Technical University of Crete

Department of Electronic and Computer Engineering

TOQL: Querying temporal information in ontologies

Master Thesis

Baratis Evdoxios

July 16, 2008

TOQL: Querying temporal information in ontologies

A thesis submitted
in partial fullfilment of the requirements
for the degree of

Master of Computer Engineering

by

Baratis Evdoxios

Abstract

We introduce TOQL, a query language for querying time information in ontologies. A distinguishing feature of TOQL is its ability to handle ontologies representing evolution of information in time (thus allowing time to affect the status of the described concepts). TOQL is designed as a high level query language that handles ontologies almost like relational databases. Queries in TOQL are issued as SQL-like statements involving time and high-level ontology concepts that vary in time. TOQL also supports expressions relating ontology concepts with time instances or time intervals. TOQL prevents users from being familiar with representation of time in ontologies. Besides TOQL syntax, this work demonstrates, full query functionality on ontologies in OWL. This includes query translation and execution of temporal queries along with a mechanism for representing time evolving concepts in ontologies. Although independent from TOQL, this work suggests a mechanism for representing time evolving concepts in ontologies based on the well known 4D perdurandist mechanism [21]. TOQL queries are translated into equivalent statements in SeRQL [11] (which are then executed on OWL). To show proof of concept, a real world temporal ontology is also implemented on which several TOQL example queries are processed and discussed. Query formulation and general user interaction with the ontology is facilitated using a Graphic User Interface (GUI).

Περίληψη

Παρουσιάζουμε την TOQL, μία υψηλού επιπέδου γλώσσα ερωτήσεων για οντολογίες που παριστάνει στατική αλλά και χρονική πληροφορία. Το κύριο χαρακτηριστικό, που ξεχωρίζει την TOQL από άλλες γλώσσες (π.χ., SPARQL, SeRQL), είναι η ικανότητά της να διαχειρίζεται οντολογίες σε που αναπαριστούν την εξέλιξη της πληροφορίας στον χρόνο (επιτρέποντας έτσι τον χρόνο να επηρεάσει την κατάσταση των κλάσεων που περιγράφονται στην οντολογία). Η TOQL έχει σχεδιαστεί ως μια γλώσσα ερωτήσεων υψηλού επιπέδου που χειρίζεται τις οντολογίες σχεδόν σαν σχεσιακές βάσεις δεδομένων. Οι ερωτήσεις στην TOQL τίθενται σαν τύπου SQL ερωτήματα που περιλαμβάνουν εκφράσεις χρόνου. Η TOQL επίσης υποστηρίζει την συσχέτιση των κλάσεων μιας οντολογίας με χρονικές στιγμές ή χρονικά διαστήματα. Στην TOQL ο χρήστης δεν χρειάζεται να είναι γνώστης του τρόπου με τον οποίο ο χρόνος αναπαρίσταται στις οντολογίες. Εκτός από την συντακτικό της TOQL, αυτή η δουλεία περιλαμβάνει μια εφαρμογή για την πλήρη υποστήριξη ερωτήσεων σε οντολογίες σε OWL. Αυτή η εφαρμογή περιλαμβάνει μετάφραση και εκτέλεση των ερωτημάτων, καθώς και ένα μηχανισμό για την αναπαράσταση των κλάσεων που εξελίσσονται στον χρόνο. Αν και ανεξάρτητο από την TOQL, η εφαρμογή ενσωματώνει τον μηχανισμό για την αναπαράσταση των χρονικά εξελισσόμενων κλάσεων σε οντολογίες που προτάθηκε από την IBM και θεωρεί ότι οι κλάσεις (έννοιες) έχουν τέσσερις διαστάσεις (four-dimensional (perdurantist) approach). Τα ερωτήματα σε TOQL μεταφράζονται σε αντίστοιχα ερωτήματα σε SeRQL (τα οποία στην συνέχεια εκτελούνται πάνω την οντολογία). Για να δείξουμε την ισχύ των παραπάνω, έχει επίσης υλοποιηθεί μια χρονική οντολογία, πάνω στην οποία εκτελούνται και συζητούνται αρκετά ερωτήματα σε TOQL. Η δημιουργία την ερωτημάτων και γενικά η επικοινωνία του χρήστη με την οντολογία διευκολύνεται με την χρήση ενός γραφικού περιβάλλοντος.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Ontologies represent a set concepts and the relationships between those concepts. Over the past few years there has been an extensive use of ontologies on domains such as artificial intelligence, software engineering and semantic web as a form of knowledge representation, replacing in many cases relational databases. Because of their increasingly important role, many ontology query languages have been proposed as tools for providing enhanced retrieval support on knowledge and for increasing the efficiency of knowledge interpretations by querying on entities, associations among entities or on properties of such entities represented in ontologies. The current state of the art requires one to submit a textual, description logic (DL) query or SQL-like query. However the logic and syntax of these querying languages necessitates a tedious effort from users before being able to write queries effectively. However, state-of-the-art languages has limited (if not at all) expressive power in handling time and concepts that vary over time in queries.

1.1 Problem Definition

A critical issue in Knowledge Representation (KR) domain is dealing with information that changes over time. Many Knowledge Representation applications need, not only to provide well organised data describing the current state of a domain, but also to provide data describing the domain's evolution. State-of-the-art information representation and reasoning methods have limited expressive power for describing real world changing processes. For example the (important and persistent) knowledge that a person will go through the stages of infant, adolescent and adult, or a company will be established, hire personnel and develop products which evolve as a result of time, cannot be adequately described using existing methods and is there-

fore unavailable to the experts. Ontology representation languages such as OWL are based on binary relations (relations connecting two instances with no time dimension) making the representation of time a difficult matter to deal with.

It is possible to enhance the capabilities of state-of-the-art information representation over semantic web and their support for information analysis and reasoning by exploiting the time dimension in the information possessed. This can be achieved by adding the concepts of time and change (evolution) in a rich semantics ontology representation enabling context aware information analysis and reasoning based on evolution over time.

Ontologies offer the means for representing high level concepts, their properties and their interrelationships. Dynamic or temporal ontologies will in addition enable representation of time evolving information in ontologies through e.g., versioning or the 4D perdurandist approach [21]. According to this approach all entities are perdurants, making no distinction between endurants (physical objects such as cars, companies, people) and occurants (events such as buying a car). The idea is that each entity is considered to be an event and has a start and an end point. An entity can be seen as a "space-time worm", with the slices of the worm being temporal parts (time slices) of the entity.A temporal ontology query language is then needed to exploit this information in searching for temporal concepts and time related information. This might not only increase the quality of searches but also add improved information interpretation capabilities to existing systems through statistical analysis, data mining and reasoning that involve time. Existing ontology query languages such as SeRQL or SPARQL fall short in handling time.

1.2 Proposed Solution

We introduce TOQL, a high-level query language for querying (time) information in ontologies. TOQL handles ontologies almost like relational databases. Queries in TOQL are issued as SQL statements involving time and high-level ontology concepts that vary in time. TOQL expands the capabilities of state-of-the-art ontology query languages such as SeRQL [11] and SPARQL [7] to handle real world changing information. Unlike SeRQL and SPARQL treats ontologies almost like relational databases, hiding from the user most of the peculiarities of an ontology. TOQL maintains the basic structure of an SQL language (SELECT, FROM,WHERE) and treats the classes and the properties of an ontology almost like tables and columns of a database. TOQL supports expressions relating concepts with time instances

or time intervals. This prevents users from being familiar with representation of time in ontologies. TOQL supports queries not only on static information in the static part of the ontology (as conventional query languages do) but also queries on time evolving information instantiated to the ontology (dynamic part). TOQL also introduces the Allen operators (BEFORE, AFTER, MEETS, METBY, OVERLAPS, OVERLAPPEDBY, DURING, CONTAINS, STARTS, STARTEDBY, ENDS, ENDEDBY, EQUALS) that allow comparisons between time intervals, and the operator AT(time point) or AT(time point, time point) that allows comparisons between an interval and a specific time interval or time point.

Besides TOQL syntax, this work demonstrates, full query functionality on ontologies in OWL. This includes query translation and execution of temporal queries along with a mechanism for representing time evolving concepts in ontologies. Although independent from TOQL, this work suggests a mechanism for representing time evolving concepts in ontologies based on the well known 4D perdurandist mechanism [21]. The 4D perdurandist mechanism is not part of the language, it is not transparent to the user (so the user need not be familiar with peculiarities of the underlying mechanism for time information representation). TOQL queries are first translated into equivalent statements in SeRQL (which are then executed on OWL). Based on the system's understanding of information, the system generates a projection (in time) of the evolution of the acquired ontology concepts.

To show proof of concept, a real world temporal ontology (for enterprise information) is also implemented on which several TOQL example queries are processed and discussed. Query formulation and general user interaction with the ontology is facilitated using a Graphic User Interface (GUI). The platform can take any temporal ontology as input and perform TOQL queries on it provided that time information is based on the 4D perdurandist mechanism.

1.3 Contributions of Present Work

The contributions of this work are summarized below:

1. We propose TOQL, a high level query language for temporal ontologies. TQOL syntax and semantics are fully specified and analyzed.

2. A TOQL interpreter capable of executing TOQL queries on any temporal ontology is described and implemented.

3. To demonstrate and objectively assess TOQL, an integrated information system capable of handling TOQL on any temporal ontologies is implemented.

1.4 Thesis Outline

Background knowledge and related research are discussed in Chapter 2. A description of OWL (Web Ontology Language) and of the four-dimensionalist (perdurantist) approach to handle time in ontologies are given. Ontology query language such as RDQL, SPARQL and SeRQL and ALLEN calculus are also presented and discussed. Finally two ontologies, the "Static Enterprise Ontology" and the "Dynamic Enterprise Ontology" are presented.

In Chapter 3 TOQL's syntax and semantics are presented and discussed. TOQL's clauses and keywords are provided. The way TOQL handles time in ontologies implementing the four-dimensionalist (perdurantist) approach is also discussed. Several query examples based on "Static Enterprise Ontology' and "Dynamic Enterprise Ontology" are also provided. A formal description of the language's syntax in BNF is given in Appendix A.

In Chapter 4 the application implemented to fully support query functionality on ontologies in OWL is presented. Issues such as query translation, ontology loading into memory and syntax and semantic errors handling are discussed. Graphic User Interface (GUI) created to facilitate query formulation and general user interaction with the ontology is also presented.

Finally conclusions and issues for further research are given in Chapter 5.

Chapter 2

Background and Related Work

2.1 Ontologies

Ontologies are specifications of the conceptualization and corresponding vocabulary used to describe a domain. They are well-suited for describing heterogeneous, distributed and semi-structured information sources that can be found on the Web. By defining shared and common domain theories, ontologies help both people and machines to communicate concisely, supporting the exchange of semantics and not only syntax. It is therefore important that any semantic for the Web is based on an explicitly specified ontology. This way, consumer and producer agents (which are assumed for the Semantic Web) can reach a shared understanding by exchanging ontologies that provide the vocabulary needed for discussion. Ontologies typically consist of definitions of concepts relevant for the domain, their relations, and axioms about these concepts and relationships. Several representation languages and systems are defined.

2.2 OWL

The future of the Web is the Semantic Web. In Sementic Web the information contained in documents is given an explicit meaning, making it easier to be processed by applications. OWL [6] can be used to represent the meaning of terms and the relationships between those terms. It is more expressive than XML, RDF and RDF-S, making it easier to represent machine interpretable content on the Web. It is a revision of DAML-OIL web ontology language and it has tree sublanguages (species):

- **OWL Lite** that supports a classification hierarchy and simple constraints.

- **OWL DL** that supports maximum expressiveness while retaining computational completeness and decidability.
- **OWL Full** that supports maximum expressiveness but no computational guarantees.

In our work the ontologies are implemented in OWL DL. OWL DL is computable while maintaining maximum expressiveness.

2.3 Representation of Time in Ontologies

Dealing with information that changes over time is a critical problem in Knowledge Representation (KR). Representation languages such as OWL, RDF (description logics), frame-based and object-oriented languages (F-logic) are all based on binary relations. The fact is that binary relations may change over time (e.g., being employee of a company) making the representation of time a difficult matter to deal with, since binary relations simply connect two instances (e.g., the employee with the company) without any temporal information. Time representation using OWL is feasible, although it is complicated. OWL-Time (formerly DAML-Time) temporal ontology describes the temporal content of Web pages and the temporal properties of Web services. Apart from language constructs for the representation of time in ontologies, there is a need for mechanism for the representation of the evolution of concepts (events) over time. Versioning suggests that the ontology has different versions (one per instance of time). When a change takes place, a new version is created. Versioning suffers from several disadvantages: (a) changes even on single attributes result in a new version of the ontology will be created (information redundancy) (b) searching for events in time instances or intervals requires exhaustive searches in multiple versions of the ontology to find the beginning and ending point time interval of interest, (c) it is clear how the relation between evolving classes are represented. In this paper the solution based on the four-dimensionalist (perdurantist) approach [21] is adopted.

Before we describe the four-dimensionalist (perdurantist) approach we should first briefly describe the three-dimensionalist (endurantist) approach. This approach distinguishes the world into two basic categories: the endurants (physical objects such as cars, companies, people) and the occurants (events such as buying a car). Endurants are supposed to exist at all times and have no time dimension, while occurants have temporal parts that exist during the times the entity exists. The main issue with this approach is that the diachronic identity (the identity that determines an entity over time) of

endurants is addressed by identifying a set of properties that do not change over time. An entity (endurant) has a set of properties that do not change over time (ex. a person's DNA) along with a set of properties that do change over time (ex. the hair's color). In these case the Leibniz's Law (i.e., X and Y are identical if and only if they share all and only the same properties) does not qualify (e.g., consider a person with black hair as X and the same person with brown hair as Y. X and Y are not identical because they do not share all and only the same properties).

The four-dimensionalist (perdurantist) approach assumes that all entities are perdurants, making no distinction between endurants and occurants. The idea is that each every entity has a start and an end point (ex. the lifetime of the sun). An entity can be seen as a four dimensional "space-time worm", with the slices of the worm being the temporal parts of the entity. With this approach the problem of diachronic entity becomes trivial since an entity is four dimensional and has temporal parts. Changes occur on the properties of the temporal part keeping the entity as a whole unchanged.

To add the time dimension to an ontology using the 4D fluent (perdurants) approach, classes *TimeSlice* and *TimeInterval*, and properties *tsTimeSliceOf* and *tsTimeInterval* are introduced as shown in Figure 2.2. Class *TimeSlice* is the domain class for all the entities' "time slices" (i.e., entities' temporal parts), while class *TimeInterval* is the domain class for all time intervals. A time interval holds the temporal information of a time slice. Property *tsTimeSliceOf* connects an instance of class *TimeSlice* with an entity, and property *tsTimeInterval* connects an instance of class *TimeSlice* with an instance of class *TimeInterval*. Properties having a time dimension are called fluent properties and connect instances of class *TimeSlice*.

The "Static Enterprise Ontology" ("StEn Ontology") of Figure 2.1 has three classes (concepts), namely *Company*, *Product* and *Employee*. Class *Company* has the datatype property *companyName* and the object properties *produces* and *hasEmployee*; class *Product* has the datatype properties *productName* and *price*, and class *Employee* has the datatype property *employeeName*.

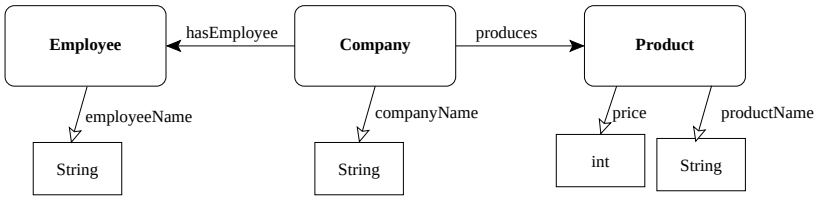


Figure 2.1: Static Enterprise Ontology

Figure 2.2 illustrates the "Dynamic Enterprise Ontology" ("DEn Ontology"), that is the temporal ontology derived by adding concepts of time (using the 4D purdurandist approach) to the Static Enterprise Ontology of Figure 2.1. *CompanyName* and *employeeName* are static properties (their value does not change over time), while properties *produces*, *hasEmployee*, *productName* and *price* are dynamic, fluent properties (their value changes over time). Because these properties are fluents their domain (and range if they are object properties) is of class *TimeSlice*. Notice that Figure 2.2 gives only the ontology without any instances. *EmployeeTimeSlice*, *CompanyTimeSlice* and *ProductTimeSlice* generic instances are provided to make clear that the domain of properties *hasEmployee*, *produces*, *productName* and *price* are time slices restricted to be slices of a specific class. For example, the domain of property *productName* is not the class *TimeSlice* but it is restricted to instances that are time slices of (*tsTimeSliceOf*) class *Product*.

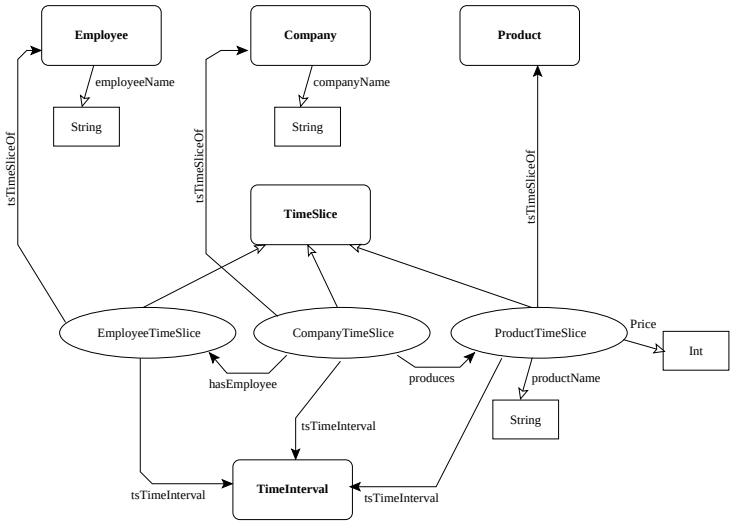


Figure 2.2: Dynamic Enterprise Ontology

2.4 Ontology Query Languages

Query languages for supporting queries on ontologies are known exist [22]. SquishQL [18] is an RDF query language, RDQL [19] is a query language first released in Jena [1], nRQL [15] is a query language for RACER, OWL-QL is a query language for the OWL-QL [14] system. SPARQL [7] is a w3c recommendation query language supported by SESAME [12],[2] and finally SeRQL [11] is the query language developed for SESAME. The following sections describe RDQL, SPARQL and SeRQL further.

2.4.1 RDQL

RDF is a directed, labeled graph data format for representing information in the Web. RDQL [19] is an SQL like object oriented query language for RDF. The purpose of RDQL is to provide a data-oriented query model. This means that RDQL only retrieves information stored in the model which contains a set of N-Triple [17] statements.

The main clauses of RDQL are SELECT and WHERE. SELECT clause identifies the variables to appear in the query results, while WHERE clause contains a set of N-Triples that define the query pattern. Assume the "StEn Ontology". The Knowledge base of this ontology is given in Turtle format [10]:

Data (Using Turtle format):

```
@prefix default: <http://www.owl-ontologies.com/Static Enterprise Ontology.owl> .

default:Company1 default:produces default:Product1
default:Company2 default:produces default:Product2

default:Product1 default:price 30
default:Product2 default:price 35
```

The following query asks for the company that has a product with value more than 32:

Query:

```
SELECT ?Company
WHERE (?Company,<ex:produces>,?Product) (?Product,<ex:price> ?price) AND ?price > 32
USING ex FOR <http://www.owl-ontologies.com/Static Enterprise Ontology.owl#>
```

Looking at the data, one can easily see that only Company2 has a product with value more than 32 (specifically 35).

Query Result:

Company
Company2

2.4.2 SPARQL

SPARQL is also a query language for RDF influenced from RDQL. SPARQL consists mainly of two parts: the SELECT clause specifies the variables to appear in the query results, and the WHERE clause provides the basic graph

pattern to match against the data graph. The FROM clause is only used to specify RDF datasets. An RDF dataset represents a collection of graphs [7]. The FROM clause specifies the dataset to be used for matching. Assume once more the "StEn Ontology". The following statements in Turtle format represent instance Company1 of the "StEn Ontology":

Data (Using Turtle format):

```
@prefix default: <http://www.owl-ontologies.com/Static Enterprise Ontology.owl> .


default:Company1 default:companyName C1
```

The following query returns the name of Company1:

Query:

```
SELECT ?companyName
WHERE
{
<http://www.owl-ontologies.com/Company1>
<http://www.owl-ontologies.com/companyName> ?companyName .
}
```

Query Result:

companyName
"C1"

2.4.3 SeRQL

SeRQL (Sesame RDF Query Language) [11] is another RDF query language that is very similar to SPARQL, but with other syntax. SeRQL, similarly to RDQL and SPARQL supports SQL syntax but in addition to RDQL and SPARQL, supports comparison between date times.

SeRQL supports two query modes, referred to as "Select Query" returning a table of values and "Construct Query" returning an RDF graph (a part of the Knowledge Base). Typically, a SeRQL "Select Query" can be built-upon from one and up to seven clauses: SELECT, FROM, FROM CONTEXT, WHERE, LIMIT, OFFSET and USING NAMESPACE. Construct queries support exactly the same clauses but start with CONSTRUCT instead of SELECT. Except from the first clause, SELECT or CONSTRUCT, the remaining clauses are optional.

SELECT clause specifies which values are returned. FROM specifies path expressions. Path expressions are expressions that match specific paths through an RDF graph. Path expression's basic forms are:

{subj} pred {obj}
{subj1} pred1 {obj1} subj2 {obj2}
{subj1} pred1 {obj1} subj2 {obj2} subj3 ...

A path expression consists of nodes and edges. The nodes and edges in the path expressions can be variables, URIs and literals. Each and every path can be constructed using a set of basic path expressions. However, there are available some short cuts to simplify path expressions:

- **Multi-value nodes:** are used to query two or more statements with identical subject and predicate. In such cases the subject and predicate do not have to be repeated. Consider the following path expression:

  FROM {subj1} pred1 {obj1, obj2, obj3}

  This path expression is equivalent to this one:

  FROM
  {subj1} pred1 {obj1},
  {subj1} pred1 {obj2},
  {subj1} pred1 {obj3}
  WHERE obj1 != obj2 AND obj1 != obj3 AND obj2 != obj3

- **Branches:** are used to query multiple properties of a single subject. Instead of repeating the subject, one can use a semi-colon to attach a predicate-object combination to the subject:

  {subj1} pred1 {obj1};
  pred2 {obj2}

  The above path exrpession in equivalent to:

  {subj1} pred1 {obj1},
  {subj1} pred2 {obj2}

- **Reified statements:** with this short cut, a path expression representing a single statement (i.e., node edge node) can be written between the curly brackets of a node:

  { {reifSubj} reifPred {reifObj} } pred {obj}

  This is equivalent to (using "rdf:" as a prefix for the RDF namespace, and "Statement" as a variable for storing the statement's URI):

  {Statement} rdf:type {rdf:Statement},
  {Statement} rdf:subject {reifSubj},
  {Statement} rdf:predicate {reifPred},
  {Statement} rdf:object {reifObj},
  {Statement} pred {obj}

WHERE clause is an optional and is useful for specifying Boolean constraints on variables. The most common boolean constrains are:

- **Value (in)equality:** values are compared using the operators "=" and "!=".

  Var1 = Var2
  Var1 != Var2

- **Numerical comparisons:** numbers can be compared to each other by using the operators "<", "<=", ">" and ">=". Notice that SeRQL allows comparison between dates and datetimes. The following example query retrieves the countries with population less than 1.000.000:

  SELECT Country
  FROM Country ex:population Population
  WHERE Population < "1000000"xsd:positiveInteger
  USING NAMESPACE ex = <http://example.org/things#>

- **The LIKE operator:** checks whether a value matches a specified pattern of characters. The following example query retrieves the country named "Belgium":

  SELECT Country
  FROM Country ex:name Name
  WHERE Name LIKE "Belgium"
  USING NAMESPACE ex = <http://example.org/things#>

- **AND, OR, NOT:** compine and negate boolean constraints. The NOT operator has the highest presedence, then the AND operator, and finally the OR operator. Parentheses can also be used

Assume the "StEn Ontology". The instances of this ontology are given in Turtle format. The following statements describe two instances of class company (Company1 with names C1, Company2 with name C2) producing Product1 with name P1 and Product2 with name P2 and whose prices are 30 and 35 respectively.

Data (Using Turtle format):

```
@prefix default: <http://www.owl-ontologies.com/Static Enterprise Ontology.owl/> .

default:Company1 default:produces default:Product1
default:Company1 default:companyName C1
default:Company2 default:produces default:Product2
default:Company2 default:companyName C2

default:Product1 default:productName P1
default:Product1 default:price 30
default:Product2 default:productName P2
default:Product2 default:price 35
```

The following query asks for companies (companies' names) producing products whose price is greater than 32. Also asks for the products' names:

Query:

```
SELECT
companyName, productName
FROM {Company} ex:name {companyName},
{Company} rdf:type {ex:Company}
{Product} rdf:type {ex:Product}
Company ex:produces Product
{Product} ex:name {productName}
{Product} ex:price {price}
WHERE price > 32xsd:int
USING NAMESPACE ex = <http://www.owl-ontologies.com/Static Enterprise Ontology.owl/>
```

Query Result:

companyName	**productName**
C2	P2

2.5 Temporal Databases

Conventional databases describe the state of concepts at a single moment of time. As new information is added to concepts, changes affect the current state of the concepts, with the old data being deleted from the database. A temporal database has built-in time aspects, representing the progression of concepts over the time.

Temporal aspects usually include valid time and transaction time. Valid time denotes the period during which a fact is true with respect to the real world, while transaction time denotes the period during which a fact is stored in the database. Both of them together form the bitemporal data. In a database table bitemporal data is often represented by four extra table-columns StartVT (start valid time) and EndVT (end valid time), StartTT (start transaction time) and EndTT (end transaction time).

2.6 Relational Databases Query Languages

Query languages for supporting queries on relational databases are known exist such as TQuel and TSQL2. Tquel is a superset of Quel and TSQL2 specifies a temporal extension to the SQL-92 language standard.

2.6.1 TQuel

TQuel [20] is a superset of Quel [13]. Quel is query language, paricularly simple, powerfull and less complex than SQL. It is a minimal extension (syntactically and semantically) of Quel which ensures that all Quel statements are also TQuel statements that have the an identical sementics in both languages and that the addinional constructs defined in TQuel have analogues in Quel.

Quel query concists of the two componets: the target list and a *where* clause. Target list specifies how the attributes of the relation being returned are computed from the attributes of the underlying relations and a *where* clause specifies which tuples participate in the derivation.

TQuel also introduces clauses such as *when* that is the temporal analogue to Quel's *where* clause, *valid* that specifies the value of an attribute in the derived relation and *as of*. *When* and *valid* clauses access valid times while as of accesses transaction times.

2.6.2 TSQL2

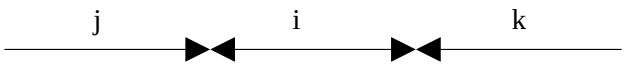
TSQL2 is a temporal extension to the SQL-92 language standard.

2.7 Allen Operators

Allen operators [9], define the complete range of intuitive relationships that can hold between time periods. A time period intuitively is the time associated with some event occuring or some property holding in the world. Assume a simple linear model of time (the future always follows the past). In this model there is one primitive relation: *Meets*. Two periods m and n meet if and only if m precedes n, there is no time between m and n, and m and n do not overlap. Assuming time periods i,j,k,l,m, the axiomatization of *Meets* is as follows [9]:

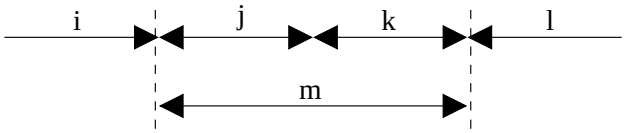
1. There is no beginning or ending of time and there are no semi-infinite or infinite periods (every period has a period that meets it and another that it meets):

$$\forall i.\exists j,k.Meets(j,i) \wedge Meets(i,k).$$



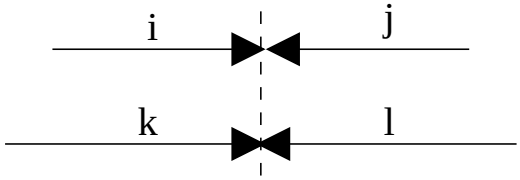
2. Any two periods that meet can be composed to produce a larger period (concatenation):

$$\forall i,j,k,l.Meets(i,j) \wedge Meets(j,k) \wedge Meets(k,l) \supset \exists m.Meets(i,m) \wedge Meets(m,l).$$



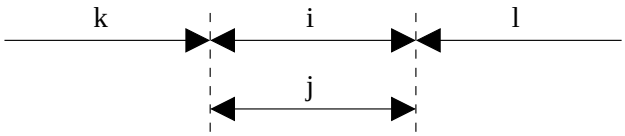
3. Periods uniquely define an equivalence class of periods that meet them. If i meets j and k, then if period l meets j must also meet k:

$$\forall i,j,k,l.Meets(i,j) \wedge Meets(i,k) \wedge Meets(l,i) \supset Meets(l,k).$$



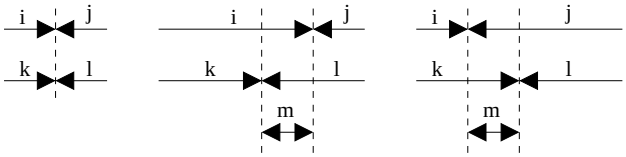
4. Equivalence classes uniquely define the periods. If two periods both meet the same period, and another period meets both them, then they are equal:

$$\forall i, j, k, l. Meets(k, i) \wedge Meets(k, j) \wedge Meets(i, l) \wedge Meets(j, l) \supset i = j$$



5. Two pairs of periods (i meets j and k meets l), either they both meets at the same "place",or the place where i meets j precedes the place where k meets l, and vice versa:

$$\forall i, j, k, l. (Meets(i, j) \wedge Meets(k, l)) \supset Meets(i, l) \otimes$$
$$(\exists m. Meets(k, m) \wedge Meets(m, j)) \otimes (\exists . Meets(i, m) \vee Meets(m, l)).$$



With this system, the complete range of intuitive relationships that could hold between time periods (Allen operators [9]) are: *Before* - *After*, *Meets* - *MetBy*, *Overlaps* - *OverlappedBy*, *Starts* - *StartedBy*, *During* - *Contains*, *Finishes* - *FinishedBy*, *Equals*. Figure 2.3 shows each of these relationships graphically:

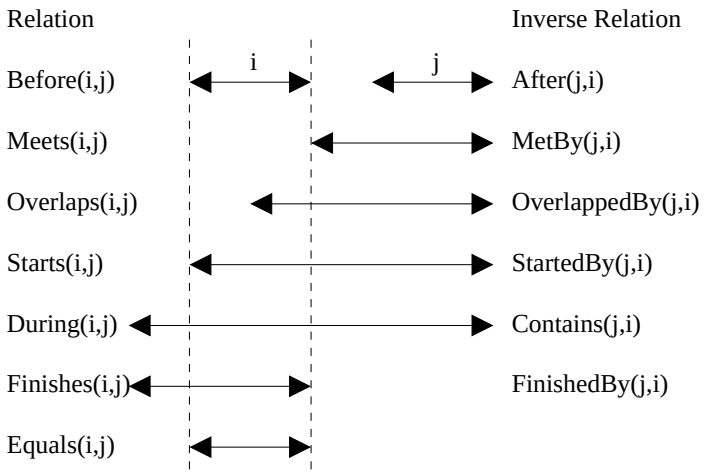


Figure 2.3: The possible relations between time periods

Chapter 3

TOQL: Syntax and Semantics

TOQL (Temporal Ontology Query Language) is an SQL-like, temporal query language for OWL. In the following sections the language (syntax and semantics) and the application developed to support this language are presented and discussed. A formal description of the language's syntax in BNF is given in Appendix A.

3.1 Basic clauses

TOQL supports most of an SQL language syntax and clauses. Namely TOQL suppports following clauses:

- **SELECT**: specifies the values to be returned.
- **FROM**: declares the class or classes to query from. Always follows SELECT.
- **WHERE**: includes logic operations and comparisons that restrict the number of rows returned by the query. Always follows FROM.
- **LIMIT**: limits the numbers of rows returned by the query (e.g., LIMIT 10 will return the first ten rows). Can be after FROM (if no WHERE clause is specified in the query), WHERE and OFFSET.
- **OFFSET**: sets the first row returned by the query (e.g., OFFSET 5 will skip the first 4 rows and will returns the others). Can be after FROM (if no WHERE clause is specified in the query), WHERE and LIMIT.

Table 3.1 summarizes TOQL syntax:

Case 1	Case 2	Case 3	Case 4
SELECT ... FROM ...	SELECT ... FROM ... OFFSET ... LIMIT ...	SELECT ... FROM ... WHERE ...	SELECT ... FROM ... WHERE ... OFFSET ... LIMIT ...

Table 3.1: Examples of TOQL syntax

Except from the above, TOQL also supports the following:

- **AS**: renames a class (if it is used in clause FROM) or a property (if it is used in clause SELECT). Renaming a class allows using more than one instance of a class in a query (e.g., FROM Company AS C1, Company AS C2). Renaming a property specifies the columns names of the returning results tables (e.g., SELECT Company.companyName AS Name).
- **AND**: connects two properties (datatype or object) in WHERE.
- **OR**: connects two properties (datatype or object) in WHERE.
- **LIKE**: compares a datatype property with a string in WHERE. Comparison is case sensitive.
- **LIKE "string" IGNORE CASE**: compares a datatype property with a string ignoring case.

Table 3.2 summarizes TOQL with the additional clauses:

Syntax
SELECT ... AS ...
FROM ... AS ...
WHERE ... LIKE ... AND ... LIKE "string" IGNORE CASE

Table 3.2: Generic TOQL syntax

Finally there are operation clauses for connecting two queries (combinatory operations) and for creating nested queries:

- **MINUS**: returns query results retrieved by the first operand exluding results retrieved by the second operand.

- **UNION**: returns the union of results returned by both operands. Duplicate answers are filtered out.
- **UNION ALL**: returns the union of results returned by both operands. Duplicate answers are not filtered out.
- **INTERSECT**: returns the intersection of results retrieved by both operands.
- **EXISTS**: this is a unary operator that has a nested SELECT-query as its operand. The operator is an existential quantifier that succeeds when the nested query has at least one result.
- **ALL**: this is an operator that has a nested SELECT-query as one of its operands. It always follows a comparison operator (i.e., "=", "!=", "<", ">", "<=", ">="). It indicates that for every value of the nested query the comparison must hold.
- **ANY**: has a nested SELECT-query as one of its operands. It always follows a comparison operator (i.e., "=", "!=", "<", ">", "<=", ">="). It indicates for at least one value of the nested query the comparison must hold.
- **IN**: has a nested SELECT-query as one of its operands. Allows set membership checking. The set is defined by the nested SELECT-query.

Table 3.3 summarizes TOQL syntax with operator clauses:

Case 1	**Case 2**	**Case 3**	**Case 4**
Query MINUS Query	Query UNION Query	Query UNION ALL Query	Query INTERSECT Query
Case 5	**Case 6**	**Case 7**	**Case 8**
SELECT ... FROM ... WHERE EXISTS (QUERY)	SELECT ... FROM ... WHERE ... CO[1] ALL (Query)	SELECT ... FROM ... WHERE ... CO[1] ANY (Query)	SELECT ... FROM ... WHERE ... IN (Query)

Table 3.3: TOQL syntax with operator clauses

[1]CO: comparison operator can be any of "=", "!=", "<", ">", "<=", ">="

3.2 Dealing with Classes and Properties

In relational databases the basic terms are relations (tables) and attribute (columns). Tables represent concepts of the world (e.g., the concept of Company, or the concept of Car) or many to many (N:N) relations between two concepts, while columns represent attributes attributes of concepts (e.g., Company name, address, ...).

In ontologies the basic terms are classes (also named concepts) and properties (object or datatype). Classes represent concepts of the world (e.g. the concept of University or the concept of Food). Properties represent relations between two concepts or between a concept and a value. Properties relating two classes (concepts) are referred to as object properties, while properties relating a class with a value are referred to as datatype properties. As an example of object property consider the relation between the Company and the Employee. These two classes can be connected with the object property *hasEmployee*. As an example of datatype property consider the name of an Employee. Class Employee can be connected with a name (string value) with datatype property *isNamed*.

TOQL not only uses SQL-like clauses and a similar syntax but also treats ontologies almost like relational databases. Tables representing concepts correspond to classes and tables representing relations correspond to object properties. Attributes correspond to datatype properties. In addition, 1:1 and 1:N relations correspond to object properties (see Section 4.3). Table 3.4 summarizes the mapping between database relations and ontology concepts used by TOQL.

Relation Database	Ontology
Table representing concept	Class
Table representing N/N relation	Object Property
1/N, 1/1 relations	Object Property
Attribute	Datatype Property

Table 3.4: Mapping between database relations and ontology concepts

In TOQL classes are declared in FROM clauses just like SQL handles tables. To access a datatype property of a class, the name of the class is followed by a dot (".") and the name of the datatype property, just like SQL handles tables and attributes:

ClassName.DatatypePropertyName

To access object properties (properties connecting two classes), the name of the domain class is followed by a dot (“.”), the name of the object property, double dot (“:”) and finally the name of range class:

DomainClassName.objectPropertyName:RangeClassName

The following query can be used to access the names of companies producing products called “x”, in the “StEn Ontology” of Figure 2.1:

SELECT Company.companyName
FROM Company, Product
WHERE Company.produces:Product
AND Product.productName LIKE “x”

Notice that the declaration of classes Company and Product in FROM is necessary. Also notice that datatype and object properties are being accessed like SQL accesses table attributes.

3.3 Dealing with Time

TOQL is a high level language hiding the implementation of time at the ontology level. As such, the user needs not be aware of the details of the 4D fluent (perdurandist) mechanism used in this work to represent time evolving information in ontologies. As mentioned in Section 2.3, an ontology implementing the 4D fluent mechanism consists of two parts: the static part which is the initial ontology (classes, properties, instances) and the dynamic part consisting of additional temporal classes needed to represent time, evolution in time as well as properties and instances of the above temporal classes (*TimeSlice* class, *TimeInterval* class, *tsTimeSliceOf* property, *tsTimeInterval* property, fluent properties). TOQL supports “high level functionality” for dealing with both static and the dynamic part while the user is aware only of the static part. TOQL automatically determines references to time related information. More specificically, TOQL,

- Retrieves all the time slices associated with a class of the static ontology.
- Determines whether a property (object or datatype) in the query is a fluent property (i.e., a property that connects time slices or a time slice with a datatype) or not (a property that connects “static” classes or a “static” class with a datatype).

- Uses the ontology's dynamic part to answer to the query, if a property specified by the query is a fluent one.
- Uses the ontology's static part to answer to the query, if a property specified by the query is not a fluent one.

As an example consider the "DEn Ontology" of Figure 2.2. Typically to retrieve companies that hired employees, one should be aware of the 4D fluent mechanism and ask for all time slices (instances) of class *Company* and all time slices of *Employee* and then query on the object property *hasEmployee* that connects those instances. In TOQL but without implementing the high level functionality described above, this is typically expressed as:

SELECT Company.companyName
FROM Company, Employee, TimeSlice AS T1 ,
TimeSlice AS T2
WHERE T1.tsTimeSliceOf:Company AND
T2.tsTimeSliceOf:Employee AND T1.hasEmployee:T2 AND
Employee.employeeName LIKE "x"

This is rather complicated expression, requires that the user be familiar with the implementation of time at the level of the ontology (the 4D fluent method in this work). However, this is not necessary in TOQL and the same query can be expressed as:

SELECT Company.companyName
FROM Company, Employee
WHERE Company.hasEmployee:Employee
AND Employee.employeeName LIKE "x"

The second query is much more easy to write than the first one. Notice that the object property *hasEmployee* is treated like its domain is class *Company* and its range is class *Employee*, while in fact it is a fluent property and has domain the class *TimeSlice* (instances of the class *TimeSlice* that are time slices of *Company*) and range the class *TimeSlice* (instances of the class *TimeSlice* that are time slices of *Employee*).

To deal with time, TOQL also introduces some other clauses. It uses additional clauses implementing the ALLEN operators (see Section 2.7). In TOQL, the implementation of ALLEN operators correspond to comparisons between fluent properties. Fluent properties connect time slices and time slices are associated with time intervals. Consequently, implementation of Allen operators correspond to comparisons between time intervals. In accordance to Allen calculus [9] the following operators are supported in TOQL:

- **BEFORE**: returns true if the first time interval is *before* the second one.
- **AFTER**: returns true if the first time interval is *after* the second one.
- **MEETS**: returns true if the first time interval *meets* the second one.
- **METBY**: returns true if the second time interval *meets* the first one.
- **OVERLAPS**: returns true if the first time interval *overlaps* the second one.
- **OVERLAPPEDBY**: returns true if the second time interval *overlaps* the first one.
- **DURING**: returns true if the first time interval is *during* the second one.
- **CONTAINS**: returns true if the first time interval is *contains* the second one.
- **STARTS**: returns true if the two time intervals *start* together.
- **STARTEDBY**: returns true if the two time intervals *start* together.
- **ENDS**: returns true if the two time intervals *end* together.
- **ENDEDBY**: returns true if the two time intervals *end* together.
- **EQUALS**: returns true if the first time interval *equals* the second one.

The following TOQL query retrieves the name of the company that hired employee "x" and *then* employee "y":

SELECT Company.companyName
FROM Company, Employee AS E1, Employee AS E2
WHERE Company.hasEmployee:E1 BEFORE Company.hasEmployee:E2
AND E1.employeeName like "x" AND E1.employeeName LIKE "y"

TOQL also introduces the clause AT which compares a fluent property (i.e., the time interval in which the property is true) with a time period (time interval) or time point. Notice that AT clause retrieves data only explicitly defined in the Knowledge Base. Assume the "DEn Ontology" and consider that at time point 5 the price of Product1 is 10 and that there is no information about its price after time point 5. If a query asks for the

price of Product1 at time point 6 TOQL will return nothing. A reasonable answer would be 10 (the last known price in the KB). Answering such queries effectively requires combining TOQL with a reasoner.

- **AT(time point)** operation returns true if the time interval holds true this point of time .
- **AT(start time point, end time point)** operation returns true if the time interval holds true for *all* the time interval (start time point - end time point).

The following TOQL query retrieves the name of the company employee "x" was working for, from time=3 to time=5:

SELECT Company.companyName
FROM Company, Employee
WHERE Company.hasEmployee:Employee AT(3,5)
AND Employee.employeeName LIKE "x"

As mentioned above in TOQL the user does not have to be aware of the implementation details of the 4D fluent mechanism. Because TOQL is independent of the mechanism implementing time, there is no way to directly access class *TimeInterval* (i.e., the class holding values of time). In order TOQL to return time the the keyword TIME is introduced. It follows datatype or object properties and can be used only in SELECT. It returns the start and end time point (if any) in which the property is true (the time interval in which the property is true). If no end point exists it returns only start point.

- **TIME** clause returns the time interval (start and end point) in which a property holds true.

As an example, the following TOQL query retrieves the time for which a company had employee "x"

SELECT Company.hasEmployee.TIME
FROM Company, Employee
WHERE Company.hasEmployee:Employee AND
Employee.employeeName LIKE "x"

A more formal description of TOQL syntax in BNF is given in appendix A. Notice that all keywords are *case insensitive*.

3.4 Special Cases

This section describes some TOQL special features. These are related to the way TOQL deals with:

- Class keys (Section 3.4.1).
- Wildcard (*) (Section 3.4.2).
- Scope (Section 3.4.3).

3.4.1 Dealing with keys

In relational databases each table's tuple is uniquely characterized by a key. A key can refer to more than one attributes (compound key). Consider a relational database that has the table Company and that this table uses the attribute ID as key. To access this key, in SQL, a user should write:

SELECT Company.ID

In OWL, each class instance and each property have a unique name. This unique name is considered to be equivalent to the unique key of relational databases. The difference is that this unique name is not an ordinary datatype property, and so it can not be treated the way described in Section 3.2 (i.e., it cannot be accessed by writing the name of the class followed by a dot '.' and the datatype property). In TOQL, the (unique) name of a class instance is accessed using the name of the class itself (without reference to a property). For example, to access the unique name of a company we write:

SELECT Company

Assume the "DEn Ontology". The following TOQL expression can be used to access the (unique) name that company, named "x", has in the time interval (3,5):

SELECT Company
FROM Company
WHERE Company.companyName
LIKE "x" AT(3,5)

3.4.2 Dealing with wildcard (*)

In TOQL wildcard (*) can be used only in SELECT. In SQL the presence of wildcard (*) in SELECT implies that all the columns of all the tables declared in clause FROM will be returned. If the wildcard (*) follows a table (*tableName*.*) all the columns of the specific table will be returned.

In TOQL the presence of wildcard (*) in SELECT implies that all the datatype properties of *all* the classes declared in FROM will be returned. If the wildcard (*) follows a class all the datatype properties of the specific class will be returned. Notice that the class unique name is not returned (only its datatype properties are returned).

Assume the ontology "StEn Ontology" of Figure 2.1. The following query retrieves companies producing product with unique name "x", as well as the product's name.

SELECT *
FROM Company, Product
WHERE Company.hasProduct:Product
AND Product LIKE "x"

3.4.3 Dealing with scope

TOQL supports set combination operations in queries as well as nested queries. Both set operations and nested queries imply that a TOQL query may be composed of more than one "subqueries". Each "subquery" has its own class declarations, as well as class and property usage and this fact introduces the need of handling the different scopes. This section discusses how TOQL handles scopes and sub queries.

TOQL treats composite queries combined by set operators and nested queries in a different query. Queries combined by set operators belong to completely different scopes. Classes declared in any of them are local to this query and are not visible to the others. The following query retrieves names of "Company_1" and also names of "Company_2" using the "DEn Ontology":

SELECT C1.companyName
FROM Company As C1
WHERE C1 like "Company_1"
UNION
SELECT C1.companyName
FROM Company As C1
WHERE C1 like "Company_2"

This TOQL expression specifies two sperate queries combined by the set operator UNION. Each of them has a different scope: classes declared in the first "subquery" are not visible to the second one. Even if the same class is to be used by the second "subquery", it must be declared again.

In TOQL, a nested query inherits all the classes declared in the query it is nested into, so a nested query can use all of these classes, but can not (re)declare any of them. Assume the "Dynamic Enterprise Ontology" and consider the following TOQL query, that retrieves the product with price higher than (or equal to) 10 and is not smaller than any other product in the Knowledge Base:

SELECT P1
FROM Product As P1
WHERE P1.price >= 10 AND NOT
P1.price < Any
(**SELECT** P2.value **FROM** Product As P2)

In this TOQL query there is a nested query following clause ANY. Notice that both query use class "Product" and this class is given different names within each subquery (i.e., "P1" and "P2" respectively).

Notice that in both queries the same class is used (Product) but different names are assigned to it (P1 and P2), otherwise a semantic error will occur (see Section 4.1.2).

3.5 Examples

In this section an ontology/knowledge base and a set of query examples are provided along with their results. The knowledge base, containing instances of the defined by DEN ontology of Figure 2.2 are given in Turtle[10] format:

Data (Using Turtle format):

```
@prefix ex1: <http://www.owl-ontologies.com/Static Enterprise Ontology.owl/> .

ex1:Company1 ex1:companyName "C1" .
ex1:Company2 ex1:companyName "C2" .

ex1:Employee1 ex1:employeeName "John" .
ex1:Employee2 ex1:employeeName "Mark" .
ex1:Employee3 ex1:employeeName "John" .
```

Data (cont):

```
ex1:Company1TimeSlice1 ex1:tsTimeSliceOf ex1:Company1 .
ex1:Company1TimeSlice1 ex1:produces ex1:Product1TimeSlice1 .
ex1:Company1TimeSlice1 ex1:hasEmployee ex1:Employee1TimeSlice1 .
ex1:Company1TimeSlice1 ex1:tsTimeInterval ex1:TimeInterval1 .

ex1:Company1TimeSlice2 ex1:tsTimeSliceOf ex1:Company1 .
ex1:Company1TimeSlice2 ex1:produces ex1:Product2TimeSlice1 .
ex1:Company1TimeSlice2 ex1:hasEmployee ex1:Employee2TimeSlice1 .
ex1:Company1TimeSlice2 ex1:tsTimeInterval ex1:TimeInterval2 .

ex1:Company2TimeSlice1 ex1:tsTimeSliceOf ex1:Company2 .
ex1:Company2TimeSlice1 ex1:produces ex1:Product3TimeSlice1 .
ex1:Company2TimeSlice1 ex1:hasEmployee ex1:Employee3TimeSlice1 .
ex1:Company2TimeSlice1 ex1:tsTimeInterval ex1:TimeInterval3 .

ex1:Employee1TimeSlice1 ex1:tsTimeSliceOf ex1:Employee1 .
ex1:Employee1TimeSlice1 ex1:tsTimeInterval ex1:TimeInterval1 .

ex1:Employee2TimeSlice1 ex1:tsTimeSliceOf ex1:Employee2 .
ex1:Employee2TimeSlice1 ex1:tsTimeInterval ex1:TimeInterval2 .

ex1:Employee3TimeSlice1 ex1:tsTimeSliceOf ex1:Employee3 .
ex1:Employee3TimeSlice1 ex1:tsTimeInterval ex1:TimeInterval3 .

ex1:Product1TimeSlice1 ex1:tsTimeSliceOf ex1:Product1 .
ex1:Product1TimeSlice1 ex1:tsTimeInterval ex1:TimeInterval1 .
ex1:Product1TimeSlice1 ex1:productName “P1” .
ex1:Product1TimeSlice1 ex1:price 10.0 .

ex1:Product2TimeSlice1 ex1:tsTimeSliceOf ex1:Product2 .
ex1:Product2TimeSlice1 ex1:tsTimeInterval ex1:TimeInterval2 .
ex1:Product2TimeSlice1 ex1:productName “P2” .
ex1:Product2TimeSlice1 ex1:price 15.0 .

ex1:Product3TimeSlice1 ex1:tsTimeSliceOf ex1:Product3 .
ex1:Product3TimeSlice1 ex1:tsTimeInterval ex1:TimeInterval3 .
ex1:Product3TimeSlice1 ex1:productName “P3” .
ex1:Product3TimeSlice1 ex1:price 20.0 .

ex1:Product3TimeSlice2 ex1:tsTimeSliceOf ex1:Product3 .
```

Data (cont):

```
ex1:Product3TimeSlice2 ex1:tsTimeInterval ex1:TimeInterval4 .
ex1:Product3TimeSlice2 ex1:productName "P3x" .
ex1:Product3TimeSlice1 ex1:price 22.0 .

ex1:TimeInterval1 ex1:startValue 1 .
ex1:TimeInterval1 ex1:endValue 5 .

ex1:TimeInterval2 ex1:startValue 6 .
ex1:TimeInterval2 ex1:endValue 10 .

ex1:TimeInterval3 ex1:startValue 3 .
ex1:TimeInterval3 ex1:endValue 7 .

ex1:TimeInterval4 ex1:startValue 8 .
ex1:TimeInterval4 ex1:endValue 13 .
```

Query example 1:

```
SELECT Product, productName
FROM Product
WHERE price > 10.0
```

This TOQL query asks for all the products (both product key and product name) with value greater than 10.0. TOQL returns:

Query Result:

Product	**productName**
Product2	P2
Product3	P3
Product3	P3x

There are only two products (Product2 with name P2 and Product3 whose name was initially P3 and then changed to P3x) satisfying this query.

Query example 2:

```
SELECT Product, productName
FROM Product
WHERE price > 10.0
MINUS
SELECT Product, productName
FROM Product
WHERE price > 17.0
```

The above query asks for products (product key and product name) with value higher than 10.0 and less than 17.0. The same asks also the next query, in a different way:

Query example 3:

```
SELECT Product, productName
FROM Product
WHERE price > 10.0 AND price <= 17.0
```

TOQL returns product P2 for both queries:

Queries Result:

Product	**productName**
Product2	P2

Query example 4:

```
SELECT Company, Company.hasEmployee.TIME
FROM Company, Employee AS E
WHERE Company.hasEmployee:E AND E.employeeName LIKE "John"
```

This query asks for the companies with employees whose name is "John" and the time period in which he was employeed. TOQL returns:

Query Result:

Company	**hasEmployee_startValue**	**hasEmployee_endValue**
Company2	3	7
Company1	1	5

Company1 had an employee named "John" between times 1 and 5, while *Company2* had an employee named "John" between times 6 and 10.

Query example 5:

```
SELECT Company, Company.companyName
FROM Company, Product AS Prod1, Product AS Prod2
WHERE Company.produces:Prod1 AND Prod1.productName LIKE "P1" AT(3)
AND Company.produces:Prod2 AND Prod2.productName LIKE "P2" AT(8)
```

This query asks for all the companies (company key and company name) that produced a product named "P1" at time point 3 and product named "P2" at time point 8. TOQL returns:

Query Result:

Company	**companyName**
Company1	C1

Company1, named "C1" satisfies this query.

Query example 6:

```
SELECT Product
FROM Product
WHERE Product.productName LIKE "P3" BEFORE Product.productName LIKE "P3x"
```

This query asks for a product (product key) that at some time point it was named "P3" and then it was renamed to "P3x". TOQL returns:

Query Result:

Product
Product3

TOQL returns Product3 whose name was initially P3 and then changed to P3x.

Query example 7:

```
SELECT Product, Product.productName
FROM Product
WHERE price AT(5) < price AT(10)
```

This query asks for a product (product key and name) whose price at time point 5 is smaller than its price at time point 10 . TOQL returns:

Query Result:

Product	**productName**
Product3	P3

Product3 is the only product satisfying this query (its value at time point 4 is 20 and then changed to 20 at time 10).

Query example 8:

```
SELECT Product, Product.productName
FROM Product
WHERE price AT(10) >= ALL (SELECT price FROM Product AS P1)
```

This query asks for a products (product key and name) that at time point 10 are more expensive than all other products now and in the past.

Query Result:

Product	**productName**
Product3	P3x

Product3 at time point 10 had price 22.0 which is the higher price of the kownledge base.

Chapter 4

Implementation-Application

To fully support TOQL an application has been created (Figure 4.1). This application includes an interpreter, an ontology parser and a Knowledge Base querying component. A Graphic User Interface (GUI) has also been implemented. The application is implemented in JAVA and uses external libraries such as SESAME [12], [2] and JENA [1] to load and to query ontologies and JGraph [3] to display ontologies graphs. The interpreter's input is a TOQL query while its output is a query in SeRQL [11]. The query is submitted to the knowledge base and the result is returned to the user. The knowledge base is in OWL [6]. The following sections describe in detail the application and all the implementation issues. Section 4.1 describes the interpreter, Section 4.2 gives an overview of the application and describes the Graphic User Interface (GUI) and Section 4.3 describes the ontology abstract view that is used through the whole application and represents the component that visualizes this view in order to facilitate the query writting.

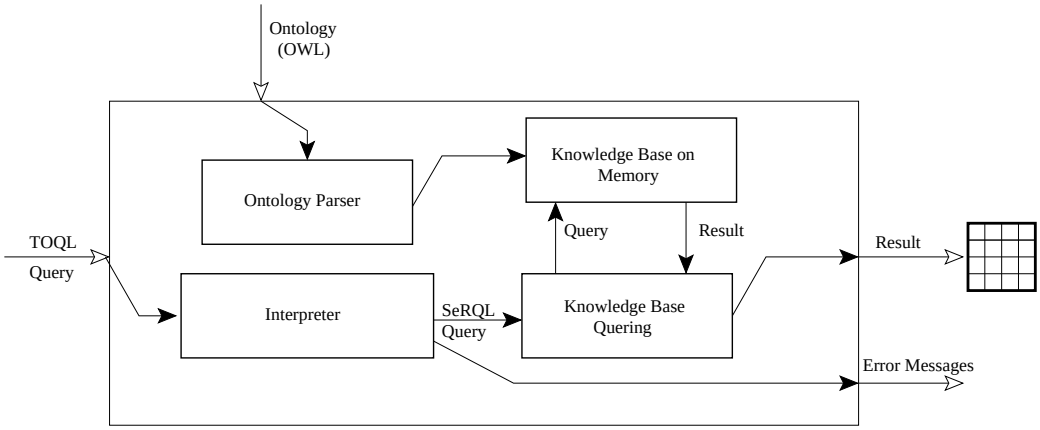


Figure 4.1: TOQL query processing system architecture

4.1 Interpreter

Apart from proposing TOQL, an interpreter for TOQL has been implemented as well. This interpreter takes as input queries written in TOQL and it outputs queries written in SeRQL [11]. Instead of writing a program for executing TOQL queries on OWL (which can be complicated) we preferred the solution with the interpreter that transforms TOQL to SeRQL. SeRQL is then used to query the KB. TOQL and SeRQL have different syntax. SeRQL was chosen as a target language for TOQL for two reasons:

- Similar to TOQL, SeRQL is an SQL like language.
- SeRQL supports comparison between datetimes which is very usefull since TOQL's main goal is to support queries on time and time operators.

Notice that SeRQL does not support any other TOQL's time features such as ALLEN operators, AT clause and 4D fluent mechanism. The interpreter creates rather complicated SeRQL queries to support all these special features.

Well known from compilers theory [8], the procedure of language interpretation can be roughly divided into four steps:

- Lexical analysis
- Syntax analysis
- Semantic analysis
- Code generation (follows intermediate code generation)

An input TOQL query is initially lexically, syntactically and semantically analyzed before it is translated into a equivalent SeRQL one. For lexical, syntax and semantic analysis JFlex 1.4.1 [4], [16] and Byacc/J 1.14 [5] have been used. Both of these tools create Java code. Figure 4.2 illustrates the procedure followed to convert a TOQL query into SeRQL.

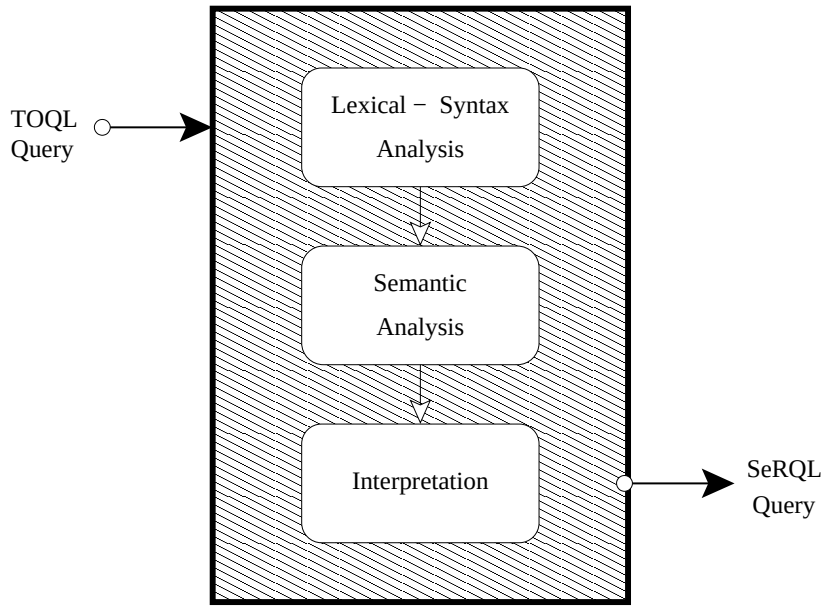


Figure 4.2: Converting a TOQL query into SeRQL

4.1.1 Lexical and Syntax analysis

Lexical analysis is the process of converting a sequence of characters into tokens (a sequence of characters that has been given a meaning-label). The input query is parsed by the lexer, a program created by the JFlex [4], [16] and tokens in the query expresssion are recognized. Each token is given a predefined meaning. For example the token SELECT is given the meaning SELECT while the token IBM is given the meaning NAME.

The next step is syntax analysis (parsing). Parsing is the process of analyzing a sequence of tokens to determine grammatical structure with respect to a given formal grammar. Parser captures the implied hierarchy of the input query and transforms it into a syntax tree, a form suitable for further processing. The sequence of tokens produced by the lexer is being processed by the parser created by Byacc/J [5]. The parser checks if the tokens form an allowable expression. This is done with reference to the TOQL's grammar (Appendix A), which recursively defines components (tokens) that can make up an expression and the order in which they must appear in the expression.

If lexical or syntax analysis end up with errors (the query has wrong syntax or uses invalid keywords) query processing terminates and error messages are returned. These error messages help the user to correct the query and resubmit it. If the query is lexically and syntactically correct, the query translation proceeds with semantic analysis.

4.1.2 Semantic analysis

Semantic analysis is the process of adding semantic information to the parse tree and building up the symbol table. It needs a complete and correct parse tree which means that follows lexical and syntax analysis and that lexical and syntax analysis must end up with no errors.

This phase is parsing a number of semantic checks. There are two types of semantic checks. The first type are all those checks that need no external knowledge. Three errors may occur in this category: the first is to use a class in SELECT clause or in WHERE clause without having declared it first in FROM clause (Table 4.1), the second is to use a property in SELECT clause or in WHERE clause without a class preceding it (table 4.2) and the third type of error is to use more than one properties in the SELECT clause of a nested query. Notice that, the second case will not cause an error if there is only one class declared in FROM.

Wrong Query	Correct Query
SELECT Company.companyName **FROM** Employee **WHERE** Company.hasEmployee:Employee AND Employee.employeeName LIKE “x”	**SELECT** Company.companyName **FROM** Company, Employee **WHERE** Company.hasEmployee:Employee AND Employee.employeeName LIKE “x”

Table 4.1: Example of parse error: Class Company used but not declared

Wrong Query	Correct Query
SELECT companyName **FROM** Company, Employee **WHERE** Company.hasEmployee:Employee AND Employee.employeeName LIKE “x”	**SELECT** Company.companyName **FROM** Company, Employee **WHERE** Company.hasEmployee:Employee AND Employee.employeeName LIKE “x”

Table 4.2: Example of parse error: Property companyName must follows a Class

The second category includes all those checks that need external knowledge (the ontology). This requires that the ontology is first loaded into the main memory (see also Section 4.2). In this section let us assume that the ontology is loaded into the memory and that semantic analyzer uses this information to check the queries for errors. The semantic analyzer checks if a class or property used in a query exists in the ontology, if a property is property of a specific class and finally if it is a fluent property so that keyword TIME can be applied to it. Table 4.3 contains all the error messages that can be returned by semantic checker and their meaning. If the semantic analysis ends up with errors query processing terminates.

Error Message	Meaning
Property is not fluent: *property name*	A static property is used as fluent
Class is not declared: *class name*	A class is used but it is not declared
Property is named after table: *property name*	A property has the same name with a class
Property declaration without table: *property name*	A property is used with no preceding class
Only one property can be used in a nested query: *property name*	A nested query can have only one property in SELECT
Class *class* does not have datatype property: *property name*	A datatype property follows a class that is not its domain
Class *class* does not have object property: *property name*	An object property follows a class that is not its domain
Class *class* does not have property: *property name*	A property follows a class that is not its domain
Class *class* does not have dynamic property: *property name*	A static property is used in place of a dynamic
Class does not exist: *class name*	A class is used that does not exist in ontology
Class is already declared: *class name*	A class is declared twice in FROM

Table 4.3: List of semantic errors

The most important part of the semantic analysis is the *symbol table*. *Symbol table* is a data structure that holds declarations so that inconsistencies and misuses can be detected. In TOQL, the symbol table is mainly used to detect multiple declarations of a class in the same query (see Section 3.4.3) and to handle the scopes of the sub-queries.

Many different implementations of symbol table are known to exist using different data structures (Binary Search Trees (BST), Linked Lists, Hash Tables). Some, of these implementations use one large symbol table for all symbols while others use separated, hierarchical symbol tables one for each

different scope.

In this work, separated symbol tables, one for each scope, are created. Each symbol table is implemented as a hash table. Besides a key (class name in our case), the hash table holds additional information for each class indicating whether class has been renamed or not (boolean value) and initial class name (e.g., Company AS C1, Company is the initial class name and C1 the class name). Symbol table stores class declarations. Whenever the parser returns a class declaration, the symbol table of the current scope is searched (*lookuped*) and the class is added into the symbol if not already there. If it is already there, there is a duplicate declaration and an error is returned. Similarly, whenever the parser returns a class usage, the symbol table is searched (*lookuped*). The difference between this and first case is that an error is returned if the class is not exist in the symbol table (class usage without having been declared). Finally a stack of symbol tables is created. Each node of this stack points to a separate symbol table corresponding to a different scope. Whenever the parser enters a nested query a new node is added at the top of the stack and whenever the parser exits a subquery the top node is removed. When parsing finishes, the stack is empty. Figure 4.3 illustrates the symbol table.

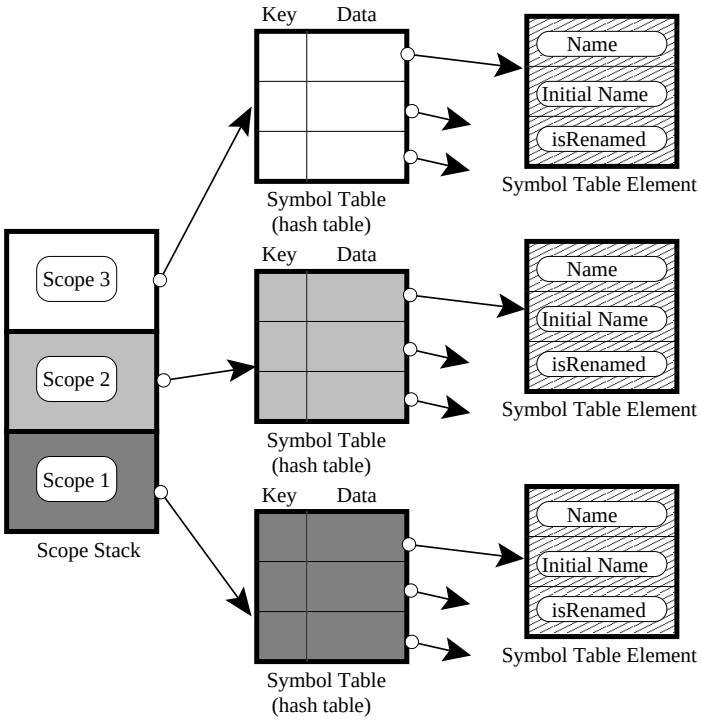


Figure 4.3: Symbol Table

4.1.3 Code generation

The last phase of query processing is the actual translation of a TOQL into an equivalent SERQL query. It follows lexical, syntax and semantic analysis and requires that no error occurred during these phases. Code generation is performed in steps as follows (Figure 4.4):

- Intermediate code generation.
- Intermediate code parsing and Java objects instantiation. These Java objects represent the TOQL query.
- Java objects processing and expansion with 4D fluent elements.
- Java objects processing and mapping to Java objects representing the SeRQL query. SeRQL query creation out of these objects.

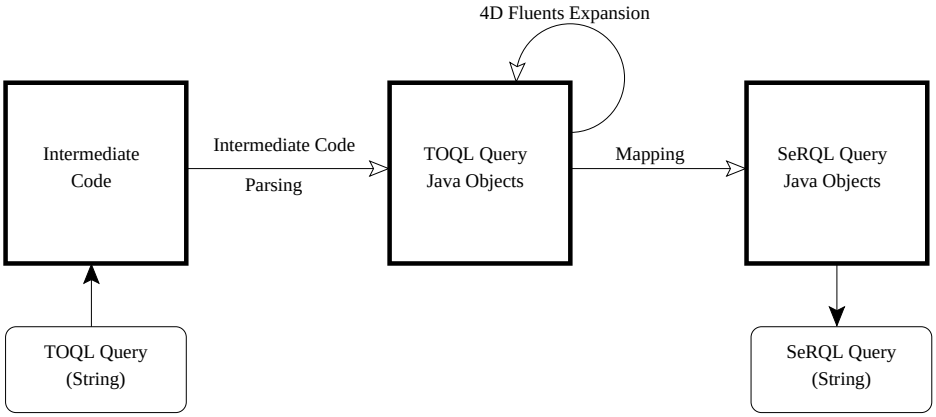


Figure 4.4: Code Generation

Intermediate code generation

The input to code generation is the parse tree. Intermediate code is a linear sequence of instructions in an intermediate language. This linear sequence of instructions is a direct convertion of the parse tree and is prior to final code generation. In this work intermediate code is implemented as a list. Each list node is a Java object that has four fields (**type**, **field1**, **field2** and **field3**). Table 4.4 presents all the different types of nodes and the use of each one of them.

Type	Field1	Field2	Field3	Description
QUERY	-	-	-	Designates the start of a query
END QUERY	-	-	-	Designates the end of a query
BLOCK	-	-	-	Designates the start of a subquery
END BLOCK	-	-	-	Designates the end of a subquery
SELECT	- or DISTINCT	-	-	Designates the start of a SELECT clause
END SELECT	-	-	-	Designates the end of a SELECT clause
FROM	-	-	-	Designates the start of a FROM clause
END FROM	-	-	-	Designates the the end of a FROM clause
WHERE	-	-	-	Designates the the start of a WHERE clause
END WHERE	-	-	-	Designates the the end of a WHERE clause
PAR	"(" or ")"	-	-	Represents a parenthesis
UNION	-	-	-	The keyword UNION
UNION ALL	-	-	-	The keyword UNION ALL
MINUS	-	-	-	The keyword MINUS
INTERSECT	-	-	-	The keyword INTERSECT
NODE	a property's name	- or a class name	- or a property's alternative name	Represents a declaration of type class.property AS "string"
TIMENODE	a property's name	- or a class name	- or a property's - alternative name	Represents a declaration of type *class.property.TIME AS "string"*
NODED	a property's name	- or a class name	-	Represents a declaration of type *class.property* in WHERE clause
NODEC	a class name	-	-	Represents a declaration of type *class* in WHERE clause
LIMIT	integer value	-	-	The keyword LIMIT. Field1 holds the value
OFFSET	integer value	-	-	The keyword OFFSET. Field1 holds the value
OR	-	-	-	The boolean operator OR
AND	-	-	-	The boolean operator AND
NOT	-	-	-	The keyword NOT
VALUE	a value (string, integer,date,...)	the value's datatype	-	Holds a value and its datatype
ALL	-	-	-	The keyword ALL
ANY	-	-	-	The keyword ANY
IN	-	-	-	The keyword IN
EXISTS	-	-	-	The keyword EXISTS
AT	start date	- or end date	-	The operator AT. Field1 holds the start date and Field2 the end date (if any)
ALLEN	BEFORE, AFTER,EQUALS...	-	-	Represents an ALLEN operator
TRUE	-	-	-	The keyword TRUE
FALSE	-	-	-	The keyword FALSE
COMP	"<","<sup>","=", "<=",">=","!="	-	-	A comparison operator
LIKE	a string	- or IGNORECASE	-	The Keywords LIKE

Table 4.4: Intermediate Code Nodes

Table 4.5 illustrates an example TOQL query specifying two sub-queries. This query asks for all the names of Company1 as well as the time period it had each name. Table 4.6 illustrates the intermediate code generated in response to this query.

```
SELECT C1.companyName.TIME as T,
C1.companyName
FROM Company As C1
WHERE C1 like "Company1"
```

Table 4.5: TOQL query example

```
QUERY - - -
BLOCK - - -
SELECT - - -
TIMENODE companyName C1 T
NODE companyName C1 -
END SELECT - - -
FROM - - -
NODE - Company C1
END FROM - - -
WHERE - - -
NODEC localName(C1) - -
LIKE "Company1" - -
END WHERE - - -
END BLOCK - - -
END QUERY - - -
```

Table 4.6: Intermediate code generated in response to the query of Table 4.5

Each line corresponds to a node with fields "Type", "Field1", "Field2", "Field3" (in this order). Empty values are denoted by "-". The first line of intermediate code is the node QUERY and the last one is the node END QUERY. Each "subquery" is surrounded by the nodes BLOCK and END BLOCK. This information is passed to the intermediate code parser and is used to identify sub-queries in the query. Nested queries are treated accordingly.

Intermediate code parsing

In this phase intermediate code is parsed and Java objects are instantiated. This parsing facilitates further processing of the query since it is described in a more "object oriented" way. The parser initially finds the nodes with types BLOCK and END BLOCK, signifying the beginning and ending of subqueries respectively, and reads the nodes in between them. Each subquery is then parsed separately and its nodes data are loaded in the memory. This process is straitforward since every intermediate code node maps to a different Java object (see Figure 4.5). Table 4.7 summarizes the mapping between intermediate code nodes and Java objects:

Node's Type	Action
QUERY	a new **Query** object is created
END QUERY	intermediate code parsing is terminated
BLOCK	a new **SubQuery** object is created
END BLOCK	sub query parsing is terminated
SELECT	Select clause parsing is initiated
END SELECT	Select clause parsing is terminated
FROM	From clause parsing is initiated
END FROM	From clause parsing is terminated
WHERE	Where clause parsing is initiated
END WHERE	Where clause parsing is terminated
PAR	a new **Parenthesis** object is created
UNION	a new **SetOperator** object is created
UNION ALL	a new **SetOperator** object is created
MINUS	a new **SetOperator** object is created
INTERSECT	a new **SetOperator** object is created
NODE	If Select clause parsing is initiated, a new **SelectNode** object is created If From clause parsing is initiated, a new **FromNode** object is created If Where clause parsing is initiated, a new **ObjectProperty** object is created
TIMENODE	a new **SelectNode** object is created
NODED	a new **DatatypeProperty** object is created
NODEC	a new **ClassId** object is created
LIMIT	a new **QuantifierNode** object is created
OFFSET	a new **QuantifierNode** object is created
OR	a new **LogicOperator** object is created
AND	a new **LogicOperator** object is created
NOT	a new **LogicOperator** object is created
VALUE	a new **Value** object is created
ALL	a new **All** object is created. It points to a SubQuery.
ANY	a new **Any** object is created. It points to a SubQuery.
IN	a new **In** object is created. It points to a SubQuery.
EXISTS	a new **Exists** object is created. It points to a SubQuery.
AT	a new **AT** object is created
ALLEN	a new **AllenOperator** object is created
TRUE	a new **BooleanConstant** object is created
FALSE	a new **BooleanConstant** object is created
COMP	a new **ComparisonOperator** object is created
LIKE	a new **Like** object is created

Table 4.7: Mapping between intermediate code and Java objects

Figure 4.5 illustrates the whole structure of Java objects that have been

created to represent a TOQL query. The description of these objects follows. Figure 4.6 illustrates the Java objects instantiated to represent the TOQL query of Table 4.5.

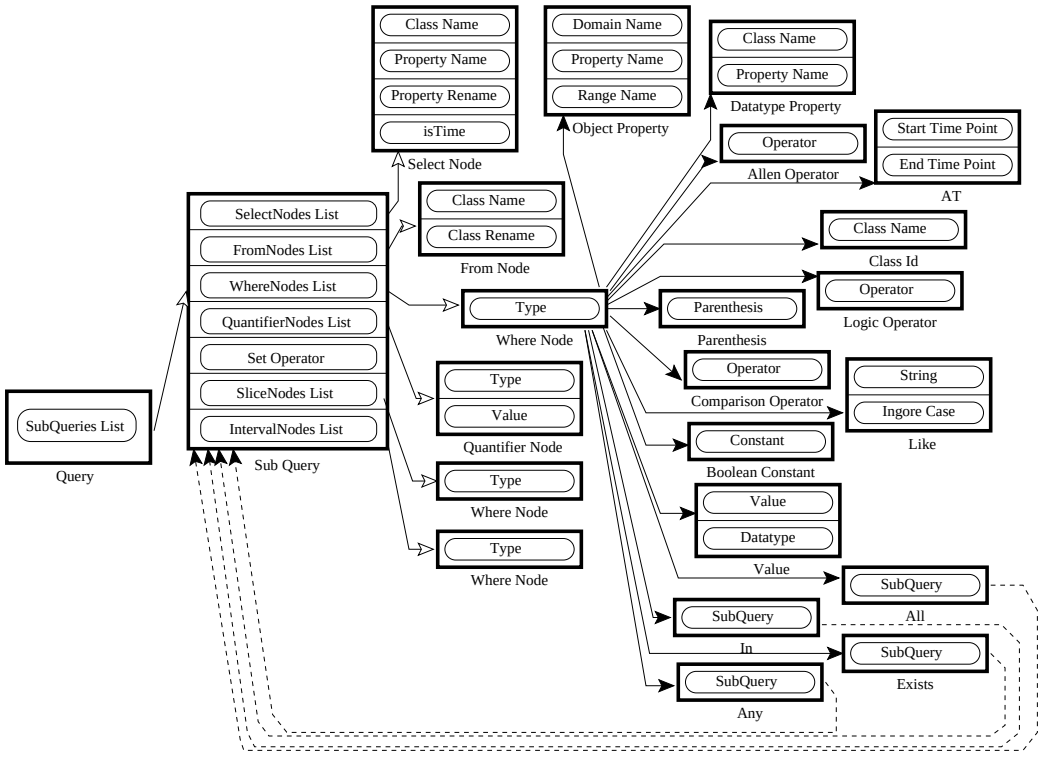


Figure 4.5: Java objects created to represent a TOQL query

The first Java object is the *Query*. *Query* represents a TOQL query and simply has a list of *"Subquery"*. The *"Subquery"* has the following fields:

- **Select Nodes List:** a list of Select Nodes. The elements that may appear after clause SELECT are datatype properties, so a **SelectNode** keeps information about the class name of a datatype property (e.x. *Company*.name) the property name (e.x. Company.*name*), the property's alternative name (e.x. Company.name AS *CompanyName*) and finally whether or not the keyword TIME is used (e.x. Company.name.*TIME*).

- **From Nodes List:** a list of From Nodes. The elements that may appear after clause FROM are class declarations, so a **FromNode** keeps information about the class name (e.x. FROM *Company*, ...) and the class alternative name (e.x. FROM Company AS *C1*).

- **Where Nodes List:** a list of Where Nodes. The elements that may appear after clause WHERE vary, so **WhereNode** is the parent class for many others which are specialized in all these different elements. The only field that **WhereNode** has, is field type that determines the type of the element (node). The classes extending **WhereNode** are:
  - **ObjectProperty:** keeps information about the object properties that appear in clause WHERE (domain class name, property name and range class name).
  - **DatatypeProperty:** keeps information about the datatype properties that appear in clause WHERE (class name, property name).
  - **AllenOperator:** keeps the type of an ALLEN operator (BEFORE, AFTER, EQUALS, MEETS, OVERLAPS, DURING, STARTS, ENDS, ...).
  - **AT:** contains the start and end time point, e.x. AT(*start time point*, *end time point*) of an AT clause.
  - **ClassId:** stores the class name of those classes that appear in clause WHERE with no property following them (class ids see Section 3.4.1).
  - **LogicOperator:** stores the type of a logic operator (AND, OR, NOT).
  - **ComparisonOperator:** stores the type of a comparison operator (<,>,=,!=).
  - **Parenthesis:** stores the type of a parenthesis ("(","(")").
  - **Like:** keeps information about the like clause. This information consists of the string following LIKE clause (e.x. Company.name LIKE *"x"*) and whether or not the clause IGNORE CASE was used.
  - **BooleanConstant:** stores the type of a boolean constant (TRUE, FALSE).
  - **Value:** keeps information about values that appear in WHERE. This information consists of the value (e.x. Product.Price = *5*) and the value's datatype.
  - **All:** clause ALL is followed by a nested query, so class **All** points to a "sub query".
  - **In:** clause IN is followed by a nested query, so class **In** points to a "sub query".

- **Exists:** clause EXISTS is followed by a nested query, so class **Exists** points to a "sub query".
- **Any:** clause ANY is followed by a nested query, so class **Any** points to a "sub query".

- **Quantifier Nodes List:** a list of Quantifiers Nodes. Quantifier nodes are named the keywords LIMIT and OFFSET that may appear at the end of "sub query". So a **QuantifierNode** simply has one field, the type of this node (LIMIT or OFFSET).
- **Set operator:** keeps the set operator (if any) following a "sub query". So **SetOperator** takes one of the following values: UNION, UNION ALL, INTERSECT, MINUS.
- **Slice Nodes List:** a list of Slice Nodes. It is used in Time Slice extension 4.1.3.
- **Interval Nodes List:** a list of Interval Nodes. It is used in Time Slice extension (Section 4.1.3).

The next figure illustrates the Java objects instantiated to represent the TOQL query of Table (Section 4.5).

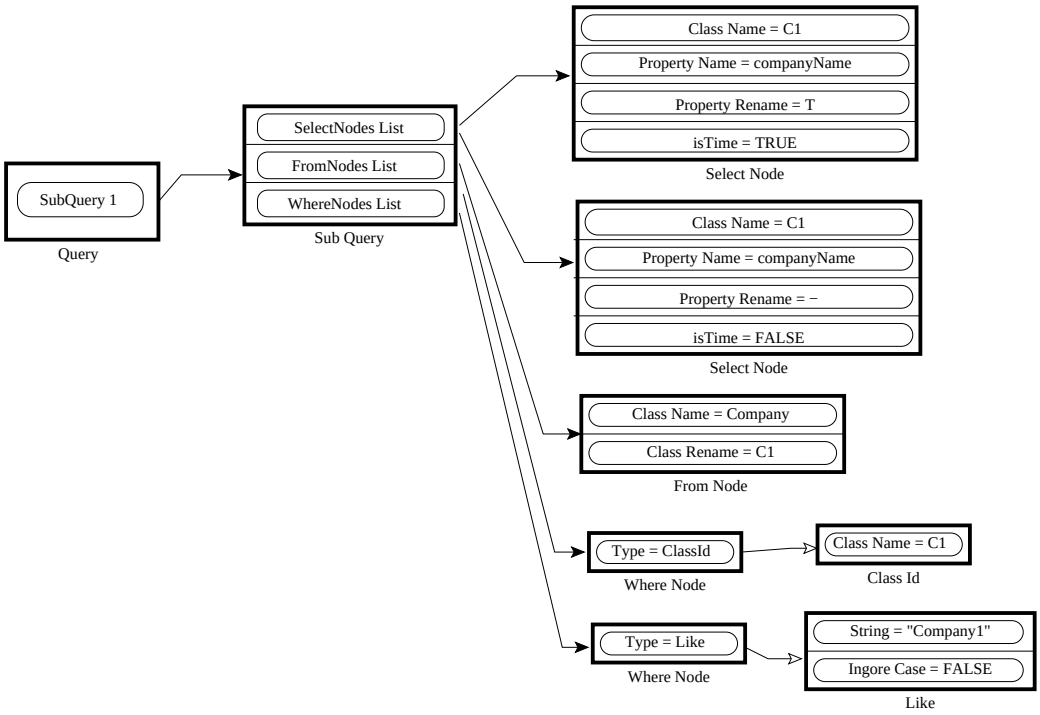


Figure 4.6: TOQL query of Table 4.5 represented by Java objects

For this query, one "SubQuery" object is created. It has two Select Nodes, one From Node and two Where Nodes. The field values of each object are also shown in the Figure 4.6 (e.g. Class Name = C1).

Time Slice extension

In this phase the Java objects representing the TOQL query are being processed. This processing has two goals:

- The first goal is to detect the 4D fluent properties and associate them with corresponding time slices at ontology level. As mentioned in Section 2.3, a fluent property interconnects two time slices or a time slice with a datatype. In TOQL, fluent properties are hadled like the "static" ones. This means that the user does not have to refer to the time slices of a class in order to use a fluent property (see Section 3.3). Consider for example the fluent property productName of "DEn Ontology". To access this property in TOQL, we write *Product.productName*, although the domain of productName is the class *TimeSlice*. This phase detects and handles all these properties.

- The second goal is to replace ALLEN operators and AT clauses with a number of clauses supported by SeRQL, since SeRQL does not them.

This phase uses the fields Slice Nodes List and Interval Nodes List of Java object **SubQuery** (see Figure 4.5). These two lists store additional **WhereNode** instances created by this processing. These instances are used by the next phase of query processing that ends up with the equivalent SeRQL query.

The next figure illustrates the Java objects instantiated to represent the TOQL query of Table 4.5 after time slice extension:

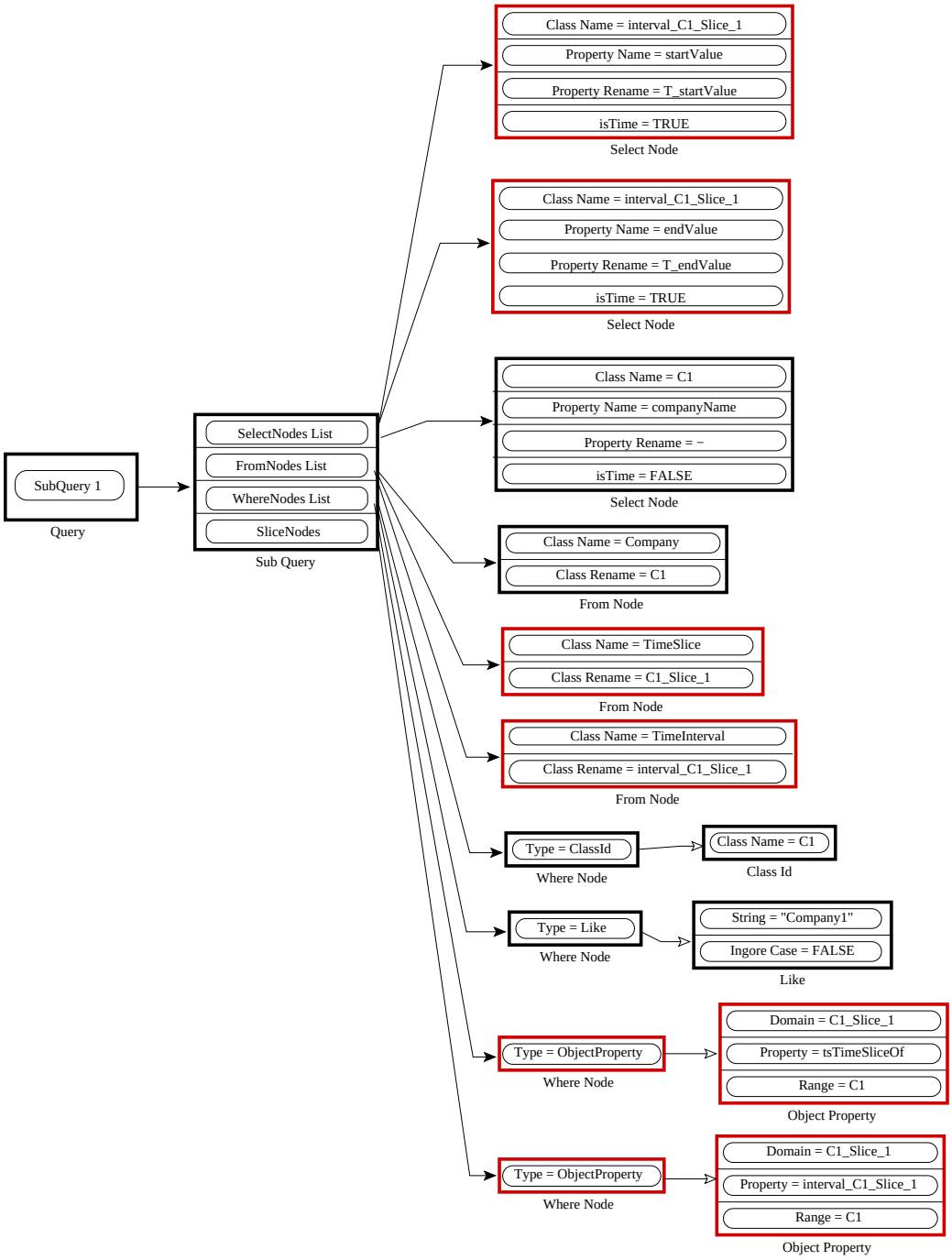


Figure 4.7: TOQL query of Table 4.5 represented by Java objects

The objects with red border have been added/altered by this phase.

SeRQL query generation

The SeRQL's SELECT clause specifies values to be returned, the FROM clause specifies the path expressions and the WHERE clause specifies boolean constraints on variables (Section 2.4.3). In TOQL, (see Chapter 3) the SELECT clause specifies datatype properties that will be returned, the FROM clause specifies the class or classes from which data will be retrieved and the WHERE clause specifies logic operations and comparisons that restrict the number of rows returned by the query.

The translation of a TOQL's SELECT clause to a SeRQL's one is straightforward, since both specify values to be returned. The next table presents a TOQL SELECT clause and how it is transformed into a SeRQL SELECT clause:

TOQL Query	SeRQL Query
SELECT Company.companyName, Employee.employeeName	SELECT companyName_Company, employeeName_Employee

To create a SeRQL SELECT clause from a TOQL one, the interpreter removes the class preceding a property (e.g., *Company*.companyName) and also changes the property's name by extending it with its domain class name (e.g., companyName becomes companyName_Company). This property's name changing ensures that the correct values will be returned by the SeRQL query. Consider the next example:

TOQL Query	SeRQL Query
SELECT C1.companyName, C2.companyName FROM Company AS C1, Company AS C2	SELECT companyName_C1, companyName_C2

This SELECT clause asks for the names of two companies (namely C1 and C2). Extending the property's *companyName* with the domain class (C1 and C2), ensures that the correct values will be returned.

The convertion of TOQL's FROM and WHERE clauses to a SeRQL's one is complicated. A TOQL's FROM clause declares the classes that will be used in the query and a WHERE clause specifies logic operations and comparisons, while a SeRQL's FROM clause specifies the path expressions and a WHERE clause specifies boolean constraints on variable. As mentioned in Section 2.4.3, path expressions can be expressed either in their basic form or by using shortcuts. In this work no shortcuts are used, since this would make the interpreter's implementation rather difficult. Besides every shortcutted path expression can be expressed by a set of basic path expressions (see Section 2.4.3).

Every class and every property in a TOQL query is represented by a path expression in SeRQL. A class is expressed by a path expression consisting of the nodes **class name** and ***namespace*[1]:Class** and the edge **rdf:type**. This path expression relates a class name in a TOQL query with its Class in the ontology. An object property is expressed by a path expression consisting of the nodes **domain class name** and **range class name** and the edge ***namespace*[1]:Object Property**. This path expression relates two classes with an object property. Finally datatype property is expressed by a path expression consisting of the nodes **domain class name** and **datatype property name** and the edge ***namespace*[1]:Datatype Property**. This path expression relates a class with a datatype property. Table 4.1.3 summarizes the way classes and properties are expressed as path expressions.

Resource	Path Expression
Class	{Class name} rdf:type {*namespace*[1]:Class}
Object Property	{DomainClassName} *namespace*[1]:ObjectPropertyName {RangeClassName}
Datatype Property	{DomainClassName} *namespace*[1]:DatatypePropertyName {DatatypePropertyName}

Table 4.8: Classes and properties expressed as path expressions

Let us now present a TOQL query based on the "StEn Ontology" of Figure 2.1 and its equivelant SeRQL:

TOQL Query	SeRQL Query
SELECT C1.companyName, FROM Company AS C1 WHERE C1 LIKE "Company_1"	SELECT companyName_C1, FROM {C1} rdf:type {ex:Company}, {C1} ex:companyName {companyName_C1} WHERE localName(C1) LIKE "Company_1"

This query asks for the Company_1 name. The class C1 is expressed by the path expression {*C1*} *rdf:type* {*ex:Company*}. This expression declares that C1 is of type Company. Similarly property companyName is expressed by the path expression {*C1*} *ex:companyName* {*ex:companyName_C1*}. This expression declares that property companyName has as domain the class C1. Let us now present a more complex query, also based on the "StEn Ontology", and its equivelant SeRQL:

TOQL Query	SeRQL Query
SELECT Product.productName, FROM Company AS C1, Product WHERE C1 LIKE "Company_1" AND C1.produces:Product	SELECT productName_Product, FROM {C1} rdf:type {ex:Company}, {Product} rdf:type {ex:Product}, {Product} ex:productName {productName_Product}, {C1} ex:produces {Product} WHERE localName(C1) LIKE "Company_1"

[1]The namespace of the resource

This query asks for the Products' names produced by Company_1. The class C1 is expressed by the path expression {*C1*} *rdf:type* {*ex:Company*}. This expression declares that C1 is of type Company. Similarly the class Product is expressed by the path expression {*Product*} *rdf:type* {*ex:Product*}, which declares that Product is of type Product. The property productName is expressed by the path expression {*Product*} *ex:productName* {*ex:productName_Product*}. This expression declares that property productName has as domain the class Product. Finally the object property produces is expressed by the path expression {*C1*} *ex:produces* {*Product*}. This expression declares that property produces has as domain the class C1 and as range the class Product. The last path expression actually replaces the expression in WHERE clause *AND C1.produces:Product*.

The transformations of Table 4.1.3 are achieved by parsing the Java objects representing a TOQL query (Figure 4.5) and mapping to the Java objects created to represent a SeRQL query (Figure 4.8). This mapping process follows the rules of Table 4.1.3. The Java objects created to represent a SeRQL query are presented bellow.

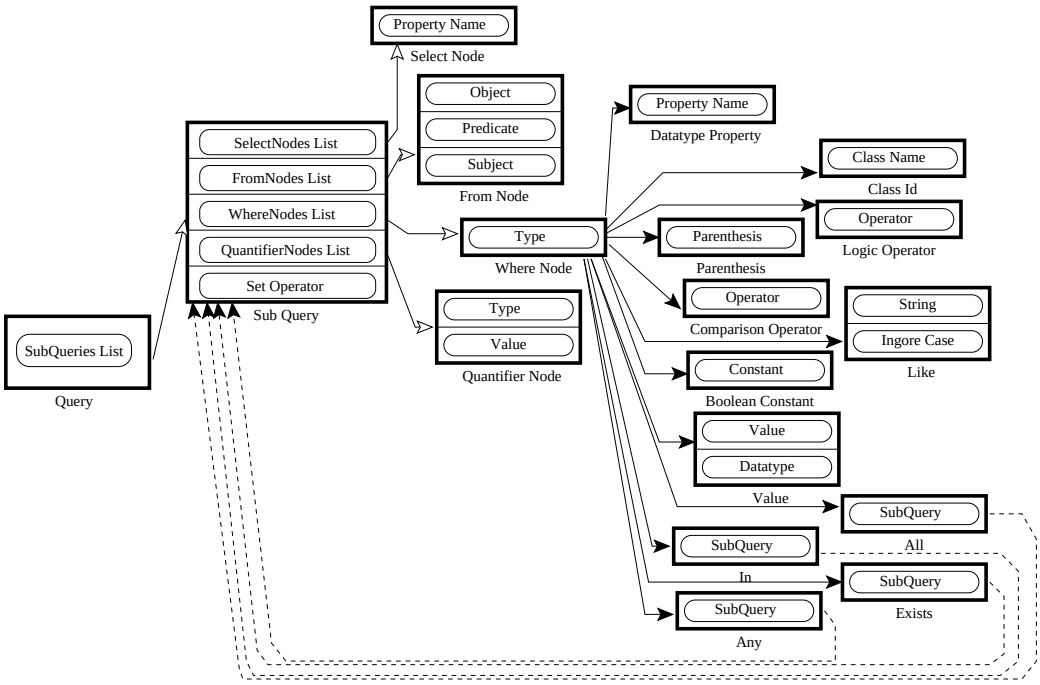


Figure 4.8: Java objects created to represent a SeRQL query

The first Java object is the *Query*. *Query* represents a SeRQL query and simply has a list of *"Subqueries"*. The *"Subqueries"* has the following fields:

- **Select Nodes List:** a list of Select Nodes. In SeRQL only property names may appear after clause SELECT, so a **SelectNode** keeps information only about property name.
- **From Nodes List:** a list of From Nodes. The FROM clause of SeRQL specifies path expressions. In our work a path expression can only consist of a subject, a predicate and an object, so a **FromNode** keeps information about these three fields.
- **Where Nodes List:** a list of Where Nodes. The elements that may appear in WHERE vary, so **WhereNode** is the parent class for many others that are specialized in all these different elements. The only field that **WhereNode** has, is field type that determines the type of the element (node). The classes extending **WhereNode** are:
  - **DatatypeProperty:** keeps information about the datatype properties that appear after WHERE clause (property name).
  - **ClassId:** stores the class name of those classes that appear after WHERE clause with no property following them (class ids see Section 3.4).
  - **LogicOperator:** stores the type of a logic operator (AND, OR, NOT).
  - **ComparisonOperator:** stores the type of a comparison operator (<,>,<=,>=,=,!=).
  - **Parenthesis:** stores the type of a parenthesis ("(","(").
  - **Like:** keeps information about the like clause. This information consists of the string following LIKE clause (e.x. Company.name LIKE *"x"*) and whether or not the clause IGNORE CASE was used.
  - **BooleanConstant:** stores the type of a boolean constant (TRUE, FALSE).
  - **Value:** keeps information about values that appear after clause WHERE. This information consists of the value (e.x. Product.Price = *5*) and the value's datatype.
  - **All:** clause ALL is followed by a nested query, so class **All** points to a "sub query".
  - **In:** clause IN is followed by a nested query, so class **In** points to a "sub query".

  - **Exists:** clause EXISTS is followed by a nested query, so class **Exists** points to a "sub query".
  - **Any:** clause ANY is followed by a nested query, so class **Any** points to a "sub query".

- **Quantifier Nodes List:** a list of Quantifiers Nodes. Quantifier nodes are named the keywords LIMIT and OFFSET that may appear at the end of "sub query". So a **QuantifierNode** simply has one field, the type of this node (LIMIT or OFFSET). A list is used because both of them may appear at the end of a query.

- **Set operator:** keeps the set operator (if any) following a "sub query". So a **SetOperator** takes one of the values: UNION, INTERSECT, MINUS.

The next figure illustrates the Java objects instantiated to represent the SeRQL that is equivalent to the TOQL query of Table 4.5.

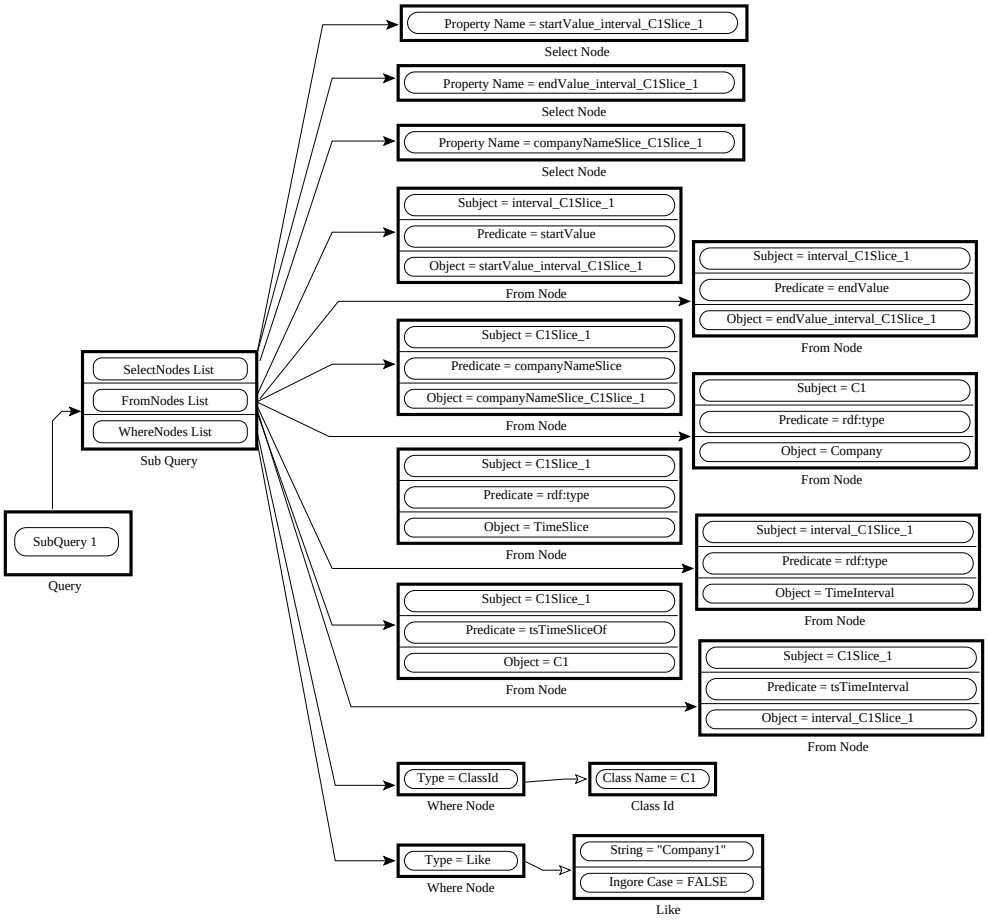


Figure 4.9: SeRQL query, equivalent to TOQL query of Table 4.5

The last step of SeRQL query generation is to parse the objects representing the SeRQL query and to create a string with the actual SeRQL query. The parser processes each Sub Query separately and appends keywords, operators, path expressions and properties to the output string. Table 4.9 summarizes the mapping between the Java objects representing a SeRQL query and the corresponding strings.

Object	String
SubQuery	one or more of “SELECT”, “FROM”, “WHERE”
SelectNode	a property name in SELECT (e.g., “companyName”)
FromNode	a path expression in FROM, subject predicate object (e.g., “Company ex:produces Product”
DatatypeProperty	a datatype property name in WHERE (e.g., “productName”)
ClassId	a class in WHERE (e.g., “Company”)
LogicOperator	a logic operator in WHERE (“AND”, “OR”, “NOT”)
ComparisonOperator	a comparison operator in WHERE (“<”,“>”,“<=”,“>=”,“=”,“!=”)
Parenthesis	an opening of closing parenthesis in WHERE (“(”,“)”)
Like	the keyword “LIKE” followed by a string in quotes and the keyword “IGNORE CASE” (optional) e.g, “LIKE “Product1” IGORE CASE”
BooleanConstant	a boolean constant in WHERE (“TRUE”, “FALSE”)
Value	a value and its datatype in WHERE (e.g., “5xsd:int”)
All	the keyword “ALL” in WHERE
In	the keyword “IN” in WHERE
Exists	the keyword “EXISTS” in WHERE
Any	the keyword “ANY” in WHERE
QuantifierNode	one of the keywords “LIMIT” and “OFFSET”
SetOperator	one of the keywords “UNION”, “UNION ALL”, “MINUS” and “INTERSECT”

Table 4.9: Mapping between Java SeRQL objects and corresponding string

Table 4.10 contains the produced SeRQL query for the TOQL query of Table 4.5. Notice that the clause USING NAMESPACE has been added at the end of the query even if the user has not provide it with the TOQL query. Namespace uniquely characterizes ontology resourses. Namespaces are retrieved automatically from the ontology and are given a symbolic name (at this case the symbolic name is ex1). More than one namespaces may be utilized in one query.

4.2 Application - GUI

An overview application (query translation, knowledge base querying) implemented in this work is presented below. The application is implemented in Java. Also the Graphic User Interface (GUI), developed for this application, is described as well. Section 4.2.1 describes the application, while Section 4.2.2 describes the GUI.

4.2.1 Application

The interpreter is the main component of the application. Apart from that, the application incorporates components for ontology parsing and loading into the memory, knowledge base querying, error reporting and presentation of results. Figure 4.1 illustrates the system architecture with all its components.

SELECT startValue_interval_C1Slice_1,
endValue_interval_C1Slice_1, companyName_C1Slice_1
FROM {interval_C1Slice_1} ex1:startValue {startValue_interval_C1Slice_1},
{interval_C1Slice_1} ex1:endValue {endValue_interval_C1Slice_1},
{C1Slice_1} ex1:companyName {companyName_C1Slice_1},
{C1} rdf:type {ex1:Company},
{C1Slice_1} rdf:type {ex1:TimeSlice},
{interval_C1Slice_1} rdf:type {ex1:TimeInterval},
{C1Slice_1} ex1:tsTimeSliceOf {C1},
{C1Slice_1} ex1:tsTimeInterval {interval_C1Slice_1}
WHERE localName(C1) Like "Company_1"
USING NAMESPACE
ex1= <http://www.owl-ontologies.com/Ontology1197730146.owl#>

Table 4.10: SeRQL query example

The input is a query written in TOQL and an ontology in OWL. Notice that the ontology must be in RDF/XML or in RDF/XML-ABBREV syntax. The query is translated into SeRQL by the interpeter, following the process described in Section 4.1. The ontology is parsed using JENA and SESAME libraries and is loaded into the main memory (see Section 4.3). The ontology is checked for consistency with the 4D fluent mechanism. This parser implements the following list of checks:

1. If the ontology implements the 4D fluent mechanism (2.3), *TimeSlice* must be the name used for time slices.

2. If the ontology implements the 4D fluent mechanism (2.3), *TimeInterval* must be the name used for time intervals.

3. If the ontology implements the 4D fluent mechanism (2.3), *startValue* and *endValue* must be the property names used to declare the start and end time point of time intervals.

4. If the ontology implements the 4D fluent mechanism (2.3), the domain of the fluent object and fluent datatype properties as well as the range of fluent object properties must be a restriction of type *only* on property *tsTimeSliceOf*.

5. If the ontology implements the 4D fluent mechanism (2.3), class *TimeSlice* must not be domain or range of any property except for *tsTimeSliceOf*.

6. If the ontology implements the 4D fluent mechanism (2.3), time slices must be connected with the time intervals via the object property *tsTimeInterval*.

7. If the ontology implements the 4D fluent mechanism (2.3), class *TimeInterval* must not be domain or range of any property except for *tsTimeInterval*, *startValue* and *endValue*.

8. If the domain and the range of a property is a restriction it must be of type *only* or *some*.[2]

Table 12 illustrates the list of error messages that may arise during this process:

[2]Currently the application does not support other restriction types.

Error Message
PropertyDomainError.Error while loading ontology: Class *className* is not concidered a class Try using on of the: *list of class* As domain of property *propertyName*
PropertyRangeError. Error while loading ontology: Class *className* is not concidered a class Try using on of the: *list of class* As range of property *propertyName*
PropertyRestrictionError. Error while loading ontology: Domain restriction of *propertyName* is not curently supported Try using on of the: All Values From (only), Some Values From (some)
TimeSliceRestrictionError. Error while loading ontology: Restriction of *propertyName* must be of type All Values From (only)
RangeNotFluentError. Error while loading ontology: Range of *propertyName* must be a restricion of type All Values From (only) on property tsTimeSliceOf
RangeFluentError. Error while loading ontology: Range of *propertyName* must not be a restricion of type All Values From (only) on property tsTimeSliceOf
TimeSliceClassError. Error while loading ontology: Domain of *propertyName* must be class TimeSlice but it is class *className*
TimeIntervalClassError. Error while loading ontology: Domain of *propertyName* must be class TimeInterval but it is class *className*

Table 4.11: List of error messages in response to ontology loading into main memory

The Knowledge Base Quering unit uses the SeRQL query created by the interpreter to query the ontology. We should mention that when we refer to ontology we mean both the ontology (structure - ABOX) and the knowledge base (instances - TBOX). The application's output is a table with the results returned by the ontology. If errors have been encountered during interpretation, SeRQL query is an empty string and the application's output is one or more error messages.

To run the application one has two choices:

- The first choice is to run the application on a shell by typing

java -jar TOQL.jar *ontology path query path*

The results or the error messages are printed on the screen.

- The second choice is to run the application using the Graphic User Interface (GUI). To invoke the GUI the user should write

  java -jar TOQL.jar

  A brief description of GUI is given in Section 4.2.2

4.2.2 GUI

GUI (Graphic User Interface) accommodates the creation of TOQL queries as it supports syntax highlighting. It also support loading the ontology into the main memory (especially useful of processing a series of queries). Figure 4.10 illustrates the GUI.

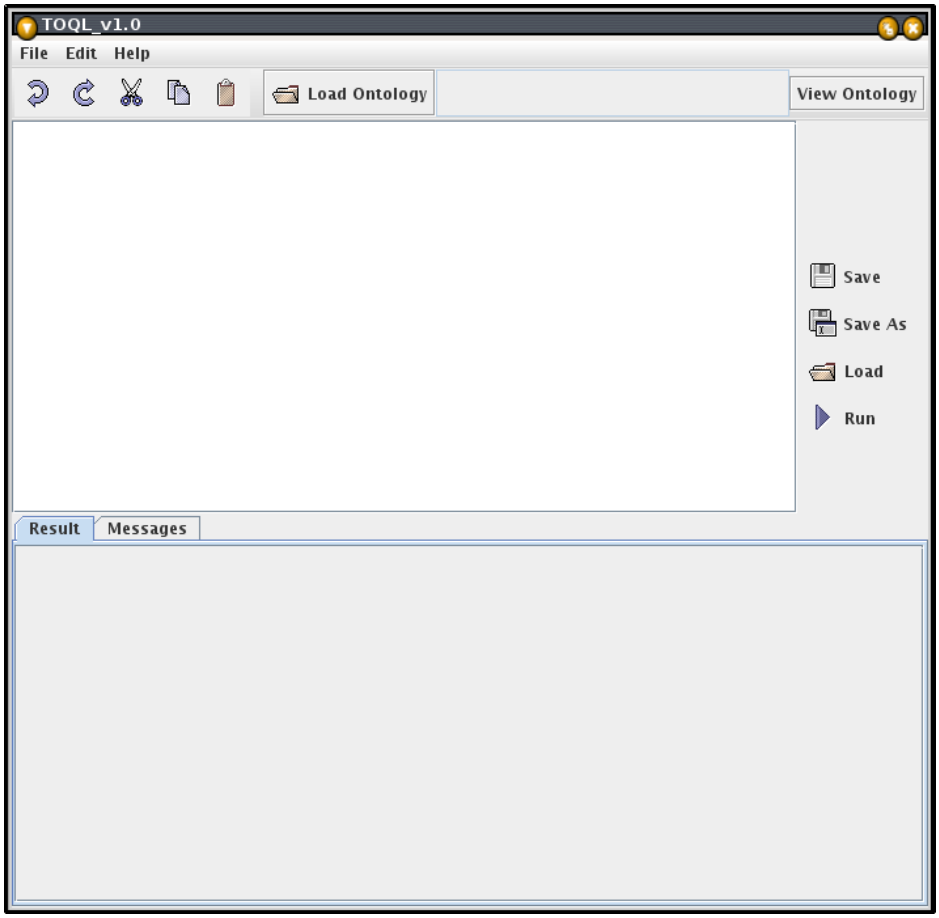


Figure 4.10: Application's GUI

The GUI is rather simple. It is divided into three panels:

- The toolbar panel
- The query editing panel
- The results panel

The toolbar panel contains buttons useful for querying editing (undo, redo, copy, cut, paste). These operations can also be accessed through the menu toolbar. It contains a button for ontology loading (Load Ontology) and button for ontology viewing (View Ontology). The usage of button "Load Ontology" is obvious while the usage of button "View Ontology" will be explained in Section 4.3.

The query editing panel is also splitted into two subpanels. The first is the actual query editor while the second one is a toolbar panel that contains

buttons useful for querying editing (save, save as, load query, run). These operations can also been accessed by the menu toolbar. The query editor facilitates the query creation by introducing two operations:

- **TOQL syntax highlighting:** a syntax highlighter specifically for TOQL has been created (Figure 4.11). It recognizes clauses (SELECT, WHERE, ...), keywords and classes-properties. In order to recognize classes and properties an ontology must be loaded into the main memory first.

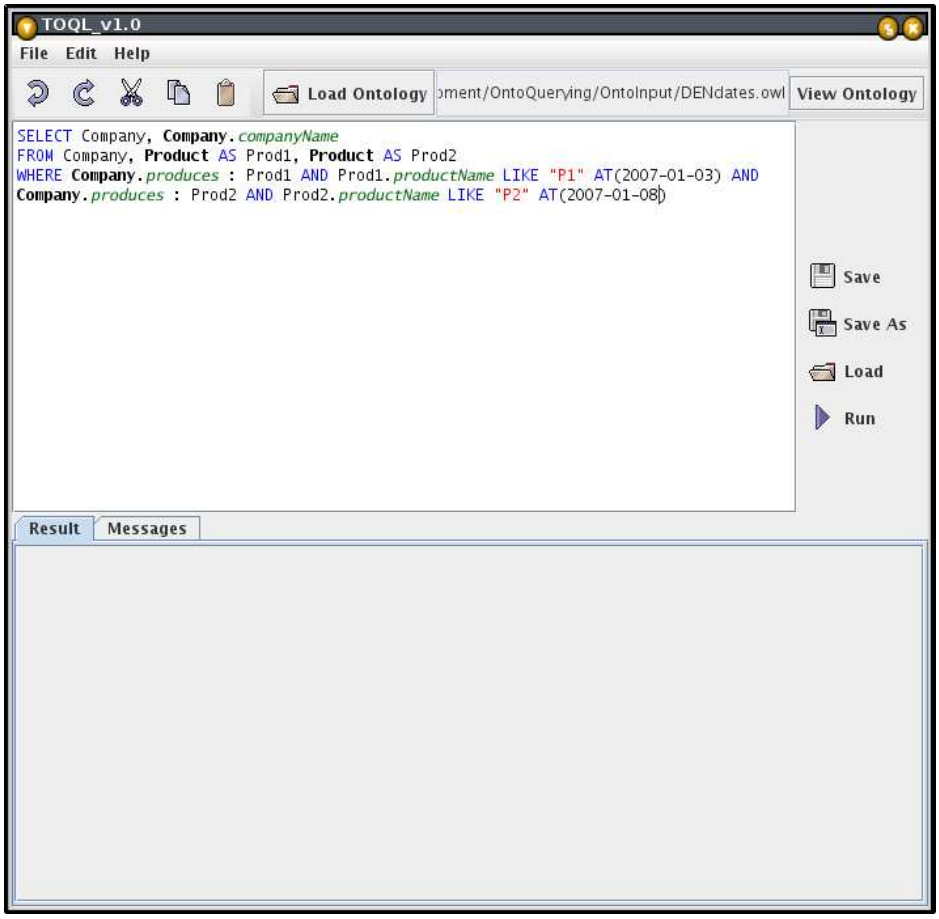


Figure 4.11: Editor highlighting

- **Code autosuggestion:** requires loading an ontology first. Each time the user writes a class name followed by a dot (".") a list with the class properties is displayed. The user can choose one of the properties (Figure 4.12).

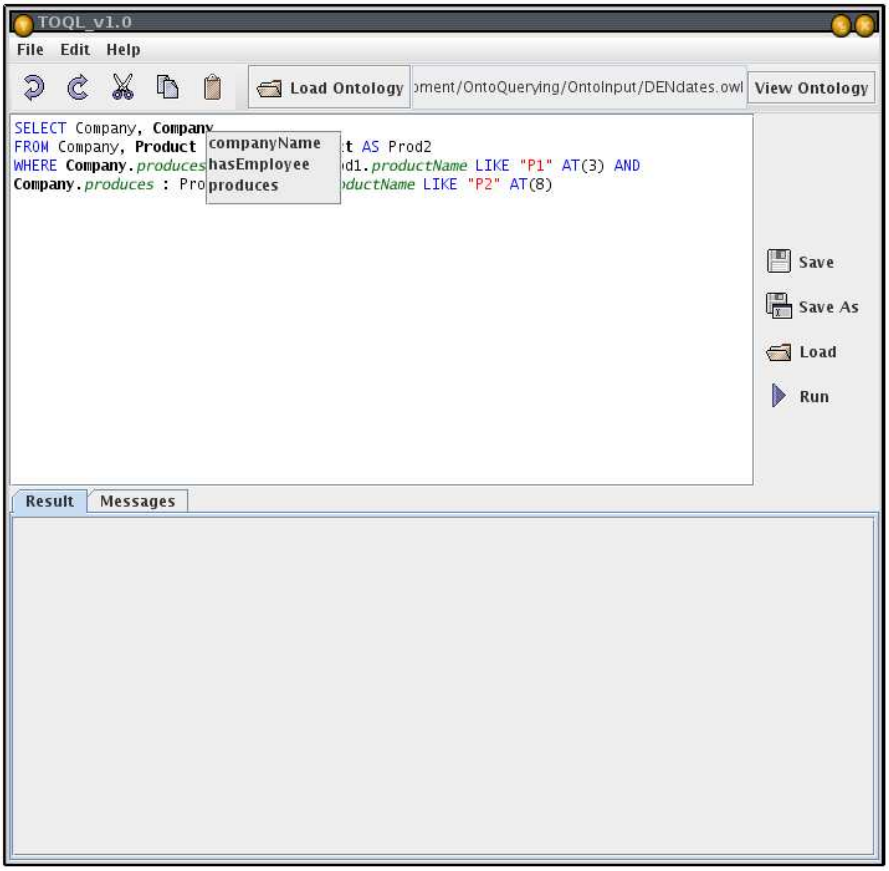


Figure 4.12: Code autosuggestion

Finally the results panel has two tabs. The first one displays the results returned by the query (Figure 4.13), while the second one displays the errors be returned as the result of query parsing. These errors can be either due to inconsistencies with the 4D fluent representation or due errors in TOQL syntax . (Figure 4.14).

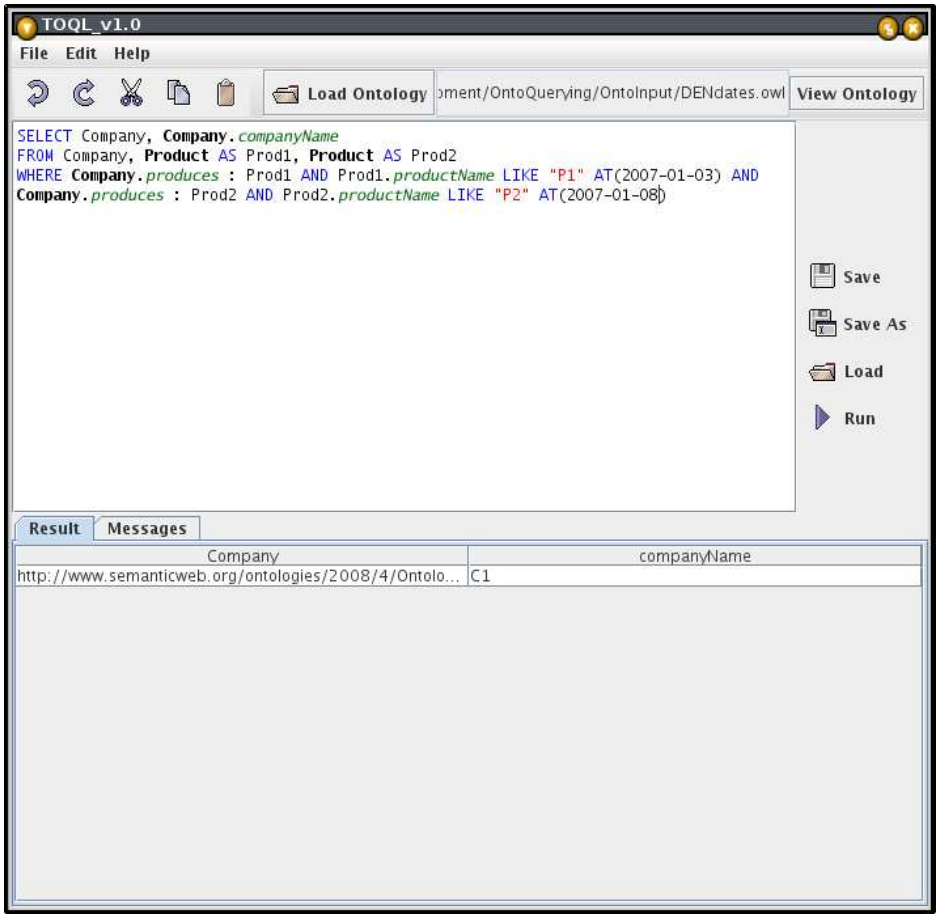


Figure 4.13: Displaying results

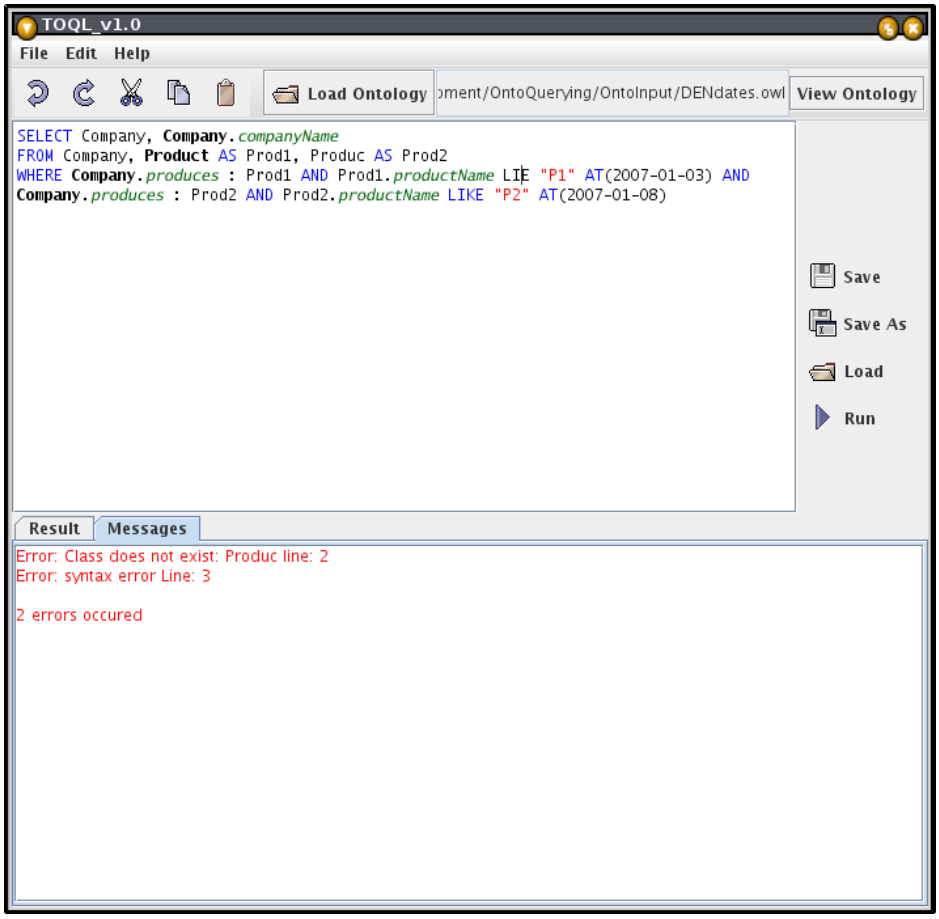


Figure 4.14: Displaying errors

4.3 Ontology Abstract View

As mentioned before, TOQL is high level language. It is designed to support querying on ontologies that implement the 4D fluent mechanism [21], and it is designed in a way that the user does not have to be aware of this mechanism. But how the user queries the ontology if he is not aware of this mechanism? The answer is that the user has to aware of an abstract form of the ontology. This abstract ontology is designed to match TOQL's syntax and semantics. Using this ontology the user is able to query on ontologies that implement this mechanism. Section 4.3.1 presents its design and Section 4.3.2 its implementation and its Graphic User Interface (GUI).

4.3.1 Design

The goal is to create a mechanism in which the input will be a 4D fluented ontology and the output is an abstract view of this ontology that will match TOQL's syntax and semantics. To fulfill this goal, all the class and properties that the 4D fluent mechanism introduces are excluded from viewing. The fluent properties that connect Time Slices are now considered to connect directly the static classes *but* they also have and a Time dimension.

Consider the "Dynamic Enterprice Ontology" of Figure 2.2. The abstract ontology that will come out from this one will have three classes, namely *Company*, *Product* and *Employee* and two datatype properties *companyName* and *employeeName*. Apart from these it will also have the datatype properties *productName* and *Price* and the object properties *hasEmployee*, *produces* that will have an extra attribute, TIME Notice that the abstract ontology does not refer to the knowledge base. The user only needs to know that "abstractly" the property *hasEmployee* interconnects the classes *Company* and *Employee*, which is true, and that it has a time dimension, which is also true,since it connects *TimeSlice* classes (see Figure 4.15. Properties with time dimension are markes with blue color).

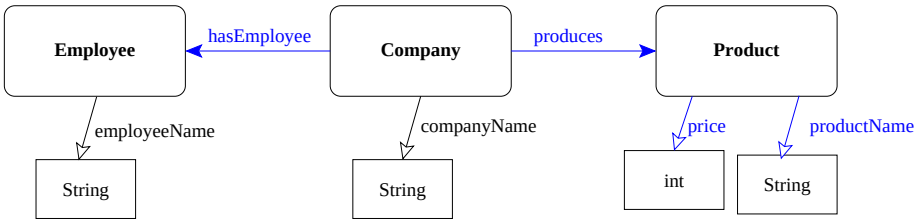


Figure 4.15: Figure's 2.2 ontology abstract view

Notice that this ontology view is actually used by the semantic analysis (Section 4.1.2) and by code autosuggestion (Section 4.2.2) although it was not mentioned there. In Section 4.3.2 the Java objects created to handle this "abstract" view and the Graphic User Interface (GUI), implemented to illustrate this it to the user and to accommodate query creation, are described.

4.3.2 Implementation-GUI

As mentioned in Section 4.2.1, ontology is parsed using JENA and is loaded on memory. Java objects have been created that map the ontology to the abstract ontology described above. Notice that TOQL does not support querying on the structure of an ontology (e.x. ISA relations) so there is no

need for this relations to be mapped into the Java objects. The Java objects that have been created are:

- **Resource:** It is the parent class of all the classes. It has two fields, *name* and *namespace*.
- **Concept:** Represents the concepts. It has two lists, one with all the datatype properties of the concept and one with all the object properties.
- **Property:** Represents the properties and it is the parent class of *Datatype* and *Object*. It has two fields, *domain* (which is of type *Concept*) and *hasTime* (boolean).
- **Datatype:** Represents the datatype properties. It has one field, *range*.
- **Object:** Represents the object properties. It has one field, *range* which is of type *Concept*.

A component that creates a graph out of these Java classes has been created. This component uses the library JGraph [3] . Clicking the button "View Ontology" (see Figure 4.1 displays this graph. Figure 4.16 illustrates this abstract view for a test ontology.

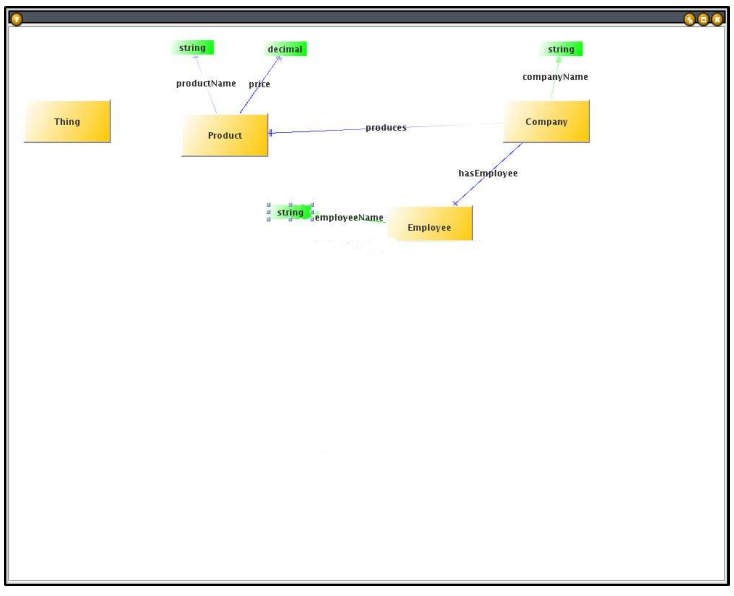


Figure 4.16: Ontology Abstract View

Chapter 5

Conclusions and Future Work

We introduce TOQL (Temporal Ontology Query Language), an ontology query language capable of querying ontologies and temporal information in ontologies. Although independent from the language, temporal concepts are assumed to be represented in OWL (or RDF) using the 4D perdunadist approach [21] implementing events occurring at specific points in time, in time intervals or evolve in time. The language supports a powerful set of operations for such temporal information including Allen operators. An interpreter, translating TOQL queries into SeRQL [11], compined with a GUI, are also implemented. We choose to introduce TOQL because of the ontologies' increasingly important role in Knowledge Representation (KR) domain and because of the fact that dealing with information that changes over time is a critical issue in KR.

A TOQL query is lexically, syntactically and semantically parsed and analyzed. A TOQL query is first tranlated into intermediate code. This interermediate code is then parsed to identify classes addressed by the query. Finally, the TOQL query is translated into an equivalnent SeRQL query which is executed on the ontology. An example TOQL query is provided as well as the intermediate code and the SeRQL query that are a response to it. Also two ontologies "Static Enterprise Ontology" and "Dynamic Enterprise Ontology" along with instances (KB) are provided. Many TOQL queries have been posed on these KBs and the results are presented and discussed.

Converting the interpreter to execute TOQL queries directly on OWL is an extension that allows adding new features in TOQL, currently not available because SeRQL does not support them, such as INSERT, UPDATE, DELETE, ORDER BY, GROYP BY, is possible. Compining TOQL with a reasoner to better support queries on time information is another extension. Future work includes also extending TOQL's syntax to handle queries on ontology structure (i.e., sub-classes and super-classes).

Appendix A

BNF

TOQL grammar:

Query	::= SubQuerySet
SubQuerySet	::= SubQuery [SetOperator SubQuerySet]
SubQuery	::= “(” SubQuerySet “)”
	| Select
SetOperator	::= “union” [“all”]
	| “minus”
	| “intersect”
Select	::= “select” [“distinct”] Projection [QueryBody]
Projection	::= “*”
	ProjectionElem (“,” ProjectionElem)*
ProjectionElem	::= ValueExpr [“as” <String>]
QueryBody	::= “from” FromExprList
	[“where” WhereExprList]
	[“limit” <Int>]
	[“offset” <Int>]
FromExprList	::= FromExpr (“,” FromExprList)*
FromExpr	::= <Name> [“as” <String>]
ValueExpr	::= <Name> [“.” <Name>] [“.” “time”]
	::= <Name> [“.” “*”]
WhereExprList	::= OrExpr
OrExpr	::= AndExpr [“or” WhereExprList]
AndExpr	::= WhereElem [“and” AndExpr]
	::= WhereElem2 AtOperator [“and” AndExpr]
	::= WhereValueExpr AtOperator CompOp
	WhereValueExpr AtOperator [“and” AndExpr]
	::= WhereValueExpr AtOperator CompOp “all”

TOQL grammar (cont):

	“(”SubQuerySet“)” [“and” AndExpr]
	::= WhereValueExpr AtOperator CompOp “any”
	“(”SubQuerySet“)” [“and” AndExpr]
	::= WhereValueExpr AtOperator “in”
	“(”SubQuerySet“)” [“and” AndExpr]
	::= WhereValueExpr AtOperator CompOp
	WhereValueExpr2 [“and” AndExpr]
	::= WhereElem2 AllenOperator WhereElem2
	[“and” AndExpr]
WhereElem	::= “(” WhereExprList “)”
	::= “true”
	::= “false”
	::= “not” WhereElem
	::= WhereValueExpr CompOp (“any”—“all”)
	“(”SubQuerySet“)”
	::= WhereValueExpr “in” “(”SubQuerySet“)”
	::= “exists” “(”SubQuerySet“)”
	::= WhereElem2
WhereElem2	::= WhereValueExpr “like” <String> [“ignore case”]
	::= WhereValueExpr CompOp WhereValueExpr2
	::= <Name> “.” <Name> “:” <Name>
WhereValueExpr	::= <Name> [“.” <Name>]
CompOp	::= “=” | “!=” | “<” | “<=” | “>” | “>=”
WhereValueExpr2	::= WhereValueExpr | <Float> | <Int>
AtOperator	::= “at” “(” <Int> [“,” <Int>] “)”
AllenOperator	::= “before” | “after” | “equals” | “meets”
	| “metby” | “overlaps” | “overlappedby” | “during”
	| “contains” | “starts” | “startedby” | “ends” | “endedby”

Bibliography

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

[2] http://www.openrdf.org/.

[3] http://www.jgraph.com/.

[4] http://jflex.de/.

[5] http://byaccj.sourceforge.net/.

[6] Owl web ontology language overview. *W3C Recommendation*, February 2004.

[7] Sparql query language for rdf. *W3C Recommendation*, January 2008.

[8] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.

[9] James F. Allen and George Ferguson. Actions and events in interval temporal logic. Technical Report TR521, 1994.

[10] David Beckett and Tim Berners-Lee. Turtle - terse rdf triple language. *W3C Recommendation*, 2008.

[11] Aduna B.V. *User Guide for Sesame 2.1*, chapter 9. The SeRQL query language (revision 3.0). 2002-2008.

[12] Aduna B.V. *User Guide for Sesame 2.1.* 2002-2008.

[13] HELD G. D., STONEBRAKER M. R., and WONG E. Ingres-a relational data base system. 44:409–416, 1975.

[14] Richard Fikes, Patrick Hayes, and Ian Horrocks. Owl-ql – a language for deductive query answering on the semantic web. (KSL 03-14), 2003.

[15] Volker Haarslev, Ralf Moller2, and Michael Wessel. Querying the semantic web with racer + nrql. 2004.

[16] Gervin Klein. *JFlex – The Fast Lexical Analyser Generator – JFlex User's Manual*, June 2008.

[17] Brian McBride. Rdf test cases. *W3C Recommendation*, 2004.

[18] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three implementations of squishql, a simple rdf query language. Technical Report HPL-2002-110, April 2002.

[19] Andy Seaborne. Rdql - a query language for rdf. *W3C Recommendation*, January 2004.

[20] Richard Snodgrass. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[21] Christopher Welty, Richard Fikes, and Selene Makarios. A reusable ontology for fluents in owl. IBM Research Report RC23755 (W0510-142), October 2005.

[22] ZHIJUN ZHANG. Ontology query languages : A performance evaluation. August 2005.