Kaynak Kod Sorgulamada Ontoloji Kullanımı

Önder Keskin[1] Ebru Sezer[2]

[1,2]Bilgisayar Mühendisliği Bölümü, Hacettepe Üniversitesi, Ankara
[1]e-posta: okeskin@cs.hacettepe.edu.tr [2] e-posta: esezer@cs.hacettepe.edu.tr

Özetçe

Yazılım mühendisliğinde, yazılım üzerinde birçok işin gerçekleştirilebilmesi için, kaynak kodun anlaşılabilmesi hayati önem taşımaktadır. Kaynak kod sorgulama araçları, yazılımın farklı kod kesimleri arasındaki ilişkilerden faydalanarak, bu kesimler arasında gezinebilmeye, dolayısı ile kodun anlaşılabilmesine imkân verir. Bu çalışma kapsamında, bir ontoloji-tabanlı kaynak kod sorgulama aracı, Eclipse geliştirme ortamına eklenti olarak geliştirilmiştir. Bu araçta bilgi tabanı olarak OWL (Web Ontology Language) ile gösterilmiş ontoloji, sorgulama dili olarak ta SPARQL (SPARQL Protocol and RDF Query Language) kullanılmıştır. Araç geliştirme süresince; ilk olarak, Java ile hazırlanmış kaynak kodlar için ontoloji oluşturulmuş, sonra ilgilenilen Java projesi için otomatik olarak ontoloji olgularını oluşturabilen bir ayrıştırıcı geliştirilmiştir. Son olarak, kullanıcının projeyi sorgulayabilmesi için bir sorgu görünümü ve bu sorguların işlenmesi sonucunda elde edilen sonuçların gösterildiği bir sonuç görünümü tasarlanmıştır. Sonuç görünümünde listelenen sonuçlarla ilgili olan kod kesimleri editör üzerinde işaretlenerek kullanıcıya gösterilmektedir.

1. Giriş

Kaynak kod, yazılımın anlaşılmasında baskın rol oynayan, birçok durumda bakım işlemlerinde programcıların ulaşabileceği tek uygulama yapısıdır. Yazılım endüstrisinde, yazılım hakkında çok basit bazı bilgilerin elde edilmesinden, karmaşık çıkarımların otomatik olarak oluşturulmasına kadar bir çok etkinliğin temelinde kaynak kodun sorgulanabilmesi ihtiyacı vardır. Sim, Clarke ve Holt'un çalışmasında [1], yazılım geliştiricilerin kod sorgulama alışkanlıklarını konu alan bir araştırma yapılmış, sorgulamanın en çok hata belirleme, kodun yeniden kullanımı (herhangi bir işlevselliği gerçekleştiren kod kesiminin aranması), programı anlama (ilgilenilen metod veya değişken bildirimlerinin ve bunların kod üzerindeki kullanımlarının incelenmesi), etki analizi (yapılacak bir değişikliğin kodun diğer kesimlerini nasıl etkileyeceğinin incelenmesi), kod temizleme (*main* metod tarafından dolaylı yada dolaysız olarak hiç çağrılmayan metodların belirlenmesi) gibi etkinliklerde ihtiyaç duyulduğu sonucuna varılmıştır.

Kaynak kod sorgulama ve arama, birçok yazılım mühendisliği aracında etkin biçimlerde kullanılıyor olmasına rağmen; amaç, hedef ve stratejiler bağlamında halen yenilikçi yorumlara açık bir konu olarak görülmektedir [2]. Yazılım geliştirme araçları (Eclipse, Visual Studio vb.), geliştiricilerin başlıca ihtiyaçlarını karşılamak üzere, kod üzerinde sorgulama imkânları sunarlar. Fakat bu ortamlardaki sorgu yapıları sabit olup, geliştiricilerin yeni sorgu türleri tanımlayıp, kullanabilmelerini sağlayacak biçimde esnek değilleridir. Örneğin Eclipse, Java programcılarına; herhangi bir sınıf veri üyesine (class data field) hangi kod kesimlerinde okuma ve yazma amaçlı erişimler yapıldığı, proje içerisindeki herhangi bir arayüzü hangi sınıf yada sınıfların gerçekleştirdiği gibi sorgulamaları yapabilmesine imkân tanır. Ancak bu tür sorgu kalıpları sabittir ve sayıları sınırlıdır.

Sonuç olarak, yazılım geliştirme ortamları için esnek sorgulama ihtiyacı önemli bir araştırma konusu haline gelmiş, bu konuda birçok çalışma yapılmış ve ilerleyen kesimde özetlenmiştir. Yapılan çalışmaların genel özelliklerine bakıldığında, iki önemli nokta üzerinde yoğunlaşıldığı görülür. Bu noktaların ilki, kaynak kod bilgilerinin saklanma biçimi ve kullanılan bilgi tabanının yapısı, ikincisi ise bu bilgi tabanı üzerinden istenilen sorgulamaların yapılabilmesine imkân verecek olan sorgulama mekanizmasıdır.

İlgili en eski çalışmalardan olan Linton'un OMEGA [3] sisteminde, program bilgileri INGRES adındaki veritabanı sisteminde tutulmaktadır. Sistem, kaynak kod hakkındaki 58 farklı ilişkiyi ve bu ilişkilerle ifade edilebilecek ayrıntılı bilgileri saklayabilmektedir. Sorgulamalar ise SQL benzeri bir dil olan QUEL sorgulama dili kullanılarak gerçekleştirilebilmektedir. Bu yöndeki sonraki çalışma olan CIA [4] ise, OMEGA'yı performans yönünden geliştirmiştir. CIA'da veritabanında kullanılan ilişki sayısı azaltılmış, böylece sorgu sonuçlarının üretilmesindeki performans arttırılmıştır. Bu iki çalışma günümüzdeki tüm beklentileri karşılamaktan uzak olsalar da, yazılım endüstrisinde program bilgilerinin veritabanında saklanması akımına öncülük etmişlerdir. XL C++ Browser [5], kaynak kod sorgulamada mantıksal programlama dili kullanımının gerçekleştirildiği ilk çalışmalardan biri olmuştur. Uygulama bir Prolog sisteminin üzerine kurulmuş, kaynak kod hakkındaki tüm olgular (facts) ana bellekte tutulmuştur. ASTLog [6], tanımlanan sorguların C++ soyut sözdizim ağaçları üzerinde çalıştığı, diğer bir mantıksal progamlama dilidir. Hızlı sonuç üretme özelliğinden dolayı, Microsoft tarafından bir çok projede kullanılmıştır. Çalışmamıza ana örnek olarak gösterebileceğimiz Jquery [7], Eclipse geliştirme ortamına eklenti olarak geliştirilmiş bir kod sorgulama aracıdır. Kod bilgilerinin oluşturulmasında ve sorgulanmasında Prolog benzeri bir mantıksal programlama dili olan TyRuBa kullanılmıştır. Yine ana örneklerden biri olan CodeQuest [8] kapsamında, kod bilgilerini saklamak için bir ilişkisel veritabanı sistemi, sorgulama için ise Prolog'un bir alt türevi olan Datalog kullanılmıştır. CodeQuest, ölçeklenebilirlik ve esnek sorgulama açısından önemli bir çalışmadır.

Kaynak kodun gösterimi, taranması ve sorgulanması alanlarında yapılmış çok sayıda çalışma bulunmaktadır. Bu

kısımda verdiğimiz örnekler, bizim çalışmamıza en yakın olarak değerlendirdiklerimizdir.
Bu çalışma kapsamında, CODONTO adındaki Java kodu sorgulama aracı, Eclipse geliştirme ortamına eklenti olarak geliştirilmiştir. Yukarıda bahsedilen çalışmalardan farkları, bilgi tabanı olarak OWL ile gösterilmiş bir ontoloji, sorgulama dili olarak ta SPARQL kullanılmış olmasıdır. Bilgisayar bilimlerinde ontoloji, herhangi bir alan kapsamındaki kavramlar ve bu kavramlar arasındaki ilişkileri temsil eden bir veri modeli olarak tanımlanabilir.
Bilgi tabanı olarak OWL ontolojisi kullanmamızdaki temel nedenler:

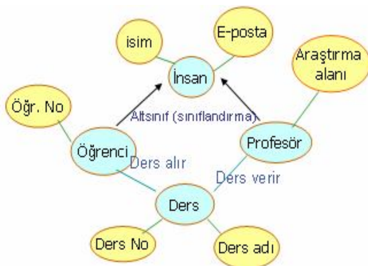
- nesne yönelimli bir programlama dili olan Java için, temel kaynak kod kavramları (paket, sınıf, metod gibi) ve bunlar arasındaki ilişkilerin ifade edilebilmesi,
- Sparql ile sorgulama imkanı sunması,
- sağladığı geçişken (transitive) veya birbirini kapsayan ilişkiler gibi imkanlar ile dolaylı çıkarsamalara imkan vermesi ve özyinelemeli sorgular yazmayı gereksiz kılması
- var olan kavramların mantıksal işleçler ile birbiri cinsinden ifade edilebilmesi gibi güçlü ifadesel özelliklere sahip olması
- OWL ontoloji ve olgularının paylaşılabilme özelliğinde olması

olarak gösterilebilir.

2. Ön bilgi

Anlamsal web ve ontoloji

Anlamsal web bilgilerin, bilgisayar ve insanların işbirliği içerisinde çalışabilmesine imkân verecek şekilde, iyi tanımlanmış anlamlarda verildiği bir WWW (World Wide Web) eklentisidir [9]. Tüm anlamsal web uygulamalarının temelinde ontoloji kullanımı bulunmaktadır. Ontoloji, herhangi bir alana ait kavramsal modelin, açık ve biçimsel tanımıdır [10]. Burada iki önemli nokta üzerinde durulmaktadır. İlki, kavramsallaştırma biçimseldir, dolayısı ile bilgisayar tarafından anlaşılabilir. İkincisi ise, ontoloji herhangi bir ilgi alanı için tasarlanır.
Ontolojiler kavramlar (sınıflar), ilişkiler (nitelikler), olgular ve aksiyomlardan oluşur. Şekil 1'de basit bir ontoloji kesiti verilmiştir.



Şekil 1: Basit bir ontoloji kesiti

Şekil 1 için, "Öğrenci" bir kavram (sınıf), "Öğr. No" ve "Ders alır" birer ilişki (nitelik), "Matematik" adındaki bir ders olgu, "Öğrenci sınıfı, İnsan sınıfının bir altsınıfıdır" ifadesi ise bir aksiyom örneği olarak gösterilebilir.

RDF (Resource Description Framework), RDFS (RDF Schema) ve OWL (Web Ontology Language)

RDF, World Wide Web üzerinde bulunan kaynaklar hakkındaki bilgileri ifade edebilmek için geliştirilmiş bir veri modelidir. Bu model, varlık-ilişki modelinde olduğu gibi bir kavramsal modelleme yaklaşımı olup, RDF terminolojisinde üçlüler (triples) olarak adlandırılan, kaynak-özellik-değer üçlemeleri ile ifade edilir. Kaynak, üzerinde konuşulan herhangi bir varlıktır (yazar, kitap, yer, kişi gibi). Her kaynak bir URI'ye (Uniform Resource Identifier) sahiptir. URI, kaynak için biricik olan değerdir. Bu değer bir internet adresi olabileceği gibi bir kimlik numarası da olabilir. Özellikler, özel türde kaynaklar olup, yine kaynaklar arasındaki ilişkiyi tanımlar. Değer, kaynakların özelliklerinin aldığı değerdir. Basit veri türünde olabileceği gibi, başka URI'lerde değer olarak kullanılabilir. RDF ifadelerinin XML' de yazımı, gösterimi, ve uygulamalar arası taşınması için RDF/XML sözdizimi standardı kullanılır.
RDFS, RDF veri modelini genişleten bir tür sistemi olup, RDF ile ifade edilen modelin sözlüğünün oluşturulmasında kullanılır. Bu sözlükte, ilgili alanda kullanılacak olan varlıklar, varlıklar arasındaki ilişkiler, özellikler, özellikler arasındaki ilişkiler ve özelliklerin alabilecekleri değerler tanımlanır. Şekil 2'de bir RDFS örneği Türkçe'ye çevrilerek verilmiştir.

```
<?xml version="1.0"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xml:base="http://www.cicekler.fake/cicekler#">
<rdf:Description rdf:ID="Çiçek">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf
     -schema#Class"/>
</rdf:Description>
<rdf:Description rdf:ID="Papatya">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf
    -schema#Class"/>
         <rdfs:subClassOf rdf:resource="#Çiçek"/>
</rdf:Description>
</rdf:RDF>
```

Şekil 2: RDFS örneği[11]

Örnekte "Papatya, bir Çiçek'tir" ifadesi modellenmiştir. Bu ifadede "Çiçek" ve "Papatya" varlık adları olup, bu varlılar arasında kalıtım (is-a) ilişkisi vardır.
RDFS, RDF sözlükleri oluşturmak için basit yetenekler sunmaktadır. Gelişmiş ontolojiler oluşturabilmek için bu yetenekleri genişleten üst seviye dillere ihtiyaç duyulmuştur. DAML, OIL ve OWL gelişmiş ontoloji dilleri olup, çokluk kısıtlamaları, geçişken ve simetrik özellikler, ayrık sınıflar gibi tanımlamalara imkan verirler. [12]
OWL, W3C (World Wide Web Consortium) tarafından tanımlanmış bir gösterim dilleri ailesi olup, OWL Lite, OWL DL ve OWL Full olmak üzere üç türü vardır. Çalışmamızda kullandığımız OWL DL, tanımlama mantığı (description logic) olarak bilinen birinci dereceden mantık (first order logic) kuralları üzerine geliştirilmiştir. [13]

SPARQL (SPARQL Query Language for RDF)

SPARQL anlamsal web için bir sorgulama dili ve veri erişim protokolüdür. W3C tarafından RDF veri modeli için tanımlanmıştır. RDF sorgulama dilleri üzerindeki çalışmalar son birkaç yıldır devam etmektedir. Bu süreç içerisinde RDQL, Squish, Versa gibi farklı yaklaşımların kullanıldığı diller geliştirilmiştir. RDQL ve Squish'e benzer olarak SQL sözdizimini örnek alan bir dil olan SPARQL kendisine geniş bir kullanım alanı bulmuştur. RDF ve OWL sorgulama araçlarının büyük çoğunluğu SPARQL desteği sunmaktadır.

```
# ön ek bildirimleri
    PREFIX foo: <http://example.com/resources/> ...
# sonuç yancümlesi
    SELECT ...
# sorgu kalıbı
    WHERE {    ...   }
# sorgu değiştiricileri
    ORDER BY ...
```

Şekil 3: SPARQL sorgu biçimi [14]

Şekil 3'te verilen SPARQL sorgu biçimi bileşenleri şu şekilde açıklanabilir:

- URI'leri, yani biricik kaynak tanıtıcılarını kısaltmada kullanılan *ön ek bildirimleri*,
- sorgudan elde edilmek istenen bilgilerin belirtildiği *sonuç yancümlesi*,
- veri kümesi üzerinde yapılacak sorgunun belirtildiği *sorgu kalıbı*,
- sıralama, bölme gibi sonuç kümesi üzerinde düzenleme yapmaya yarayan *sorgu değiştiricileri*.

Elementlere ait isim, sembol, renk, atom numarası ve kütle bilgilerinin yer aldığı periyodik tablo veri seti [15] üzerinden, elementlere ait isim, atom numarası ve renk bilgilerini, atom numarasına göre sıralanmış biçimde çekebilmek için tanımlanmış örnek sorgu şekil 4'te verilmiştir.

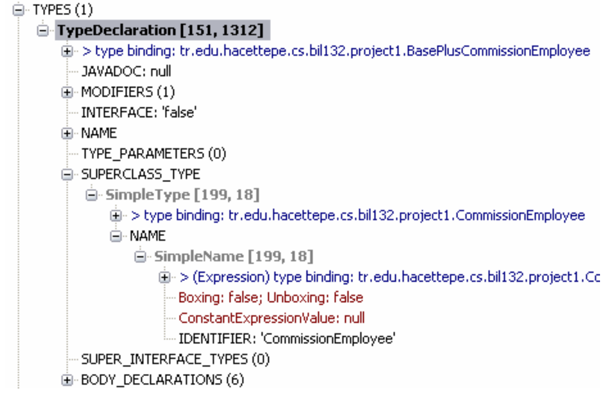
```
PREFIX table:
<http://www.daml.org/2003/01/periodictable/PeriodicTable#>
SELECT ?name ?number ?color
WHERE
{
    ?element table:name ?name.
    ?element table:symbol ?symbol.
    ?element table:atomicNumber ?number.
    OPTIONAL { ?element table:color ?color. }
}ORDER BY ?number
```

Şekil 4: SPARQL sorgu örneği[16]

SPARQL herhangi bir sorguyu, yapısal yada yarı-yapısal RDF çizge-kalıpları üzerinden eşleştirme yolu ile sonuçlandırır. Periyodik tablo örneği için, bazı elementlerin renk bilgilerinin olmadığı bilinmektedir. Buna rağmen şekil 4'te belirtilen sorguda, renk bilgileri olmayan elementlerin de sonuç kümesi içinde yer alması OPTIONAL anahtar sözcüğü ile sağlanmaktadır.
Benzer şekilde UNION, SPARQL için önemli bir işleç olup, sonuç kümeleri üzerinde mantıksal VEYA işlemini gerçekleştirir.

Soyut sözdizim ağacı (Abstract syntax tree-AST)

Soyut sözdizim ağacı, Eclipse gibi geliştirme ortamlarında, yeniden düzenleme (refactoring), hızlı onarım (quick fix), hızlı yardım (quick assist) gibi yardımcı araçlar için temel teşkil eden bir altyapı ve kaynak kodun ağaç gösterimidir. Ağaç yapısı, düz metin-tabanlı yapıya göre, kodu çözümleme ve değiştirme bakımından daha güvenilir ve uygundur. Eclipse ortamında java kaynak kodu soyut sözdizim ağaçlarının nasıl göründüğü, *ASTView* [17] eklentisi ile incelenebilir. Şekil 5'de *ASTView* eklentisi tarafından oluşturulmuş, bir Java kaynak kütüğüne ait soyut sözdizim ağacı görülmektedir.



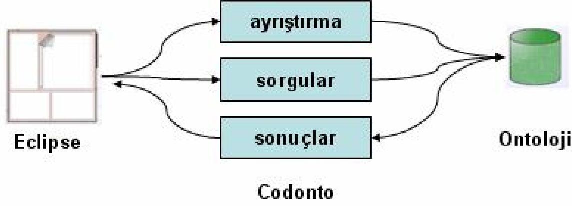
Şekil 5: Soyut sözdizim ağacı örneği

Herbir Java kaynak kütüğü, AST düğümlerinden oluşan bir ağaç olarak temsil edilebilir. Ağaç üzerindeki tüm düğümler, *ASTNode* sınıfının alt sınıflarıdır. Java programlama dilindeki her kod biriminin ağaç üzerinde bir karşılığı vardır ve her birim bir *ASTNode* alt sınıfı ile temsil edilir. Örneğin, metod bildirimleri *MethodDeclaration* altsınıfı ile, değişken bildirimleri ise *VariableDeclarationFragment* altsınıfı ile temsil edilirler. En sık kullanılan düğümlerden bir tanesi *SimpleName* düğümüdür. Örneğin, $i = j + 7;$ gibi bir atama işleminde, i ve j birer *SimpleName* türünde düğüm ile temsil edilirler.
Kaynak kodun, soyut sözdizim ağacı üzerinden her türlü analizi yapılıp, istenilen tüm bilgi ve çıkarımlar elde edilebilir. Bunun için Eclipse'in sunduğu kütüphaneler (org.eclipse.jdt.core gibi) kullanılıp, amaca uygun yazılım geliştirilmelidir.
Soyut sözdizim ağacı, kaynak kodun en alt düzeyde detaylandırılmış mantıksal modelidir ve kod ile ilgili tüm işlemlerde doğrudan kullanılabilir.
Bizim çalışmamızda, proje kapsamındaki herbir kaynak kod kütüğüne ait soyut sözdizim ağacı, geliştirdiğimiz ayrıştırıcı tarafından analiz edilip, kod hakkındaki amaca uygun tüm kavram ve ilişkiler elde edilerek ontoloji olguları oluşturulur. Bu şekilde kod bilgileri daha kolay sorgulanabilir bir model üzerine oturtulmuş olur.
Proje içerisindeki herhangi bir sınıfa ait "A" ismindeki soyut (abstract belirtece sahip) bir metodu geçersiz kılan tüm metodlara ulaşabilmek için soyut sözdizim ağacı kullanılarak geliştirilen yazılım hem sadece belli bir amaca hizmet edecek, hem de programlama açısından karmaşık işlemleri gerektirecektir. Bunun yerine proje kapsamındaki tüm metod olgularının, değişim belirteçleriyle ve ait oldukları sınıflarla

ilişkilendirildiği bir ontoloji olgusu üzerinden gerçekleştirilecek bir sorgu ile istenilen sonuç elde edilebilir.

3. CODONTO

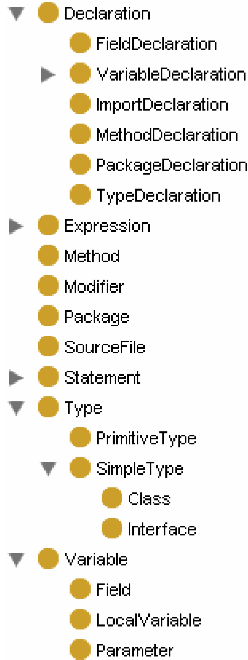
Bu bölümde, CODONTO'nun geliştirilme aşamaları ayrıntılı biçimde anlatılmış, uygulamaya ait bazı örnekler sunulmuştur. Şekil 6'da Codonto çözüm modeli görülmektedir.



Şekil 6: Codonto çözüm modeli

Java ontolojisi

İlk olarak, Java dil belirtimi 3 (JLS 3) referans alınarak, kaynak kod kavram ve ilişkilerinin temsil edildiği bir OWL ontolojisi oluşturulmuştur. Ontolojinin oluşturulmasında OWL, SPARQL gibi teknolojiler için programlama arayüzü sağlayan ve anlamsal web uygulamaları geliştirmek için tasarlanmış Jena [18] kütüphanesi kullanılmıştır.



Şekil 7:Java ontolojisi sınıfları

Sınıfları şekil 7'de görülen ontolojide, Java diline ait *Class, Interface, Type, Field, Parameter* gibi kavramlar, hiyerarşik bir şekilde sınıflandırılmışlardır. Örneğin *Class* ve *Interface* sınıfları, *SimpleType* sınıfının alt türleridir. Bu şekildeki bir hiyerarşik yapı, herhangi bir java kütüğü içerisinde tanımlanmış tüm basit türlerin listelenmesini sağlayacak olan SPARQL sorgusunun, ilgili Java kütüğü içerisinde tanımlanmış tüm sınıf ve arayüzlerin listelenmesi ile sonuçlanmasını sağlayacaktır.

Ontolojiyi oluşturan diğer elemanlar, kavramlar arasındaki ilişkileri temsil eden nesne nitelikleri (object properties) ve herbir OWL sınıfını tanımlamada kullanılan veri türündeki niteliklerdir (datatype properties). Tablo 1'de, tanımlanan bazı nesne ve veri türü nitelikleri görülmektedir. Tablo içerisindeki ilk beş satırda bulunan nitelikler veri türü, diğerleri nesne türü niteliğidir.

OWL Nitelik İsmi	Açıklama
Interface_name	Arayüze ait isim bilgisi
Parameter_length	Parametrenin karakter olarak uzunluğu
Parameter_start_pos	Parametrenin kaynak kütüğü üzerindeki başlangıç konumu
Method_source_file	Metodun tanımlandığı kaynak kütüğünün ismi
Method_is_constructor	Metodun yapıcı olup olmadığını gösteren bilgi
Method_has_Parameter	Metod ve parametrelerini ilişkilendirir.
Parameter_has_Type	Parametrelerin türünü belirtir. int, String gibi.
Class_SuperInterFaceType_ Interface	Sınıf ve gerçekleştirdiği arayüzü ilişkilendirir.
Method_returntype_Type	Metod'un dönüş tipini belirtir.
Method_implements_Method	Metodu implement ettiği metod ile ilişkilendirir.
Method_has_Modifier	Metodu sahip olduğu erişim belirteçleri ile ilişkilendirir.
Class_SuperClassType_Class	Sınıf ile altsınıfını ilişkilendirir.

Tablol 1: Java Ontolojisi nitelik tablosu

Örneğin, *Fied_has_Type* nesne niteliği, *Field* ve *Type* olarak tanımlanmış kavramlar arasındaki bir ilişki olup, kod içerisindeki herbir sınıf veri üyesi ile bu veri üyelerine ait olan tür bilgilerini ilişkilendirmek için kullanılır. 'Parameter_length' veri türü niteliği ise, herbir metod parametresinin ait olduğu kod kütüğü içerisindeki uzunluk bilgisini ifade eder.

Proje içerisinde, "getName" isminde birden fazla metod olabilir. Bu metodları temsil eden tüm olgular için "getName" ismi kullanılamaz. Çünkü, ontoloji örneği içerisindeki tüm olgu isimleri biricik olmalıdır. Bu nedenle, herbir olgu için farklı isim üretilmesini sağlayan bir yöntem geliştirilmiştir. Oluşturulan her olgu için, olgunun ait olduğu sınıf ismi ile rastgele ve daha önce başka bir olgu tarafından kullanılmamış bir sayının birleşimi, isim olarak atanır ("Method_3422" gibi).

Java projesi ontoloji olgusunun otomatik olarak oluşturulmasını sağlayan ayrıştırıcı tasarımı

Java dili için bir ontoloji oluşturulduktan sonra, üzerinde çalışılan herhangi bir Java projesi ontoloji olgusunun otomatik olarak oluşturulmasını sağlayan bir ayrıştırıcı geliştirilmiştir.
Ayrıştırıcı tasarımında Eclipse'in sunduğu "java modeli" ve "AST" yapıları kullanılmıştır. Eclipse Java modeli, java eleman ağacının incelenebilmesi için arayüz sunar. Java eleman ağacı, herhangi bir java projesinin sadece üst düzey elemanlarını içeren bir yapıdır. Bunlar kod kütüğünü temsil eden derlem birimi (compilation unit), paket, tip (sınıf, arayüz gibi), method, veri üyesi gibi elemanlardır. [19] Herhangi bir Java kodunun AST'si *ASTParser* sınıfının *createAST* metodu tarafından tarafından oluşturulur. *ASTParser* sınıfına kaynak kod dosyası, *ICompilationUnit* türünde parametre olarak geçilir. *ICompilationUnit*, java modelde kaynak kod dosyasında karşılık gelir.

Ayrıştırıcının çalışma biçimi şu şekildedir:

- İşleme sokulacak olan java projesi için Eclipse'in sunduğu java modeli kullanılarak proje içerindeki tüm .java kütükleri içeren paketler, sonra da bu paketler içerisindeki tüm derlem birimleri (kod kütükleri) çekilerek bir listeye atılır.
- Liste içerisinde bulunan herbir derlem birimi sıra ile işleme sokulur.
- İlk olarak, sıradaki derlem birimi için *ASTParser* sınıfı tarafından oluşturulan soyut sözdizim ağacının kök düğümü *derlem birimi yapısal özellik ayrıştırıcısı* olarak nitelendirilebilecek sınıfa parametre olarak geçilir.
- Herbir yapısal özellik ayrıştırıcısı sınıfı farklı türde AST düğümünü analiz etmeye yarar. Bunlardan *derlem birimi yapısal özellik ayrıştırıcısı* sınıfı, ilgili derlem birimini ontolojideki kavram ve ilişkiler bazında analiz ederek, ilgili OWL olgularının oluşturulmasını sağlayacak olan sınıflardan ilkidir. Bu sınıf, *CompilationUnit* türündeki AST düğümünü parametre olarak alır ve düğüme ait tüm yapısal özellikleri çeker. Bunlar, paket bildirimini içeren "package", içe aktarma bildirimlerini içeren "imports" ve sınıf ve arayüz bildirimlerini içeren "types" özellikleridir.
- Aktif olan *derlem birimi yapısal özellik ayrıştırıcısı* sınıfı parametre olarak aldığı *CompilationUnit* türündeki AST düğümünün özelliklerini çektikten sonra bu özelliklerden elde ettiği değerleri kullanarak ilgili türdeki olguları oluşturur. Örneğin, işleme sokulan derlem birimi içerisinde bir sınıf tanımlanmış ise "types" özelliğinin değeri *TypeDeclaration* türündeki AST düğümü olacaktır. Bu durumda ilk olarak *TypeDeclaration* türünde bir OWL olgusu yaratılacaktır. Ayrıştırma işlemine *tip bildirimi yapısal özellik ayrıştırıcısı*, *TypeDeclaration* türündeki AST düğümünü parametre alarak devam eder.
- *Tip bildirimi yapısal özellik ayrıştırıcısı*, parametre olarak aldığı *TypeDeclaration* düğümünün yapısal özelliklerini çeker ve herbir özellik için gerekli işlemleri gerçekleştirir. Örneğin, "name" basit türde bir özellik olup, sınıfın ismini belirtir. Sıra bu özelliğin işlenmesine geldiğinde ilk olarak *Class* türünde OWL olgusu yaratılır. Sonra, bu sınıfa ait isim, uzunluk, başlangıç pozisyonu, paket ismi, ait olduğu kod kütüğü ismi gibi bilgiler çekilerek ve ontolojideki ilgili veri türü nitelikleri kullanılarak gerekli OWL ifadeleri oluşturulur ve ontolololoji olgusuna eklenir.
- *Tip bildirimi yapısal özellik ayrıştırıcısı* sınıfının işlemesi gereken diğer özelliklerden bazıları şunlardır: "superclassType", "superInterfaceTypes" ve "bodyDeclarations". Bunlardan "superclassType" özelliği, işlenen sınıf başka bir sınıftan türememiş ise "null" değerine, türemiş ise *SimpleType* türündeki AST düğümü türünde bir değere sahiptir. Bu durumda, *basit tür yapısal özellik ayrıştırıcısı* sınıfı devreye girer ve türetilen sınıf için bir OWL olgusu oluşturur. Veri türü niteliklerini kullanarak bu sınıfı tanımlamak için gerekli OWL ifadelerini oluşturabilmek için düğümün özelliklerini çeker. "superInterfaceTypes" özelliği için ise yine *basit tür yapısal özellik ayrıştırıcısı* sınıfı kullanılır ve bu sefer işlenen sınıfın gerçekleştirdiği tüm arayüzler için OWL olguları oluşturulur. "bodyDeclarations" özelliği *tip bildirimi yapısal özellik ayrıştırıcısı* sınıfının işlediği en kapsamlı özelliktir. Burada, metod ve veri üyesi bildirimleri analiz edilir. Örneğin herbir metod bildirimi için yapılacak işlemler, ilk olarak *MethodDeclaration* türünde OWL olgusu oluşturulup, özelliğin sahip olduğu *MethodDeclaration* türündeki AST düğümünden metoda ait kod kütüğü üzerindeki uzunluk, başlangıç pozisyonu gibi bilgilerin çekilerek ve ilgili OWL veri türü nitelileri kullanılarak, metod bildirimi olgusunu tanımlayan OWL ifadelerinin oluşturulmasıdır. Bu işlemler yapıldıktan sonra, belirtilen metodun sahip olduğu özelliklerin analiz edilebilmesi için (parametreler, dönüş tipi, erişim belirteçleri gibi) *method bildirimi yapısal özellik ayrıştırıcısı* sınıfı devreye girer ve ayrıştırmayı devam ettirir.
- Anlaşılacağı üzere, ayrıştırma işlemi farklı türde AST düğümlerini işleyebilecek şekilde tasarlanmış yapısal özellik ayrıştırıcısı sınıfları tarafından sıradüzensel biçimde gerşekleştirilir. Her ayrıştırıcı sınıf, işlediği AST düğüm türü ile ilgili olan OWL sınıf ve veri türü niteliklerini kullanarak gerekli işlemleri gerçekleştirir ve java projesi OWL olgusunu günceller.
- Dikkat edilirse buraya kadar olan aşamalarda sadece OWL olgularının yaratılıp, ontolojideki veri türü niteliklerinin kullanılarak bu olguları tanımlayan OWL ifadelerinin oluşturulmasından bahsedilmektedir. Tüm derlem birimlerinin ayrıştırılıp, proje kapsamındaki tüm olgular oluşturulduktan sonra bu olgular arasındaki ilişkilerin kurulması, bunun için de analiz işleminin baştan alınması gerekir. İkinci kez yapılacak olan analizde kullanılmak üzere, olguların oluşturulmasında kullanılan yapısal özellik ayrıştırıcı sınıflara benzer şekilde çalışan ve var olan olgular arasındaki ilişkileri kuracak olan sınıflar tasarlanmıştır. Bu sınıflar da sıradüzensel bir şekilde çalışırlar ve hangi durumlarda hangi tür olgular arasında hangi ilişkilerin kurulacağı bilgileri ile kodlanmışlardır.
- Son olarak, kod içerisindeki kavramlar ve bu kavramlar arasındaki ilişkilerin temsil edildiği ontoloji olgusu, kalıcı olarak OWL formatında kaydedilir.

İlk olarak tüm OWL olgularının oluşturulması, sonraki adımda ise analizin baştan alınarak, olgular arasındaki ilişkilerin kurulması şu şekilde açıklanabilir: Belirtilen iki işlemin paralel olarak gerçekleştirilmesi, bazı problemlere yol açabilecektir. Örneğin, proje içerisindeki bir A sınıfı, bir B sınıfını kalıtıyor olabilir. Bu durumda ontoloji örneğinde,

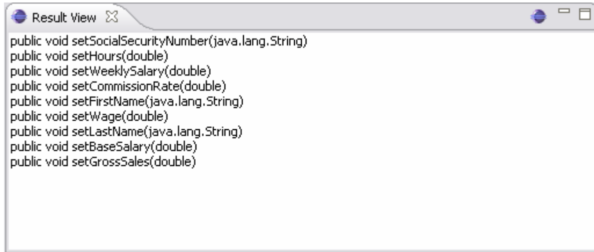
A sınıf olgusu ile B sınıf olgusu, *Class_SuperClassType_Class* nesne niteliği ile ilişkilendirilmelidir. Fakat, analiz sırasında B sınıf olgusunun henüz oluşturulmamış olması, bu ilişkinin kurulmasını engeller. Bu nedenle ilk olarak tüm olguları oluşturup, sonraki adımda olgular arasındaki ilişkileri kurmak uygun bir yöntem olacaktır.

Sorgu ve sonuç görünümleri ve örnek sorgulamalar

Java Projesi ontoloji olgusunun otomatik olarak üretilip, OWL formatında kaydedilebilmesinden sonra, Eclipse geliştirme ortamı ile ontoloji olgusu arasında bir sorgulama arayüzüne ihtiyaç duyulmuştur. Bu arayüz, eklenti olarak geliştirilen sorgu ve sonuç görünümleri ile sağlanmıştır.

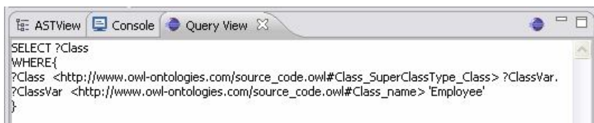
Şekil 8: Sorgu görünümü

SPARQL sorguları şekil 8'deki gibi sorgu görünümüne düz metin olarak yazılmaktadır. Örnekteki sorguda, "yapıcı olmayan ve parametre alan tüm metodlar"ın listelenmesi istenmektedir. Şekil 9'da, ilgili sorgunun işletilmesi sonucunda elde edilen sonuç listesi, sonuç görünümü üzerinde görülmektedir.



Şekil 9: Sonuç görünümü

Çıkarsamanın örneklendiği diğer bir sorgu şekil 10'da görülmektedir. Bu sorguda, "Employee sınıfından dolaylı ve dolaysız olarak türeyen tüm sınıflar"ın listelenmesi istenmektedir.



Şekil 10: Çıkarsama örneği için sorgu görünümü

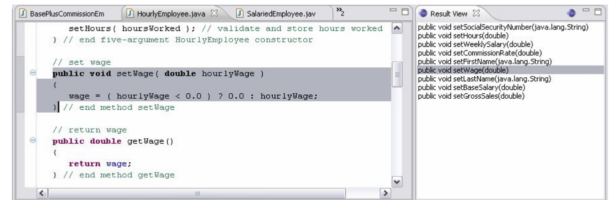
Şekil 11 'de, şekil 10'daki sorgunun işletilmesi sonucunda elde edilen sonuç listesi görülmektedir. Sonuçlar içerisinde bulunan "SalariedEmployee", "HourlyEmployee" ve "CommissionEmployee" sınıfları "Employee" sınıfından doğrudan türeyen sınıflardır. "BasePlusCommissionEmployee" sınıfı ise "CommissionEmployee" sınıfından doğrudan, "Emplyee" sınıfından ise dolaylı olarak türemiştir. Bu şekilde bir sonucun elde edilmesinde "Class_SuperClassType_Class" nesne niteliğinin geçişken (transitive) olarak tanımlanması etkili olmuştur.

Şekil 11: Çıkarsama örneği için sonuç görünümü

Aracın cevap verebildiği diğer sorgulara: "herhangi bir arayüzünü gerçekleştiren sınıflar", "herhangi bir paket içerisinde bulunan tüm sınıf ve arayüzler", "erişim belirteci *public* olup *final* olmayan tüm metodlar", "herhangi bir soyut metodu geçersiz kılan tüm metodlar", "belirtilen türde sınıf veri üyesine sahip tüm sınıflar" vb. örnek olarak verilebilir.

Sorgu görünümü üzerine girilen herhangi bir sorgunun işlenip, listelenmesine kadar olan süreç şu adımlarla açıklanabilir:

- Sonuç görünümünün sağ üst köşesinde bulunan düğme ile eklenti tetiklenir.
- Eklenti, sorgu görünümü üzerindeki sorgu metnini karakter dizisi olarak alıp ayrıştırarak, sonuç görünümü üzerinde listelenecek olan kod elemanı türünü çeker. Şekil 8'deki örnek için, listelenecek kod elemanları türünün *Method* olduğu görülmektedir.
- Jena kütüphanesinde bulunan *QueryExecution* sınıfının, *execSelect()* metodu ile sorgu işletilerek sonuç kümesi elde edilir.
- Sonuç kümesindeki herbir olgu için, sonuç görünümünde olguyu temsil edecek olan bir nesne oluşturulur. Bu nesne içerisinde olguya ait isim, kod kütüğü üzerindeki başlangıç konumu, uzunluğu, ait olduğu kaynak kütüğü ismi gibi bilgiler bulunmaktadır. Tüm bu bilgiler, sonuç görünümündeki herbir elamanın kod editörü üzerinde gösterilmesinde kullanılır.
- Olgular için oluşturulan nesnelerin tümü, sonuç görünümünün içerik sağlayıcı (content provider) sınıfına geçilerek, olgu isimlerinden oluşan bir listenin gösterilmesi sağlanmış olur.



Şekil 12: Seçilen sonucun editör üzerinde gösterilmesi

Sonuç görünümü üzerindeki listede bulunan elemanlardan birine çift tıklandığında, seçilen elemanın bulunduğu kod

kütüğü içerisindeki yeri işaretlenmiş olarak editör üzerinde gösterilir.

4. Sonuç

Bu çalışmada, CODONTO adındaki ontoloji-tabanlı kaynak kod sorgulama aracı, Eclipse geliştirme ortamına eklenti olarak geliştirilmiştir. Bilgi tabanı olarak OWL ontolojisi, sorgulama dili olarak ta SPARQL kullanılmış ve Java programlama dili ile geliştirilmiş projeler üzerinde esnek sorgulamaların yapılabilmesi amaçlanmıştır.

Çalışmamızda, OWL'nin ifadesel olarak güçlü bir üst model olması, tanımlama mantığı üzerine kurulduğundan çıkarsamalara imkan vermesi ve sorgulama imkanı sunması gibi özellikleri kullanılmış, herhangi bir java projesi olgusuna sadece projenin geliştirildiği ortamda (yerelde) ulaşma imkanı sunulmuştur. Bunun yanında, OWL ontoloji ve olgularının paylaşılabilmesi özelliklerinden faydalanılarak, geliştirdiğimiz araca web üzerinden gelen sorgulara cevap verip, ilgilenilen proje bilgilerinin sunulması yeteneğinin kazandırılması, çalışmamızın bir sonraki aşamasıdır.

Aracın esnek sorgulara cevap verebildiği görülmüştür fakat aynı amacı gerçekleştiren diğer uygulamalar ile karmaşıklık ve performans karşılaştırılması yapılmamıştır. Bu konudaki çalışmalarımız devam etmektedir.

5. Kaynakça

[1] S. E. Sim, C. L. A. Clarke & R. C. Holt, “Archetypal Source Code Searches: A Survey of Software Developers and Maintainers,” *International Workshop on Program Comprehension* (1998).

[2] Bajracharya, S., Ngo, T., Linstead, E., Rigor, P., Dou, Y., Baldi, P. ve Lopes, C. Sourcerer: A Search Engine for Open Source Code. International Conference on Software Engineering (ICSE 2007).

[3] Mark A. Linton. Implementing relational views of programs. In Peter B. Henderson, editor, Software Development Environments (SDE), sayfa 132–140, 1984.

[4] Yih Chen, Michael Nishimoto, ve C. V. Ramamoorthy. The C information abstraction system. IEEE Transactions on Software Engineering, 16(3):325–334, 1990.

[5] Shahram Javey, Kin'ichi Mitsui, Hiroaki Nakamura, Tsuyoshi Ohira, Kazu Yasuda, Kazushi Kuse, Tsutomu Kamimura, ve Richard Helm. Architecture of the XL C++ browser. In CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research, sayfa 369–379. IBM Press, 1992.

[6] Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In USENIX Conference on Domain-Specific Languages, sayfa 229–242, 1997.

[7] Doug Janzen ve Kris de Volder. Navigating and querying code without getting lost. In 2nd International Conference on Aspect-Oriented Software Development, sayfa 178–187, 2003.

[8] E. Hajiyev, M. Verbaere, ve O. de Moor. CodeQuest: scalable source code queries with Datalog. In D. Thomas, editor, Proceedings of ECOOP, volume 4067 of Lecture Notes in Computer Science, sayfa 2–27. Springer, 2006.

[9] T. Berners-Lee, J. Hendler, ve O. Lassila, “The Semantic Web,” Scientific Am., May 2001, sayfa 34–43.

[10] T.R. Gruber, “A Translation Approach to Portable Ontologies,” *Knowledge Acquisition*, vol. 5, no. 2, 1993, sayfa 199–220.

[11] http://www.w3schools.com/rdf/rdf_schema.asp

[12] http://www.w3.org/TR/rdf-primer/

[13] Davies J., Studer R. & Warren P. (2006) Semantic Web Technologies: trends and research in ontology-based systems. John Wiley & Sons Ltd., Chichester, sayfa 4.

[14] http://www.cambridgesemantics.com/2008/09/sparql-by-example/

[15] http://www.daml.org/2003/01/periodictable/PeriodicTable#

[16] http://www.xml.com/pub/a/2005/11/16/introducing-sparql-querying-semantic-web-tutorial.html

[17] http://www.eclipse.org/jdt/ui/astview/index.php

[18] http://jena.sourceforge.net/

[19] http://www.eclipse.org/jdt/core/index.php