논문 2011-1-4

코드클론 표본 집합체 자동 생성기

이효섭*, 도경구**

Automatic Generation of Code-clone Reference Corpus

Hyo-Sub Lee*, Kyung-Goo Doh**

요 약

프로그램 내의 코드클론을 찾아주는 도구나 기술들을 평가하기 위해서는 해당 도구가 탐지하는 못하는 클론이 있는지 확인해야 한다. 이를 위해서 샘플 소스코드에 대해서 코드클론을 모두 모아놓은 표준 표본 집합체가 필요하다. 그런데 기존의 코드클론 표본 집합체는 여러 클론탐지 도구의 결과들을 참조해 수작업으로 구축하지만 평가 기준으로 사용하기에는 빠져있는 표본이 많다. 본 연구에서는 자동으로 코드클론 표본 집합체를 생성하는 방법을 제안하고 도구를 구현하였다. 이 도구는 프로그램 소스를 핵심구문트리로 변환한 뒤, 트리를 샅샅이 비교하여 클론 패턴을 찾아낸다. 본 도구는 오탐이 없으며, 특정한 패턴을 제외하고 미탐도 없어서 코드클론 표본 집합체를 자동으로 생성하기 적합하다. 실험결과 상용도구인 CloneDR에서 찾아낸 클론을 모두 포함하면서 2-3배 더 많은 클론들을 찾아내었고, Bellon의 기존 표본 집합체의 클론들을 거의 대부분 포함(93-100%)하면서 자동 구축한 표본 집합체의 크기가 훨씬 크다.

Abstract

To evaluate the quality of clone detection tools, we should know how many clones the tool misses. Hence we need to have the standard code-clone reference corpus for a carefully chosen set of sample source codes. The reference corpus available so far has been built by manually collecting clones from the results of various existing tools. This paper presents a tree-pattern-based clone detection tool that can be used for automatic generation of reference corpus. Our tool is compared with CloneDR for precision and Bellon's reference corpus for recall. Our tool finds no false positives and 2 to 3 times more clones than CloneDR. Compared to Bellon's reference corpus, our tools shows the 93%-to-100% recall rate and detects far more clones.

한글키워드 : 코드클론, 탐지성(recall), 표본 집합체, 트리 패턴

*,** 한양대학교 컴퓨터공학과
(email: {leehs, doh}@plasse.hanyang.ac.kr)
접수일자: 2011.4.3 수정완료: 2011.5.7
※ 본 연구는 교육과학기술부/한국연구재단 우수연구센터 육성사업의 지원으로 수행되었음(과제번호 2011-0000974).

1. 서론

소스 프로그램에서 구문적 생김새가 동일한 코드 조각을 코드클론이라고 한다. 소프트웨어에

코드클론들이 발견되는 가장 큰 이유는 코드 개발자들이 새로운 코드를 작성하기보다는 비슷한 기능을 가진 소스코드들을 복사하고 붙이기(copy-and-paste)한 뒤 그대로 또는 일부분만 수정하여 사용하기 때문이다. 정확히 말해, 코드클론 자체는 시스템에 문제를 일으키지 않는다. 그러나 클론의 일부만 수정한 경우 일괄성을 유지하지 못할 수 있고, 버그가 있는 코드를 그대로 복사하여 재사용하였다면 삽입한 코드에도 동일한 버그가 있을 가능성이 높아진다. 그리고 소스코드에 코드클론이 많으면 프로그램의 유지보수 비용이 증가하고 시스템의 질이 떨어진다[4].

기존의 연구에서 밝혀진 바에 따르면 산업용 소프트웨어의 5-20%[7], 대용량 시스템의 10-15%[5] 그리고 객체지향 COBOL의 50%이상이 클론이라고 한다[9,10]. 소스 프로그램에서 코드클론들을 찾아내는 가장 손쉬운 방법은 사람의 눈으로 하나하나 검사하는 것이다. 그러나 검사자의 능력에 따라 결과가 다를 수 있고, 대규모 시스템의 경우에는 많은 인력과 시간이 소요되어 경제적이지 못하다. 따라서 효율적이며 신뢰성이 높은 자동화된 코드클론 탐지 도구가 필요하다.

기존의 코드클론 탐지 자동화 도구들은 다양한 코드클론 탐지 기법들을 이용해서 구현하였다. 도구 사용자는 종종 정확한 클론탐지를 위해 탐지 속도가 조금 늦어지는 것을 용인하기도 하지만, 반대로 빠른 시간 내에 결과를 얻기 위해서 클론의 정확성을 일부 포기하거나 비교적 낮은 수준의 클론들을 요구하는 등 기대치를 낮추기도 한다. 이와 같이 기존의 많은 도구들 중에서 사용자 요구를 만족하는 도구를 선택하고 사용하기 위해서는 각 도구의 성능을 정확히 알아야 하고, 제대로 성능평가가 이루어져야 한다. 코드클론 탐지 도구들의 성능평가를 위해 기존에 많이 사용하는 5가지 평가 요소들(parameters)을 살펴보면 다음과 같다[10].

- 이식성(portability) : 다양한 언어들을 쉽게 지원할 수 있는가
- 정확성(precision) : 도구에서 찾아낸 클론은 정확한가(오탐은 없는가)
- 탐지성(recall) : 코드내의 모든 클론들을 빠짐없이 탐지하는가(미탐은 없는가)
- 확장성(scalability) : 복잡하고 대용량 코드들을 수행할 수 있는가
- 견고성(robustness) : 코드 복제 유형의 다양한 타입[1]들을 탐지하는가

위의 5가지 평가 요소 중에서 이식성, 정확성, 확장성, 견고성은 해당 도구의 결과만을 독립적으로 분석하여 성능을 평가할 수 있다. 그러나 탐지성은 해당 도구가 탐지하지 못하는 클론이 있는지 확인하기 위해 비교할 “절대 클론”들의 모음이 필요하다. 이런 절대 클론들의 모음을 코드클론 표본 집합체(code clone reference corpus) 라고 한다. 기존 연구에서는 여러 클론 탐지 도구들의 결과물들을 참조하여 수작업으로 모아서 신뢰할만한 코드클론 표본 집합체를 구축하였다[2,3]. 이와 같이 수동으로라도 다양한 도구의 클론들을 모아 코드클론 표본 집합체를 만드는 이유는 도구마다 찾아내는 클론이 다르기 때문이다. 도구마다 찾아내지 못하는 코드클론(미탐 : false negatives)이 있고, 찾아냈더라도 정확하지 않은 코드클론(오탐 : false positives)이 있어서 한 개의 도구에서 찾아낸 클론들만으로는 표준이 될 만한 코드클론 집합체를 구축하기 어렵다.

1) Roy와 Cordy는 코드 복제의 유형을 네 가지로 분류하였다[10]. Type 1은 공백과 주석을 제외하고는 동일한 코드, Type 2는 식별자, 상수, 타입을 제외하고 구조적/ 외형적으로 동일한 코드, Type 3는 type 2에 명령어가 삽입 또는 삭제된 코드, 마지막으로 Type 4는 구조는 다르지만 의미적으로 계산이 동일한 코드이다.

본 연구에서는 클론탐지 도구들의 성능 평가 요소인 탐지성을 평가하기 위해 반드시 필요한 코드클론 표본 집합체를 구축하는 방법을 제안한다. 그리고 이 방법을 이용해 자동화 도구를 만들어 수동으로 작성한 기존의 코드클론 표본 집합체와 비교 실험하였다. 비교결과 우리의 자동화 도구는 비교하는 소스 프로그램의 클론들을 오탐없이(no false positives) 잘 찾을 뿐 아니라, 기존 표본 집합체의 클론들을 거의 대부분 포함하면서 기존의 수작업으로는 찾을 수 없었던 클론 표본들을 많이 찾을 수 있었다.

본 논문의 구성은 다음과 같다. 2절에서는 기존의 코드클론 탐지 기법과 지금까지 수동으로 작성한 코드클론 표본 집합체들에 대해 살펴보고, 3절에서는 비교하는 프로그램에서 클론들을 빠짐없이 정확하게 찾아내는 우리 도구의 알고리즘을 설명한다. 4절에서는 우리 도구의 정확성 평가를 위해 상용도구인 CloneDR의 결과와 비교하고, 탐지성 평가를 위해 수동 구축한 Bellon의 코드클론 표본 집합체와 비교 실험한다. 그리고 5절에서 결론과 추후연구기술로 마무리 한다.

2. 기반기술

기존의 코드클론 표본 집합체는 여러 클론 탐지 도구의 결과를 분석하여 수작업으로 구축하였다. 여러 도구들의 결과들을 모아 코드클론 표본 집합체를 만드는 이유는 각각의 코드클론 탐지 도구가 오탐과 미탐의 집합이 서로 상이하여 한 개의 도구에서 찾아낸 클론들만으로는 표본 집합체 구축이 어렵기 때문이다.

다음은 기존의 코드클론 표본 집합체 구축을 위해 여러 도구들이 채용한 코드클론 탐지 기법들로, 비교단위와 구조에 따라 크게 6가지로 분류한다[10].

- 문자열기반(string-based) : 프로그램을 문자의 나열인 텍스트로 보고 문자를 기본 단위로 비교하는 방식이다.
- 토큰기반(token-based) : 어휘분석(lexical analysis)으로 생성한 토큰들의 나열인 토큰열이 비교대상이며 토큰을 기본단위로 비교하는 방식이다.
- 트리기반(tree-based) : 구문분석(parsing)을 통해 변환한 추상구문트리(abstract syntax tree : AST)가 비교대상이며 하위트리(subtree)를 기본 단위로 비교하는 방식이다.
- 계량치기반(metric-based) : 코드의 구문적 특징을 나타내는 다양한 계량치를 비교하여 코드클론을 찾아낸다. 주로 이름이나 레이아웃, 표현식 등을 계량치로 나타낸다.
- 의존그래프기반(PDG-based) : 프로그램의 제어 흐름 및 데이터 흐름을 표현하는 프로그램 의존 그래프(program dependency graph)가 비교대상이다. 부분 그래프를 기본 단위로 비교하는 방식이다.
- 혼합방식(hybrid) : 위의 방식들을 적절히 조합하여 코드클론을 판별하는 방식이다.

기존의 코드클론 탐지 기법들 중에서 일부는 복사 후 붙이기 오류 탐지나 프로그램 표절탐지, 리팩토링, 소프트웨어 형상관리와 같은 응용 분야에 그대로 적용하는데 적절하지 않다. 문자열기반은 프로그램의 줄 변화, 빈칸 삽입, 식별자의 이름변경과 같이 거의 의미 없는 차이도 구별하기 때문에 오탐(false negative)이 많이 생길 수 있어서 클론 탐지의 효용성이 떨어진다. 토큰기반은 단순한 토큰의 나열을 비교하기 때문에 구문 구조를 고려하지 않아서 오탐이 많이 생길 수 있다. 계량치기반이나 의존그래프기반은 모양이 다른 코드들이지만 그래프나 계량치가 동일하면

코드클론으로 탐지하여 오탐이 생길 수 있다. 반면 트리기반은 프로그램의 AST를 대상으로 비교하기 때문에 프로그램의 구문 구조를 비교 단위에 포함해서 코드클론을 찾는다. 따라서 프로그램의 골격이 비슷한 경우를 정확히 판별하기 위해서는 코드의 구문 구조 정보가 포함되어 있는 AST를 비교하는 방식이 적절하다.

다양한 클론탐지 도구들의 성능을 비교 평가하기 위해서 Bailey와 Burd[3]는 5개의 도구들(토큰기반 CCFinder와 JPlag, 트리기반 CloneDR, 계량치기반 Covert, 웹 서비스를 지원하는 Moss)을 이용해서 수작업으로 집합체를 만든 후 각 도구의 정확성과 탐지성을 계산하였다. Bellon[2]은 6개의 도구(문자열기반 Duploc, 토큰기반 Dup와 CCFinder, 트리기반 CloneDR, 계량치기반 CLAN, 의존그래프기반 Duplix)을 선택해서 정확성과 탐지성 그리고 코드 복제 유형을 확인해 도구들의 성능을 비교 평가하였다. Bellon의 코드클론 표본 집합체는 결과를 수집할 6개의 도구에 공동으로 있는 클론(jointly clone)을 수집하고 일부 도구에서만 탐지한 클론들을 수동으로 병합(union) 또는 교차(intersection)해 구축하였다.

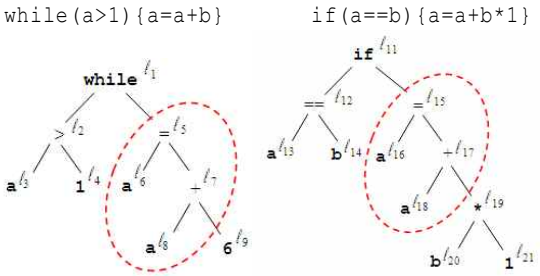
3. 코드클론 탐지

이 절에서는 비교 대상인 두 프로그램에서 코드클론을 찾는 새로운 방법에 대해 자세히 기술한다. 우선 비교 단위인 클론 패턴(clone pattern)에 대해 알아보고, 다음에는 동일한 모양의 클론 패턴들을 합병해서 만든 클론 패턴 클래스(clone pattern class)에 대해 알아본다.

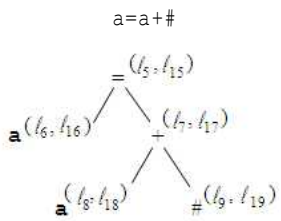
3.1 클론 패턴과 클론 패턴 클래스

비교 대상이 되는 프로그램 텍스트는 어휘분석과 구문분석을 통해 핵심구문트리(abstract syntax tree : AST)로 변환한다. 이때 핵심구문트리의 각 노드에는 프로그램 텍스트에서의 고유한 위치정보인 라인과 컬럼의 번호를 꼬리표(label)로 붙인다.

트리조각은 트리에서 서로 연결된 노드들의 덩어리이다. 두 비교대상 핵심구문트리에서 가장 큰 공통 트리조각(the largest common substructure of two trees)을 클론 패턴이라고 한다. 클론 패턴은 트리의 모양을 그대로 유지하고 있으며, 추가적으로 구멍(hole, '#'로 표시)이라는 특별한 리프(leaf) 노드가 있다. 구멍은 비교 대상 핵심구문트리에서 서로 다른 부분의 하위트리를 대체한 것이다. 클론 패턴의 각 노드에는 비교대상인 두 핵심구문트리 해당 노드의 위치정보가 쌍으로 묶여서 꼬리표로 달려있다.



(a) 비교대상 핵심구문트리

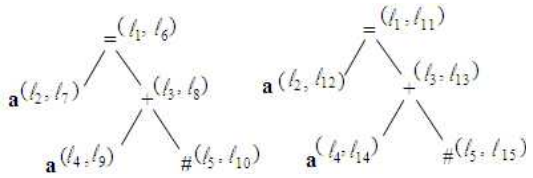


(b) 클론 패턴

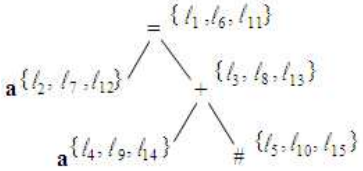
그림 1. 두 핵심구문트리와 클론 패턴

예를 들어, 그림 1(a)의 두 핵심구문트리 `while(a>1){a=a+b}`와 `if(a==b){a=a+b*1}`에서 가장 큰 공통 트리조각인 클론 패턴은 '='이 루트노드인 그림 1의 (b)과 같이 표현한다.

동일한 모양의 클론 패턴들을 클론 패턴 클래스(clone-pattern class)로 모아서 표현할 수 있다. 각 노드의 위치정보는 리스트로 표현하는데, 이 때 각 위치정보의 순서는 중요하다. 그림 2는 동일한 모양의 두 클론 패턴과 이들을 병합한 클론 패턴 클래스의 예를 보여준다.



(a) 동일한 모양의 클론 패턴



(b) 클론 패턴 클래스

그림 2. 동일한 모양의 두 개의 클론 패턴과 이를 병합한 클론 패턴 클래스

3.2 알고리즘

두 핵심구문트리에서 가장 큰 공통된 클론 패턴을 찾기 위해서는 트리의 각 노드를 루트노드로 하는 가능한 모든 트리들의 쌍을 대상으로 비교해야 한다. 각 비교대상 트리 쌍에 대해서 루트 노드부터 하향, 너비우선으로 각 노드를 비교하면서 가장 큰 공통된 패턴을 찾는다. 이미 찾아낸 패턴이 포함하는 작은 패턴은 비교 대상에서 제외하여 최대한 불필요한 계산을 줄여서 실행시간을 줄이도록 알고리즘을 설계하였다[6].

입력으로 두 개의 핵심구문트리를 받아서 클론 패턴 클래스의 집합을 내주는 이 알고리즘은 크게 두 단계로 나뉜다. 처음은 핵심구문트리에서 클론 패턴을 찾는 패턴 구축(Algorithm 1 : Pattern Building)이고, 다음은 동일한 모양을 가진 클론 패턴들을 병합해서 클론 패턴 클래스를 생성하는 패턴 클러스터링(Algorithm 2 : Pattern Clustering)이다.

우선 첫 번째 알고리즘인 패턴 구축에서는 특별한 2차원 배열인 작업 테이블(work table)을 사용한다. 작업 테이블에는 비교하는 노드 쌍의 방문 여부를 표시하는데, 이미 방문한 노드들의 중복 비교를 피하기 위해 비교가 끝난 두 트리의 노드 쌍들을 배열에 기록한다. 따라서 초기값이 'true'인 작업 테이블의 노드는 비교가 끝나면 'false'로 값이 바뀐다.

클론 패턴은 루트 노드부터 비교하는 하향식 너비우선 검색으로 트리의 노드들을 비교해 구축한다. 핵심구문트리의 두 노드가 동일하면 트리의 하위트리 노드는 비교를 계속하고, 클론 패턴의 노드에는 (핵심구문트리의 해당 노드 각각의 위치정보를) 하나의 쌍으로 묶은 꼬리표를 달아 클론 패턴을 구축한다. 반대로 두 노드가 서로 다르면 해당 노드의 하위트리는 더 이상 비교하지 않고 멈춘다. 이때 클론 패턴의 해당 노드는 구멍(#)으로 표시한다. 이 과정은 두 핵심구문트리의 모든 부분트리 쌍들을 대상으로 하지만, 작업 테이블의 값이 'false'인 노드를 루트노드로 하는 부분트리의 쌍은 비교대상에서 제외한다. 이와 같이 두 트리에서 구멍이 있는 클론 패턴을 찾는 방법은 전통적으로 anti-unification이라고 하며, 오래 전 Plotkin [8]과 Reynolds [9]가 처음 제안하였다.

Algorithm 1. Pattern Building

```
1  function anti-unify (t1, t2 : tree, worklist : work-table)
2                       where t1 = s1^l1(u1, … , um), m ≥ 0
3                         and t2 = s2^l2(v1, … , vn), n ≥ 0
4                       returns tree-pattern
5  type
6     work-table = array [L1][L2] of boolean
7                  where Li is the set of labels in ti, i=1 or 2
8     tree-pattern = map of S to P(P)
9                  where S is the set of function symbols of arity≥1
10                   and P is the set of patterns
11 begin
12    ∀l1∈L1, ∀l2∈L2, worklist[l1][l2] ← false; (* default is true *)
13    if s1 ≠ s2 then return #^(l1, l2)
14    else (* s1 = s2 and m = n in AST *)
15       if m = n = 0 then
16          return s1^(l1, l2) (* t1 are t2 are leaves *)
17       else (* m = n ≠ 0 *)
18          return s^(l1, l2)(anti-unify(u1 , v1 , worklist),⋯,
19                            anti-unify(um, vm, worklist))
20 end
```

Anti-unification을 구체화한 것이 *anti-unify* 함수로, 비교 대상인 두 핵심구문트리에서 가장 큰 공통 트리조각인 클론 패턴들을 내준다. 클론 패턴의 각 노드에는 비교하는 두 텍스트 프로그램의 위치정보를 하나의 쌍으로 묶어 꼬리표를 단다. 이때 클론 패턴의 특별한 노드인 구멍(#)에는 위치정보 외에 하위트리의 노드 개수를 나타내는 '구멍크기(hole mass)' 정보를 추가한다. 추가한 '구멍크기'는 기존 위치정보와 쌍으로 묶는다. 그림 3은 그림 1의 (b) 패턴에서 구멍인 '#'에 구멍크기 정보를 추가한 모양을 보여준다. '#'에서 기존의 꼬리표 l_9 , l_{19} 와 쌍으로 묶은 1과 3은 각각 하위트리의 노드 개수를 나타낸다.

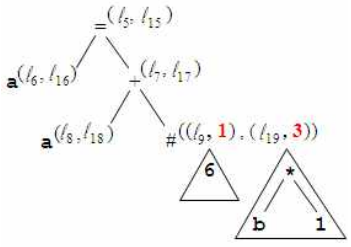


그림 3. 클론 패턴
('#'에 구멍크기 정보 추가)

Algorithm 2. Pattern Clustering

```
1  function clustering ( ps : set of patterns )
2                 returns set of pattern classes as a pattern-cluster
3  type
4     pattern-cluster = map of S to P(C)
5                  where S is the set of function symbols of arity ≥ 1
6                    and C is the set of pattern classes
7  var
8     new : boolean
9     p : pattern
10    c : pattern class
11    cs : set of pattern classes
12 begin
13    if ps ≠ ∅ then
14       begin
15          c ← pick one pattern out from ps and then
16                make it into a singleton pattern class ;
17          cs ← { c }
18       end
19    while ps ≠ ∅ do
20       p ← pick one pattern out from ps ;
21       new ← true ;
22       for each c in cs while new do
23          if p and c are the same shape then
24             begin
25                merge the labels of p into c ;
26                new ← false ;
27             end
28          if new then
29             begin
30                c ← make p into a singleton pattern class ;
31                cs ← cs ∪ { c } ;
32             end
33       return cs
34 end
```

두 번째 알고리즘인 패턴 클러스터링에서는 동일한 모양을 가진 클론 패턴들을 모아 클론 패턴 클래스로 합병한다. 함수 *clustering*은 다양한 모양의 클론 패턴들을 입력받아 동일한 모양의 클론 패턴들을 모은 클론 패턴 클래스들의 집합을 내준다. 그리고 클론 패턴 클래스의 각 노드에는 하나로 합병한 패턴들의 위치정보 (및 구멍크기) 리스트가 붙는다. 여기서 클론 패턴 클래스는 (구멍을 제외한) 노드의 총 개수, 구멍 노드의 총 개수, 최대 구멍 크기와 같은 정보로 각각의 클론 패턴 클래스를 표현한다.

그림 4는 그림 2의 (b)의 클론 패턴 클래스에서 구멍인 '#'에 구멍크기 정보를 추가한 모양을 보여준다. 이 클론 패턴 클래스의 구멍에는 기존의 세 개 위치정보 (l_9, l_{19}, l_{25})와 세 개의 구멍크기를 각각 쌍으로 묶어 리스트 형태로 꼬리

표를 단다. 이 클론 패턴 클래스는 (구멍을 제외한) 노드의 개수가 4개이고, 구멍 노드의 개수는 1개이고, 그 구멍 노드에서 가장 큰 구멍크기는 4 이다.

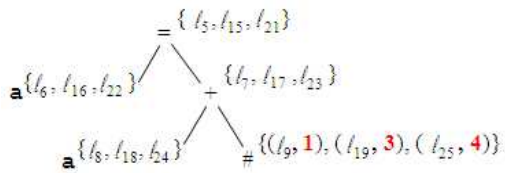


그림 4. 클론 패턴 클래스
('#'에 구멍크기 정보 추가)

4. 실험 및 평가

이 절에서는 제 3절의 알고리즘으로 구현한 코드클론 탐지기가 높은 정확성(precision)과 탐지성(recall)을 가지고 있는지 실험을 통해 비교 평가한다.

4.1 구현 및 환경설정

제 3절의 알고리즘을 기반으로 Objective Caml 3.09와 Python 2.5.1을 사용해서 프로그램 트리 패턴을 자동으로 찾아내는 도구를 구현하였으며, 구문 분석은 Joust 0.8(Objective Caml로 작성된 Java 파서)[2] 이용하였다.

이 도구는 사용자가 요구하는 제한조건을 만족하는 클론 패턴들을 모아서 클론 패턴 클러스터를 구축한다. 사용자 맞춤형 클론 패턴 클러스터를 구축하기 위해 우리 도구에서 요구하는 세 개의 입력 파라미터는 다음과 같다.

- `MinTN`(Minimum Token Number) : 크기가 너무 작아서 클론으로서의 의미가 없는 패턴들을 배제하기 위해 허용하는 최소 토큰의 개수
- `MaxHN`(Maximum Hole Number) : 하나의 패턴 클러스터에서 최대 수용 가능한 구멍의 개수 (비교하는 두 프로그램 코드에서 서로 다른 부분의 양을 표시)
- `MaxHM`(Maximum Hole Mass) : 패턴 클러스터의 각 구멍이 허용하는 최대 구멍크기 (구멍이 너무 크면 클론이라고 보기 어려움)

따라서 우리의 도구로 모은 클론 패턴 클래스들은 모두 `MinTN`보다 토큰개수가 더 많고, `MaxHN` 보다는 hole개수가 더 작고, 모든 hole은 `MaxHM` 보다 hole 크기가 작다. 우리 도구에서 기본으로 세팅한 입력 파라미터 값은 `MinTN=20`, `MaxHN=6`, `MaxHM=5` 이다. 세 개의 파라미터의 기본값 중에서 `MinTN=20`은 평균적으로 3줄 이하의 코드에 해당하는 크기지만, 대상언어나 요약방법에 따라 작은 크기를 가진 코드도 클론으로 의미가 있을 수 있기 때문에 비교적 작은 값인 20을 `MinTN`의 기본값으로 정했다[5]. `MaxHN`와 `MaxHM`의 기본값은 코드클론을 탐지하는 다른 도구(CloneDR[1])에서 이미 사용하고 있는 값들을 참고하여 기준으로 정하였다.

우리 도구를 평가하기 위해 이미 코드클론 표본 집합체가 공개되어 있는 네 개의 Java 응용 프로그램들의 소스코드를 비교 실험 대상으로 정했다. 대상 프로그램의 크기는 약 0.7-8.3MB이며 대략 1만4천-2십만 줄이다. 표 1은 각 응용 프로그램들의 크기(MB), 파일 수, 줄 수를 보여준다.

표 1. 비교대상인 응용 프로그램들

응용 프로그램	크기 (MB)	파일 수	줄 수
`netbeans-javadoc`	0.69	97	14,301
`eclipse-ant`	1.35	149	29,880
`eclipse-jdtcore`	6.37	687	135,675
`j2sdk1.4.0-javax-swing`	8.32	533	202,943

2) http://www.cs.cmu.edu/~ecc/joust.tar.gz

4.2 정확성

코드클론 표본 집합체 자동 생성기로 찾아낸 코드클론들이 정확한지 알아내기 위해 먼저 도구에서 찾은 클론들을 육안으로 하나하나 검사하였다. 그리고 오탐이 없는 것으로 알려져 있는 트리기반 상용도구 CloneDR[1]의 결과와 비교하여 우리 도구가 찾아내지 못하거나 틀린 코드클론이 있는지 모두 확인하였다.

CloneDR은 5개의 입력 파라미터를 사용한다. 이 입력 파라미터들은 우리 도구에서 사용하는 파라미터와 비교했을 때 의미나 항목이 일부분만 일치한다. 따라서 여러 번의 모의실험을 통해 두 도구가 거의 동일한 조건에서 실험할 수 있도록 파라미터들의 값을 조절하였다. CloneDR이 사용한 파라미터들의 값은 다음과 같다.

- similarity threshold = 95%
- maximum parameters = 6
- minimum clone mass = 6
- number of characters per node = 16
- initial starting depth = 2

그리고 CloneDR은 의미 있는 클론으로 보기엔 크기가 작은 3줄 이하의 코드클론은 무시한다. 따라서 동일한 조건으로 비교하기 위해 우리 도구에 최소 소스코드 줄 수를 표시해주는 새로운 파라미터인 MinLS(Minimum Line Scope)를 추가하여 4로 초기화하였다.

표 2는 우리 도구와 CloneDR에서 찾아낸 클론 클래스들의 개수이다. 탐지한 클론들을 육안으로 직접 확인할 결과, 우리 도구는 CloneDR과 마찬가지로 오탐이 없었다. 그리고 우리 도구에서 찾아낸 클론들은 상용 도구인 CloneDR에서 찾아낸 클론들을 100% 포함하면서 약 2.5-3배 더 많은 코드클론들(코드클론 클래스들)을 찾았다.

표 2. 우리 도구와 CloneDR에서 찾아낸 코드클론 클래스의 개수 비교

응용 프로그램	우리도구	CloneDR
netbeans-javadoc	177	66
eclipse-ant	247	122
eclipse-jdtcore	2,791	930
j2sdk1.4.0-javax-swing	1,517	529

두 도구는 서로 다른 환경에서 실행하였다. 우리의 코드클론 표본 자동 생성기는 Mac Pro Work station에서 Mac OX X Server 10.5.7, Xeon 2*2.8 Quad Core Processor, 4GB RAM에서 구현 및 실행하였지만, CloneDR은 Window에서만 사용가능한 제한된 라이선스를 실험용으로 제공받았기 때문에 Windows XP를 사용하는 Intel Core 2 Duo CPU 2.66GZ, 2GB RAM인 개인용 컴퓨터에서 실행하였다. 동일하지 않은 환경에서 실행하였으므로 실행시간을 비교하는 것이 큰 의미가 있다고 할 수는 없지만, 참조용으로 탐지시간을 재 보았다. 표 3은 표 2의 코드클론 클래스들을 탐지할 때 소요된 두 도구의 실행시간이다. 우리 도구는 CloneDR과 비교해서 약 2.5-5배 느리다.

표 3. 우리 도구와 CloneDR의 실행시간

응용 프로그램	실행시간	
	우리도구	CloneDR
netbeans-javadoc	0h 01m 40s	0h 00m 38s
eclipse-ant	0h 03m 56s	0h 01m 22s
eclipse-jdtcore	2h 40m 11s	0h 28m 39s
j2sdk1.4.0-javax-swing	5h 06m 09s	0h 57m 34s

4.3 탐지성

탐지성은 우리의 코드 클론 탐지도구가 코드 내의 클론을 빠짐없이 탐지하는가(미탐이 어느 정도인가)를 나타내준다. 따라서 우리 도구에서

탐지한 클론들을 이미 구축되어 있는 Bellon의 코드클론 표본 집합체[2]와 비교하여 확인하였다. 이 집합체는 5개의 널리 알려진 도구로 찾아낸 코드클론들의 결과를 참고로 하여 수작업으로 구축한 것이다.

표 4는 Bellon의 코드클론 표본 집합체(기존 표본 집합체)와 우리의 자동 생성기가 자동으로 생성해낸 클론 패턴 클래스들을 응용 프로그램별로 비교한 결과이다. 네 개의 응용 프로그램 중에서 프로그램의 크기(표본 집합체의 개수는 각각 1,345와 777 개)가 너무 커서 수작업으로 비교가 현실적으로 불가능한 두 개의 프로그램은 전체의 5%(67과 38개)만 살펴보았다. 비교대상 집합체 선정 방법은 단순하게 일련번호가 큰 5%를 선정하였다. 우리 도구는 MinTN=20, MaxHN=25, MaxHM=25, MinLS=4인 파라미터들의 값으로 실험하였다. 이 값들은 우리 도구가 Bellon의 코드클론 표본 집합체를 최대한 포함하는 파라미터들의 최소값이다.

비교결과 표 4와 같이 우리 도구에서 찾아낸 코드클론들은 Bellon의 코드클론 표본 집합체를 93-100%를 포함하였다. 결과적으로 우리 도구에서 찾은 클론들은 기존 Bellon의 표본 집합체를 거의 대부분 포함한다고 할 수 있다.

표 4. 우리 도구가 포함한 Bellon의 코드클론 표본 집합체의 개수

응용 프로그램	기존 표본 집합체	우리 자동 생성기
netbeans-javadoc	55	54 (98%)
eclipse-ant	30	30 (100%)
eclipse-jdtcore	67	62 (93%)
j2sdk1.4.0-javax-swing	38	38 (100%)

Bellon의 클론 표본 집합체에서 우리 도구가 탐지하는 못한 클론 표본은 총 6개이다. 표 5의 예제는 Bellon의 코드클론 표본 집합체에 있는 netbeans-javadoc에 존재하는 클론으로 우리 도구로는 찾아내지 못한 클론이다. 소스코드 내에 한줄 씩 다른 코드(밑줄 친 78, 80, 85줄)가 끼어있는데 우리 도구는 큰 덩어리가 아닌 여러 개의 작은 크기로 쪼개진 클론을 찾는다. 이와 같이 우리 도구는 코드 내의 작게 나눠진 클론들을 모두 탐지하지만 MinLS=4 일 때 표 5와 같은 큰 크기의 클론은 탐지하지 못했다.

표 5. 예제 : 우리도구에서 찾지 못하는 기존의 코드클론 표본 집합체

Reference Number : 579
File Name : DocumentationSettings.java

```
75  public ServiceType getServiceTypeOfDocument() {
76      JavadocSearchType.Handle st = (JavadocSearchType.Handle)
                                      getProperty( PROP_SEARCH );
77      JavadocSearchType t = null;
78      int x = 15;
79      if (st != null) {
80          x++;
81          t = (JavadocSearchType)st.getServiceType(); }
82      if (t == null) {
83          --x;
84          if (isWriteExternal()) {
85              x = x > 15 ? 15 : x;
86              return null; }
87      return (JavadocSearchType)Lookup.getDefault().
                lookup(org.netbeans.modules.javadoc.search.
                        Jdk12SearchType.class); }
88  return t; }
```

```
192 public ServiceType getSearchEngine() {
193     JavadocSearchType.Handle searchType =
            (JavadocSearchType.Handle)getProperty(PROP_SEARCH);
194     JavadocSearchType type = null;
195     if (searchType != null) {
196         type=(JavadocSearchType)searchType.getServiceType();}
197     if (type == null) {
198         if (isWriteExternal()) {
199             return null; }
200         return (JavadocSearchType)Lookup.getDefault().
                    lookup(org.netbeans.modules.javadoc.search.
                            Jdk12SearchType.class); }
201     return type;  }
```

그러나 우리의 도구는 기존 표본 집합체에 없는 클론들을 많이 찾아낸다. 표 6은 우리 도구에서 탐지한 코드클론 클래스의 개수로, Bellon의 코드클론 표본 집합체를 구축할 때 사용한 도구들의 파라미터들과 비슷하게 입력 파라미터 값(MinTN=20, MaxHN=15, MaxHM=5 , MinLS=6들을 주고 얻어낸 결과이다. 표 6의 통계치는 우

리의 도구의 경우 코드 클론 클래스의 개수인 반면, Bellon의 코드클론 표본 집합체의 경우 코드 클론 쌍의 개수이므로 수치상으로 직접적인 비교는 공평하지 않다. 왜냐하면 n개의 클론으로 구성된 클론 클래스 1개는 총 ${}_nC_2$개의 서로 다른 클론 쌍과 대응할 수 있기 때문이다. 예를 들어, Bellon의 코드클론 표본 집합체에서는 j2sdk 1.4.0-javax-swing에서 JList.java(2322-2329줄)의 클론으로 총 4쌍(Reference Number : 2699, 5533, 7322, 7457)을 제시하고 있는데, 우리 도구에서 찾아낸 해당 클론 클래스는 이 5개의 클론을 포함해서 총 37개의 클론으로 구성되어 있다.

표 6. 우리 도구에서 찾아낸 클론 클래스의 개수

응용 프로그램	Bellon의 표본 집합체	우리 자동 생성기
	클론 쌍의 개수	클론 클래스의 개수
netbeans-javadoc	55	113
eclipse-ant	30	146
eclipse-jdtcore	1,345	2,618
j2sdk1.4.0-javax-swing	777	2,196

5. 결론 및 향후 연구방향

본 논문에서는 클론 패턴을 단위로 정확하면서도 빠짐없이 코드클론을 찾아내는 방법을 제안하였고, 이 방법을 이용해 코드클론 표본 집합체를 자동 생성하는 도구를 구현하였다. 이 도구는 비교해서 클론 패턴으로 찾은 프로그램 트리의 노드들을 중복 비교하지 않으면서 동일한 모양의 클론 패턴들을 병합해 코드클론 표본 집합체를 생성한다. 자동 생성한 코드클론 표본 집합체는 오탐이 없으며, 트리기반 상용 클론 탐지도구 CloneDR과 비교했을 때 CloneDR이 탐지한 클론을 모두 포함할 뿐만 아니라 2-3배 더 많은 클론들을 찾았다. 그리고 수동으로 구축한 Bellon의 코드클론 표본 집합체를 거의 대부분 포함(93-100%)하면서 더 많은 클론을 자동으로 구축할 수 있음을 보였다.

향후연구에서는 클론 클래스를 탐지하는 우리 도구가 기존 코드클론 표본 집합체의 클론 쌍보다 얼마나 더 많은 클론을 찾는지 조사할 예정이다. 우리 도구의 장점은 파서만 있다면 어떤 언어에도 적용해 자동으로 코드클론 표본 집합체를 생성할 수 있다는 점이다. 따라서 기존의 Java파서 외에 다른 언어들의 파서를 우리 도구에 붙이고 다양한 응용 프로그램들을 대상으로 실험해서 알고리즘과 도구의 성능을 확인할 것이다.

신뢰할만한 코드클론 표본 집합체를 자동으로 생성함으로써 앞으로 클론탐지도구들의 성능 비교평가에 유용하게 쓰일 수 있을 것이라 기대한다.

참 고 문 헌

[1] Baxter, I., Yahin, A., Moura, L., and Anna, M., “Clone Detection Using Abstract Syntax Trees”, In ICSM, pp. 368-377, 1998.

[2] Bellon, S., Koschke, R., Krinke,J., and Merlo,E. “Comparison and Evaluation of Clone Detection Tools”, IEEE TSE, Vol.33(9), pp. 577-591, 2007.

[3] Burd, E. and Bailey, J., “Evaluating CLone Detection Tools for Use during Preventative Maintenance”, In ICSM, pp.35-43, Montreal, Canada, October 2002.

[4] Giesecke. Simon, "Generic modelling of code clones", In Proceeding of Duplication, Redundancy, and Similarity in Software, ISSn 16824405, Dagstuhl, Germany, July 2006.

[5] Kapser, C. and Godfrey, M., "Supporting the Analysis of Clones in Software Systems: A Case Study", Journal of

Software Maintenance and Evolution: Research and Practice, Vol. 18(2): 61-82, March 2006.
[6] Lee, Hyo-Sub and Doh, Kyoug-Goo, "Tree-pattern- based duplicate code detection", In DSMM, pp.7-12, 2009.
[7] Liu, C., Chen, C., Han, J., and Yu, P., "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis, In KDD, pp.872-881, 2006.
[8] Plotkin, G.D., "A note on inductive generalization", Machine Intelligence, 5:153-163, 1970.
[9] Reynolds, J.C., "Transformational systems and the algebraic structure of atomic formulas", Machine Intelligence, 5:135-151, 1970.
[10] Roy, C.K. and Cordy, J.R., "A survey on software clone detection research", Queen's School of computing TR 2007-541, 115 pp, 2007.
[11] Roy, C.K., Cordy, J.R., and Koschke, R., "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach", Journal of Software Maintenance and Evolution, Vol.21, iss. 2, pp. 143-169, Mar/Apr. 2009.

저 자 소 개

이 효 섭

1997년 대진대학교 컴퓨터공학과 학사
2000년 광운대학교 정보통신공학과 석사
2000년~현재 : 한양대학교 컴퓨터공학과 박사과정

<주관심분야 : 프로그래밍언어, 프로그램 분석 및 검증, 소프트웨어 보안>

도 경 구

1980년 한양대학교 산업공학과(학사)
1987년 미국 아이오와주립대학교 전산학과(석사)
1992년 미국 캔사스주립대학교 전산학과(박사)
1993.4~1995.8, 일본 Aizu 대학 교수
2000.9~2001.12, 스마트카드연구소 대표이사
2005.3~2006.2, 미국 캘리포니아대학 데이비스캠퍼스 방문교수
1995.9~현재, 한양대학교 안산캠퍼스 컴퓨터공학과 교수

<주관심 분야> 프로그래밍언어, 프로그램 분석, SW 보안, 프로그래밍언어 의미표기법