클래스 수준 뮤테이션 분석을 위한 동등 뮤턴트 검출 기법

(An Equivalent Mutation Detection Method for Class-Level Mutation Analysis)

장 원 호 † 마 유 승 ††
(Won Ho Jang) (Yu Seung Ma)

권 용 래 †††
(Yong Rae Kwon)

요 약 클래스 수준 뮤테이션 분석(class-level mutation analysis)의 경우, 동등 뮤턴트(equivalent mutant)는 전체 뮤턴트들의 개수에서 많은 비중을 차지하기 때문에 동등 뮤턴트의 검출은 뮤테이션 수행 비용 절감에 매우 중요하다. 하지만 현재까지 클래스 수준 뮤테이션을 대상으로 동등 뮤턴트를 검출하는 연구는 미미한 실정이다. 본 논문에서는 클래스 수준 뮤테이션을 대상으로 의존성 그래프(dependency graph)를 이용하여 동등 뮤턴트를 검출하기 위한 기법을 제안한다. 제안한 기법은 인트라-클래스(intra-class) 수준에서 정적분석을 수행함으로써 인트라-메소드(intra-method) 수준의 분석 방법을 사용하는 기존의 연구들이 검출할 수 없었던 클래스 수준 동등 뮤턴트를 검출할 수 있었다.

키워드 : 테스팅, 뮤테이션분석, 동등뮤턴트

Abstract Mutation testing is known as a very useful technique for measuring the effectiveness of a test data set and finding weak points of the test set. An equivalent mutant degrades the effectiveness of mutation testing. Elimination of equivalent mutants is a very important problem in mutation testing.

In this paper, we proposed kinds of methods for detecting class-level equivalent mutants. These methods judge the equivalency of mutants through structural informations and behavioral information of the original program and mutants using static analysis. We found that our approach can detect not a few of equivalent mutants and expected that the cost of mutation testing can be saved considerably.

Key words : testing, mutation analysis, equivalent mutant

1. 서 론

뮤테이션 분석(mutation analysis)[1-3] 기법은 시험 데이터의 효율성 판단을 위해 널리 쓰이는 방법으로, 시험 대상 프로그램에 오류들을 의도적으로 삽입한 프로그램인 뮤턴트(mutant)들을 생성한 뒤 삽입된 오류들을 검출하는 시험 데이터를 좋은 시험 데이터로 판단한다. 클래스 수준 뮤테이션 분석(class-level mutation analysis)은 객체지향 프로그램에서 상속(inheritance), 다형성(polymorphism) 등의 객체지향 특성으로 인해 발생하는 오류들을 대상으로 한다.

뮤테이션 분석은 그 효율성이 뛰어남이 입증되었으나, 동등 뮤턴트(equivalent mutant) 생성과 다수의 뮤턴트 수행이라는 문제로 인해 고가의 수행비용이 소요되는 단점이 있다[2-6]. 특히, 클래스 수준 뮤테이션 분석의 경우 동등 뮤턴트의 비중이 높음이 밝혀졌지만[7], 이들의 검출에 대한 연구는 미미한 실정이다. 따라서, 본 논문에서는 클래스 수준 뮤테이션을 대상으로 동등 뮤턴트의 검출 능력을 향상시키기 위한 방법을 제안한다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문의 배경 지식인 동등 뮤턴트의 검출 조건과 클래스 수준 뮤테이션 연산자(mutation operator)에 대해 간략히 기술한다. 3장에서는 본 논문에서 제안하고자 하는 의존성 그래프(dependency graph)를 이용한 동등 뮤턴트 검출 기법을 설명한다. 4장에서는 제안한 기법을 통한 실험 결과를 기술한다. 관련 연구는 5장에서 기술하였으며, 6장에서는 결론 및 향후 연구에 대해 기술한다.

2. 배경 지식

2.1 동등 뮤턴트(Equivalent Mutant) 검출 조건

동등 뮤턴트는 원 프로그램(original program)과 완벽히 동일한 기능을 수행하는 프로그램이다. 동등 뮤턴트는 원 프로그램과 동일한 행위를 하므로, 서로 다른 실행 결과를 산출하는(동등 뮤턴트를 원 프로그램과 구

· 이 논문은 제36회 추계학술발표회에서 '클래스 수준 뮤테이션 분석을 위한 동등 뮤턴트 검출 기법'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : KAIST 전산학과
jwh1807@gmail.com
†† 정 회 원 : ETRI
ysma@etri.re.kr
††† 종신회원 : KAIST 전산학과 교수
YongRaeKwon@kaist.ac.kr

논문접수 : 2009년 12월 21일
심사완료 : 2010년 2월 22일

정보과학회논문지: 컴퓨팅의 실제 및 레터 제16권 제5호(2010.5)

분해내는) 시험 데이터는 존재하지 않는다. 하지만 뮤테이션 분석에서 시험 데이터들의 효율성은 전체 뮤턴트 개수에서 검출된 뮤턴트의 개수 비율로 산출이 되므로, 동등 뮤턴트가 제거되지 않은 상태에서는 시험 데이터 집합의 효율성이 제대로 계산될 수 없다[5].

시험 데이터가 뮤턴트를 원 프로그램과 구별해 내기 위해서는 아래의 3가지 조건을 순차적으로 모두 만족해야 한다. 만일, 이 중 하나의 조건이라도 만족되지 않았다면 그 뮤턴트는 원 프로그램과 구별해 낼 수 없게 되고 동등 뮤턴트로 판정할 수 있다[5].

① Reachability condition(D_R) : 뮤테이션이 일어난 문장을 반드시 실행해야 한다.

② Necessity condition(D_N) : 뮤테이션이 일어난 문장이 실행된다면, 문장 실행 직후의 프로그램 상태가 원 프로그램과 달라야 한다.

③ Sufficiency condition(D_S) : 프로그램의 최종 상태가 원 프로그램과 달라야 한다.

3. 동등 뮤턴트 검출 기법

본 논문에서 제안하는 동등 뮤턴트 검출 기법은 뮤테이션 대상 클래스에 대한 메소드(method) 수준 의존성 그래프(dependency graph)[8]를 추출하여 클래스 멤버 간의 의존성 관계를 파악한 뒤, 뮤테이션을 통한 클래스 변형이 영향을 미치는 범위를 분석한다.

본 장에서는 의존성 그래프가 본 기법을 위해 어떻게 해석 되는지를 기술하고, 이를 이용한 2가지 동등 뮤턴트 검출 기법 - 알고리즘 기법과 휴리스틱 기법 - 을 제안한다. 휴리스틱 기법은 D_N 조건을 위배하는 동등 뮤턴트들을 검출하며, 알고리즘 기법은 D_R과 D_S조건을 위배하는 동등 뮤턴트들을 검출한다. 이 두 기법은 서로 다른 동등 뮤턴트 검출 조건을 다루므로 두 기법은 상호보완적으로 사용된다.

3.1 동등 뮤턴트 검출을 위한 의존성 그래프

본 장에서는 제안하고자 하는 기법에 사용되는 의존성 그래프의 각 요소들과 함수들의 의미에 대해 기술한다.

3.1.1 의존성 그래프 요소

의존성 그래프는 노드(node), 에지(edge), 진입점(entry point), 외부점(external point)의 4개의 요소들로 구성되며 그 의미는 아래와 같다.

- 노드 : 클래스의 멤버 - 필드(field) 또는 메소드 - 들은 각각의 노드로 표현된다.
- 에지 : 각 노드들 사이의 의존성을 표현하며 방향성이 있다. 메소드 간의 호출 관계, 메소드와 필드 간의 데이터 종속 관계, 필드 간의 데이터 종속 관계가 있을 때 의존성이 있으며, 에지로 표현된다..
- 진입점 : 클래스 외부에서 해당 클래스 멤버에 접근하는 경로를 추상화하는 노드로, 클래스 당 하나의 진입점이 존재한다.
- 외부점 : 클래스에서 외부의 상태를 변화시키는 경로를 추상화 하는 노드이다.

상속 관계를 갖는 클래스의 경우, 부모 클래스를 기반으로 자신의 확장된 구조를 추가하는 점진적 형태를 지니게 된다[9]. 의존성 그래프 역시 부모 클래스의 의존성 그래프를 기반으로 자신의 구조를 추가하는 점진적 형태를 가질 수 있다. 그림 2는 그림 1에 기술된 Parent와 Child 클래스에 대한 의존성 그래프들이다. Child 클래스는 Parent 클래스의 자식 클래스로, 그림 2에서와 같이 Parent 클래스의 의존성 그래프를 기반으로 표현됨을 알 수 있다.

```
class Parent {
  public void method() {
    System.out.println("parent");
  }
}
class Child extends Parent {
  private int a = 0;
  public void method() {
    a++;
    super.method();
  }
}
```

그림 1 상속 관계를 갖는 Parent와 Child 클래스

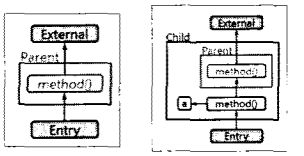

그림 2 Parent와 Child의 의존성 그래프

3.1.2 의존성 함수 의미

클래스 멤버들 간에 직간접적인 의존관계파악을 위해, 이행적 폐쇄(transitive closure)를 추출해낸다. 본 논문에서는 3.1.1장에서 기술한 의존성 그래프를 통해 이행적 폐쇄를 추출하기 위한 두 가지 함수를 제안한다.

- *Def*(G, V) : 의존성 그래프 G에서, 노드 V로 들어오는 방향 에지들의 이행적 폐쇄를 의미한다.
- Ref(G, V) : 의존성 그래프 G에서, 노드 V에서 나가는 방향 에지들의 이행적 폐쇄를 의미한다.

그림 3은 그림 2의 Child 클래스의 의존성 그래프 G를 용해 Child의 필드인 a에 영향을 미칠 수 있는 노드들인 Def(G, a)를 도식화한 것이다.

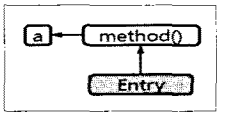

그림 3 Def(G, a)

3.2 동등 뮤턴트 검증 기법 1: 알고리즘 기법

뮤턴트 내의 변형이 일어난 코드를 어떠한 입력으로도 실행시킬 수 없는 경우($D_R = \varnothing$인 경우)나, 어떠한 입력으로도 결과값에 영향을 미칠 수 없는 경우($D_S = \varnothing$인 경우)를 만족하는 동등 뮤턴트 중의 일부는 의존성 그래프를 사용하여 검출해낼 수 있다.

그림 4는 뮤턴트의 코드 영역을 실행 가능한 부분(Ref(G, V_{ENT}))과 외부에 영향을 미치는 부분(Def(G, V_{EXT}))을 기준으로 나눈 것이다.

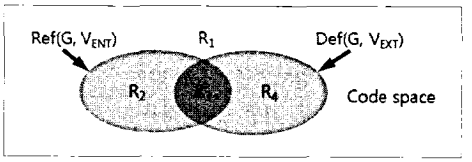

그림 4 코드 영역

- R1 : 실행되지 않고, 외부에 영향도 없음
- R2 : 실행되지만, 외부에 영향이 없음
- R3 : 실행되고, 외부에 영향을 미침
- R4 : 외부에 영향을 미칠 수 있지만, 실행되지 않음

뮤테이션이 일어난 부분에 해당하는 노드 V가 속하는 영역이 코드영역의 어느 부분이냐에 따라 동등 여부를 판가름할 수 있다. 직관적으로, 노드 V가 R3에 속하지 않는다면 해당 뮤턴트는 최소한 D_R 또는 D_S 중의 하나의 조건을 위배하므로 동등 뮤턴트로 판정할 수 있다.

o $D_R \cap D_S = \varnothing \Rightarrow V \notin R3 = Ref(G, V_{ENT}) \cap Def(G, V_{EXT})$

3.3 동등 뮤턴트 검증 기법 2: 휴리스틱 기법

본 장에서는 클래스 수준의 뮤테이션 연산자 중에서 뮤테이션 코드 수행 직후의 상태가 원 프로그램과 다르지 않은 동등 뮤턴트($D_N = \varnothing$인 경우) 검출 조건을 본 저자의 경험에 의거하여 13가지 클래스 뮤테이션 연산자에 대해 기술한다.

표 2는 13가지 클래스 뮤테이션 연산자에 대해 휴리스틱 기법을 이용해 동등 뮤턴트를 검출해내는 조건이다. 이해를 돕기 위해, IOP 클래스 뮤테이션 연산자를 이용한 예시를 기술한다. IOP 연산자는 메소드 오버라이딩(overriding) 구현 시 발생하는 실수를 다룬다 [10]. IOP는 super 키워드를 이용한 오버리든 메소드의 호출 순서를 바꾸는데 그림 5는 그 예이다. IOP 연산자는 행위적(behavioral) 변경을 발생시키지만, 메소드 수준 의존성 그래프는 그림 6으로 동일하다. 이 경우, 표 2의 조건을 기반으로 동등 뮤턴트를 검출할 수 있다. IOP 연산자는 오버라이딩 메소드의 확장된 코드(① : Ref(G-$E_{SUPER,V}$,V))와 super 메소드 호출(② : Ref(G_P, V_P)) 순서를 바꾸는데, 예제에서처럼 이 두 부분이 서로

표 2 휴리스틱 기법의 동등 조건들

연산자	동등 조건
IHI	$Def(G',V_P)\cap Def(G',V')=\varnothing \wedge Ref(G',V_P)\cap Ref(G',V')=\varnothing \wedge \nexists Init(C',V_P)$
IHD	$Def(G,V_P)\cap Def(G,V)=\varnothing \wedge Ref(G,V_P)\cap Ref(G,V)=\varnothing \wedge \nexists Init(C,V_P) \wedge \nexists Init(C,V)$
IOD	$Ref(G-E_{SUPER,V},V)\not\supseteq V_{EXT} \wedge (Ref(G_P-E_{SUPER,V_P},V_P)\not\supseteq V_{EXT} \vee \exists E_{SUPER,V})$
IOP	$Ref(G-E_{SUPER,V},V)\not\supseteq V_{EXT} \vee Ref(G_P,V_P)\not\supseteq V_{EXT}$
ISI	No local variable with same symbol ∧ No hiding attribute
ISD	$(Ref(G,V)\not\supseteq V_{EXT} \wedge \exists E_{SUPER,V}) \vee$ No overriding method
IOR	$Ref(G,V)\not\supseteq V_{EXT} \wedge Ref(G',V')\not\supseteq V_{EXT}$
IPC	$Ref(G,V_{Default\ constructor})\not\supseteq V_{EXT} \wedge Ref(G,V_{Called\ constructor})\not\supseteq V_{EXT}$
JTI	No local variable having same symbol
JTD	No local variable having same symbol
JSI	$Ref(G,V)\not\supseteq V_{EXT} \vee Def(G,V)\not\supseteq V_{ENT}$
JSD	$Ref(G,V)\not\supseteq V_{EXT} \vee Def(G,V)\not\supseteq V_{ENT}$
JDC	$Ref(G,V)\not\supseteq V_{EXT}$

G : 원본 클래스의 의존성 그래프
G′ : 뮤턴트의 의존성 그래프
G_P : 부모 클래스의 의존성 그래프
V : 변형이 일어날 부분의 노드
V′ : 변형이 일어난 후의 노드
Init(C, V) : 원본 클래스 C에서 V의 초기화 코드
$E_{SUPER,V}$: V노드의 super 호출에 해당하는 에지
V_{ENT} : 진입점 노드
V_{EXT} : 외부점 노드

```
class Child extends Parent {
  private int a = 0;
  public void method() { // 원본 코드
    super.method();      // a++;
    a++;                 // super.method();
  }
}
```

그림 5 IOP가 적용된 Child 클래스의 뮤턴트

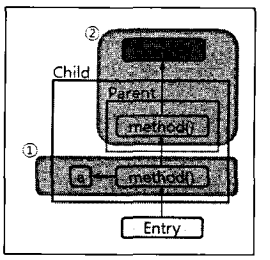

그림 6 IOP 조건

영향을 미치지 않는 독립적인 경우라면, IOP의 동등 조건(최대 두 부분중 하나만 외부에 영향을 미치는 경우)에 부합하게 되므로 동등 뮤턴트로 판정할 수 있다.

4. 실 험

현존하는 클래스 뮤테이션 시험 도구인 MuJava[11]는

일부 동등 뮤턴트를 검출하고 있다[12]. 따라서, 본 장에서는 MuJava에 내장된 동등 뮤턴트 검출 기법[12]과 본 논문에서 제안한 기법을 이용하여 동등 뮤턴트 검출 개수를 비교한 뒤 제안한 기법의 효율성을 검증한다.

4.1 실험 도구

제안한 기법의 효율성 검증을 위해, Java 뮤테이션 분석 도구인 MuJava[11]를 확장하여 본 기법을 적용하는 도구를 구현하였는데 그 구조는 그림 7과 같다.

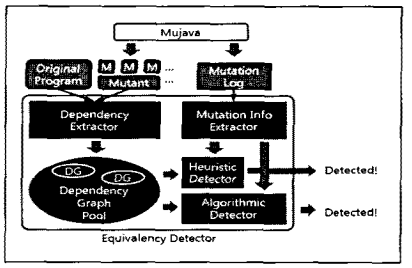

그림 7 동등 뮤턴트 검출 도구 구조

4.2 실험 결과

본 실험에서는 다양한 크기와 다양한 용도를 가지는 자바 기본 패키지 일부를 대상으로 하였으며, 그 내용은 표 3과 같다. 표 3의 뮤턴트 개수는 MuJava에서 동등 뮤턴트 제거 기능(이하 MuJava의 필터링)을 동작시키지 않고 생성한 뮤턴트들의 개수를 의미한다.

표 3 실험 대상과 생성된 뮤턴트 수

패키지	파일 수	뮤턴트 개수
java.math	5	495
java.net	54	1515
java.security	64	421
java.sql	29	106
java.util.zip	18	495
합계	170	3032

실험 결과, 전체 뮤턴트들 중에 휴리스틱(OH: 8.6%) 및 알고리즘(OA:9.9%), 동시적용(OB:14.5%) 기법을 적용해서 최대 14.5%의 동등 뮤턴트를 검출할 수 있었다. 또한 MuJava에서 필터링(OF:32.8%) 후에 휴리스틱(OFH:35.9%) 및 알고리즘(OFA:39.9%), 동시적용(OFB: 40.1%) 기법을 적용해서 최대 40.1%의 동등 뮤턴트를 검출할 수 있었다. 즉, MuJava에서의 필터링을 적용 후에도 제안한 기법을 사용하여 추가로 검출 효율을 향상시킬 수 있었다. 각 패키지 별로 편차는 존재하지만 MuJava의 필터링, 본 논문의 휴리스틱, 알고리즘 기법을 중첩 적용함으로써 검출 효율이 증가하는 것을 볼 수 있다.

MuJava의 필터링 후에는 검출 성과가 줄어들기는 하지만 여전히 유효한 것으로 보인다. MuJava의 필터링에서는 996개의 동등 뮤턴트를 검출한 반면, 본 논문에서 제안한 기법을 중첩 적용시에는 추가로 220개의 동등 뮤턴트를 검출함으로써 약 22.1%의 검출능력 향상을 확인할 수 있었다.

- O : MuJava의 필터링 없이 생성한 뮤턴트 개수
- OH : 휴리스틱 기법 적용
- OA : 알고리즘 기법 적용
- OB : 두 기법 동시 적용
- OF : MuJava의 필터링 적용
- OFH : OF 후 휴리스틱 기법 적용
- OFA : OF 후 알고리즘 기법 적용
- OFB : OF 후 두 기법 동시 적용

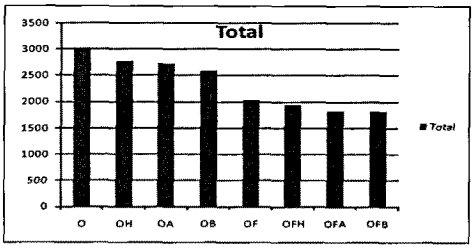

그림 8 동등 뮤턴트 제거 후 뮤턴트 개수
* 남은 뮤턴트 수이므로 적을수록 좋음 *

- F : MuJava의 필터링시 제거 개수
- H : 휴리스틱 적용시 검출 개수
- A : 알고리즘 적용시 검출 개수
- B : 두 기법 동시 적용시 검출 개수

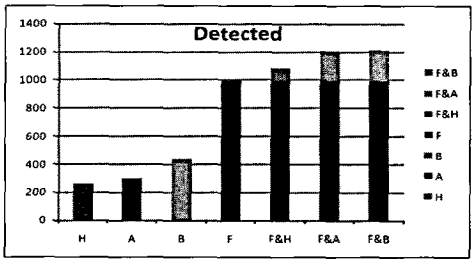

그림 9 검출된 개수
* 검출한 동등 뮤턴트 수이므로 많을수록 좋음 *

4.3 실험 검증 & 개선점

본 실험의 검증을 위해, 제안된 기법만의 검출 개수 441개 중 40개를 무작위 선택하여 수작업으로 동등성 여부를 판단하였다. 그 결과 35%에 해당하는 14개는 잘못된 결과로 확인되었다. 35%의 부정확함은 제안된 기법의 문제가 아닌 의존성 그래프 추출의 부정확함에 기인한 것으로 확인되었으며, 의존성 그래프 추출 방법을 개선함으로써 검출 정확성 향상을 꾀할 수 있을 것이다.

5. 관련 연구

뮤턴트가 동등 뮤턴트임을 증명하는 문제는 두 프로그램 사이의 등가성(equivalency)을 입증하는 문제로 귀결되며, 이는 해결 불가능한(undecidable) 문제이다[4,5]. 따라서, 뮤턴트의 검출은 특정 조건을 가지는 부분해(partial solution)들에 대한 알고리즘들을 개발하여 자동화하고, 나머지 동등 뮤턴트들은 수작업에 의하여 검출되어져 왔다. 순차 프로그램을 대상으로 동등 뮤턴트 검출에 대한 다양한 연구가 진행되어왔다.

논문 [4]에서는 컴파일러의 프로그램 최적화 알고리즘이 동일한 결과를 내는 프로그램의 코드를 변화시킨다는 점에 착안하여, 6가지 최적화 패턴을 뮤테이션 연산자와 비교하여 동등 뮤턴트 검출 기법을 제안하였다. 이 기법은 모든 인트라-메소드 수준의 동등 뮤턴트 중에서 약 9%에 해당하는 동등 뮤턴트를 검출해 내었다.

논문 [5]에서는 실행 불가능 경로(infeasible path)에서 발생하는 뮤테이션은 모두 동등 뮤턴트라는 점에 착안하여, 실행 불가능 경로상의 동등 뮤턴트를 찾고자 하였다. 이 기법은 약 48%의 동등 뮤턴트를 검출해 내었다.

논문 [6]에서는 프로그램 슬라이싱(program slicing) 기법을 사용하여 뮤테이션과 관련된 코드만 추출하는 기법을 제안하였다. 그러나 이 기법의 자동화된 검출 기법은 아이디어 수준의 제안에 그치고 있어서, 실제 실험 결과는 존재하지 않는다.

하지만, 이 기법들은 전통적 뮤테이션 연산자에 적용할 수 있는 기법들[5,6]로, 클래스 수준 뮤테이션 연산자들에 적용할 수 없거나 그 효과가 제한적이다. 또한, 객체지향 프로그램에 특화된 동등 뮤턴트 검출 연구는 미미한 실정이다.

6. 결론 및 향후과제

본 논문에서는 클래스 수준 동등 뮤턴트를 검출하는 기법을 제안하였다. 제안한 기법은 의존성 그래프를 사용한 메소드 레벨의 구조적 분석을 통하여 객체지향 관점에서 새로이 추가된 다형성, 상속성의 특징을 반영할 수 있었다.

또한, 제안한 기법을 실제 도구로 구현한 뒤, 널리 쓰이고 있는 Java 표준 API 소스를 대상으로 그 효용성을 검증하는 실험을 수행하였다. 실험 결과, 제안한 기법은 기존 MuJava보다 약 22.1% 많은 동등 뮤턴트를 검출함으로써 그 효율성을 입증할 수 있었다. 클래스 수준 뮤테이션 연산자는 많은 수의 동등 뮤턴트를 생성함을 확인할 수 있었다. 즉, 클래스 수준 뮤테이션 분석에서 동등 뮤테이션 검출은 뮤테이션 분석에서 비용 절감의 큰 요소임을 알 수 있었다.

향후 연구는 크게 두가지 방향으로 이루어질 수 있다. 먼저, 의존성 그래프의 불완전함(다형성, 다이나믹 바인딩 등으로 인함)을 보완하여 더욱 정확한 의존성 그래프를 얻음으로써 검출의 정확도를 향상시키는 일이다. 다음으로, 더 많은 뮤테이션 연산자를 대상으로 휴리스틱 기법의 조건들을 탐색하여 휴리스틱 기법의 검출 성능을 향상시키는 일이 될 수 있다.

참 고 문 헌

[1] J. H. Andrews, L. C. Briand and Y. Labiche. Is mutation an appropriate tool for testing experiments? *Proceedings of the 27th international conference on Software engineering*, pp.402-411, 2005.

[2] A. J. Offutt, J. Pan, K. Tewary and T. Zhang. *An experimental evaluation of data flow and mutation testing. Software Practice and Experience*, 26: 165-176, 1996.

[3] P. G. Frankl, S. N. Weiss and C. Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. J. Systems software 1997, 38:235-253, 1997.

[4] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. Software Testing, Verification and Reliability, volume 4 issue 3, pp.131-154, 1994.

[5] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. Software Testing, Verification and Reliability, volume 7, pp.165-192, 1997.

[6] R. Hierons, M. Harman and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. Software Testing, Verification and Reliability, volume 9, pp.233-262, 1999.

[7] Y. S. Ma, M. J. Harrold and Y. R. Kwon. Evaluation of mutation testing for object-oriented programs. ICSE'06, May 20-28, 2006.

[8] Wikipedia the free encyclopedia, Dependency Graph, http://en.wikipedia.org/wiki/Dependency_graph

[9] P. Wegner and S. B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. Proceedings of ECOOP'88, pages 55-77, 1988.

[10] Y. S. Ma, Y. R. Kwon and A. J. Offutt. Inter-class mutation operators for Java. Proceedings of the 13 th International Symposium on Software Reliability Engineering(ISSRE'02), 2002.

[11] Y. S. Ma, A. J. Offutt and Y. R. Kwon. MuJava: an automated class mutation system. Software Testing, Verification and Reliability, volume 15, pp.97-133, 2005.

[12] A. J. Offutt, Y. S. Ma and Y. R. Kwon. The class-level mutants of MuJava. *Proceedings of the 2006 international workshop on Automation of software test*, pp.78-84, 2006.