIR benchmark내의 10개 프로그램에 대해 FEAST와 대표적 CBFL기법인 Tarantula를 적용하여 실험을 수행하였다. 실험으로부터 FEAST는 평균적으로 약 14.62%의 실행된 구문을 검사함으로써 결함을 찾을 수 있다는 것을 확인하였는데, 이는 FEAST가 Tarantula보다 평균적으로 26.55% 더 정확한 결과이다.

키워드: 커버리지 기반 결함 위치추정 기법, Coincidentally correct test case, 결함 가중치, 디버깅

Abstract Fault-localization is a very expensive step of the whole debugging process, because it usually requires human developers to reason about the differences between passed runs and failed runs step-by-step. Thus, there have been active researches regarding automated fault localization based on test coverage. One main difficulty for precise fault localization is due to the coincidentally correct test cases (CCTs), which are passed test cases that execute a faulty statement. In this paper, to overcome this limitation, we propose Fault-wEight bASed localization Technique (FEAST), which can reduce the negative effects of CCTs by considering fault weights on test cases, which indicate "likelihood"

· 본 연구는 미래부가 지원한 2013년 정보통신·방송(ICT) 연구개발사업, 지식경제부/한국산업기술평가관리원 IT R&D 프로그램[10041752, 초소형·고신뢰 OS와 고성능 멀티코어 OS를 동시 실행하는 듀얼 운영체제 원천 기술 개발], 그리고 미래부/한국연구재단의 중견연구자지원사업-핵심연구(2012046172)의 연구비 지원으로 수행되었음

† 학생회원 : KAIST 전산학과
seokhyeon.mun@gmail.com
(Corresponding author임)
vita500@gmail.com
†† 종신회원 : KAIST 전산학과 부교수
moonzoo@cs.kaist.ac.kr

논문접수 : 2013년 5월 21일
심사완료 : 2013년 7월 22일

정보과학회논문지: 소프트웨어 및 응용 제40권 제10호(2013.10)

of the statements executed by the test cases to be faulty statements. To evaluate the accuracy of the fault weight metric and the precision and stability of the suspiciousness metric of FEAST, we have performed a series of experiments by applying both FEAST and a representative fault localization technique called Tarantula on the 10 SIR benchmark programs. From the experiments, we confirm that the fault weight metric can recognize CCT accurately and the precision and stability of FEAST are higher than Tarantula. For example, FEAST identifies a fault after examining 14.62% of the target program code on average, which is 26.55% more precise result compared to Tarantula.

Keywords: coverage-based fault localization, coincidentally correct test case, fault weight, debugging

1. 서 론

다양한 종류의 프로그램 테스팅 기법이 연구되어 왔지만, 프로그래머들은 여전히 프로그램 오류의 원인인 결함(fault)을 찾는데 매우 많은 시간을 사용한다. 결함 위치추정(fault localization)은 디버깅 과정 중 가장 많은 시간과 노력을 요구하는 것으로 알려져 있는데[1], 이는 프로그래머가 직접 실패한 실행에서 실행된 코드를 모두 추적하면서 결함으로 의심되는 코드들을 찾아야 하기 때문이다. 따라서, 이러한 노력을 줄이기 위해, 프로그래머가 결함을 찾는 과정을 지원할 수 있는 자동화된 결함 위치추정 기법이 연구되어 왔다.

많은 종류의 결함 위치추정 기법 연구 중 각광받는 연구 방향은 커버리지 기반 결함 위치추정(CBFL: Coverage Based Fault Localization)기법이다. CBFL 기법은 프로그램을 실행한 후 얻은 커버리지(e.g., 구문 커버리지, 분기 커버리지)와 테스트 케이스의 실행 결과(성공, 실패)를 이용하여, 각 코드(e.g., 구문, 분기 등)가 결함일 가능성을 계산하여 각 코드에 부여함으로써, 프로그래머가 결함을 찾는 일을 도와준다(3.1절 참조). CBFL 기법의 기본 아이디어는 각 프로그램 코드와 실패한 테스트 케이스들간의 상관관계를 이용하는 것으로, 높은 상관관계를 가지는 코드일수록 결함일 가능성이 높은 것으로 간주된다. CBFL 기법은 기존에 제시된 다른 종류의 결함 위치추정 기법에 비해서 일반적으로 더 효과적인 것으로 알려져 있다[2].

하지만, CBFL 기법을 실제 프로그램에 적용했을 때 결함과 실패한 테스트 사이의 상관관계가 결함을 정확히 찾을 수 있을 만큼 높지 않은 문제가 발생한다. 그 상관관계를 낮추는 가장 큰 요소로 알려진 것은 결함 코드를 실행했지만 오류를 발생시키지 않는 테스트 케이스인 Coincidentally Correct Test case (CCT)[3]이다. 결함 코드가 CCT에 의해서 많이 실행될수록 해당 코드는 결함 코드가 아닌 것으로 간주될 가능성이 점점 더 높아지는데, 이는 프로그램 실패를 발생시키지 않는 테스트 케이스, 즉 성공 테스트 케이스가 결함 코드를 많이 실행할수록 결함 코드는 정상인 코드로 간주되기 때문이다. CCT에 의해서 CBFL기법의 정확성이 떨어지는 문제는 다른 연구들에서 이미 지적되었으며[4-6], 대상 프로그램에 따라 많은 수의 CCT가 발생 가능한 것으로 알려져 있다[6-8].

본 연구에서는 CCT로 인해서 발생하는 CBFL 기법의 정확성 저하를 극복하기 위해 결함 가중치에 기반한 결함 위치추정 기법(Fault-wEight bASed localization Technique: FEAST)을 제안한다. FEAST는 CBFL기법에서 사용되는 각 테스트 케이스에 결함 가중치(fault weight)를 부여함으로써 CBFL 기법의 정확성을 향상시킨다. FEAST기법에서 각 테스트 케이스의 결함 가중치는 각 테스트 케이스가 결함 구문을 실행했을 확률로 정의된다(3.2 절 참조). 이를 기반으로, FEAST는 구문 s의 의심도를 전체 실패 테스트 케이스 수 중 구문 s를 실행한 실패 테스트 케이스 수의 비율과, 구문 s를 실행한 테스트 케이스들의 평균 결함 가중치를 동시에 고려하여 계산한다. 즉, FEAST 기법은 구문 s를 실행한 테스트 케이스들의 결함 가중치를 이용함으로써 CBFL 기법의 정확성을 향상시킨다.

본 논문에서 우리들이 제안한 결함 가중치(fault weight)와 FEAST기법의 효과성을 측정하기 위해서 다음과 같은 연구 질문을 지정하였다.

RQ1: 각 테스트 케이스들에 부여된 FEAST기법의 결함 가중치가 얼마나 정확한가? 즉, CCT와 non-CCT의 결함 가중치 값이 얼마나 다른가?

RQ2: Tarantula 기법과 비교했을 때 FEAST기법은 얼마나 더 효과적인가? 즉, 의심도 순으로 정렬된 실행된 구문들을 순서대로 검사했을 때, 한 개의 결함 구문을 찾기 위해서 몇 %의 실행된 구문들을 검사해야 하는가?

RQ3: FEAST 기법은 얼마나 안정적인가? 즉, Tarantula 기법과 비교했을 때, 결함을 찾기 위해 검사해야 하는 코드의 비율이 가장 적은 결함 버전과 가장 많은 결함 버전 사이의 차이는 얼마인가?

우리는 위의 연구 질문들에 대한 답을 위해 FEAST와 Tarantula 기법을 SIR benchmark[9]의 10개 프로그램에 대해 적용하고 해당 기법의 정확성을 측정하였다. 10개 프로그램들 중 7개는 SIEMENS benchmark의 프로그램이며, 3개는 실제 사용자들에 의해 사용되는 큰 크기(9000 줄 이상)의 프로그램들이다(flex 2.4.7,

grep 2.2, space). 본 논문의 실험결과는 결함 가중치를 이용하는 FEAST가 Tarantula기법에 비해 더 정확하게 결함 구문의 위치를 추정한다는 것을 보여준다(5.2절 참조). Tarantula 기법과 비교했을 때 평균적으로 26.55% 더 정확하게 결함 구문의 위치를 추정하며, 특히 세 개의 큰 프로그램(i.e., flex, grep, space)에서는 Tarantula 보다 57.72% 더 정확한 결과를 보여 준다.

본 논문의 연구 성과는 다음과 같다.

- CCT에 의한 CBFL 기법의 정확성 저하를 줄일 수 있는 각 테스트 케이스에 대한 결함 가중치를 정의(3.2절 참조).
- Tarantula 기법보다 더 정확하고 더 안정적인 새로운 의심도 공식을 제안(3.3장 참조).
- 우리가 정의한 결함 가중치의 정확성과, 새로운 CBFL 기법인 FEAST의 정확성을 실험을 통해 증명(5장 참조).

해당 논문의 나머지 구성은 다음과 같다. 2장에서는 결함 위치추정 기법과 관련된 연구들에 대해서 설명하고 FEAST와 다른 위치추정 기법들을 간단히 비교한다. 3장에서는 본 논문에서 제안한, 결함 가중치를 이용한 새로운 CBFL 기법인 FEAST에 대해서 설명한다. 4장에서는 SIR benchmark의 10개 프로그램을 대상으로 수행한 실험의 환경 및 실험 설계에 대해서 설명한다. 5장에서는 실험 결과를 제시하고 분석하며, 6장에서는 본 논문의 결론과 향후 연구 방향에 대해 논의한다.

2. 관련 연구

Jones와 그의 동료들이 Tarantula[10]라고 불리는 효과적인 CBFL 기법을 제안한 이후, CBFL 기법의 정확성을 향상시키기 위한 많은 연구들이 진행되었다. Tarantula 기법은 구문 커버리지와 테스트 케이스의 실행 결과를 이용하여 각 구문의 의심도를 계산하는데(3.1절 참조), 의심도를 계산하는 의심도 공식에 따라 CBFL 기법의 정확성에 차이가 있을 수 있다. 이러한 맥락에서, Abreu와 그의 동료들은[11] CBFL 기법의 정확성 향상을 위해 생물학 분야에서 사용되는 Ochiai coefficient[12]를 이용하여 각 프로그램 구문의 의심도를 계산한다. 구문 커버리지를 이용하지 않는 방법으로, Liblit과 그의 동료들은[13,14] 프로그램의 구문 대신에 프로그램의 각 술어(predicate, e.g., branch)가 실패한 실행과 성공한 실행에서 실행되는 패턴의 차이 정도에 따라 각 술어에 의심도를 부여하고, 높은 의심도의 술어를 우선순위화함으로써 결함을 찾고자 한다. Liu와 그의 동료들은[15] 통계적 모델을 도입하여 Liblit[13,14]의 연구를 발전시킨다. Dallmeier와 그의 동료들은[16] 자바 프로그램 실행시의 함수 호출 순서를 이용하여 결함을 찾고자 한다. Santelices와 그의 동료들은[17] 기존 Tarantula 기법을 프로그램 구문 커버리지 뿐만 아니라 분기 커버리지와 정의-사용 쌍 커버리지를 동시에 사용함으로써 CBFL기법의 정확성을 향상시킨다. 해당 기법은 분기문들과 정의-사용 쌍들을 프로그램 구문들에 맵핑시키고, 맵핑된 결과를 이용하여 각 구문에 대한 평균 의심도를 계산한다.

비록 많은 수의 CBFL 기법들이 제안되었지만, 제안된 대부분의 기법들은 Coincidentally Correct Test case (CCT) 때문에 그 정확성이 떨어지는 문제점을 가지고 있다. 이는 CCT가 많을수록 CBFL 기법에서 계산되는 결함 코드의 의심도가 떨어지기 때문인데(3.1절 참조), 해당 문제를 극복하기 위해서 다양한 방법들이 제안되었다.

Marsi와 그의 동료들은[18] 테스트 케이스 중 CCT인 테스트 케이스를 예측하고 CCT를 제거한 후 CBFL기법을 적용함으로써, Tarantula기법의 정확성을 향상시키고자 한다. 그들은 성공한 테스트 케이스들이 실행한 구문들에 기반하여 테스트 케이스들을 분류한다. 만약 특정 그룹으로 분류된 테스트 케이스들(실행한 구문들이 비슷한 그룹)이 실행한 구문들의 평균 의심도가 다른 그룹보다 높을 경우, 해당 그룹은 CCT로 분류되어 제거된다. 이 기법은 CCT를 잘못 예측했을 때 결함 구문이 아닌 다른 구문의 의심도가 높아질 수 있어서 CBFL 기법의 정확성이 크게 떨어질 수 있다. Bandyopadhyay와 그의 동료들은[19] CBFL기법을 사용하는 사용자로부터의 피드백을 이용하여 CCT를 제거하고자 한다. 사용자에게 가장 높은 의심도를 가지는 구문을 제시하고, 제시된 구문이 결함 구문이 아니라고 사용자가 보고하면, 보고된 정보를 이용하여 CCT라고 의심되는 테스트 케이스를 제거한다. 하지만 이 기법 역시 CCT를 잘못 예측하고 제거할 경우 CBFL기법의 정확성이 오히려 더 떨어질 수 있다. Wang과 그의 동료들은[8] CCT에 의해서 얻어진 커버리지를 제거함으로써 CCT에 의한 정확성 저하를 줄이고자 한다. 하지만 프로그램 실패가 발생했을 때, 해당 실패의 원인이 어떤 종류인지 (e.g., missing function call, missing assignments, etc.) 알아야 할 뿐만 아니라 발생한 실패가 생성되는 프로그램의 실행 패턴(e.g., data flow, dynamic control flow)을 알고 있어야 CCT에 의해 얻어진 커버리지를 제거할 수 있다는 단점이 있다. 우리가 제안하는 FEAST 기법은 실패가 발생하는 프로그램의 실행 패턴을 알 필요 없이, 각 테스트 케이스에 자동으로 부여되는 결함 가중치를 이용함으로써 CCT로 인해 발생하는 CBFL 기법의 정확성 저하를 감소시킬 수 있다.

결함 위치추정 기법의 성능을 향상시키기 위한 다른 방법으로, Artzi와 그의 동료들은[20] 커버리지 기반 결함 위치추정 기법을 위한 테스트 케이스를 CONCOLIC testing[21,22]을 이용하여 자동으로 생성한다. 기본 아

이디어는 실패한 테스트 케이스와 최대한 비슷한 테스트 케이스를 생성하여, 생성된 테스트 케이스 중 성공 테스트 케이스가 실행하지 않은 구문들의 의심도를 높이는 것이다. 이를 위해서 실패 테스트 케이스의 기호 경로 제약(symbolic path constraint)을 조작하여 새로운 테스트 케이스를 생성해 낸다. 해당기법은 생성해낸 테스트 케이스중 CCT의 수가 많을 경우 결함 구문의 의심도를 높이지 못할 수 있기 때문에, 효과적이지 못할 수 있다. Bandyopadhyay와 그의 동료들은[23] CC-proximity[24]에 기반한 테스트 케이스의 결함 가중치를 제안한다. 해당 테스트 케이스 결함 가중치 메트릭은 Ochiai의심도 공식[11]의 성공 테스트 케이스 항을 대체하는데 사용되기 때문에, 사용자에 의해 주어지는 임계값(threshold)을 필요로 한다. 반면 본 논문에서는 사용자에 의해 주어지는 임계값을 요구하지 않는 테스트 케이스에 대한 가중치를 제안하며, 이를 이용한 새로운 의심도 공식을 제안한다. 또한, FEAST 기법의 정확성 측정을 위한 실험들을 수행한다.

3. 결함 가중치에 기반한 결함 위치추정 기법 (FEAST: Fault wEight bASed localization Technique)

3.1 배경

커버리지 기반 결함 위치추정 기법(Coverage Based Fault Localization: CBFL)은 프로그램을 실행하고 얻은 커버리지와 테스트 결과(성공 또는 실패)를 이용하여 프로그램 각 코드(e.g., 구문, 분기 등)가 결함 코드일 가능성을 나타내는 의심도를 계산한다. 각 코드의 의심도는 해당 코드를 실행한 성공 테스트 케이스와 실패 테스트 케이스의 수에 기반하여 계산되는데, 일반적으로 한 개의 코드가 많은 수의 실패 테스트 케이스에 의해 실행되는 동시에 적은 수의 성공 테스트 케이스에 의해 실행되면 높은 의심도를 가지게 된다. Tarantula[10] 기법은 구문 커버리지를 기반으로 다음의 의심도 공식을 이용하여 구문 s의 의심도를 계산한다.

$$\mathrm{Susp(s)_{Tarantula}} = \frac{\frac{|\mathrm{failed(s)}|}{|\mathrm{T_f}|}}{\frac{|\mathrm{failed(s)}|}{|\mathrm{T_f}|} + \frac{|\mathrm{passed(s)}|}{|\mathrm{T_p}|}}$$

공식에서 T_f와 T_p는 각각 실패와 성공 테스트 케이스의 전체 집합, failed(s)와 passed(s)는 각각 구문 s를 실행한 실패와 성공 테스트 케이스의 집합을 나타낸다.

Tarantula기법은 위의 공식을 이용하여 각 구문의 의심도를 계산한 후, 계산된 의심도를 이용하여 프로그램의 구문들을 의심도가 큰 것부터 작은 것 순으로 정렬한다. 정렬된 구문들은 사용자에 의해 의심도 순으로 검사됨으로써, 사용자가 결함 구문을 찾기 위해 검사해야 하는 구문의 개수를 줄여 준다.

그림 1은 위의 의심도 공식이 어떻게 결함 구문을 찾는지 보여 준다. 예제의 첫 번째 행에는 tc1부터 tc6까지 6개의 테스트 케이스가 주어져 있고, 소스 코드 3번 줄 구문의 올바른 구문은 'x=x+2' 이기 때문에, 현재 3번 줄의 구문(i.e., 'x=x-2')은 결함 구문이다. 각 테스트 케이스에 의해서 실행된 구문들은 그림 1에서 검정색 동그란 점으로 표시되어 있다. tc1은 3을 반환(i.e., 8번 줄)하는데 이는 3번 줄에 결함이 없을 때(i.e., 'x=x+2')의 반환값인 1과 다르므로, 실패 테스트 케이스이다. tc2, tc3, tc4, tc5, tc6은 옳은 값을 반환 하므로 성공 테스트 케이스이다(3번 줄이 'x=x+2'일 때의 반환값과 같은 값을 반환한다). '성공'과 '실패'의 결과는 그림 1의 가장 마지막 행에 각각 'Pass'와 'Fail'로 표시되어 있다. 그림 1에서 tc2, tc3, tc4는 결함구문(i.e., 3번 줄)을 실행했음에도 실패하지 않으므로(즉, 결함을 실행하는 성공 테스

int example (int x, int y){	tc1 (5,1)	tc2 (9,1)	tc3 (9,0)	tc4 (10,1)	tc5 (1,2)	tc6 (4,7)	Tarantula susp.	Tarantula rank	FEAST susp.	FEAST rank
1 int ret = 0;	●	●	●	●	●	●	0.50	7	0.92	6
2: if (x>y){	●	●	●	●	●	●	0.50	7	0.92	6
3: x = x-2;//should be x = x+2;	●	●	●	●			0.63	3	**0.95**	**2**
4: ret = ret+1;	●	●	●	●			0.63	3	0.95	2
5: } else{ y = y+2; }					●	●	0.00	8	0.36	8
6: if (x<y+6){	●	●	●	●	●	●	0.50	7	0.92	6
7: ret = ret+2; }	●				●	●	**0.71**	**1**	0.90	7
8: return ret; }	●	●	●	●	●	●	0.50	7	0.92	6
Pass/Fail status	Fail	Pass	Pass	Pass	Pass	Pass				

그림 1 커버리지 기반 결함 위치추정 기법 예제

Fig. 1 Example of coverage-based fault localization

트 케이스) Coincidentally Correct Test case(CCT)이다.

Tarantula 의심도 공식에서 7번 줄의 구문은 가장 높은 의심도를 가지므로($\frac{\frac{1}{1}}{\frac{1}{1}+\frac{2}{5}}$=0.71), 1위로 순위가 메겨진다. 반면에 결함 구문(3번 줄)은 0.63($\frac{\frac{1}{1}}{\frac{1}{1}+\frac{3}{5}}$=0.63)의 의심도를 가지기 때문에 3위로 순위가 메겨진다. 이는 7번 줄을 실행하는 성공 테스트 케이스(tc5, tc6)의 수보다 결함 구문(i.e., 3번 줄)을 실행하는 성공 테스트 케이스(tc2, tc3, tc4)의 수가 더 많기 때문이다. Tarantula의심도 공식에서는 tc2, tc3, tc4와 같은 CCT들로 인해 결함 구문의 의심도가 낮아지는데 이는 Tarantula 공식에서 분모에 있는 |passed(s)|항의 값이 CCT의 수가 많을수록 증가하기 때문이다. 따라서 CCT의 숫자가 많으면 많을수록 결함 구문의 의심도는 감소하고, 이로인해 CBFL 기법의 정확성은 점점 더 떨어질 수 있다. 본 논문에서 제안하는 새로운 CBFL 기법인 FEAST는 각 테스트 케이스에 결함 가중치를 부여함으로써 CCT에 의한 CBFL 기법의 정확성 저하를 감소시킨다(3.3절 참조).

3.2 테스트 케이스에 대한 결함 가중치

FEAST 기법에서 테스트 케이스 t가 결함 구문을 실행했을 가능성을 나타내는 결함 가중치는 다음과 같이 정의 된다.

$$w_{fault}(t) = \frac{\sum_{t_f \in T_f} \frac{|stmt(t) \cap stmt(t_f)|}{|stmt(t_f)|}}{|T_f|}$$

stmt(t)는 테스트 케이스 t에 의해서 실행된 구문의 집합, T_f는 실패 테스트 케이스의 집합을 나타낸다. $w_{fault}(t)$는 테스트 케이스 t에 의해 실행된 구문들의 집합(i.e., stmt(t))과 각 실패 테스트 케이스 $t_f \in T_f$가 공통으로 실행한 구문들의 비율(i.e., $\frac{|stmt(t) \cap stmt(t_f)|}{|stmt(t_f)|}$의 합을 전체 실패 테스트 케이스의 수(i.e., $|T_f|$)로 나눈 값이다.[1] 따라서, 테스트 케이스 t가 실패 테스트 케이스가 실행한 구문들을 많이 실행하면 할수록 $w_{fault}(t)$의 값은 증가하고, 이는 테스트 케이스 t가 결함 구문을 실행했을 가능성이 높음을 의미한다. 그림 1에서 제시된 6개의 테스트 케이스에 대해서 다음과 같이 결함 가중치를 계산할 수 있다.

- $w_{fault}(tc1) = \frac{\frac{|stmt(tc1) \cap stmt(tc1)|}{|stmt(tc1)|}}{1} = \frac{\frac{7}{7}}{1} = 1.$
- $w_{fault}(tc2) = \frac{\frac{|stmt(tc2) \cap stmt(tc1)|}{|stmt(tc1)|}}{1} = \frac{\frac{6}{7}}{1} = 0.86.$
- $w_{fault}(tc3) = w_{fault}(tc4) = w_{fault}(tc2) = 0.86,$ ($\because$ stmt(tc2) = stmt(tc3) = stmt(tc4)).
- $w_{fault}(tc5) = \frac{\frac{|stmt(tc5) \cap stmt(tc1)|}{|stmt(tc1)|}}{1} = \frac{\frac{5}{7}}{1} = 0.71.$
- $w_{fault}(tc6) = w_{fault}(tc5) = 0.71$, ($\because$ stmt(tc5) = stmt(tc6).

각 테스트 케이스의 결함 가중치 값에서 확인할 수 있듯이 결함 구문을 실행한 테스트 케이스들(tc2,tc3,tc4)의 결함 가중치(0.86)는 그렇지 않은 테스트 케이스들(tc5,tc6)의 결함 가중치(0.71)보다 더 높은 값을 가진다.

3.3 FEAST기법의 의심도 공식

FEAST기법은 구문 s의 의심도를 각 테스트 케이스에 부여된 결함 가중치를 이용하여 다음과 같이 계산한다.

$$Susp_{FEAST}(s) = \left(\frac{|failed(s)|}{|T_f|} + \frac{\sum_{t \in test(s)} w_{fault}(t)}{|test(s)|}\right)/2$$

test(s)는 구문 s를 실행한 테스트 케이스들의 집합이다(i.e., test(s) = failed(s) ∪ passed(s)).

FEAST의심도 공식의 $\frac{|failed(s)|}{|T_f|}$항은 구문 s가 많은 수의 실패 테스트 케이스에 의해서 실행될수록 구문 s의 의심도를 증가시킨다. 오른쪽항인 $\frac{\sum_{t \in test(s)} w_{fault}(t)}{|test(s)|}$는 구문 s가 높은 결함 가중치를 가진 테스트 케이스들에 의해 실행될수록 구문 s의 의심도를 증가시킨다. 구문 s의 의심도를 [0, 1]로 한정하기 위해서 위 두 개 항을 더한 값을 2로 나눠준다.

예를 들어, 그림 1에서 결함 구문인 3번 줄의 의심도는 다음과 같이 계산된다.

$$\left(\frac{1}{1} + \frac{w_f(tc1) + w_f(tc2) + w_f(tc3) + w_f(tc4)}{4}\right)/2$$
$$= \left(1 + \frac{\frac{7}{7}+\frac{6}{7}+\frac{6}{7}+\frac{6}{7}}{4}\right)/2 = 0.95$$

그림 1의 예제에서 FEAST는 Tarantula기법보다 더 정확한 것을 확인할 수 있다. 예제에서 FEAST 기법을 적용했을 때는 결함 구문을 찾기 위해서 최대 2개(4, 3번 줄)의 구문을 검사해야 하는 반면, Tarantula기법에서는 결함 구문을 찾기 위해 최대 3개(7, 4, 3 번 줄)의 구문을 검사해야 한다(의심도가 같은 구문은 서로에 대해 우선순위가 없기 때문에, 결함을 찾기 위해 최대로 검사해야 하는 구문의 수는 우선순위가 같은 구문의 개수를 모두 포함해야 한다). FEAST기법은 각 테스트 케이스에 부여된 결함 가중치를 이용하여 각 구문들의 의심도를 계산하기 때문에 Tarantula 기법보다 좀 더 효과적으로 결함의 위치를 추정할 수 있다.

본 논문에서 제안하는 FEAST기법은 기존에 제안되었던 방법들(e.g., [8,18])과는 다르게, 어떤 성공 테스트

1) $|T_f|$ 의 값은 항상 0 이상인데, 이는 주어진 테스트 케이스 중 실패 테스트 케이스가 없을 경우 결함의 위치를 추정할 수 없기 때문이다.

케이스가 CCT인지 아닌지를 정확하게 구분하지 않고, "확률적"으로 예측한다. 이를 통해 CCT가 아닌 테스트 케이스를 CCT로 잘못 예측했을 때 생길 수 있는 심각한 성능저하를 줄일 수 있다. 예를 들어, 어떤 특정 non-CCT(결함 구문을 실행하지 않은 성공 테스트 케이스)의 결함 가중치가 CCT의 결함 가중치보다 높다 하더라도, 전체 CCT의 결함 가중치 평균은 전체 non-CCT의 결함 가중치 평균보다 더 높은 값을 가지므로(5.1장 참조), 잘못 예측된 테스트 케이스로 인한 성능 저하를 줄일 수 있다.

4. 실험 환경

본 논문에서는 FEAST 기법과 제시된 결함 가중치 공식의 정확성을 확인하기 위해 아래와 같은 연구 질문들을 지정하였다.

- **RQ1:** 각 테스트 케이스들에 부여된 FEAST 기법의 결함 가중치가 얼마나 정확한가? 즉, CCT와 non-CCT의 결함 가중치 값이 얼마나 다른가?

테스트 케이스에 부여된 결함 가중치는 FEAST의 의심도 공식에서 사용되므로, 결함 가중치가 충분히 정확하게 결함 구문을 실행한 성공 테스트 케이스(CCT)와 결함 구문을 실행하지 않은 성공 테스트 케이스(non-CCT)를 구분할 수 있어야 한다. 따라서 실험에서 사용되는 모든 테스트 케이스에 대해서 결함 구문을 실행한 테스트 케이스와 그렇지 않은 테스트 케이스의 결함 가중치를 계산하고 그 차이를 비교한다.

- **RQ2:** Tarantula기법[10]과 비교했을 때 FEAST 기법은 얼마나 더 정확한가? 즉, 의심도 순으로 정렬된 실행된 구문들을 순서대로 검사했을 때, 한 개의 결함 구문을 찾기 위해서 몇 %의 실행된 구문들을 검사해야 하는가?

결함 위치추정 기법의 효과를 나타내는 중요한 척도 중 하나는 정확도이다. 정확도는 결함 위치추정 기법의 대상이 되는 프로그램의 내부 실행 구조, 사용되는 테스트 케이스, 그리고 대상 프로그램의 결함의 종류에 라 그 값이 다르게 측정될 수 있다. 따라서, 실험을 수행할 때 결함 위치추정 기법의 정확성에 영향을 미칠 수 있는 요인들을 다양하게 하면서, 정확도를 측정해야 한다.

- **RQ3:** FEAST 기법은 얼마나 안정적인가? 즉, Tarantula 기법과 비교했을 때, 결함을 찾기 위해 검사해야하는 코드의 비율이 가장 적은 결함 버전과 가장 많은 결함 버전 사이의 차이는 얼마인가?

안정성은 결함 위치 추정기법의 효과성을 측정할 수 있는 또 다른 척도이다. 결함 위치 추정 기법 flt1이 또 다른 기법 flt2보다 평균적으로 더 정확하게 결함의 위치를 추정한다고 가정해보자. 하지만 만약 flt1의 결함 위치 추정 정확성(결함 구문을 찾기 위해 검사해야하는 실행된 구문의 비율)이 대상 프로그램의 종류에 따라 flt2에 비해 크게 달라진다면, 실제 결함 위치 추정 기법 사용시에는 flt2가 flt1보다 더 유용할 수 있다. 왜냐하면 flt2를 사용했을 때 결함을 찾기 위해 소모되는 비용을 flt1에 비해 좀 더 정확하게 추정할 수 있기 때문이다. 특히, 높은 안정성을 가지는 결함 위치추정 기법은 프로젝트의 시간과 자원이 제한된 산업 프로젝트에서 좀 더 유용할 수 있다.

FEAST 기법의 유용성을 측정하기 위해 설정한 위의 연구 질문들에 대한 답을 위해 FEAST와 Tarantula 기법을 SIR benchmark[9]의 10개 프로그램에 대해 적용하여 실험을 수행하였다. 다음 장에서는 실험의 구체적 수행 환경에 대해 설명한다.

4.1 실험 대상

실험을 위해 사용된 프로그램은 SIR benchmark의 10개의 프로그램(표 1)이며, 그 중 7개 프로그램은 SIEMENS benchmark의 프로그램들이며, 3개의 프로그램은 실제 사용자에 의해 사용되는 큰 크기(9000 줄 이상)의 프로그램이다(flex 2.4.7, grep 2.2, space). SIR

표 1 실험 대상 프로그램
Table 1 Experiment objects

Target program	LOC	# of faulty ver. used	# of test cases	Description
print_tokens	472	7	4130	lexical analyzer
print_tokens2	399	9	4115	lexical analyzer
replace	512	30	5542	pattern replacement
schedule1	292	5	2650	priority scheduler
schedule2	301	9	2710	priority scheduler
tcas	141	41	1608	altitude separation
tot_info	440	23	1052	information measure
flex	13273	16	567	lexical analyzer generator
grep	14659	3	805	pattern matcher
space	9564	28	13585	ADL interpreter
Average	4005.3	17.1	3676.4	

표 2 실험 대상 프로그램에서 사용된 테스트 케이스의 결함 가중치
Table 2 Fault weights of test cases for the target programs

Target program	Failed test cases		Passed test cases				Total test cases used	
			CCT		NonCCT			
	Avg. # of test cases	Avg. fault weight	Avg. # of test cases	Avg. fault weight	Avg. # of test cases	Avg. fault weight	Avg. # of test cases	Avg. fault weight
print_tokens	69.14	0.88	1960.14	0.66	2100.71	0.50	4130.00	0.58
print_tokens2	229.33	0.90	1187.89	0.64	2697.78	0.74	4115.00	0.72
replace	100.63	0.90	2218.83	0.81	3222.53	0.57	5542.00	0.67
schedule1	142.80	0.98	1431.20	0.95	1069.00	0.63	2643.00	0.82
schedule2	29.00	0.96	2518.89	0.90	153.56	0.21	2701.44	0.86
tcas	39.66	0.98	845.56	0.90	722.78	0.49	1608.00	0.71
tot_info	83.74	0.89	680.13	0.81	288.13	0.34	1052.00	0.69
flex	207.44	0.90	108.25	0.64	229.69	0.42	545.38	0.65
grep	318.00	0.88	46.33	0.28	386.67	0.46	751.00	0.63
space	1954.21	0.91	2302.14	0.79	9328.64	0.63	13585.00	0.70
Average	317.40	0.92	1329.94	0.74	2019.95	0.50	3667.28	0.70

benchmark의 프로그램들은 그 목적에 따라 내부 로직이 서로 다를 뿐 아니라, 대상 프로그램 및 결함 버전에 따라 존재하는 결함 구문의 종류가 서로 달라서 결함 위치추정 기법의 정확성을 측정하기 위한 다양한 실험 환경을 조성할 수 있다.

표 1은 실험에 사용된 프로그램과 각 프로그램의 결함 버전 수 및 테스트 케이스의 수를 나타낸다. 각각의 결함 버전들은 한 개의 결함(한 개의 구문만이 결함이 없는 버전과 다른 것) 또는 다중 결함(두 개 구문 이상이 결함이 없는 버전과 다른 것)을 가지고 있다. 결함들은 다른 연구자들에 의해서 프로그램에 심어 졌으며, 결함 구문을 실행하는 테스트 케이스가 없거나 실패 테스트 케이스가 없는 결함 버전들은 본 논문의 실험에서 제외 하였다. 본 실험에서는 SIR benchmark가 제공하는 모든 테스트 케이스를 사용하였으며, 해당 테스트 케이스들은 카테고리 파티션 방법[25]으로 생성된 기능 테스트 케이스와 추가 분기 커버리지(additional branch coverage)달성을 위한 테스트 케이스 및 랜덤으로 생성된 테스트 케이스들로 구성되어 있다[9]. 표 1의 3번째 열은 본 논문의 실험에서 사용한 결함 버전의 개수를 나타내며 4번째 열은 benchmark에서 제공하는 테스트 케이스의 수를 나타낸다.

4.2 실험 구현

본 저자들은 FEAST와 Tarantula[10] 기법을 2056 줄의 C++코드로 구현하였다. 실험 대상 프로그램은 gcc 컴파일러를 이용하여 컴파일 되었고, 대상 프로그램을 benchmark에서 제공하는 테스트 케이스를 이용하여 실행 한 후 gcov를 이용하여 구문 커버리지를 측정하였다. 본 논문에서 실험을 위해 FEAST와 Tarantula 기법을 구현한 도구(tool)는 gcov에 의해 얻어진, 각 테스트 케이스의 구문 커버리지를 기록한 파일을 파싱한 후 어떤 테스트 케이스가 어떤 구문을 실행하였는지를 나타내는 테이블(그림 1)을 생성한다. 각 테스트 케이스의 실행 결과(성공 또는 실패)는 동일한 테스트 케이스를 결함이 없는 버전에서 실행한 후 관찰할 수 있는 실행 결과값이 서로 다를 경우 '실패', 같을 경우 '성공'으로 기록된다. 본 실험에서는 실험대상 프로그램에서 테스트 케이스들에 의해 실제 실행된 구문들만 고려하며, 실행된 구문들만이 FEAST와 Tarantula 기법에 의해 각자 자신의 의심도를 가지게 된다. 실행된 구문들은 의심도가 높을수록 높은 순위를 가진다. 모든 실험은 Debian 6.05 운영체제와 Intel i5 3.6Ghz, 8GB 메모리를 장착한 하드웨어 환경에서 수행되었다.

4.3 실험의 타당도에 대한 위협 요인

우리 실험의 타당도에 대한 가장 큰 외부적 위협 요인(threat to external validity)은 본 논문의 실험에서 대상으로 하는 프로그램들의 대표성이다. 본 실험에서는 10개의 c 프로그램들만을 대상으로 실험을 수행하기 때문에 해당 프로그램들이 실제 사용자에 의해 사용되는 프로그램들의 특성들을 모두 반영하지 못할 수 있다. 하지만, 다른 연구들[2,8,10,11,18,19,23]에서 결함 위치추정 기법의 정확성을 측정하기 위해서 사용한 벤치마크인 SIEMENS benchmark내의 모든 프로그램을 실험 대상에 모두 포함하였을 뿐만 아니라, 실제 사용자에 의해 사용되는 프로그램에 대해 FEAST 기법의 정확성을 측정하기 위해 flex, grep, space와 같은 큰 크기(9000 줄 이상의 프로그램)의 프로그램들도 실험대상에 포함하였다. 타당도에 대한 두 번째 외부적 위협 요인은 개발자가 실행된 구문들을 의심도 순으로 검사하고, 개발자 스스로가 검사하는 구문이 결함 구문인지 아닌지를 정확

히 판단할 수 있다는 가정이다. 일반적으로 개발자는 결함의 위치를 찾기 위해 프로그램 내부의 로직을 고려하면서 특정 구문과 관련된 여러 구문들을 검사하여 결함의 위치를 추정하기 때문에 위의 가정이 완전히 성립하지 않을 수 있다. 하지만, 결함 위치추정 기법은 프로그램 디버깅시에 사용되므로 사용자가 특정 구문이 결함 구문인지 아닌지를 정확히 판단할 수 있도록 지원 해줄 수 있는 다양한 기법이 존재한다(e.g., [26,27]). 또한 위의 가정은 이미 관련 연구(e.g., [2,8,10,11,17-19,23,28]) 들에서 일반적으로 사용되므로, 우리들은 위의 가정이 다른 관련연구와 본 논문의 연구를 공평하게 비교하기 위해 필요한 가정이라고 생각한다.

가장 큰 내부적 위협 요인(threat to internal validity)은 FEAST와 Tarantula 기법을 구현한 도구에 존재할 수 있는 결함이다. 해당 위협 요인을 제어하기 위해 우리는 FEAST와 Tarantula기법을 구현한 도구의 정확한 작동 여부를 광범위하게 테스트 하였다.

5. 실험 결과 및 분석

5.1 RQ1에 대한 고찰: 테스트 케이스에 부여된 결함 가중치의 정확성

표 2는 결함 구문을 찾기 위해서 실험에서 사용한 테스트 케이스들에 대한 통계를 나타낸다. 실패 테스트 케이스 대 성공 테스트 케이스의 비율은 약 1:10.6 $\left(=\frac{1329.94+2019.95}{317.40}\right)$이며, CCT와 non-CCT의 비율은 1:1.5$\left(=\frac{2019.55}{1329.94}\right)$이다. 또한 표 2의 마지막 행에 나타나 있듯이 CCT의 결함 가중치 평균(0.74)은 non-CCT의 결함 가중치 평균(0.50)보다 높고, 실패 테스트 케이스의 결함 가중치 평균(0.92)보다는 낮다.

그림 2는 표 1의 3번째 열에 나타나 있는 총 171개의 결함 버전(faulty version)들에서 사용된 테스트 케이스들의 결함 가중치 값에 따른 CCT와 non-CCT의 비율을 보여준다. 그림 2에서 테스트 케이스들의 가중치가 높아 질수록 CCT의 비율이 증가함을 확인할 수 있다. 예를 들어, 0.9에서 1사이((그림 2) 가로축의 0.95)의 결함 가중치를 가지는 성공 테스트 케이스들 중 87.7%는 CCT이고 12.3%는 non-CCT이다. 즉, 0.90이상의 결함 가중치를 가지는 테스트 케이스일 경우 0.877의 확률로 coincidentally correct test case(CCT)인 반면, 0.5에서 0.6 사이의 결함 가중치를 가지는 성공 테스트 케이스들은 0.218의 확률로 coincidentally correct test case (CCT)다.

그림 3은 결함 가중치에 따른 CCT와 non-CCT 누적 비율을 나타낸다. 그림 3에 나타나 있는 바와 같이 55%의 CCT가 0.8이상의 가중치를 가지고 있는 반면,

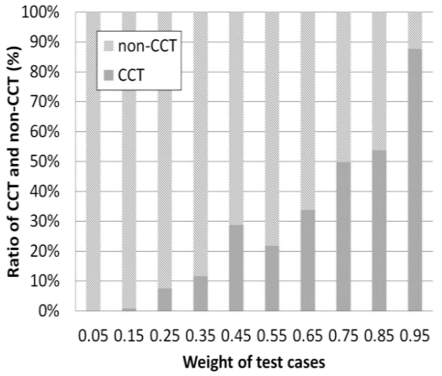


그림 2 결함 가중치에 따른 CCT와 non-CCT의 비율
Fig. 2 Ratios of CCTs and non-CCTs for different fault weights

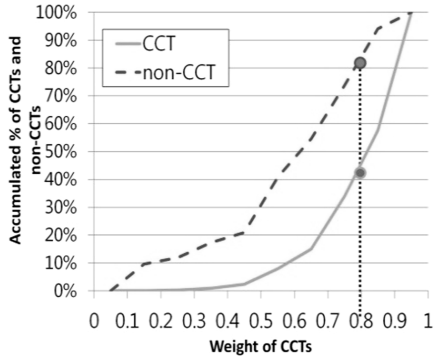


그림 3 결함 가중치에 따른 CCT와 non-CCT의 누적 비율
Fig. 3 Accumulated ratios of CCTs and non-CCTs for different fault weights

18%의 non-CCT만이 0.8이상의 가중치를 가지고 있다.

위의 사실로부터 본 논문에서 제안한 결함 가중치 공식이 CCT와 non-CCT를 충분히 구분한다는 것을 확인할 수 있다.

5.2 RQ2에 대한 고찰: FEAST의 정확성

표 3은 본 실험에서 사용된 10개 대상 프로그램에 대해 FEAST와 Tarantula를 적용한 결과를 나타낸다. 표 3에서 FEAST기법이 Tarantula에 비해 더 정확한 것을 확인할 수 있다(더 정확하게 결함 구문의 위치를 추정하는 기법은 **볼드체** 및 밑줄로 표시되어 있음). FEAST는 replace와 schedule1 프로그램을 제외한 모든 프로그램에서 결함 구문의 위치를 Tarantula보다 더 정확하게 추정한다. 예를 들어, 3번째 행의 print_tokens2에서 FEAST는 Tarantula에 비해 81.28% $\left(=\frac{12.88-2.41}{12.88}\right)$ 더 정확하다. 하지만 replace와 schedule1에 대해서는 Tarantula가 FEAST보다 더 정확하다.[2)]

표 3 실험 대상 프로그램에서 결함 구문을 찾기 위해 검사해야 하는 실행된 구문의 평균 비율(%)

Table 3 % of code examined to localize a fault in the target programs

Target program	% of code examined		Relative Imprv.(%)
	Taran.	FEAST	
print_tokens	28.09	**11.83**	57.87
print_tokens2	12.88	**2.41**	81.28
replace	**9.19**	12.08	-31.37
schedule1	**5.95**	10.32	-73.59
schedule2	52.85	**45.87**	13.21
tcas	27.81	**26.89**	3.28
tot_info	27.30	**15.92**	41.67
flex	28.18	**15.47**	45.12
grep	28.54	**1.48**	94.83
space	5.86	**3.91**	33.22
Average	22.67	**14.62**	26.55

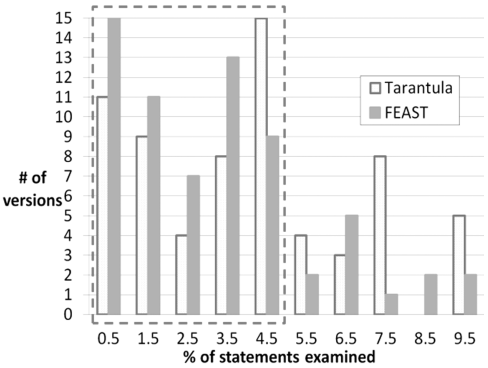


그림 4 매우 정확하게 결함 구문의 위치를 찾을 수 있는 결함 버전의 개수

Fig. 4 # of versions for which fault localization is very precise

실제로 FEAST와 Tarantula 기법을 이용하여 결함 구문의 위치를 추정할 때 FEAST는 Tarantula 기법에 비해 더 유용할 수 있다. 이는 **매우 정확하게** (i.e., 5% 미만의 실행된 구문을 검사하면 결함 구문을 찾을 수 있을 때) 결함 구문의 위치를 추정할 수 있는 결함 버전들의 숫자가 Tarantula에 비해 FEAST가 더 많기 때문이다. 그림 4에 나타나 있는 바와 같이 Tarantula 기법에서 5% 미만의 실행된 구문들을 검사함으로써 결함 구문을 찾을 수 있는 결함 버전들은 전체 171개 버전 중 47(=11(0~1%) + 9(1~2%) + 4(2~3%) + 8(3~4%) + 15(4~5%))개인 반면, FEAST기법에서는 171개 버전 중 55(=15+11+7+13+9)개의 버전에서 5%미만의 실행된 구문들을 검사하면 결함 구문을 찾을 수 있다. 더욱이 3% 미만의 실행된 구문을 검사함으로써 결함 구문을 찾을 수 있는 버전의 숫자는 Tarantula 기법에서는 24개인 반면 FEAST 기법에서는 33개로, FEAST가 Tarantula기법보다 37.5%(=(33-24)/24) 더 많다. 많은 수의 줄로 구성된 프로그램에서 결함 구문을 찾기 위해 검사해야 하는 실행된 구문의 비율이 3~5% 이상일 경우에는 결함의 위치추정을 위해 고려해야 하는 요소(e.g., 4.3장의 가정)들이 많아서, 결함 위치추정 기법의 가치가 크게 떨어질 수 있다[29]. 이런 면에서 FEAST는 Tarantula 기법보다 더 유용하다.

표 4 결함 구문의 위치가 더 정확하게 추정된 버전의 개수

Table 4 # of versions where a fault is localized more precisely

Target program	# of versions where a fault is localized more precisely		
	Taran.	FEAST	Tie
print_tokens	0	**5**	2
print_tokens2	1	**5**	3
replace	**12**	**12**	6
schedule1	2	**3**	0
schedule2	1	**7**	1
tcas	7	**26**	8
tot_info	0	**18**	5
flex	1	**15**	0
grep	0	**3**	0
space	2	**18**	8
Average	2.6	**11.2**	3.3

또 다른 흥미로운 결과 중 하나는 FEAST가 replace를 제외한 다른 모든 실험대상 프로그램에서 Tarantula 기법보다 더 정확하다는 것이다. 표 4는 FEAST기법이 더 정확하게 결함 구문의 위치를 추정한 결함 버전의 개수를 보여준다. 예를들어, FEAST 기법은 tot_info의 총 23개 버전 중 18개 버전에서 더 높은 정확성을 보였고, 5개 버전에서는 Tarantula 기법과 동일한 정확성을 나타내었다.

5.3 RQ3에 대한 고찰: FEAST기법의 안정성

표 5는 Tarantula와 FEAST기법에서 각 프로그램에서 결함을 찾기 위해 가장 적은 비율의 실행된 코드를 검사해야하는 best 버전과 가장 많은 비율의 실행된 코드를 검사해야하는 worst 버전을 보여준다. 표 5에서 확인할 수 있듯이, FEAST가 best 버전과 worst 버전에서 검사해야하는 코드의 비율 차이의 평균인 41.53%

2) schedule1에 대해서 FEAST 기법이 더 낮은 정확성을 이는 이유는 schedule1의 7번 결함 버전에서 한 개의 결함 구문을 찾기 위해 FEAST는 40.79%의 실행된 구문을 검사해야 하는 반면, Tarantula는 0.66%의 실행된 구문을 검사해야 하기 때문이다. 이는 해당 결함 버전에 여러 개의 결함이 있기 때문인데(5.3 절 참조), Tarantula 기법을 이용하여 결함 구문의 의심도를 계산하면 결함 구문을 실행하는 성공 테스트 케이스의 수가 매우 적어서 결함 구문이 매우 높은 의심도를 가진다.

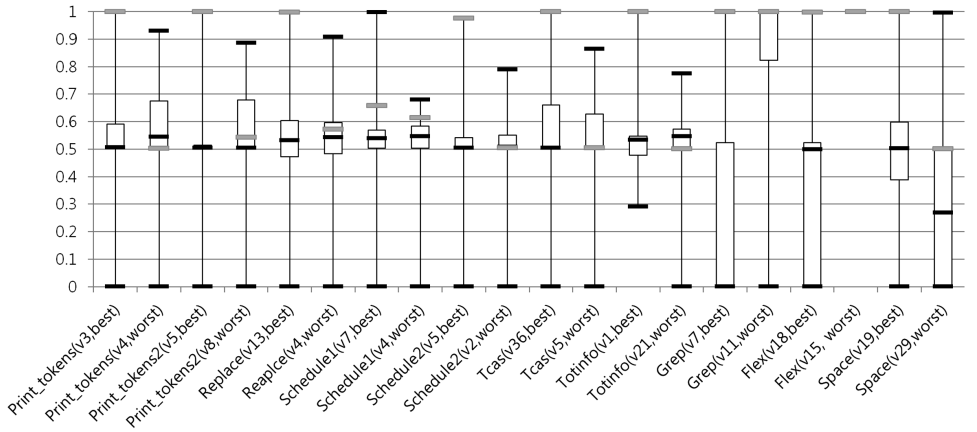


그림 5 각 프로그램에서 best version과 worst version의 FEAST기법 의심도 분포
Fig. 5 Distribution of FEAST's suspiciousness scores on the best and worst versions of the target programs

표 5 Best 버전과 Worst 버전에서 결함을 찾기 위해 검사해야하는 코드의 비율
Table 5 % of code examined for best and worst cases for Tarantula and FEAST

Target Program	% of code examined					
	Tarantula			FEAST		
	Best ver.	Worst ver.	Diff-rence	Best ver.	Worst ver.	Diff-rence
p_tokens	0.56	85.64	85.08	1.11	39.78	38.67
p_tokens2	0.47	39.44	38.97	0.47	8.49	8.02
replace	0.43	41.20	40.77	0.43	32.76	32.33
schedule1	0.66	12.84	12.18	1.36	40.79	39.43
schedule2	0.71	88.32	87.61	5.11	73.53	68.42
tcas	1.56	84.38	82.82	1.56	73.44	71.88
tot_info	0.88	73.68	72.80	0.88	61.40	60.52
flex	1.93	72.92	70.99	0.30	53.22	52.92
grep	1.02	73.61	72.59	0.39	3.41	3.02
space	0.03	36.36	36.33	0.03	40.08	40.05
Average	0.83	60.84	60.01	1.16	42.69	41.53

는 Tarantula의 60.01%에 비해 작다((표 5)의 마지막 행 참조). 이는 FEAST가 Tarantula에 비해 평균적으로 더 안정적임을 보여 준다. 그림 5는 각 프로그램의 best 버전과 worst 버전에서의 의심도 분포를 box plot 그래프로 보여 준다. 결함 구문의 의심도는 노란색 막대로 표시되어 있다. 예를 들어, print_tokens의 best 버전과 worst 버전의 의심도 분포는 해당 그래프의 가장 왼쪽에 나타나 있다. 그림에 나타나 있듯이, FEAST 기법에서는 best 버전과 worst 버전에 관계없이 결함 구문의 의심도는 매우 높다(대부분의 결함 구문은 print_tokens와 space의 worst version을 제외하고 0.9이상의 의심도를 가진다).[3)]

5.4 FEAST기법의 제한사항

FEAST가 Tarantula기법보다 평균적으로 26.55% 더 정확하지만((표 3)의 가장 오른쪽 아래 항 참조), 실험에서 사용된 171개 버전 중 20개 버전에서 FEAST는 결함 구문의 위치를 충분히 정확하게 추정하지 못한다(해당 20개 버전에서 결함 구문을 찾기 위해 실행된 구문들의 40%이상을 검사해야 한다). 우리는 위의 20개 버전을 정확성 저하 원인에 따라 다음과 같이 3개의 종류로 분류한다((표 6) 참조).

- 한 개의 결함 버전에 여러 개의 결함 구문이 존재(6개 버전)
- 결함 구문의 의심도가 가장 높은 의심도를 가진 구문과 크게 차이가 나지 않음(7개 버전)
- 결함 구문이 많은 수의 구문들로 구성된 basic block에 위치해 있음(7개 버전)

우리는 위의 제약 사항들이 커버리지 기반 결함 위치 추정 기법이 일반적으로 가질 수 있는 문제라고 생각한다.

1) 한 개의 결함 버전에 여러 개의 결함 구문이 존재(6개 버전): 예를 들어, FEAST기법을 적용했을 때 schedule1의 7번 버전에서 한 개의 결함 구문을 찾기 위해 검사해야 하는 실행된 구문의 비율은 40.79%이다. 이러한 현상의 가장 큰 원인은 해당 버전이 4개의 결함 구문을 가지고 있기 때문이다. 다중 결함 구문은 FEAST 기법을 이용하여 한 개의 결함 구문 s_f의 의심도를 계산할 때, FEAST의심도 공식의 왼쪽 항$\left(\frac{|\text{failed}(s_f)|}{|T_f|}\right)$의 값을 한 개의 결함 구문을 가진 schedule1의 다른 결함 버전들에 비해 낮은 값을 가지도록 만든다. 왜냐하면 결

3) Tarantula worst version의 결함 구문 평균 의심도는 FEAST기법보다 매우 낮은 약 0.6의 값을 갖는다.

표 6 부정확한 결함 위치 추정의 원인
Table 6 Reasons for the imprecise fault-localization results

Reasons for imprecise localization	# of ver.	Avg. % of code examined	Avg. susp of fault
Multiple faults	6.00	52.710	0.875
Neglible difference in the top susp. score	7.00	61.150	0.949
Fault in a large basic block	7.00	71.210	0.904

함 구문이 여러 개일 경우 s_f는 모든 실패 테스트 케이스에 실행되지 않을 수 있는 반면, 결함 구문이 하나일 경우 해당 결함 구문은 모든 실패 테스트 케이스에 의해 실행되기 때문이다($\frac{|failed(s_f)|}{|T_f|}$의 값이 1이 된다). 해당 버전(schedule1 버전 7)에는 모든 실패 테스트 케이스에 의해 실행되는, 결함 구문이 아닌 구문들 정상 구문들이 존재한다. 이로 인해 정상 구문들의 의심도를 계산할 때의 왼쪽 항은 결함 구문들의 의심도를 계산할 때의 왼쪽 항보다 큰 값을 가진다. 특히, 해당 버전에서 152개의 실행된 구문들 중 89개 정상 구문들의 왼쪽 항의 값은 1인 반면, 4개 결함 구문들의 왼쪽 항의 값은 평균적으로 0.65이다. 따라서 결함 구문들의 순위는 정상 구문들보다 낮아진다. 우리는 FEAST의 정확성이 다중 결함 구문으로 인해 총 6개 결함 버전에서 감소됨을 확인하였다(결함 구문을 찾기 위해 40%이상의 실행된 구문을 검사해야 한다).

다중 결함 구문에 의해 CBFL 기법의 정확성이 떨어지는 문제의 해결을 위해서 여러가지 방법들이 이미 제안되었다. 예를 들어 [30]에서는 각 결함의 위치를 해당 결함에 의해 발생된 실패 테스트 케이스들만을 이용하여 추정함으로써, 프로그램에 한 개의 결함 구문이 있을 때 결함 위치추정기법을 적용하는 것과 동일한 효과를 얻고자 한다. 이를 위해 동일 결함 구문에 의해 발생한 실패 테스트 케이스들을 클러스터링 기법을 이용하여 추정한다.

2) 결함 구문의 의심도가 가장 높은 의심도를 가진 구문과 크게 차이가 나지 않음(7개 버전): 7개 버전에서는 결함 구문과 가장 높은 순위를 가진 정상 구문(correct statement)의 의심도 차이가 크게 나지 않았다. 즉, 결함 구문의 의심도가 매우 큼에도 불구하고 결함 구문의 순위는 낮았는데, 이는 결함 구문의 의심도보다 조금 높은 의심도를 가진 정상 구문들의 수가 많았기 때문이다.

표 7은 flex버전 6과 schedule2의 버전 2, 3, 7, 8, 10, 그리고 tcas 버전 14에서 결함 구문을 찾기 위해서 40% 이상의 실행된 구문들을 검사해야 함을 보여 준다(평균적으로 58.36%의 실행된 구문들을 검사해야함). 하지만, 표에 나타난 바와 같이 결함 구문들의 의심도와 정상 구문들의 의심도 차이는 매우 작다(평균적으로 0.029의 값을 가진다).

표 7 결함 구문의 의심도와 가장 높은 순위를 가진 구문의 의심도 차이
Table 7 Differences between the suspiciousness scores of the faulty statements and high-rank non-faulty statements DIFFERENCES

Target program	Ver-sions	% of code examined	Highest susp.	Susp. Of fault	Diff-rence
flex	6	52.75	0.984	0.939	0.045
schedule2	2	61.30	0.977	0.941	0.036
	3	73.50	0.980	0.943	0.037
	7	61.59	0.970	0.940	0.030
	8	65.20	0.969	0.938	0.031
	10	50.40	0.968	0.942	0.026
tcas	14	43.75	0.981	0.980	0.001
Average		58.36	0.975	0.946	0.029

예를 들어, tcas버전 14의 경우 결함 구문의 의심도는 가장 높은 의심도를 가지는 구문과 거의 같지만, 결함 구문보다 조금 큰 의심도를 가진 27개의 정상 구문들 때문에 결함 구문을 찾기 위해서 검사해야 하는 실행된 구문의 비율이 크다(43.75%).

해당 문제의 해결을 위해 좀 더 정확한 결함 가중치 공식과 개선된 의심도 공식을 개발해야 할 것으로 판단된다(6장 결론 및 향후 연구 참조).

3) 결함 구문이 많은 수의 구문들로 구성된 basic block에 위치해 있음(7개 버전): 나머지 7개의 버전들은 tcas의 3, 5, 12, 13, 26, 27, 34번 버전들이다. 해당 결함 버전(faulty version)들의 결함 구문은 main 함수의 basic block에 위치해 있다. 해당 basic block은 26개의 구문들로 구성되어 있으며, 이는 tcas에서 실행 가능한 구문들의 약 40%를 차지한다. 즉, 결함 구문의 의심도가 1이라고 할지라도, 결함 구문을 찾기 위해서는 적어도 40%의 실행된 구문들을 검사해야 한다(결함 구문과 같은 basic block에 위치한 26개의 구문들은 결함 구문과 같은 의심도를 가지기 때문에, 26개의 구문들은 결함 구문과 같은 우선순위를 가지게 된다).

이러한 문제가 생기는 가장 큰 원인은 구문 커버리지 기반의 결함 위치 추정 기법에서는 같은 basic block에 속하는 구문들이 모두 같은 의심도를 가지기 때문이다. 이 문제의 해결을 위하여 정의-사용쌍 커버리지(definition-use pair coverage)를 사용하거나[17], 프로그램 슬라이싱(slicing)[23] 기법을 CBFL기법과 함께 사용함으로써 같은 basic block내의 구문들 중 프로그램 실패와 더 큰 상관관계가 있는 구문에 더 높은 의심도를 부여함으로써 정확성을 좀 더 향상시킬 수 있다.

6. 결론 및 향후 연구

본 논문에서는 CCT에 의한 CBFL 기법의 정확성 저하를 감소시킬 수 있는, 결함 가중치에 기반한 결함 위치 추정 기법(Fault wEight bASed localization Technique: FEAST)을 제안했다. FEAST는 CBFL 기법에서 사용되는 각 테스트 케이스에 결함 가중치를 자동으로 부여함으로써 CCT에 의한 CBFL 기법의 정확성 저하를 감소시킨다. 실제 사용자에 의해 사용되는 3개의 프로그램을 포함하는, SIR benchmark의 10개 프로그램 171개 결함 버전(faulty version)에 대한 실험을 통해 FEAST가 Tarantula기법에 비해 평균적으로 26.55% 더 높은 정확성을 가지는 것을 확인할 수 있었다.

우리들은 본 논문에서 제안한 테스트 케이스의 결함 가중치와 FEAST기법의 정확성이 주어진 테스트 케이스가 달성하는 프로그램 커버리지의 종류(e.g., branch coverage, path coverage)에 따라 어떻게 변하는지를 실험적으로 측정하여, 특정 커버리지를 달성하는 테스트 케이스 모음(test case suite)에 적합한 결함 가중치와 의심도 공식을 개발할 예정이다. 특히, 테스트 케이스의 결함 가중치를 계산할 때 테스트 케이스의 구문 커버리지뿐만 아니라 실행된 구문의 수행 순서와 구문들 사이의 의존성(e.g., 데이터 의존성 등)을 고려함으로써 테스트 케이스에 대한 결함 가중치의 정확성(CCT와 non-CCT를 구분하는 정도)을 향상시킬 계획이다.

References

[1] I. Vessey, "Expertise debugging computer programs: A process analysis," *IJMMS*, vol.23, pp.459-494, 1985.

[2] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," *ASE*, pp.273-282, 2005.

[3] D. J. Richardson and M. C. Thompson, "An analysis of test data selection criteria using the RELAY model of fault detection," *TSE*, vol.19, pp.533-553, 1993.

[4] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," *ICSE*, pp.82-91.

[5] W. E. Wong and Y. Qi, "Effective program debugging based on execution slices and inter-block data dependency," *JSS*, vol.79, pp.891-903, 2006.

[6] W. Masri, R. Abou-Assi, M. El-Ghali, and N. Al-Fatairi, "An empirical study of the factors that reduce the effectiveness of coverage-based fault localization," *DEFECTS*, pp.1-5, 2009.

[7] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of data-flow-and controlflow-based test adequacy criteria," *ICSE*, pp.191-200, 1994.

[8] X. Wang, S. Cheung, W. Chan, and Z. Zhang, "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization," *ICSE*, pp.45-55, 2009.

[9] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *ESE*, vol.10, pp.405-435, 2005.

[10] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," *ICSE*, pp.467-477, 2002.

[11] R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund, "On the accuracy of spectrum-based fault localization," *MUTATION*, pp.89-98, 2007.

[12] A. da Silva Meyer, A. A. F. Garcia, A. P. de Souza, and C. L. de Souza, "Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (Zea mays L.)," *GMB*, vol.27, pp. 83-91, 2004.

[13] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," *PLDI*, pp.141-154, 2003.

[14] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *PLDI*, vol.40, pp.15-26, 2005.

[15] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: statistical model-based bug localization," *ESEC/FSE*, vol.30, pp.286-295, 2005.

[16] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for Java," *ECOOP*, pp.733-733, 2005.

[17] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," *ICSE*, pp.56-66, 2009.

[18] W. Masri and R. A. Assi, "Cleansing test suites from coincidental correctness to enhance fault-localization," *ICST*, pp.165-174, 2010.

[19] A. Bandyopadhyay and S. Ghosh, "Tester Feedback Driven Fault Localization," *ICST*, 2012.

[20] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," *ISSTA*, Trento, Italy, pp.49-59, 2010.

[21] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," *PLDI*, pp.213-223, 2005.

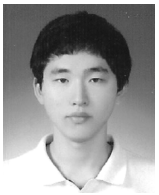
[22] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/FSE*, 2005.

[23] F. Tip, "A survey of program slicing techniques," JPL, vol.3, pp.121-189, 1995.

[24] C. Liu, X. Zhang, and J. Han, "A systematic study of failure proximity," *TSE*, vol.34, pp.826-843, 2008.

[25] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *Communications of the ACM*, vol.31, pp.676-686, 1988.

[26] H. Cleve and A. Zeller, "Locating causes of program failures," *ICSE*, pp.342-351, 2005.
[27] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *ACM SIGPLAN Notices*, pp.246-256, 1990.
[28] W. Eric Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *JSS*, vol.83, pp.188-208, 2010.
[29] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Practical fault localization for dynamic web applications," *ICSE*, pp.265-274, 2010.
[30] J. A. Jones, J. F. Bowring, and M. J. Harrold, "Debugging in parallel," *ISSTA*, pp.16-26, 2007.



문 석 현
2012년 경북대학교 컴퓨터공학과 학사
2012년~현재 KAIST 전산학과 석사과정 재학. 관심분야는 소프트웨어 Fault localization 및 디버깅

김 윤 호
정보과학회논문지 : 소프트웨어 및 응용 제 40 권 제 2 호 참조

김 문 주
정보과학회논문지 : 소프트웨어 및 응용 제 40 권 제 2 호 참조