KIPS Trans. Softw. and Data Eng.
Vol.12, No.5 pp.199~206
ISSN: 2287-5905 (Print), ISSN: 2734-0503 (Online)

https://doi.org/10.3745/KTSDE.2023.12.5.199

Quality Visualization of Quality Metric Indicators based on Table Normalization of Static Code Building Information

Chansol Park† · So Young Moon†† · R. Young Chul Kim†††

ABSTRACT

The current software becomes the huge size of source codes. Therefore it is increasing the importance and necessity of static analysis for high-quality product. With static analysis of the code, it needs to identify the defect and complexity of the code. Through visualizing these problems, we make it guild for developers and stakeholders to understand these problems in the source codes. Our previous visualization research focused only on the process of storing information of the results of static analysis into the Database tables, querying the calculations for quality indicators (CK Metrics, Coupling, Number of function calls, Bad-smell), and then finally visualizing the extracted information. This approach has some limitations in that it takes a lot of time and space to analyze a code using information extracted from it through static analysis. That is since the tables are not normalized, it may occur to spend space and time when the tables(classes, functions, attributes, Etc.) are joined to extract information inside the code. To solve these problems, we propose a regularized design of the database tables, an extraction mechanism for quality metric indicators inside the code, and then a visualization with the extracted quality indicators on the code. Through this mechanism, we expect that the code visualization process will be optimized and that developers will be able to guide the modules that need refactoring. In the future, we will conduct learning of some parts of this process.

Keywords : Software Engineering, Database Normalization, Code Metrics, Static Analysis, Software Visualization

정적 코드 내부 정보의 테이블 정규화를 통한 품질 메트릭 지표들의 가시화를 위한 추출 메커니즘

박 찬 솔† · 문 소 영†† · 김 영 철†††

요 약

현대 소프트웨어의 규모는 커지고 있다. 이에 따라 고품질 코드를 위한 정적 분석의 중요성이 커지고 있다. 코드에 대한 정적 분석을 통해 결함과 복잡도를 식별하는 것이 필요하다. 이를 가시화하여 개발자 및 이해 관계자가 알기 쉽게 가이드도 필요하다. 기존 코드 가시화 연구들은 정적 분석의 코드 내부 정보들을 데이터베이스 테이블에 저장하여 및 품질 지표(CK Metrics, Coupling, Number of function calls, Bed smell)에 대한 계산을 질의어화 하고 추출된 정보를 가시화하는 과정을 구현하는 것에만 초점을 두었다. 이러한 연구들은 방대한 코드로부터 추출한 정보를 이용하여 코드를 분석할 때 많은 시간과 자원이 소모된다는 한계점이 있다. 또한 각 코드 내 정보 테이블들이 정규화되지 않았기 때문에 코드 내부의 정보(클래스, 함수, 속성 등)들에 대한 테이블 조인 연산 시 메모리 공간과 시간 소비가 발생할 수 있다. 이러한 문제들을 해결하기 위해, 데이터베이스 테이블의 정규화된 설계와 이를 통한 코드 내부의 품질 메트릭 지표에 대한 추출 및 가시화 메커니즘 제안한다. 이러한 메커니즘을 통해 코드 가시화 공정이 최적화되고, 개발자가 리팩토링해야 할 모듈을 가이드 할 수 있을 것으로 기대한다. 앞으로는 부분 학습도 시도할 예정이다.

키워드 : 소프트웨어 공학, 데이터베이스 정규화, 코드 메트릭, 정적 분석, 소프트웨어 가시화

1. 서 론

현재 소프트웨어는 다양한 분야에서 다양한 종류로 개발되고 있다[1]. 이러한 소프트웨어들의 품질을 측정하고 개선하여 고품질화하는 것은 매우 중요하다. 이를 위해서는 코드에 대한 정적/동적 분석을 통해 코드의 결함을 식별하고, 데이터 및 제어 흐름 복잡도에 대해 계산해야 한다. 또한 분석 결과를 소프트웨어 개발 프로젝트의 개발자 및 이해 관계자가 알기 쉽도록 가시화해야 한다.

※ 이 논문은 2022 년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. 2021R1I1A3050407, No. 2021R1I1A1A01044060).
※ 이 논문은 2022년 한국정보처리학회 ACK 2022의 “코드 내부 정보의 정규화 기반 효율적인 코드 정적 분석 및 가시화”의 제목으로 발표된 논문을 확장한 것임.
† 준 회 원 : 홍익대학교 소프트웨어융합학과 석사과정
†† 정 회 원 : 홍익대학교 소프트웨어융합학과 초빙교수
††† 정 회 원 : 홍익대학교 소프트웨어융합학과 정교수
Manuscript Received : December 20, 2022
First Revision : February 9, 2023
Accepted : February 28 2023
* Corresponding Author : R. Young Chul Kim(bob@hongik.ac.kr)

하지만 기존의 우리의 도구는 소프트웨어의 규모가 커지고 소프트웨어의 개수가 많아짐에 따라 분석을 통해 추출한 데이터의 규모도 커져 적용하기에 어려워졌다[2]. 따라서 본 논문은 ACK 2022 우수논문을 확장하여, 코드 내부의 정보를 추출하여 저장하는 데이터베이스의 테이블 구조를 정규화하여 코드에 대해 효율적인 정적 분석 및 가시화 기법을 제안한다[3]. 이 설계를 통해 코드 가시화 공정이 최적화되고, 개발자가 리팩토링해야 할 모듈을 가이드 하기를 기대한다.

본 논문의 2절에서는 관련 연구로서 소프트웨어 역공학을 통한 코드 정적 분석과 기존의 코드 정적 분석을 통한 가시화 툴 체인을 소개한다. 3절에서는 코드 내부 정보의 테이블 정규화를 통한 품질 메트릭 지표들의 가시화를 위한 추출 메커니즘을 설명한다. 4절에서는 해당 메커니즘을 JAVA로 작성된 코드에 적용한 사례를 소개하고, 5절에서 결론 및 추후 연구를 제시하며 마무리한다.

2. 관련 연구

2.1 코드 정적 분석과 소프트웨어 역공학

코드 정적 분석은 코드를 실제 실행하지 않고 코드를 분석하는 기법이다[4]. 코드 정적 분석은 소스 코드 혹은 소스 코드의 컴파일 과정에서 산출되는 Abstract Syntax Tree (AST) 혹은 목적 파일 등의 분석을 통해 코드로부터 내부의 정보를 추출한 후 가공하여 복잡도를 계산하거나 결함을 검출하는 과정을 일컫는다.

소프트웨어의 개발 공정은 Fig. 1과 같이 일반적으로 요구사항 분석, 설계, 구현, 테스트, 배포 총 5개의 단계로 나뉜다. 역공학은 소프트웨어의 개발 공정의 역순으로 하위 단계의 산출물로부터 상위 단계의 산출물을 복원하는 작업이다[5]. 정적 분석을 통해 소스 코드로부터 정보를 추출한다. 이를 통해 설계 단계의 산출물인 설계 도면을 복원할 수 있다.

2.2 기존의 코드 정적 분석을 통한 가시화 툴 체인

본 연구실에서는 다양한 방식으로 정적 분석을 통한 코드 설계 복원 및 복잡도 가시화 연구가 진행 중이다[6,7,8,9]. 우선 코드에 대한 정적 분석을 통해 내부의 정보를 추출한다. 이를 통해 CK Metrics, Coupling, Function call의 횟수, Bed smell 등의 복잡도를 계산하고, 소프트웨어 설계 도면을 복원한다. 이 기법의 연구는 Open Source 기반 툴 체인화를 통한 코드 정적 분석 연구이다[6].

해당 연구에서 사용한 툴 체인은 Fig. 2와 같이 Source Navigator 도구를 통해 소스 코드를 분석하여 추출된 Source Navigator Database(SNDB)의 파일 시스템 구조를 기반으로 테이블을 생성한 후 JAVA Parser 도구를 통해 SNDB_BY 테이블을 보충하는 구조이다.

분석 과정에서 SNDB 파일 구조 그대로 데이터베이스 테이블들을 생성하여 Fig. 3과 같이 테이블을 통해 코드 내부의 요소 간의 종속성을 표현하기 어렵다. 또한 각 테이블의 속성이 파일에 저장된 정보를 그대로 반영하여 데이터베이스 테이블이 정규화하지 않은 문제점이 있다. 이러한 한계로 인해 Table 1과 같이 복잡도를 추출하기 위해 복잡한 조건의 Join 구문을 사용해야 하며, 두 테이블을 Join 하더라도 더미 데이터와 같은 잘못된 데이터가 조회되는 데이터베이스 이상 현

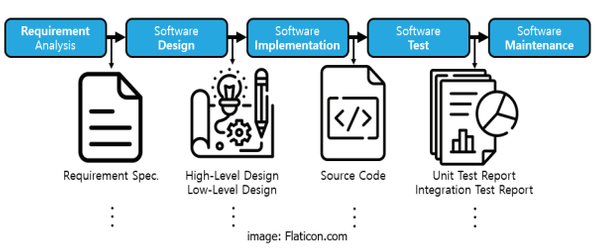

Fig. 1. Software Development Process and Byproducts

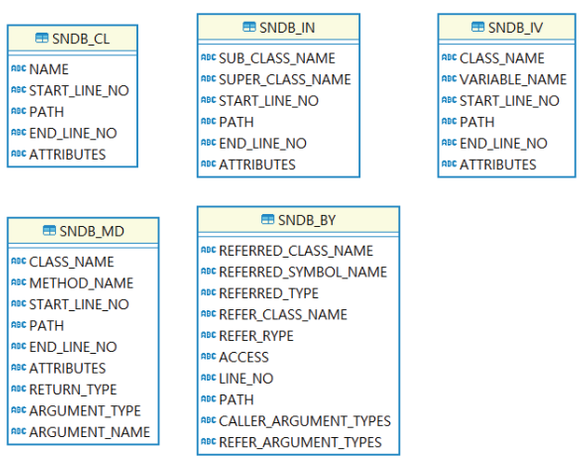

Fig. 2. Previous Database Table Design

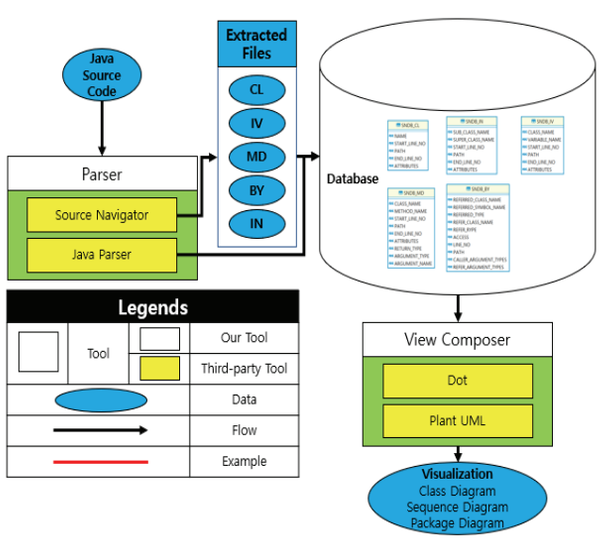

Fig. 3. First Previous Tool-chain Structure for Code Static Analysis

상이 발생할 가능성이 있다는 심각한 문제점이 존재한다.

두 번째로 설명할 연구는 역공학을 통한 UML 설계 추출 자동화 연구이다[7]. 해당 연구에서 사용한 툴 체인은 Fig. 4와 같이 JAVA Parser 라이브러리를 통해 코드를 분석하여 추출한 정보를 따로 데이터베이스 구조에 저장하지 않고 정적 분석 및 가시화를 진행하는 구조이다. 이러한 방법의 한계는 코드로부터 추출 및 분석한 모든 정보를 메모리에 저장하므로 대규모 코드에 적용하기 어렵다. 또한, 분석한 정보를 다이어그램 및 그래프 외의 방법으로는 저장하지 않으므로 지속적인 분석을 통한 시계열 데이터를 형성하는 데에도 어려움이 있다.

Table 1. Example Query to Extract Complexity from Previous DB

```
select R.class, R.symbol_name, R.ref_class, R.ref_symbol,
R,filename, R.ref_type, R.position
from RefersTo as R LEFT JOIN
MethodImplementation as M
on R.ref_symbol = M.name
where R.ref_type = 'ud' and R.class <> '#'
and M.name IS NULL and R.ref_argument_types = ''
and R.access <> 'p' order by R.class, R.symbol_name,
R.position;
```

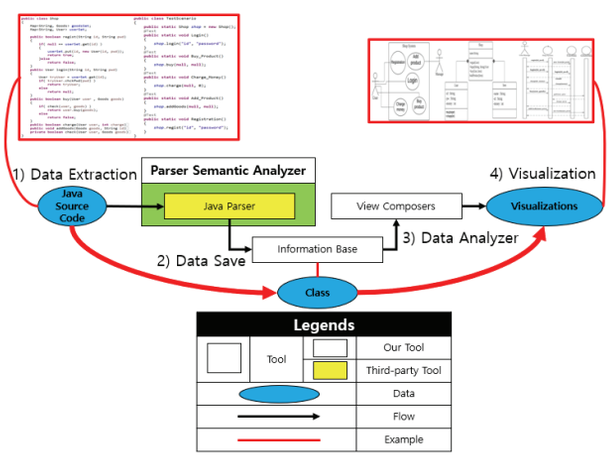

Fig. 4. Second Previous Tool-chain Structure for Code Static Analysis

3. 코드 내부 정보의 테이블 정규화를 통한 품질 메트릭 지표들의 가시화를 위한 추출 메커니즘

현재 제안하는 코드 정적 분석 툴 체인의 구조는 Fig. 5와 같다. 툴 체인은 코드 분석 도구, 데이터베이스, 코드 복잡도 분석 도구, 가시화 도구로 나누어져 있다.

-코드 분석 도구는 분석 대상 코드가 작성된 언어에 적합한 정적 분석 도구를 이용하여 코드로부터 정보를 추출한다. 이후 코드의 클래스, 함수, 속성과 같은 기본적인 정보와 클래스 간의 관계, 함수 간의 호출 관계 등과 같이 코드 요소 간 관계를 데이터베이스에 삽입하는 도구이다. JAVA를 분석하기 위해 JAVA parser 라이브러리를 이용했다. 또한 Solidity를 분석하기 위해 JAVA AST Parser for Solidity(JACS)를 개발했다[10]. 해당 도구는 Solidity의 컴파일러인 Solc 도구를 통해 분석 대상의 Solidity 프로그램의 AST를 파일 형태로 추출한다. 이후 해당 AST 파일을 읽어 JAVA 언어의 객체 구조에 저장하여, JAVA를 통해 Solidity로 작성된 프로그램을 정적 분석할 수 있도록 해주는 도구이다. 이외에도 C++, C# 그리고 Python과 같은 객체지향 언어도 정적 분석 도구를 구현하여 적용할 수 있다.

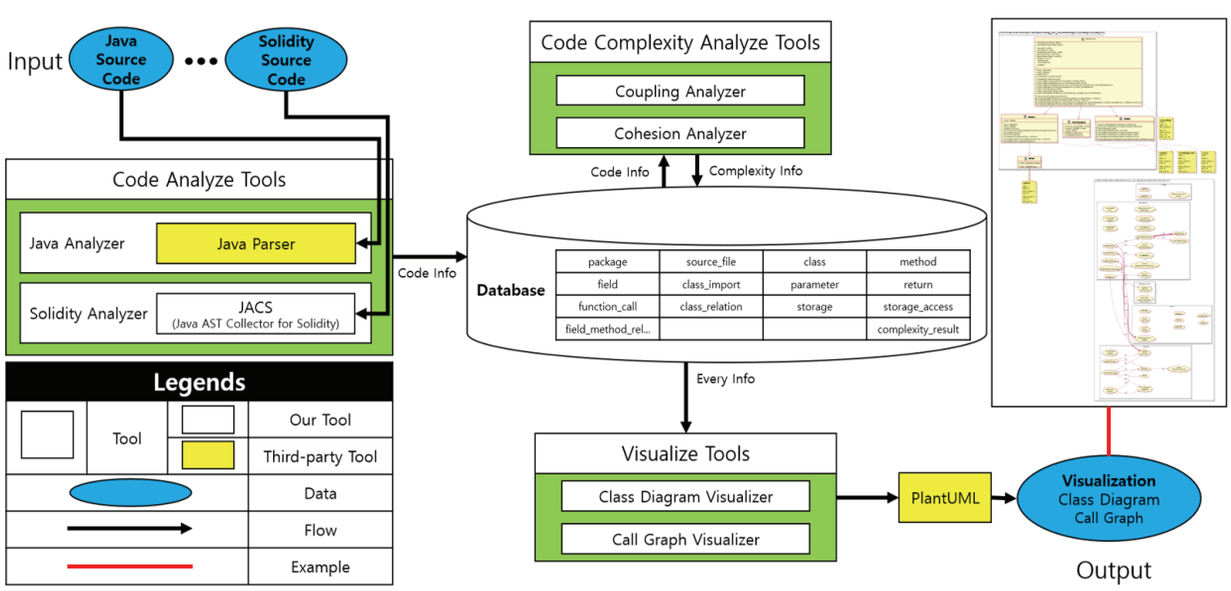

Fig. 5. Improved Tool-chain Structure for Code Visualization

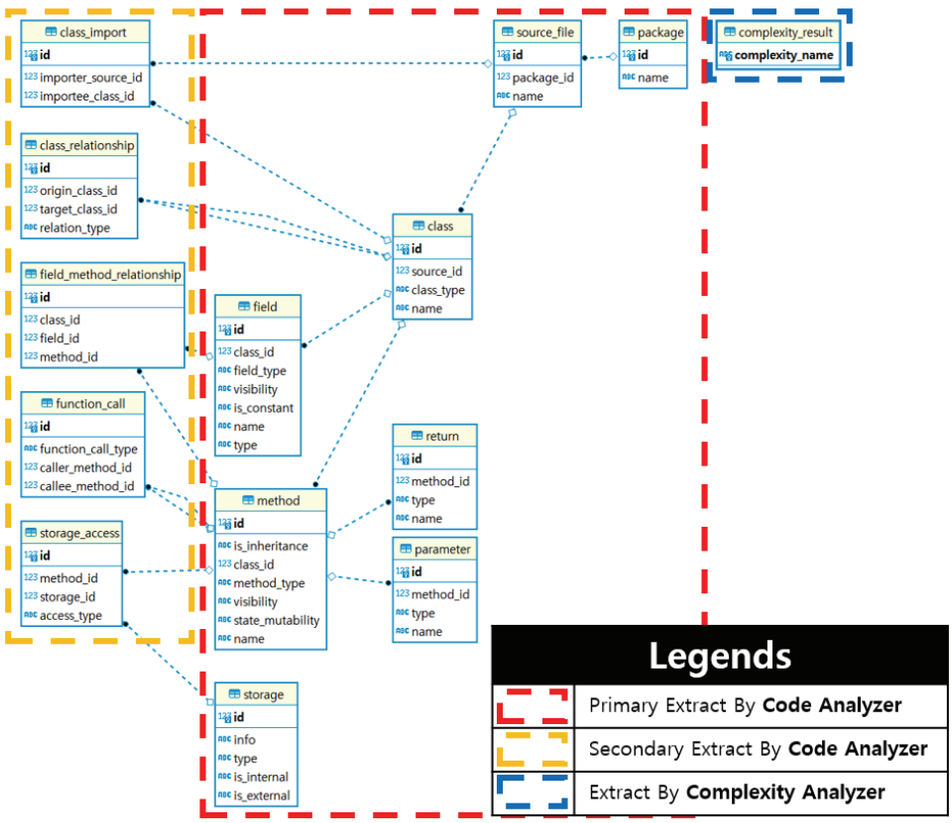

Fig. 6. Normalized Table Design of Each Code Information

-코드 복잡도 분석 도구는 데이터베이스로부터 코드 정보를 읽어와 질의를 통해 복잡도(결함도, 응집도)를 계산한다. 이렇게 계산된 복잡도를 다시 데이터베이스에 복잡도 테이블에 추가하는 도구이다.

-가시화 도구는 추출 및 분석되어 데이터베이스에 저장된 정보를 통해 PlantUML 스크립트를 작성하는 도구이다. 이 과정에서 작성된 스크립트는 PlantUML 도구를 통해 다이어그램과 그래프로 가시화된다. 현재 Class Diagram과 Call Graph 가시화 도구가 구현되었다.

개선된 데이터베이스의 테이블 구조 설계는 Fig. 6과 같이 12개의 코드 정보 테이블들과 1개의 복잡도 테이블로 이루어져 있다. 코드 분석 도구를 통해 코드 정보 테이블에서 패키지 레벨, 소스 코드 레벨, 클래스 레벨 그리고 클래스의 함수와 속성의 테이블에 정보가 저장되며 코드가 접근하는 데이터베이스와 파일 시스템과 같은 저장소 정보도 저장된다. 이때 함수의 경우 수 개의 매개변수와 언어에 따라 수 개의 반환 값을 가질 수 있으므로 이 둘의 정보를 함수 테이블로부터 각각의 테이블로 분리하여 저장한다. 이후 소스 코드 레벨과 클래스 레벨 간의 Import, 클래스와 클래스 간의 관계, 함수와 함수 간의 Function Call, 같은 클래스 내에서의 속성과 함수 간의 관계, 저장소와 함수 간의 Storage Access 정보들을 분석하여 각각의 테이블에 저장된다.

데이터베이스의 각 테이블에 포함되는 데이터는 고유 ID를 Primary Key로 갖는다. 이후 개체 간 가질 수 있는 모든 관계에 대해 ID를 통해 연결한다. 따라서 기존 연구에서의 방식인 개체의 이름을 통해서만 분석하는 방법의 경우 같은 이름을 가진 개체를 구분하는 것이 어려웠지만, 개선된 데이터베이스 구조를 이용하면, 이름을 포함한 모든 속성이 같은 개체라도 ID를 통해 구분할 수 있다.

현재 툴 체인을 통해 개발할 수 있는 복잡도는 클래스 간의 복잡도를 측정하는 Coupling Between Object(CBO), CBO′, Response For a Class(RFC), Message Passing Coupling(MPC), Data Abstraction Coupling(DAC), DAC′와 클래스 간 응집도를 측정하는 Lack Of Cohesion in Methods$_1$(LCOM$_1$)와 LCOM$_2$가 있다[11,12,13,14].

CBO는 클래스 간의 연결을 통해 클래스 간의 결합도를 측정하는 지표로서, 해당 클래스가 연결된 다른 클래스의 개수를 측정한다. CBO′는 CBO와 같이 클래스 간 결합도를 나타내는 지표이지만 상속 기반의 연결을 포함하지 않는다는 차이점이 있다. Table 2와 Table 3은 각각 한 클래스에 대한 CBO와 CBO′를 계산하는 코드이다.

RFC는 클래스의 객체에 의해 수신된 메시지에 응답하여 잠재적으로 실행될 수 있는 함수를 통해 클래스와 다른 클래스가 결합도를 나타내는 지표이다. 클래스에 대한 RFC 지표

Table 2. CBO Extraction Method

```
HashSet<Integer> relation = new HashSet<Integer>();
ResultSet targetRS = stat.executeQuery("SELECT * “
  + "FROM class_relationship “
  + "WHERE origin_class_id = "+i+";");
while(targetRS.next()) {
  relation.add(targetRS.getInt("target_class_id"));
}
targetRS.close();
ResultSet originRS = stat.executeQuery("SELECT * “
  + "FROM class_relationship “
  + "WHERE target_class_id = "+i+";");
while(originRS.next()) {
  relation.add(originRS.getInt("origin_class_id"));
}
originRS.close();
int cbo = relation.size();
```

Table 3. CBO′ Extraction Method

```
HashSet<Integer> relation = new HashSet<Integer>();
ResultSet targetRS = stat.executeQuery("SELECT * “
  + "FROM class_relationship “
  + "WHERE origin_class_id = "+i+" “
  + "AND relation_type != \'generation\' “
  + "AND relation_type != \'realization\';");
while(targetRS.next()) {
  relation.add(targetRS.getInt("target_class_id"));
}
targetRS.close();
ResultSet originRS = stat.executeQuery("SELECT * “
  + "FROM class_relationship “
  + "WHERE target_class_id = "+i+" “
  + "AND relation_type != \'generation\' “
  + "AND relation_type != \'realization\';");
while(originRS.next()) {
  relation.add(originRS.getInt("origin_class_id"));
}
originRS.close();
int cbo_prime = relation.size();
```

Table 4. RFC Extraction Method

```
HashSet<Integer> relation = new HashSet<Integer>();
ResultSet funRS = stat.executeQuery("SELECT * “
  + "FROM method “
  + "WHERE class_id = "+i+" "
  + "AND (method_type != \'event\' “
  + "AND method_type != \'modifier\' “
  + "AND method_type != \'default constructor\';");

while(funRS.next()) {
  relation.add(funRS.getInt("id"));
  Statement funCallStat = con.createStatement();
  ResultSet funCallRS =
    funCallStat.executeQuery("SELECT * "
  + "FROM function_call “
  + "WHERE caller_method_id = "+funRS.getInt("id")+" "
  + "AND function_call_type = \'function_call\';“);

  while(funCallRS.next()) {
    relation.add(funCallRS.getInt("callee_method_id"));
  }
  funCallRS.close();
  funCallStat.close();
}
funRS.close();
int rfc = relation.size();
```

Table 5. MPC Extraction Method

```
HashSet<Integer> funCall = new HashSet<Integer>();
ResultSet funRS = stat.executeQuery("SELECT * "
  + "FROM method "
  + "WHERE class_id = "+i+" "
  + "AND method_type != \'default constructor\';");

while(funRS.next()) {
  Statement funCallStat = con.createStatement();
  ResultSet funCallRS =
      funCallStat.executeQuery("SELECT * "
  + "FROM function_call "
  + "WHERE caller_method_id = "+funRS.getInt("id")+" "
  + "AND function_call_type = \'function_call\';“);

  while(funCallRS.next()) {
    funCall.add(funCallRS.getInt("id"));
  }
  funCallRS.close();
  funCallStat.close();
}
funRS.close();
int mpc = funCall.size();
```

Table 6. DAC Extraction Method

```
Statement stat = con.createStatement();
Vector<String> classes = new Vector<String>();
ResultSet classeDataRS =
    stat.executeQuery("SELECT * FROM class;");

while(classeDataRS.next()) {
  classes.add(classeDataRS.getString("name"));
}

for(int i = 1; i < classes.size()+1; i ++) {
  HashSet<Integer> variables =
      new HashSet<Integer>();

  ResultSet stVarRS =
      stat.executeQuery("SELECT * “
        + "FROM field "
        + "WHERE class_id = "+i+";");

  while(stVarRS.next()) {
    for(String classeName : classes) {
      if(stVarRS.getString("type").contains(classeName)) {
        variables.add(stVarRS.getInt("id"));
      }
    }
  }
  stVarRS.close();
  dac.add(variables.size());
}
stat.close();
```

Table 7. DAC ´ Extraction Method

```
HashSet<Integer> variables =
     new HashSet<Integer>();

ResultSet statusVariableRS =
    stat.executeQuery("SELECT * FROM field;");

Statement classeStat = con.createStatement();

while(stVarRS.next()) {
    if(stVarRS.getString("type").contains(
        classeStat.executeQuery("SELECT * "
            + "FROM class "
            + "WHERE id = \'"+i+").getString("name"))
        ){
          variables.add(statusVariableRS.getInt("id"));
        }
}
statusVariableRS.close();
int dac_prime = variables.size();
```

Table 8. $LCOM_1$ Extraction Method

```
HashMap<Integer,HashSet<Integer>> relation =
      new HashMap<Integer,HashSet<Integer>>();
ResultSet funRS =
      stat.executeQuery("SELECT COUNT(*) "
            + "FROM method "
            + "WHERE class_id = "+i+";");
int numOfFun = funRS.getInt(0);
funRS.close();

int totalMPair = 0;
for(int j = 1; j < numOfFun; j++) {
   totalMPair += i;
}

ResultSet fmrRS = stat.executeQuery("SELECT * "
      + "FROM field_method_relationship "
      + "WHERE class_id = "+i+";");

while(fmrRS.next()) {
   int fid = fmrRS.getInt("field_id");
   int mid = fmrRS.getInt("method_id");
   if(!relation.containsKey(fid)) {
      relation.put(fid,new HashSet<Integer>());
   }
   relation.get(fid).add(mid);
}

int sharedMPair = 0;
for(int fid: relation.keySet()) {
   for(int j = 1; j < relation.get(fid).size(); j++) {
      sharedMPair += j;
   }
}
fmrRS.close();
int lcom1 = totalMPair - sharedMPair;
```

Table 9. $LCOM_2$ Extraction Method

```
HashMap<Integer,HashSet<Integer>> relation = new
      HashMap<Integer,HashSet<Integer>>();
ResultSet funRS =
      stat.executeQuery("SELECT COUNT(*) "
            + "FROM method "
            + "WHERE class_id = "+i+";");

int numOfFun = funRS.getInt(0);
funRS.close();

int totalMPair = 0;

for(int j = 1; j < numOfFun; j++) {
   totalMPair += i;
}

ResultSet fmrRS =
      stat.executeQuery("SELECT * "
            + "FROM field_method_relationship "
            + "WHERE class_id = "+i+";");

while(fmrRS.next()) {
   int fid = fmrRS.getInt("field_id");
   int mid = fmrRS.getInt("method_id");
   if(!relation.containsKey(fid)) {
      relation.put(fid,new HashSet<Integer>());
   }
   relation.get(fid).add(mid);
}

int sharedMPair = 0;
for(int fid: relation.keySet()) {
   for(int j = 1; j < relation.get(fid).size(); j++) {
      sharedMPair += j;
   }
}
fmrRS.close();
int result = totalMPair - (sharedMPair*2);
if(result < 0) {
   result = 0;
}
int lcom2 = result;
```

값은 클래스의 함수 집합 M과 함수에 의해 직접 혹은 간접적으로 호출되는 함수의 집합으로 구성되는 응답 집합에 포함되는 함수의 개수이다. Table 4는 한 클래스에 대한 RFC를 계산하는 함수이다.

MPC는 클래스가 호출하는 함수를 통해 다른 클래스들과의 결합도를 나타내는 지표이다. 클래스의 MPC 지표 값은 클래스의 함수 호출 개수이다. Table 5는 한 클래스에 대한 MPC를 계산하는 함수이다.

DAC는 속성으로서 결합 되는 클래스에 관한 결합도 지표로서 DAC는 클래스가 속성으로 다루는 다른 클래스의 수이며, DAC′는 DAC와 반대로 클래스를 속성으로써 다루는 다른 클래스의 수이다. Table 6과 Table 7은 검사 대상인 모든 클래스에 대한 DAC와 한 클래스에 대해 DAC′를 계산하는 함수이다.

LCOM은 클래스 내부의 함수 간의 응집도를 측정하는 지표이다. $LCOM_1$과 $LCOM_2$은 속성과 함수 쌍을 통해 함수 간의 응집도를 나타낸다. $LCOM_1$은 공통으로 사용하는 속성이 없는 함수 쌍의 개수이다. $LCOM_2$는 공통으로 사용하는 속성이 없는 함수 쌍의 개수에서 공통으로 사용하는 속성이 있는 함수 쌍의 개수를 뺀 값이다. 단, $LCOM_2$의 계산 결과가 음수인 경우 $LCOM_2$의 값을 0으로 설정한다. Table 8과 Table 9는 각각 하나의 클래스에 대한 $LCOM_1$과 $LCOM_2$를 계산하는 함수이다.

4. 사례 연구

개선된 소프트웨어 가시화 툴 체인은 객체지향 언어로 작성된 프로그램에 대해 범용적으로 적용할 수 있다. 본 논문에서는 그중에 JAVA 언어로 작성된 프로그램에 적용한다. JAVA 언어로 작성된 소스 코드 분석에 관한 적용 사례로 직

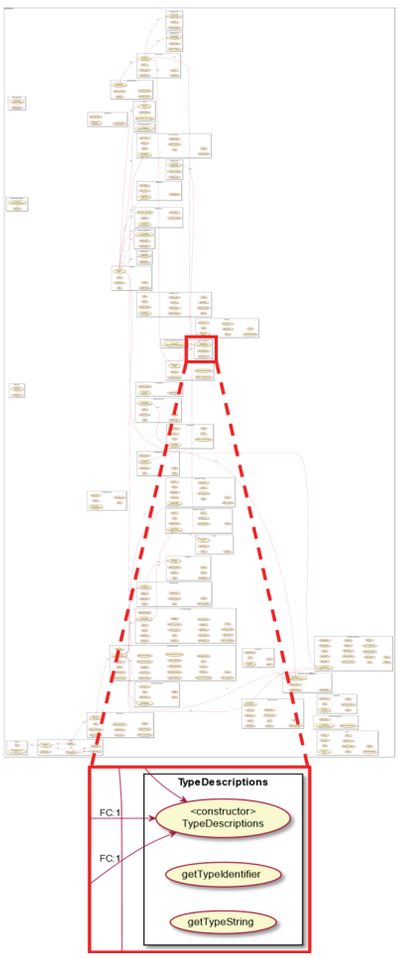

Fig. 7. Extracted Call Graph for JAVA Source Code

접 개발한 JACS 도구에 적용한다. JACS 도구는 Solidity 프로그램의 AST를 파싱하여 JAVA의 객체 구조에 저장하는 툴 체인이다. Fig. 7은 툴 체인을 통해 추출한 JAVA의 Call Graph이다. 함수 간의 호출 계 및 호출 횟수를 화살표의 두께 및 숫자로 표기했다. 이를 통해 호출이 잦은 함수를 강조하여 나타낼 수 있다.

Fig. 8은 Class Diagram과 복잡도이다. Class Diagram 설계와 클래스에 대해 계산된 복잡도를 가시화한다. 또 지정된 수치를 넘어 복잡하다고 판단된 클래스에 대해 강조하여 표시해 해당 부분을 리팩토링할 수 있도록 가이드한다.

이를 통해 예제 프로그램에 대해 설계를 추출하여 계산한 복잡도를 함께 표시했다. 연결된 선의 개수와 계산한 복잡도를 통해 프로그램의 복잡한 코드 부분을 알기 쉽게 나타낼 수 있다. 결국 기존 도구에 비해 처리 속도 향상과 분석 범위를 확장시켰다.

5. 결 론

본 논문에서는 정적 분석 툴 체인이 효율적으로 동작하여 기존보다 큰 규모의 코드에 대해 적은 자원을 통해 빠르고 정확한 정적 분석과 가시화를 할 수 있도록 데이터베이스 구조를 정규화했다. 이를 통해 코드로부터 Class Diagram과 Call Graph를 추출했다. 또한, 결합도와 응집도에 관련된 복잡도를 추출해 설계와 함께 나타내었다. 추후 코드로부터 추출해 데이터베이스 테이블에 저장한 저장공간 정보와 해당 저장공간에 대해 접근하는 함수와 접근 방식 정보를 통해, 역공학 기

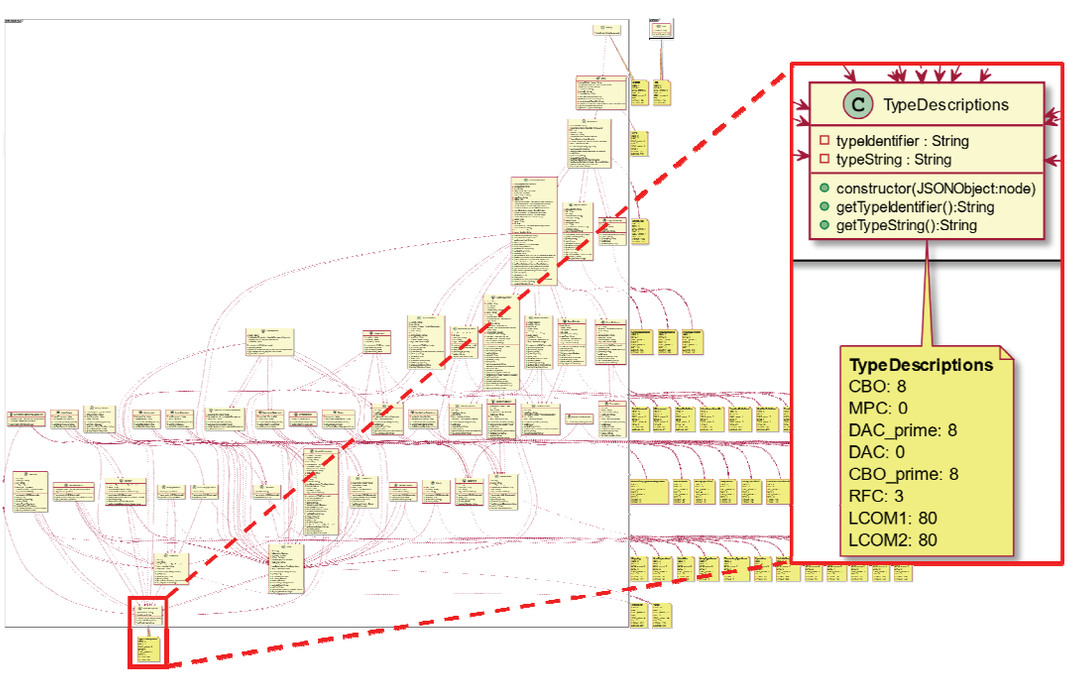

Fig. 8. Extracted Class Diagram for JAVA Source Code

반의 기능 점숫값에 대해 검증할 예정이다. 또한 정적 분석 툴 체인으로 추출한 코드의 품질에 대한 학습을 통해 나쁜 코드와 좋은 코드를 식별하는 모델에 관한 연구를 진행할 예정이다.

References

[1] J. H. Song and M. S. Ahn, "2021 SW Convergence Survey Results Report," Software Policy & Research Institute, 2022, report num.: 385001.

[2] M. H. Choi and I. S. Jeon, "2021 Software Industry Survey" Software Policy & Research Institute, 2022, report num.: 127005.

[3] C. S. Park, B. K. Jeon, and R. Y. C. Kim, "Effective code static analysis and visualization based on Normalization of internal code information," *Proceedings of the Korea Information Processing Society Conference*, Vol.29, No.2, pp.85-87, 2022.

[4] X. Rival and K. Yi, "Introduction to static analysis: An abstract interpretation perspective." Mit Press, 2020.

[5] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, Vol.7, No.1, pp.13-17, 1990.

[6] K. H. Kang, K. S. Lee, Y. S. Kim, Y. B. Park, H. S. Son, and R. Y. C. Kim, "A study on code static analysis with open source-based tool chainization," *KIISE Transactions on Computing Practices*, Vol.21, No.2, pp.148-153, 2015.

[7] S. J. Jung, J. H. Kim, W. Y. Lee, B. K. Park, H. S. Son, and R. Y. C. Kim, "Automatic UML design extraction with software visualization based on reverse engineering," *International Journal of Advanced Smart Convergence*, Vol.10, No.3, pp.89-96, 2021.

[8] H. E. Kwon and R. Y. C. Kim, "Extracting use case design mechanisms via programming based on reverse engineering," *International Journal of Applied Engineering Research*, Vol.10, No.90, pp.503-505, 2015.

[9] B. K. Park, K. H. Kang, H. S. Son, B. K. Jeon, and R. Y. C. Kim, "Code visualization for performance improvement of java code for controlling smart traffic system in the smart city," *Applied Sciences*, Vol.10, No.8, 2020.

[10] C. S. Park et al., "Design validation practices on design extraction of solidity's smart contract code based on reverse engineering," *Advanced and Applied Convergence Letters*, Vol.17, pp.26-31, 2021.

[11] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," i*n IEEE Transactions on Software Engineering*, Vol.20, No.6, pp.476-493, 1994.

[12] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," *SIGPLAN*, Vol.26, No.11, pp.197-211, 1991.

[13] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of Systems and Software*, Vol.23, No.2, pp.111-122, 1993.

[14] B. Henderson-Sellers, "Object-oriented metrics: Measures of complexity," Prentice-Hall, pp.142-147, 1996.

박 찬 솔

https://orcid.org/0009-0009-9462-1783
e-mail : c2193102@g.hongik.ac.kr
2022년 홍익대학교 소프트웨어융합학과 (학사)
2022년 ~ 현 재 홍익대학교 소프트웨어융합학과 석사과정
관심분야 : Software Visualization, Software Quality

문 소 영

https://orcid.org/0009-0000-9498-9689
e-mail : whit2@hongik.ac.kr
2007년 ~ 2012년 ㈜액트 대리
2018년 홍익대학교 전자전산공학과 (석・박사)
2017년 ~ 현 재 NIPA SP 연구원
2019년 ~ 현 재 홍익대학교 소프트웨어융합학과 초빙교수
관심분야 : Software Visualization, Requirement Engineering

김 영 철

https://orcid.org/0000-0002-2147-5713
e-mail : bob@hongik.ac.kr
2000년 IIT, Dept. of Computer Science (박사)
2000년 ~ 2001년 LG산전 중앙연구소 Embedded System 부장
2001년 ~ 현 재 홍익대학교 소프트웨어융합학과 정교수
관심분야 : Software Visualization, TMM, Requirement Engineering