JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 29 No. 10, pp. 219-228, October 2024
https://doi.org/10.9708/jksci.2024.29.10.219

A Library for Object-to-Graph Mapping with Annotations in Java

Ji-Woong Choi*

*Associate Professor, School of Computer Science and Engineering, Soongsil University, Seoul, Korea

[Abstract]

In this paper, we propose a method for constructing RDF knowledge graphs from objects in OOP. RML mapping has been the de-facto way of generating RDF graphs from heterogeneous data. However, the input to an RML mapping is limited to the data in files or databases. Our new RML implementation, designed to overcome the limit, has two differences compared to existing RML implementations. First, our implementation provides a new way to specify mapping rules in the form of special comments known as annotations in the source code. It is because existing works do not provide a means to refer to specific program elements to which the mapping rules will be applied. Second, our work provides mapping engine as a library, whereas the engines in existing studies runs in an independent process. Therefore, our mapping engine can be easily embedded in other applications to access in-memory objects to be mapped. In this system paper, we describe the proposed system in detail and present the results of RML test cases execution to confirm the usefulness of the system.

▸ **Key words**: RDF, Knoweldge graphs, RML, Declarative mapping, Java Annotations

[요 약]

본 논문에서는 OOP에서의 객체로부터 RDF 지식 그래프를 구축하는 방법을 제안한다. RML 매핑은 이기종 데이터로부터 RDF 그래프를 생성하고자 할 때 사용하는 사실상 표준 방법이다. 그러나 현재 RML 매핑에서의 입력은 파일이나 데이터베이스에 저장된 데이터로 제한된다. 이를 극복하기 위해, 제안된 RML 구현은 기존 구현들과 구별되는 두 가지 특징이 있다. 우선, 제안 방법에서는 매핑 규칙을 소스 코드에 애너테이션 형태로 작성한다. 기존 방법으로는 새로운 매핑 규칙 적용 대상이 된 프로그램 내의 특정 요소를 참조할 수단이 없기 때문이다. 다음으로, 기존 구현들에서는 매핑 엔진이 독립된 애플리케이션이지만 제안 방법에서는 매핑 엔진을 라이브러리 형태로 제공한다. 이것은 매핑 엔진이 다른 애플리케이션에 포함되어야 그 애플리케이션의 실행 시간 메모리에 존재하는 객체에 접근할 수 있기 때문이다. 본 논문에서는 제안 시스템의 구조와 동작 과정을 설명하며 유용성 확인을 위한 RML 테스트 케이스 실행 결과를 제시한다.

▸ **주제어**: RDF, 지식 그래프, RML, 선언형 매핑, 자바 애너테이션

• First Author: Ji-Woong Choi, Corresponding Author: Ji-Woong Choi
*Ji-Woong Choi (iamjwchoi@gmail.com), School of Computer Science and Engineering, Soongsil University
• Received: 2024. 09. 11, Revised: 2024. 09. 30, Accepted: 2024. 10. 04.

Copyright © 2024 The Korea Society of Computer and Information
http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

지식 그래프는 도메인 엔티티의 속성 그리고 엔티티 간의 상관을 그래프 모형으로 구축하여 도메인 시맨틱 정보를 표현한다[1]. 최근 들어, 기계 학습 기반의 서비스에서 지식 그래프의 사실 혹은 문맥 정보를 성공적으로 활용한 사례들의 발표로 인해, 지식 그래프의 구축은 다양한 분야에서 대규모화되고 있다[2-3]. 지식 그래프는 전통적으로 RDF(Resource Description Framework) 모델로 구축해 왔으며 근래에는 그래프 데이터베이스에서 사용되는 LPG(Labeled Property Graph) 모델이 주목받고 있다[4].

RDF 그래프는 주로 이기종 데이터를 그래프 요소로 매핑 규칙에 따라 변환시키는 방식으로 구축된다[5]. 이때 매핑 규칙 정의를 위해 사용되는 대표적인 언어가 R2RML(RDB to RDF Mapping Language)[6]과 RML(RDF Mapping Language)[7]이다. R2RML은 W3C 표준으로서 매핑의 소스가 관계형 데이터베이스(RDB)이다. RML은 매핑 소스를 RDB뿐 아니라 CSV, XML, JSON 포맷의 파일로 다양화한 R2RML에 대한 확장이다. RML은 R2RML을 포함한 사양이기 때문에 현재 사실상 표준의 지위에 있다[8].

본 논문은 RML 매핑 소스를 기존 파일 혹은 RDB에 저장된 데이터가 아닌 애플리케이션 실행 시간 메모리에 존재하는 데이터로 확장 시키고자 한다. 이를 위해 제안하는 RML 구현이 SDM-RDFizer[9], Morph-KGC[10], RMLStreamer[11]와 같은 기존 구현들과 구별되는 두 가지 특징이 있다. 첫째, 제안 시스템에서는 매핑 규칙을 소스 코드에 애너테이션 형태로 작성한다. 애너테이션은 모던 프로그래밍 언어들이 프로그램을 구성하는 각종 엔티티에 추가적인 기능을 손쉽게 부여할 수 있게 하는 수단으로 제공하는 구문 요소다[12]. 기존 방법에서는 매핑 규칙을 별도의 문서에 작성하며 이 문서에서 매핑 규칙이 적용될 매핑 소스를 URI로 지정한다. 매핑 소스가 파일 혹은 RDB이기 때문에 URI로 접근 가능하기 때문이다. 그러나 객체는 URI로 접근할 수 있는 대상이 아니기 때문에 제안 시스템에서는 변환 대상 객체의 원형인 클래스 정의부에 직접 애너테이션 형태로 매핑 규칙을 정의한다. 매핑 규칙을 애너테이션으로 작성함으로 인해 비롯되는 장점은 컴파일 시간에 매핑 규칙의 오류를 점검할 수 있다는 것이다. 기존 방법에서는 매핑 규칙의 오류는 실행 시간에 발견된다. 둘째, 기존 구현들에서는 매핑 엔진이 독립된 애플리케이션이지만 제안 방법에서는 매핑 엔진을 라이브러리 형태로 제공한다. 이것은 매핑 엔진이 변환 대상 객체와 같은 프로세스 공간에 있어야 객체에 접근하여 매핑을 수행할 수 있기 때문이다.

본 논문에서 제안하는 객체로부터 그래프를 생성하는 방법은 유연하고 기민한 그래프 생성을 돕는다. 기존 RML 매핑은 외부에서 로드 한 데이터 전체를 독립된 애플리케이션에서 일괄적으로 매핑해 버리는 방식이므로 처리 도중 개입이 불가능하다[13]. 그러나 제안 방법에서는 매핑 대상 객체의 선정, 매핑 결과에 대한 중간 조사 등의 제어 논리를 필요에 따라 추가할 수 있는 구조를 제공하기 때문이다. 또한, 파일 혹은 RDB 데이터가 애플리케이션이 실행 중 생산한 것을 저장한 결과일 경우 실행 시간 데이터를 외부로 저장하고 이를 매핑 목적으로 다시 메모리로 로드하는 스텝을 제거할 수 있기 때문이다.

본 논문의 나머지 부분은 다음과 같이 구성된다. 2장에서는 본 연구의 기반이 되는 기술을 요약한다. 3장에서는 제안하는 시스템의 구조 및 처리 절차를 설명한다. 4장에서는 테스트를 수행한 결과를 제시한다. 5장에서는 결론을 맺음과 동시에 향후 연구 방향을 소개한다.

II. Related Works

본 장에서는 제안 시스템의 기반이 되는 기술인 RML과 Java 애너테이션을 소개함으로써 본 장 이후에 대한 이해를 돕고자 한다. RML에 대해서는 RML 매핑 규칙 정의를 위한 구문의 구조와 의미 위주로 사례를 통해 기술하고 애너테이션에 대해서는 애너테이션 구문의 소개 및 기능의 범위, 처리 방법, 활용 사례 위주로 기술한다.

1. RML Mapping

그림 1~3은 하나의 RML 매핑 사례로써 그림 2는 RML 매핑 문서이고 그림 1과 3은 각각 매핑에 대한 입력과 출력 데이터이다. RML 매핑에서 사용 가능한 입력 데이터 형식은 행렬 구조인 CSV, RDB와 트리 구조인 XML, JSON이다. 그림 1은 CSV와 JSON 데이터를 포함한다. 그림 3은 RDF 데이터세트이다.

그림 2 매핑 문서는 RDF 직렬화 구문 중 가장 간결한 구문인 Turtle[14]로 쓰여 있다. RML 매핑 문서는 triples map의 모음이다. 그림 2에서 7~31행과 31~42행이 각각 <#TM1>과 <#TM2>라고 명명된 triples map이다. triples map은 매핑의 소스 데이터를 지정하기 위한 1개의 logical source, 주어 생성 규칙인 1개의 subject map, 술어·목적어 생성 규칙인 0개 이상의 predicate

object map으로 구성된다. predicate object map은 1개 이상의 predicate map과 1개 이상의 object map으로 구성된다. triples map의 트리플 생성 규칙은 행렬 구조 데이터의 경우 행마다 적용되며 트리 구조 데이터의 경우 객체마다 적용된다. 그림 1에서 csv 파일에는 1개의 행 그리고 json 파일에는 1개의 객체가 있다.

```
Airport.csv
1: id, city, bus, latitude, longitude
2: 6523, Brussels, 25, 50.901389, 4.484444

Landmark.json
1: { "venue": { "latitude": "50.901389",
2:              "longitude": "4.484444" },
3:   "location": { "continent": " EU",
4:                 "country": "BE",
5:                 "city": "Brussels" } }
```

Fig. 1. Input Data For RML Mapping In Fig. 2

RML은 트리플의 주어, 술어, 목적어뿐 아니라 RDF 그래프를 구성하는 각종 요소를 template, constant, reference 중 하나의 방식으로 생성할 수 있도록 한다. template 방식은 포맷 문자열 중 일부를 값으로 치환한 요소를 생성한다. 포맷 문자열 중 중괄호 부분이 치환될 부분이며 중괄호에는 치환될 값의 컬럼 혹은 속성명이 지정되어 있다. 이 예에서는 12, 37행에서 template 방식으로 주어를 생성했다. constant 방식은 입력 소스 데이터를 사용하지 않고 문서에 명시된 값으로 요소를 생성한다. 이 예에서는 rr:predicate, rr:graph, rr:language 예약어를 사용해 술어, 그래프명, 언어 태그를 constant 방식으로 생성했다. reference 방식은 명시된 컬럼 혹은 속성명에 속한 값 자체를 그대로 그래프의 요소로 취하는 방식이다. 이 예에서는 18, 23, 41행에서 목적어를 reference 방식으로 생성했다. RML에서는 다른 triples map에서 생성한 주어를 현재 생성코자 하는 목적어로 취하게 하는 방식도 제공한다. 그림 2의 24~31행이 그 예로서 <#TM2>로부터 생성된 주어를 <#TM1>이 생성하는 술어 ex:located의 목적어로 취하게 한다. 이때 28~30행에서 사용된 join condition은 <#TM1>이 현재 매핑 중인 행의 “city” 컬럼값과 같은 “location.city” 속성값을 갖는 객체에서 <#TM2>가 생성한 주어를 취해야 한다는 의미이다. 이 방식의 결과는 그림 3의 4행 문장이며 4행 문장의 목적어가 5행 문장의 주어와 같음을 확인할 수 있다.

그림 3의 1행 문장은 2~5행 문장들과 달리 문장 내 항이 4개다. 이러한 문장을 쿼드(quad)라 하며 쿼드는 주어, 술어, 목적어, 그래프명 순으로 구성된다. 쿼드는 문장 내 주어, 술어, 목적어 트리플이 그래프명으로 식별되는 그래프 소속임을 명시한 것이다. 즉, 그림 3의 5개 문장은 하나의 RDF 데이터세트에 포함되며 이 데이터세트는 디폴트 그래프에 속한 2~5행 트리플과 ex:StopsGraph에 속한 1행 트리플로 구성된 것으로 해석된다.

```
01: @prefix rml: <http://semweb../ns/rml#>.
02: @prefix ql: <http://semweb../ns/ql#>.
03: @prefix rr: <http://www../ns/r2rml#>.
04: @prefix xsd: <http://www../XMLSchema#>.
05: @prefix ex: <http://ex../ns#>.
06: @base <http://ex../ns#>.

07: <#TM1> a rr:TriplesMap;
08:   rml:logicalSource [
09:     rml:source "Airport.csv";
10:     rml:referenceFormulation ql:CSV; ];
11:   rr:subjectMap [
12:     rr:template "http://airport../{id}";
13:     rr:class ex:Stop;
14:     rr:graph ex:StopsGraph; ];
15:   rr:predicateObjectMap [
16:     rr:predicate ex:route;
17:     rr:objectMap [
18:       rml:reference "bus";
19:       rr:datatype xsd:int ] ];
20:   rr:predicateObjectMap [
21:     rr:predicate ex:lat;
22:     rr:objectMap [
23:       rml:reference "latitude" ] ];
24:   rr:predicateObjectMap [
25:     rr:predicate ex:located;
26:     rr:objectMap [
27:       rr:parentTriplesMap <#TM2>;
28:       rr:joinCondition [
29:         rr:child "city";
30:         rr:parent "location.city"; ];
31:     ]; ].

32: <#TM2> a rr:TriplesMap;
33:   rml:logicalSource [
34:     rml:source “Landmark.json" ;
35:     rml:iterator "$"; ];
36:   rr:subjectMap [
37:     rr:template "http://venue../{location.city}" ];
38:   rr:predicateObjectMap [
39:     rr:predicate ex:countryCode;
40:     rr:objectMap [
41:       rml:reference "location.country";
42:       rr:language "en" ] ].
```

Fig. 2. Example RML Mapping

```
1: <http://airport../6523> rdf:type transit:Stop ex:StopsGraph.
2: <http://airport../6523> ex:route "25"^^xsd:int.
3: <http://airport../6523> ex:lat "50.901389".
4: <http://airport../6523> ex:located <http://venue../Brussels>.
5: <http://venue../Brussels> ex:countryCode "BE"@en.
```

Fig. 3. Output Data By RML Mapping In Fig. 2

2. Annotations in Java

Java의 애너테이션은 주석(comment)처럼 소스 코드에 대한 메타데이터 즉 추가적인 정보이다. 주석은 정보의 수용자가 인간이지만, 애너테이션은 그 대상이 컴파일러 혹은 JVM(Java Virtual Machine) 즉 실행 환경이다. 따라서, 컴파일 혹은 실행 시간에 이 정보를 활용한 추가적인 기능을 프로그램에 부여할 수 있다. Python과 JavaScript에서는 'Decorator' 그리고 Swift에서는 'Attribute'처럼 명칭은 제각각이나 Java의 애너테이션과 유사한 개념의 구문 장치를 현대의 다른 프로그래밍 언어에서도 제공하고 있다.

```
@Entity
@Getter @Setter
public class Student {
    @Id
    private Long id;
    private String name;
    @ManyToOne
    private Department dept;

//public Long getId() { return id; }
//public void setId(Long id) { this.id = id; }
//public String getName() { return name; }
//public void setName(String name) { this.name = name; }
//public Department getDept() { return dept; }
//public void setDept(Department dept) { this.dept = dept; }
}

@Entity
public class Department {
    @Id
    private Long id;
    private String name;
    @OneToMany
    private List<Student> students;
}
```

Fig. 4. Java Annotation Usage Example

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Entity {
    String name() default "";
}
```

Fig. 5. Annotation Entity Source Code

그림 4는 애너테이션 사용 예다. 애너테이션은 '@' 기호를 사용한다. 그림 5는 그림 4에서 사용된 애너테이션 Entity 정의 코드이며 애너테이션 정의 시에도 애너테이션 @Target과 @Retention이 사용되었음을 확인할 수 있다. 그림 4에서 사용된 애너테이션들 중 @Getter와 @Setter는 롬복[15]이라는 라이브러리에서 제공하는 것이며 나머지들은 JPA(Java Persistence API)[16]라는 Java 표준 사양을 구현한 프레임워크에서 제공하는 것이다. 애너테이션은 정의하기에 따라 타입, 필드, 메서드, 지역 변수 등 소스 코드의 각종 구문 요소에 부여할 수 있다. 그림 4의 @Entity는 타입, 나머지는 타입 내 필드에 부여된 것이다. 그림 5의 첫 행은 @Entity를 타입에만 부여되도록 정의한 것이며 이로 인해 타입이 아닌 요소에 부여되면 컴파일 에러가 발생한다. 그림 5의 @Retention은 애너테이션의 유지 기한을 정한다. 애너테이션 Entity는 @Retention의 속성값이 'RUNTIME'이라서 실행 시간까지 사용할 수 있는 애너테이션이 된다. 반면에 @Getter와 @Setter는 유지 기한이 'SOURCE'인데 이것은 컴파일 시 이 두 애너테이션을 번역 대상에서 제외하라는 의미다. 따라서 실행 시간에는 사용할 수 없게 된다.

애너테이션은 다양한 용도로 활용된다. 롬복의 두 애너테이션은 개발자가 상용구 코드를 작성하는 수고를 덜어준다. 그림 4에서 주석 처리된 코드들은 @Getter와 @Setter를 사용하지 않았을 시 직접 입력해야 하는 코드다. 롬복은 해당 코드의 번역된 버전을 컴파일 시 바이트코드에 주입해 준다. JPA는 애플리케이션에 ORM(Object Relational Mapping) 기능을 추가해 준다. ORM은 개발자가 RDB에 종속적인 코드를 작성하는 수고를 덜어준다. ORM 도구는 @Entity가 부여된 클래스를 RDB의 특정 테이블 혹은 뷰와 대응시킴은 물론 해당 클래스의 객체와 대응된 테이블 혹은 뷰의 행 데이터도 1:1로 대응시킨다. 그리고 객체와 행 데이터 간의 상호 변환 또한 ORM 도구 내에서 처리해 준다. 따라서 개발자는 본질적으로 RDB 데이터를 다루는 프로그램이라 하더라도 객체만을 다루면 되는 편의를 누리게 된다.

III. The Proposed System

1. System Overview

본 절에서는 제안 시스템을 구성하는 요소별 역할을 사용 흐름에 따라 개괄적으로 설명한다. 그림 6은 제안 시스템의 구성도이다. 제안 시스템은 총 10개의 애너테이션을 제공한다. 사용자는 이 애너테이션들을 사용해 소스 코드에 RML 매핑 규칙을 작성할 수 있다. 그림 7~8은 그 예로서 그림 2 매핑 규칙과 의미가 동등하도록 애너테이션 형태로 재작성된 것이다. 그림 2의 <#TM1>과 <#TM2> triples map은 각각 그림 7과 그림 8에 대응한다. 애너테이션 프로세서는 컴파일 시간에 동작한다. 이 프로세서는 작성된 애너테이션들의 논리적 구성 오류와 구문 오류를 점검하여 위반 시 컴파일 에러를 발생시킨다.

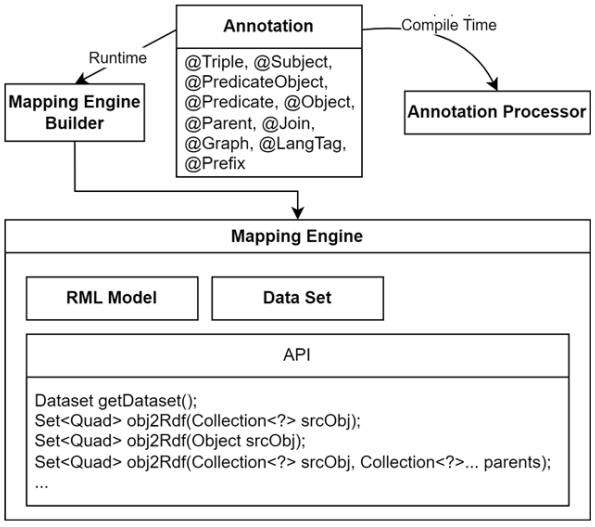

Fig. 6. System Overview

매핑 엔진은 실행 시간에 매핑을 수행하는 주체다. 매핑 엔진은 매핑 엔진 빌더로 생성한다. 사용자는 매핑 엔진 빌더를 통해 매핑 엔진을 구성할 수 있다. 구성의 핵심은 소스 코드에 애너테이션으로 작성된 매핑 규칙들 중 생성코자 하는 매핑 엔진이 사용할 매핑 규칙을 선정하는 것이다. 따라서 사용자는 서로 다른 구성의 복수 개의 매핑 엔진을 생성해 사용할 수도 있다. 그림 9는 매핑 엔진을 생성하는 코드 예다. 그림 7과 8의 애너테이션을 함께 사용하기 위하여 그림 9에서 매핑 빌더에 Airport와 Landmark 클래스를 전달하고 있다. 클래스를 전달하는 이유는 매핑 규칙 애너테이션이 클래스에 부여된 것이기 때문이다. RML 모델은 매핑 엔진 초기화 시 생성되어 매핑 엔진에 내장된다. RML 모델은 애너테이션 형태의 매핑 규칙을 객체화한 것이며 매핑 엔진은 RML 모델을 참조하여 매핑을 수행한다. RML 모델은 RML 사양의 매핑 규칙 논리 구조를 프로그래밍 객체 모델로 옮긴 것이다. 매핑 엔진에 내장된 또 하나의 구조물인 데이터세트는 매핑 엔진이 생성한 그래프를 보관하는 기능과 보관된 그래프를 외부 저장소로 내보내는 기능을 제공한다.

매핑 엔진 생성 이후 사용자는 매핑 엔진의 API를 사용해 원하는 시점에 변환을 명령할 수 있다. 그림 9의 마지막 행이 변환을 위한 API 호출 예다. API는 단일 객체이건 객체의 모음이건 상관없이 변환 가능하도록 설계하였으며 변환 결과도 API 호출에 따른 반환 값으로 바로 확인할 수 있도록 하였다. 따라서 사용자는 변환 결과를 조사한 후 선별하여 데이터세트에 보관하는 작업 순서가 가능하다.

```
@Prefix(prefix = "ex", iri = "<http://ex../ns#>")
@Prefix(prefix = "xsd", iri = Ns.XSD)

@Triple(
  s = @Subject(
        template = "http://airport../{id}",
        classes = {"ex:Stop"},
        graphs = {@Graph(constant = "ex:StopsGraph")}),

  pos = {
    @PredicateObject(
      p = @Predicate(constant = "ex:Stop"),
      o = @Object(field = "bus", dt = "xsd:int")),
    @PredicateObject(
      p = @Predicate(constant = "ex:lat"),
      o = @Object(field = "latitude")),
    @PredicateObject(
      p = @Predicate(constant = "ex:located"),
      o = @Object(
            parent = @Parent(id = "#TM2"),
            join = { @Join(
                       child = "city",
                       parent = "location.city")})})})
public class Airport {
    long id;
    String city;
    int bus;
    String latitude;
    String longitude;
}
```

Fig. 7. Class Airport Annotated

```
@Triple(
  id = "#TM2",
  s = @Subject(template = "http://venue../{location.city}"),
  pos = {@PredicateObject(
          p = @Predicate(constant = "ex:countryCode"),
          o = @Object(
                field = "location.country",
                langTag = @LangTag(constant = "en")))})
public class Landmark {
    Venue venue;
    Location location;
}

class Location {
    String continent;
    String country;
    String city;
}

class Venue {
    String latitude;
    String longitude;
}
```

Fig. 8. Class Landmark Annotated

```
MappingEngine engine = new MappingEngine.Builder()
            .baseIri("http://example.com/base/")
            .triples(Airport.class, Landmark.class)
            .build();
...
Set<Quad> quads = engine.obj2Rdf(obj);
```

Fig. 9. Mapping Engine Usage Example

2. Design of Annotations and Their Processing

본 절에서는 RML 매핑 정의를 위한 애너테이션의 구조 상세를 제시하고 설계 전략 그리고 컴파일 시간에 수행되는 애너테이션 처리 사항을 기술한다.

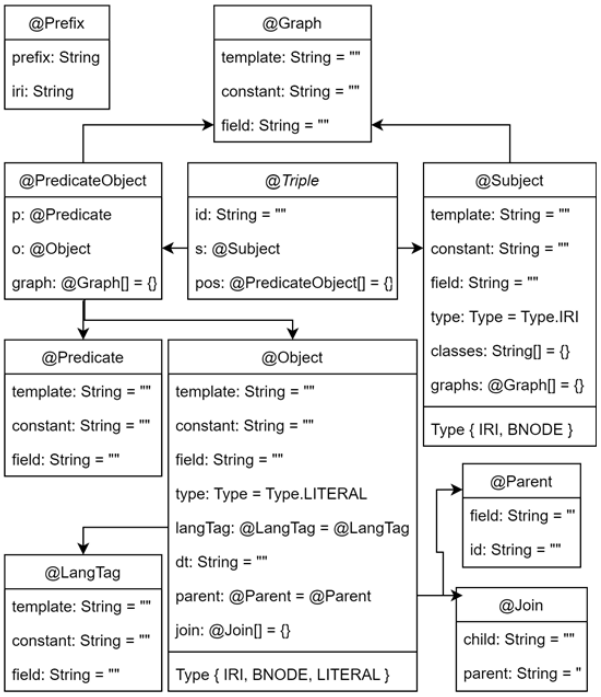

Fig. 10. Annotations For RML Mapping

그림 10은 애너테이션 각각의 상세 속성과 애너테이션 간의 참조 관계를 보여준다. 속성은 '속성명: 타입 = 기본값' 형식이다. 모든 애너테이션은 타입에만 부여될 수 있고 실행 시간까지 유지되도록 구현되었다. 그림 10의 애너테이션 간 참조 관계는 그림 7~8에서처럼 애너테이션 간 내포 관계로 사용 시 표현된다. 참조 관계의 루트이자 내포 관계의 최상위는 @Triple이다. @Prefix는 참조 관계에 포함되지 않지만 @Triple 이하 논리 구조에서 속성값으로 요구되는 IRI(Internationalized Resource Identifier)를 축약된 형식으로 사용하고자 할 때 필요하게 되는 접두어를 미리 정의해두는 용도이며 그림 7에서처럼 단일 클래스에 여러 개 부여할 수 있다. @Prefix와 마찬가지로 제안 시스템은 @Triple 또한 단일 클래스에 여러 개 부여할 수 있도록 설계하였다. 각각은 id 속성값으로 구분된다.

RML 사양은 매핑 규칙 정의를 위해 그림 3에서 볼 수 있는 'rr' 혹은 'rml' 접두어로 시작되는 어휘를 제공한다. '접두어가 'rr'이면 R2RML에서 정의한 것이며 접두어가 'rml'이면 RML에서 추가 정의한 것이다. RML은 이 어휘들을 클래스와 프로퍼티로 분류한다. 제안 시스템은 클래스는 애너테이션에 그리고 프로퍼티는 애터네이션의 속성에 대응시켜 설계함으로써 매핑 규칙을 애너테이션으로 작성하더라도 기존 문서로 작성된 매핑 규칙과 동일한 논리 구조를 유지할 수 있게 하였다.

컴파일 시간에는 사용자가 작성한 애터테이션의 오류를 점검하여 위반 시 컴파일 에러를 발생시킨다. 점검 주체는 컴파일러와 에너테이션 프로세서다. 에너테이션 프로세서는 컴파일러 수준에서 처리할 수 없는 사항들을 점검한다. 컴파일러는 그림 10에 표현된 정보에 근거하여 애너테이션 출현 위치의 적절성 그리고 출현 횟수의 적절성 등 구문에 대한 구조 차원의 사항들을 점검해 준다. 애너테이션 프로세서는 구문에 대한 의미 차원의 적절성을 점검한다. 주요 점검 사항은 다음과 같다. 첫째, @Triple의 id 속성 관련으로서 정의 시 id의 유일성 그리고 참조 시 참조되는 id의 존재 여부다. 둘째, field 속성 관련 사항으로서 field 속성값 혹은 template 문자열 내에 사용된 필드명의 존재 여부를 점검한다. 참고로, field 속성은 RML의 어휘 'rml:reference'에 대응하는데 Java에서 필드는 클래스의 멤버 변수를 뜻하며 제안 시스템에서는 reference 되는 대상이 필드로 한정되기에 속성명으로 선택했다. 셋째, template, constant, field 속성 관련 사항으로서 이 세 속성 중 한 속성만을 선택해서 사용해야 하므로 셋 중 한 속성에 기본값이 아닌 값을 할당했는지를 점검한다. 이 점검 사항에 있어서 예외는 @Object이다. @Object에서는 parent 속성과 이 세 속성은 배타적으로 사용되어야 한다. 따라서 parent 속성이 사용되면 이 세 속성은 모두 기본값이어야 한다. 마지막으로, IRI 접두어 관련 사항이다. @Prefix에서 서로 다른 IRI에 대한 같은 접두어의 중복된 사용 여부 그리고 축약된 형식의 IRI 사용 시 사용된 접두어의 존재 여부를 점검한다.

3. Object-to-Graph Mapping in Runtime

본 절에서는 제안 시스템이 실행 시간에 객체로부터 RDF 그래프를 생성하는 방법을 기술한다. 본 절에서부터는 객체를 그래프로 매핑하는 것을 OGM(Object to Graph Mapping)으로 기술하고자 한다. 범용 목적의 OGM 라이브러리인 제안 시스템을 제작할 때의 근본적인

어려움은 무슨 타입의 객체가 매핑의 입력으로 사용될지를 제작 시점에 미리 알 수 없다는 것이다. 매핑에 사용될 객체는 실행 중에 사용자에 의해 주어지기 때문에, 해당 객체의 타입을 실행 시간에 즉 동적으로 식별해 내야 한다. 그래야 객체에 올바른 매핑 규칙을 적용할 수 있기 때문이다. 제안 시스템은 Java의 리플렉션 기법을 사용해 주어진 객체의 타입을 동적으로 알아낸다. 리플렉션이란 Java의 객체가 실행 중 JVM에 의해 관리된다는 특성을 활용한 저수준 기법으로서 JVM에 직접 의뢰하여 객체의 타입과 타입의 구조 정보를 획득할 수 있게 해줌은 물론 나아가 이 정보를 바탕으로 객체의 조작까지 허용한다. 제안 시스템은 리플렉션을 통해 획득한 타입과 타입에 대한 구조 정보를 활용해 객체를 탐색해 가며 매핑 규칙에 명시된 필드값을 추출하여 RDF 그래프 생성에 사용한다. 그림 6의 매핑 엔진이 제공하는 API 중 실질적으로 OGM을 수행하는 함수들인 obj2Rdf의 인수 타입은 Java의 Object 혹은 Object의 컬렉션이다. Object 타입이 Java에서 어떤 타입의 객체이건 참조할 수 있기 때문이다. 하지만 참조 타입이 Object인 객체는 실제 타입이 무엇이건 Object 타입으로만 다루어야 한다는 제약을 obj2Rdf 함수에 리플렉션 기법을 적용함으로써 극복하였다.

(ㄱ)

name	address		hobbies
John	city	state	[Music, Running]
	Los Angeles	CA	

(ㄴ)

name	address.city	address.state	hobbies
John	Los Angeles	CA	Music
John	Los Angeles	CA	Running

Fig. 11. Flattening an Object

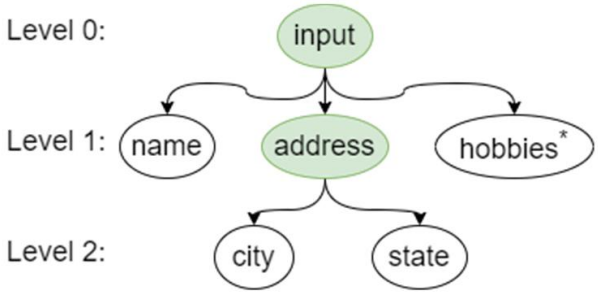

Fig. 12. A Tree Consisting of Field Names

구체적인 OGM 절차는 변환을 위한 객체가 입력되면 우선 객체의 실제 타입을 알아낸 후 해당 타입에 부여된 매핑 규칙을 골라낸다. 애너테이션 형태의 매핑 규칙들은 이미 매핑 엔진 내 RML 모델로 옮겨져 있으므로 선정된 매핑 규칙을 구성하는 하나의 주어 생성 규칙 그리고 복수의 술어·목적어 쌍 생성 규칙을 RML 모델을 대상으로 1회씩 방문한다. RML 모델의 탐색이 종료되면 매핑 규칙 당 두 가지 결과물이 산출된다. 하나는 RDF 문장들의 템플릿이며 다른 하나는 필드명들의 트리이다. 템플릿 내 RDF 문장들에는 입력 객체의 필드명이 곳곳에 포함되어 있으며 이 필드명들을 결국 모두 필드 값으로 치환시켜야 한 객체에 대한 OGM이 완료되기 때문에 치환을 위해 방문해야 하는 필드명들을 트리로 구성한 것이다. 이 두 결과물은 한 번 생성되면 동일 타입의 객체에 대해서는 계속 재사용된다.

변환 대상 객체의 필드는 그 값이 리터럴, 객체, 컬렉션 중 하나다. 컬렉션은 배열, 리스트, 집합 등을 총칭하며 컬렉션의 요소는 객체들일 수도 있으며 리터럴들일 수도 있다. 그림 11의 (ㄱ)은 name, address, hobbies라는 필드로 구성된 어느 객체를 테이블 형태로 표현한 것이다. name은 값이 리터럴인 사례이며 address는 값이 객체인 사례이고 hobbies는 값이 리터럴의 배열인 사례이다.

필드명의 필드 값으로의 치환은 결국 리터럴로 이루어지기 때문에 RDF 문장들의 템플릿 내 존재하는 필드는 그 값이 모두 리터럴이다. 그림 11의 (ㄴ)은 (ㄱ)을 평탄화시킨 결과로서 매핑에 사용될 수 있는 필드명을 모두 나열하면 name, address.city, address.state, hobbies로 도출되며 이 네 필드의 값은 모두 리터럴임을 확인할 수 있다.

값이 리터럴인 필드에 대한 처리는 템플릿 내 해당 필드명을 필드 값으로 단순히 치환하면 된다. 요소값이 리터럴인 컬렉션 필드에 대한 처리는 치환할 값이 요소의 개수만큼 존재하기 때문에 RDF 문장 템플릿에서 해당 컬렉션 필드명이 포함된 문장들만 빼내 다음 이 문장들을 하나의 단위로 취급하여 요소의 개수만큼 사본을 준비한 뒤 각각의 사본에 각각의 요소값으로 해당 필드명을 치환시킨다. 치환이 완료된 모든 사본을 원래의 RDF 문장 템플릿에 추가시키면 하나의 리터럴 컬렉션 필드에 대한 처리는 종료된다. 그림 12는 그림 11의 (ㄱ) 객체에서 name, address.city, address.state, hobbies가 매핑 시 치환할 값을 갖는 필드들이라 가정했을 때 생성되는 필드명들의 트리이다. 이 트리에서 치환할 값을 갖는 필드는 모두 말단 노드에 배치된다. 그림 12에서 hobbies의 위첨자 '*'는 hobbies가 컬렉션임을 표현한 것이다. 즉 hobbies에 해당하는 대상이 여럿 존재할 수 있음을 표현한 것이다.

```
01: Set<Quad> map(Set<Quad> template, Object obj, Node node) {
02:   fieldNameNodes = findChildren(node)
03:   fieldNameValuePairs = extractValues(obj, fieldNameNodes)
04:   foreach (pair : fieldNameValuePairs) {
05:     if (pair.value is a literal)
06:       fillValue(template, pair)
07:     else if (pair.value is a collection of literals)
08:       fillValueMultipleTimes(template, pair)
09:     else if (pair.value is a collection of objects) {
10:       Set<Quad> temp = empty
11:       foreach (element : pair.value)
12:         temp.add(map(template, element, pair.name))
13:       template = temp
14:     }
15:     else if (pair.value is an object)
16:       template = map(template, pair.value, pair.name)
17:   }
18:   return template
19: }
```

Fig. 13. Pseudo-code to Replace Fields' Name by Values

필드명 트리에서 말단 노드는 필드 값이 리터럴 혹은 리터럴들의 컬렉션인 필드명들이지만 말단 노드가 아닌 노드는 객체 혹은 객체들의 컬렉션이 값인 필드명들이다. 그리고 형제 노드들은 부모 노드에 해당하는 객체에 함께 소속된 필드명들이 된다. 이러한 노드들의 특성 그리고 노드 간 관계를 고려하여 설계한 그림 13의 필드명을 필드 값으로 치환하는 알고리즘은 하나의 재귀함수로 구성되어 있다. 이 함수의 1회 호출은 필드명 트리의 말단이 아닌 한 노드를 방문하여 수행하는 작업에 대응되며 이 알고리즘은 말단이 아닌 노드만을 깊이 우선 방식으로 탐색한다. 그림 12로 설명하면 그림 12에서 뿌리 노드인 input에 이어 address 노드만 방문하면 탐색이 종료된다.

그림 13의 1행에서 세 번째 매개변수로 선언된 node에는 필드명 트리에서 현재 방문하고자 하는 노드가 전달되며 두 번째 매개변수인 obj에는 node에 해당하는 객체가 전달된다. 한 노드를 방문하면 그 노드의 모든 자식 노드에 있는 필드명을 취합하여 노드와 함께 전달된 객체에서 취합된 필드명에 해당하는 필드값들을 추출한 다음 필드명과 필드값의 쌍을 생성한다. 이 작업이 그림 13의 2~3행에 해당한다. 그다음 각각의 필드명과 필드값 쌍에 대한 처리가 그림 13의 4~17행 반복문에 해당한다. 반복문은 4개의 분기된 작업으로 나뉜다. 필드값이 리터럴일 땐 5~6행, 리터럴의 컬렉션일 땐 7~8행, 객체들의 컬렉션일 땐 9~14행, 객체일 땐 15~16행에 있는 작업을 수행한다. 필드값이 리터럴일 때와 리터럴의 컬렉션일 때의 치환 작업은 이미 기술하였으므로 필드값이 객체일 때는 재귀 호출이 발생한다. 이 호출에는 필드명을 node로, 필드값 객체를 obj로, 호출 시점까지 치환이 이루어진 RDF 문장 템플릿을 template으로 전달한다. 재귀 호출이 완료되면 RDF 문장들의 템플릿이 반환되며 기존 템플릿을 이 반환된 템플릿으로 교체한다. 재귀 호출 과정에서 더 많은 치환이 이루어진 것이기 때문이다. 필드값이 객체 컬렉션일 때도 재귀 호출이 발생한다. 다만 각각의 요소 객체에 대하여 요소 개수만큼 호출한다. 그리고 각각의 호출 결과로 반환된 문장 템플릿들은 같은 필드를 서로 다른 값으로 치환한 것들이기 때문에 기존 템플릿을 반환된 문장 템플릿들을 모두 하나로 모은 후에 교체하는 방식을 취한다.

IV. Conformance Test

본 장에서는 제안 시스템이 다양한 매핑 테스트 케이스에서 올바른 RDF 문장들을 생성하는지를 평가하고자 한다. 본 연구에서는 RML 매핑 엔진의 정확성을 평가하고자 RML 사양을 관리하는 조직에서 개발한 테스트 케이스 세트[17]를 사용하고자 한다. 이 테스트 케이스 세트는 297개의 개별 테스트 케이스들을 60개 범주로 묶어놓았다. 범주별로 서로 다른 매핑 규칙이 할당되어 있으며 같은 범주로 묶인 테스트 케이스들은 매핑을 위한 입력 데이터 포맷만 서로 다르다. 데이터는 총 6종의 포맷으로 제공된다. 나열하면 RDBMS인 MySQL, PostgreSQL, SQLServer 그리고 파일 포맷인 XML, JSON, CSV이다. 범주마다 6가지 포맷을 모두 테스트하지는 않으며 RDBMS로만 구성된 범주도 다수 있다.

제안 시스템은 객체로부터 RDF 그래프를 생성하는 시스템이기 때문에 테스트 케이스 세트가 제공하는 다양한 포맷의 데이터를 객체로 변환하는 작업을 개별 테스트 케이스마다 선행하였다. 객체로의 변환 작업은 포맷별로 오픈 소스 도구들을 사용하였다. CSV 데이터는 OpenCSV[18], XML 데이터는 JAXB[19], JSON 데이터는 Jackson[20], RDBMS 데이터는 Spring Data JPA[21]를 사용하였다. 이 도구들은 모두 제안 시스템처럼 애너테이션을 사용한 변환 정의 인터페이스를 제공한다는 공통점이 있다. 다음으로 테스트 케이스 세트가 제공하는 매핑 규칙은 문서 형태로 제공되기 때문에 이를 애너테이션 형태로 재작성하는 작업 또한 범주별로 수행하였다. 테스트 케이스 세트는 매핑 엔진이 생성해야 할 RDF 문장들을 텍스트 파일 형태로 테스트 케이스별로 제공한다. 제안 시스템은 RDF 문장들의 집합 객체를 매핑 결과로 생산하기 때문에, 테스트 케이스 세트에서 제공하는 텍스트 파일을 읽어 들여 RDF 문장들의 집합 객체를 생성하였다. 두 비교 대상이 동일한 형태로 만들어졌으므로 집합의 동치 여부를 기계가 따지도록 하였다. 두 집합이 동치로 출력되면 해당 테스트 케이스를 통과한 것으로 판정했다.

Table 1. Test Result

Result Description	# of categories
Failed to compile due to invalid annotation syntax	4
Failed to compile due to wrong field access	8
RDF graph creation success	48

표 1은 테스트 결과를 요약한다. 범주별로 성공과 실패가 나뉘었다. 제안 시스템은 48개 범주에서 성공적으로 OGM을 수행했으며 12개 범주에서 컴파일 시간 에러가 발생함으로 인해 수행을 완료하지 못했다. 하지만 컴파일 에러가 발생한 12개 범주는 테스트 케이스 세트에서 고의로 에러가 유발되도록 설계된 것들이다. 즉 매핑 엔진이 에러 유발 사유들을 탐지할 수 있는지를 점검하고자 할당한 범주들이다. 에러 유발 원인은 크게 두 종류로 구분된다. 첫 번째는 특정 구문 요소를 누락시키거나 올바르지 않은 곳에 위치시키는 방식으로 매핑 규칙 구성이 RML 매핑 규칙 사양을 준수하지 않은 경우로서 4개의 범주가 이에 해당한다. 두 번째는 매핑 규칙 구성은 RML 사양을 준수하나 매핑의 입력 데이터 소스에 존재하지 않는 컬럼 혹은 속성을 매핑 규칙에 사용한 경우로서 8개의 범주가 이에 해당한다. 기존 매핑 엔진들은 독립적인 애플리케이션 형태로 동작하며 실행 시에야 매핑 규칙 문서와 매핑의 입력 데이터에 대한 접근이 이루어지기 때문에, 이 12개 범주 테스트에 대해 실행 시간에 예외를 발생시키는 방식으로 반응한다. 이와는 달리 제안 시스템에서는 이 12개 범주에 포함된 개별 에러 유발 사유들이 모두 컴파일 시간에 컴파일러 혹은 애너테이션 프로세서가 점검하는 항목들에 포함됐기 때문에, 컴파일 에러로 반응했다. 결과적으로 모든 테스트 케이스를 통과했으므로 제안 시스템이 매핑 규칙에 따른 정확한 그래프를 생성한다고 평가할 수 있다.

V. Conclusions

본 논문을 통해 애플리케이션 실행 시간 메모리에 존재하는 객체로부터 RDF 그래프를 생성해 주는 시스템을 제안했다. 객체의 RDF 그래프로의 변환은 사전에 정의된 매핑 규칙을 객체에 적용함으로써 수행된다. 매핑 규칙 정의 체계는 RML 사양을 따랐다. RML은 다양한 이기종 데이터로부터 RDF 그래프를 생성하기 위한 목적으로 고안된 매핑 규칙 정의 체계로서 이 분야의 사실상 표준의 지위에 있다. RML 사양이 파일 혹은 데이터데이스에 기록된 데이터만을 매핑의 대상으로 삼는다는 점을 고려할 때, 제안 시스템은 RML 매핑의 적용 범위를 객체로까지 확장 시켰다는 데 의의가 있다. 매핑 규칙의 적용 대상이 객체인 점으로 인해 제안 시스템은 기존 RML 구현들과 구별되는 두 가지 특징을 갖게 되었다. 첫째, 제안 시스템에서는 클래스 정의부에 애너테이션 형태로 매핑 규칙을 정의하도록 한다는 점이다. 매핑 규칙 정의 위치가 클래스인 까닭은 객체란 것이 실행 중 생성되는 것이므로 해당 객체에 적용할 매핑 규칙을 그 객체의 원형인 클래스에 추가하는 것이 직관적이고 합리적이라고 판단했기 때문이다. 그리고 애너테이션은 클래스 정의에 대한 메타데이터로 취급되기 때문에, 기존 로직에 대한 부작용을 최소화할 수 있다는 장점 또한 있다. 둘째, 제안 시스템에서는 매핑 엔진을 라이브러리 형태로 제공한다는 점이다. 매핑 엔진이 변환 대상 객체와 같은 프로세스 공간에 있어야 객체에 접근하여 매핑을 수행할 수 있기 때문이다. 이와 관련하여 제안 시스템은 실행 중 결정된 변환 대상 객체를 다루는 데 있어서 Java의 리플렉션 기법을 활용했다. 이로 인해 사용자가 객체에 대한 추가적인 정보를 제공하지 않고도 시스템이 스스로 해당 객체에 올바른 매핑 규칙을 찾아 적용하는 효과를 가져왔다.

본 연구에서는 제안 시스템에 RML 테스트 케이스 세트를 적용한 결과 제안 시스템이 다양한 조합의 매핑 규칙에서도 객체로부터 올바른 RDF 그래프를 생성함을 확인하였다. 다만 이 평가 도구의 목적에 입력 데이터 규모에 따른 scalability가 포함되지 않았음을 밝힌다. 향후 연구로는 본 연구를 바탕으로 객체로부터 RDF 그래프 뿐 아니라 최근 들어 주목받고 있는 Property 그래프 포맷으로도 지식 그래프를 생성해 주는 시스템을 개발하고자 한다.

REFERENCES

[1] A. Hogan, E. Blomqvist, M. Cochez, C. D'amato, G. De Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier, A. N. Ngomo, A. Polleres, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, and A. Zimmermann, "Knowledge Graphs," ACM Computing Surveys, Vol. 54, No. 4, pp. 1-37, May 2022. DOI: 10.1145/3447772

[2] Y. Han, G. Lu, S. Zhang, L. Zhang, C. Zou, and G. Wen, "A Temporal Knowledge Graph Embedding Model Based on Variable Translation," Tsinghua Science and Technology, Vol. 29, No. 5, pp. 1554-1565, October 2024, DOI: 10.26599/TST.2023.9010142

[3] S. Ji, S. Pan, E. Cambria, P. Marttinen, and P. S. Yu, "A Survey

on Knowledge Graphs: Representation, Acquisition, and Applications," IEEE Transactions on Neural Networks and Learning Systems, Vol. 33, No. 2, pp. 494-514, February 2022, DOI: 10.1109/TNNLS.2021.3070843

[4] R. Zhang, P. Liu, X. Guo, S. Li, and X. Wang, "A Unified Relational Storage Scheme for RDF and Property Graphs," Proceedings of the 16th International Conference on Web Information System and Applications, pp. 418-429, Qingdao, China, September 2019. DOI: 10.1007/978-3-030-30952-7_41

[5] A. Dimou, "High-quality knowledge graphs generation: R2rml and rml comparison, rules validation and inconsistency resolution," Applications and Practices in Ontology Design, Extraction, and Reasoning, Vol. 49, No. 4, pp. 55-72, November 2020. DOI: 10.3233/SSW200035

[6] S. Das, S. Sundara, and R. Cyganiak, R2RML: RDB to RDF Mapping Language, http://www.w3.org/TR/r2rml/.

[7] B. D. Meester, P. Heyvaert, and T. Delva, RDF Mapping Language (RML), https://rml.io/specs/rml/

[8] V. Janev, D. Graux, H. Jabeen, and E. Sallinger, "Knowledge Graphs and Big Data Processing," Springer Cham, pp. 59-72, 2020.

[9] E. Iglesias, S. Jozashoori, D. Chaves-Fraga, D. Collarana, and M. Vidal, "SDM-RDFizer: An RML Interpreter for the Efficient Creation of RDF Knowledge Graphs," Proceedings of the 29th ACM International Conference on Information & Knowledge Management, pp. 3039-3046, Virtual Event, Ireland, October 2020. DOI:10.1145/3340531.3412881

[10] J. Arenas-Guerrero, D. Chaves-Fraga, J. Toledo, M. S. Pérez, and O. Corcho, "Morph-KGC: Scalable knowledge graph materialization with mapping partitions," Semantic Web, Vol. 15, No. 1, pp. 1–20, January 2024. DOI:10.3233/SW-223135

[11] G. Haesendonck, W. Maroy, P. Heyvaert, R. Verborgh, and A. Dimou, "Parallel RDF generation from heterogeneous big data," Proceedings of the International Workshop on Semantic Big Data, pp. 1-6, Amsterdam, Netherlands, July 2019. DOI: 10.1145/3323878.3325802.

[12] E. Guerra, A. D. O. Dias, L. G. D. O. Véras, A. Aguiar, J. Choma, and T. S. D. Silva, "A Model to Enable the Reuse of Metadata-Based Frameworks in Adaptive Object Model Architectures," IEEE Access, Vol. 9, pp. 85124-85143, June 2021. DOI: 10.1109/ACCESS.2021.3087795

[13] M. Bakken, "maplib: Interactive, Literal RDF Model Mapping for Industry," IEEE Access, Vol. 11, pp. 39990-40005, April 2023. DOI: 10.1109/ACCESS.2023.3269093

[14] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers, RDF 1.1 Turtle, https://www.w3.org/TR/turtle/.

[15] Project Lombok, https://projectlombok.org/

[16] C. Tudose, and C. Odubășteanu, "Object-relational Mapping Using JPA, Hibernate and Spring Data JPA," Proceedings of the 23rd International Conference on Control Systems and Computer Science, pp. 424-431, Bucharest, Romania, May 2021. DOI: 10.1109/CSCS52396.2021.00076

[17] P. Heyvaert, A. Dimou and B. D. Meester, RML Test Cases, https://rml.io/test-cases/.

[18] OpenCSV, https://opencsv.sourceforge.net/

[19] JAXB, https://javaee.github.io/jaxb-v2/

[20] Jackson, https://github.com/FasterXML/jackson

[21] Spring Data JPA, https://spring.io/projects/spring-data-jpa

Authors

Ji-Woong Choi received the B.S., M.S. and Ph.D. degrees in Computer Science and Engineering from Soongsil University, Korea, in 2001, 2003 and 2011, respectively. Dr. Choi joined the faculty of the School of Computer Science and Engineering at Soongsil University, Seoul, Korea, in 2013. He is currently an Associate Professor in the School of Computer Science and Engineering, Soongsil University. He is interested in Data and Knowledge, Artificial Intelligence and Machine Learning.