TOQL: Temporal Ontology Querying Language

Evdoxios Baratis[1], Euripides G.M. Petrakis[1], Sotiris Batsakis[1], Nikolaos Maris[1], and Nikolaos Papadakis[1]

Department of Electronic and Computer Engineering
Technical University of Crete (TUC)
Chania, Greece
e-mail:{dakis, petrakis, batsakis}@intelligence.tuc.gr, nickmeet@gmail.com, npapadak@intelligence.tuc.gr

Abstract. We introduce TOQL, a query language for querying time information in ontologies. TOQL is a high level query language that handles ontologies almost like relational databases. Queries are issued as SQL-like statements involving time (i.e., time points or intervals) or high-level ontology concepts that vary in time. Although independent from TOQL, this work suggests a mechanism for representing time evolving concepts in ontologies based on the four-dimensional perdurantist mechanism. However, TOQL prevents users from being familiar with the representation of time in ontologies. To show proof of concept, an application has been developed that supports translation and execution of TOQL queries on temporal ontologies combined with a reasoning mechanism based on event calculus. A real world temporal ontology is also implemented on which several TOQL example queries are processed and discussed.

1 Introduction

Dealing with information that changes in time over the semantic web is a difficult problem to deal with. Recent advances in semantic web technology suggest that this can be achieved by adding the concepts of time and change in a rich semantics ontology representation, allowing time to affect the status of the described concepts [13, 6].

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 [8] or the four-dimensional perdurantist approach [15]. According to this approach, all entities are perdurants: 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 support searching for temporal concepts and time related information.

The current state of the art of ontology languages requires submitting a textual, description logic (DL) query or SQL-like query [17]. However the logic and syntax of such languages necessitates a tedious effort from users before being able to write queries effectively. State-of-the-art ontology query languages such as SeRQL [1] or SPARQL [11] have limited (if not at all) expressive power for handling time in queries (their syntax does not support temporal operators). The present work addresses all these issues.

We introduce TOQL (Temporal Ontology Querying Language), 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 (may) vary in time. 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 queries not only on static information in the static part of the ontology (as conventional ontology query languages do) but also supports queries on time evolving information instantiated to the ontology (dynamic part). TOQL supports Allen operators that allow comparisons between time intervals, and the operator AT that allows comparisons between time points or time intervals.

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 inspired by the four-dimensional perdurantist (4D fluent) approach [16]. However, TOQL syntax is independent of any temporal representation and can work with any other mechanism (e.g., versioning). As such, the 4D fluent (perdurantist) mechanism is not part of the language and it is not visible to the user (so the user need not be familiar with peculiarities of the underlying mechanism for time information representation).

In the accompanying implementation, TOQL queries are first translated into equivalent statements in SeRQL which are then executed on the underlying OWL temporal ontology. The query interpreter addresses information in the ontology to generate 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 discussed.

Related work in the field of knowledge representation is discussed in Section 2. This includes, discussion on temporal and ontology query languages along with issues related to representing time evolving information in ontologies. The TOQL language is presented in Section 3 (a formal description of the language's syntax in BNF is given in [3]). The implementation of TOQL is discussed in Section 4, followed by conclusions and issues for future work in Section 5. Several query examples are also given and discussed throughout the work.

2 Background and Related Work

Several representation languages are defined for the Semantic Web, the most important of them are referred to as the OWL-family [10] of ontology languages

(OWL-Full, OWL-DL and OWL-Lite) for ontology building and knowledge representation. OWL-S [5] is an ontology for describing properties and capabilities of web services. Within OWL-S, a sub-ontology, OWL-Time [6] has been developed that is much simpler and provides a vocabulary for expressing the most needed time-related facts.

Dealing with information that changes over time is a critical problem in Knowledge Representation (KR). Representation languages such as OWL, RDF (which are based on description logics),the same as frame-based and object-oriented languages (F-logic) are all based on binary relations. Binary relations simply connect two instances (e.g., the employee with the company) without any temporal information. Nevertheless, time representation using OWL is feasible, although complicated [16].

2.1 Representation of Time

The OWL-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 still a need for mechanisms for the representation of the evolution of concepts (events) in time.This is related to the problem of the representation of time in temporal (relational and object oriented) databases [18]. Existing methods are relying mostly on temporal Entity Relation (ER) models [19] taking into account valid time (i.e., time interval during which a relation holds), transaction time (i.e., time at which a database entry is updated) or both. Also time is represented by time points, intervals or finite sets of intervals. However, time representation in OWL differs because (a) OWL semantics are not equivalent to ER model semantics (e.g., OWL adopts the *Open World Assumption* while ER model adopts the *Closed World Assumption*) and (b) relations in OWL are restricted to binary ones. Time representation in Semantic Web can be achieved using *Temporal Description logics, Reification, Versioning or 4D-fluents.*

Temporal Description Logics (TDL) extend Description Logics (DL) with additional time representation operators and semantics such as *"until"* and *"always in the past"*. Many TDLs have been proposed [20, 21] with the most expresive of them being undecidable. Contrary to other approaches, temporal description logics offer additional semantics and reasoning mechanisms and they don't suffer from data redundancy. All other approaches except TDLs require temporal semantics to be defined using an additional set of rules combined with a reasoning mechanism, as we did in this work. TDLs disadvantage is that they require extending OWL to represent time (by introducing additional operators and semantics), while the other approaches can be implemented directly using OWL.

Reification is a general puprose technique for perpesenting n-ary relations using a language such as OWL that permits only binary relations. Specifically, an n-ary relation is represented as a new object that has all the arguments of the n-ary relation as attributes. For example if the relation R holds between objects A and B at time t, expressed as $R(A,B,t)$, this is represented in OWL using reification as an object R with attributes A, B and t. Reification suffers from two disadvantages: (a) data redundancy, because a new object is created whenever a temporal relation has to be represented (this is a problem common to all approaches based on non temporal Description Logics such as OWL-DL) and (b) offers limited OWL reasoning capabilities [16].

Versioning [8] 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 require that a new version of the ontology be created leading to information redundancy (b) searching for events occurred at time instances or during time intervals requires exhaustive searches in multiple versions of the ontology,(c) it is not clear how the relation between evolving classes is represented. Furthermore, ontology languages such as OWL [10] are based on binary relations (relations connecting two instances) with no time dimension.

The *4D-fluent* (perdurantist) approach [15] shows how temporal information can be represented effectively in OWL. Notice though that it still sufferers from data redundancy. Concepts in time are represented as 4-dimensional objects with the 4th dimension being the time. Time instances and time intervals are represented as instances of a *time interval* class which in turn is related with time concepts varying in time. Changes occur on the properties of the temporal part of the ontology keeping the entities of the static part unchanged.

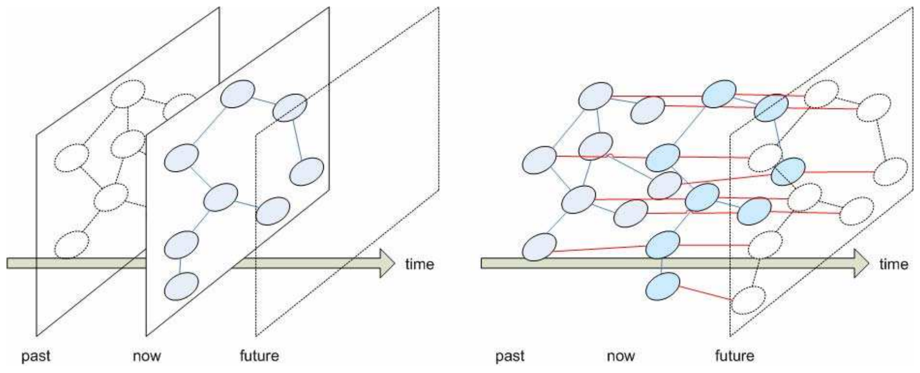

Fig. 1. Schematic representation of the concept of time determined ontology.

As illustrated in Figure 1[1] a development in time can only be described by a series of snapshot ontologies each superimposing itself on the previous version of the described reality (left). The 4D-fluent (perdurantist) ontology (on the right) allows the concepts of time and change to become integral parts of the ontology. In TOQL we opt for the later type of representation based on 4D fluents.

[1] The figure is from "Annex1: Description of Work" document of project TOWL http://www.towl.org

Following the approach by Welty and Fikes [15], to add the time dimension to an ontology, classes *TimeSlice* and *TimeInterval* with properties *tsTimeSliceOf* and *tsTimeInterval* respectively are introduced. Class *TimeSlice* is the domain class for entities representing temporal parts (i.e., "time slices") and class *TimeInterval* is the domain class of 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.*

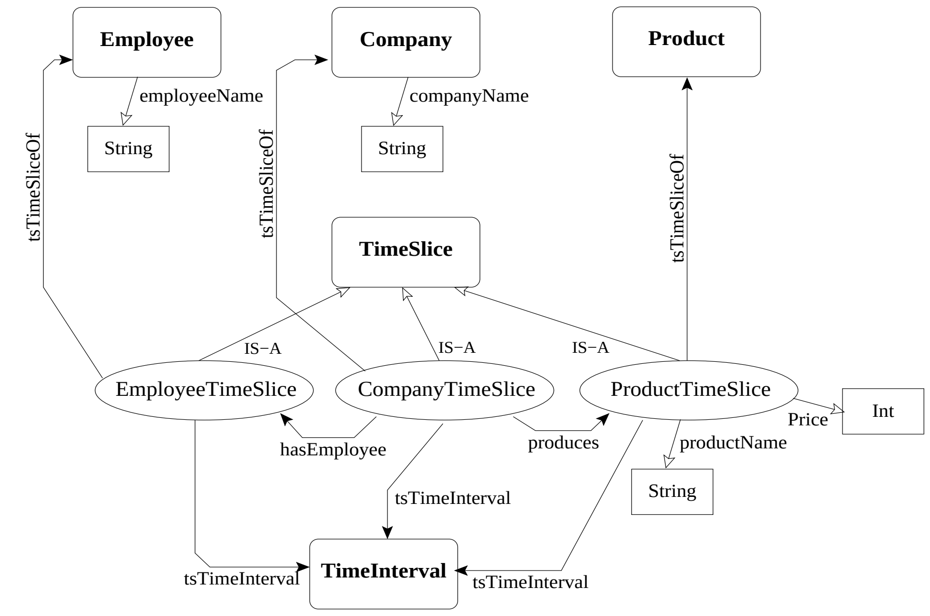

Fig. 2. Dynamic Enterprise Ontology

Figure 2 illustrates the so called "Dynamic Enterprise Ontology" ("*DEn* Ontology") defined in this work: a temporal ontology with classes *Employee* with datatype property *employeeName*, *Company* with datatype property *companyName* and *Product* with datatype properties *price* and *productName.* In this example, *CompanyName* and *EmployeeName* are static properties (their values do not change in time), while properties *produces*, *hasEmployee*, *productName* and *price* are dynamic (fluent) properties whose values may change in time. Because they are fluent properties, their domain (and range) is of class *TimeSlice*. *EmployeeTimeSlice*, *CompanyTimeSlice* and *ProductTimeSlice* are instances of class *TimeSlice* and are provided to denote 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 class *TimeSlice* but it is restricted to instances that are time slices of class *Product*. A knowledge base with the instances of the *DEn* ontology used in the work can be found in [3] or can be downloaded from http://www.intelligence.tuc.gr/∼petrakis/downloads/TOQL.zip.

2.2 Temporal Query Languages

The main goal of temporal query languages is to maintain simplicity of expression while the time dimension is added. Other desirable features include, temporal upward compatibility (i.e., conventional queries and modifications on temporal relations act on the current state), temporal aggregation (i.e., possibility to request the history of something), point and interval-based views of data, expressive power and ease of implementation.

Examples of temporal query languages for temporal databases include TQuel [14], TSQL2 [9] and ATSQL [4]. Query languages for handling time information in ontologies (e.g., time evolving entities) besides TOQL, are not known to exist. Nevertheless, query languages for RDF and OWL ontological representations are of particular interest as they form the basis for developing the new type of temporal ontology query languages. SeRQL [1] and SPARQL [11] are good representatives of this category of query languages. SPARQL [11] is a W3C recommendation query language. SeRQL is a RDF/RDFS query language combining features of other (query) languages (e.g., RQL [7], RDQL [12], N-Triples, N3). Important features of SeRQL are: Graph transformation, RDF and XML Schema data type support, expressive path expression syntax and optional path matching. SeRQL supports comparison between date times and more query types than SPARQL, which has limitations in the "where" clause, since it doesn't support nested queries.

3 TOQL: Syntax and Semantics

TOQL (Temporal Ontology Query Language) is an SQL-like language for OWL, supporting the basic structure of SQL (SELECT - FROM - WHERE) and treats classes and properties of an ontology almost like tables and columns of a database. The new language takes into account differences in the type of relations in the two representations and also supports time operators: Allen operators (BEFORE, AFTER, EQUALS, MEETS, OVERLAPS, DURING, STARTS, ENDS) and operators AT(time point) and AT(time point, time point). Allen operators [2] compare datatype properties e.g., *A.B like "x" before C.D like "y"*. The language also supports additional functionalities such as LIMIT, OFFSET that limit the number of answers to be returned, and nested queries. A formal description of the language's syntax in BNF can be found in [3] (all keywords are *case insensitive*). TOQL supports most of an SQL language syntax and clauses (see cite[3] for a complete list), the most important of them being:

– **SELECT**: specifies the object property values to be returned.

– **FROM**: declares the class or classes to query from. Always follows SELECT.
– **WHERE**: includes logic operations and comparisons between object property values that restrict the number of answers returned by the query. Always follows FROM.

TOQL supports the following operators:

– **AS**: renames a class (in a FROM clause) or a property (in a SELECT clause). Renaming of a class allows using more than one instances of a class in a query (e.g., FROM Company AS C1, Company AS C2). Renaming of a property allows changing its name in the results (e.g., SELECT Company.companyName AS Name).
– **AND**: connects two expressions involving properties (datatype or object properties) in WHERE, returns objects satisfying both expresssions.
– **OR**: connects two expressions involving properties (datatype or object properties) in WHERE and returns objects satisfying at least one.
– **LIKE**: checks whether a datatype property value matches a specified string in WHERE. Comparison is case sensitive.
– **LIKE "string" IGNORE CASE**: checks whether a datatype property value matches a specified string ignoring case.

Table 1 summarizes TOQL syntax:

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

Table 1. Generic TOQL syntax.

There are operation clauses connecting two (or more) queries in a nested query:

– **MINUS**: returns query results retrieved by the first operand, excluding 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 2 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[2]
	ALL (Query)
		SELECT ...
		FROM ...
		WHERE ... CO[2]
		ANY (Query)
			SELECT ...
			FROM ...
			WHERE ...
			IN (Query)

Table 2. TOQL syntax with operator clauses.

3.1 Dealing with Classes and Properties

In ontologies the basic terms are classes (also named concepts) and properties (object or datatype). Classes represent concepts of the world. 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 are connected with the object property *hasEmployee*. As an example of datatype property consider the name of an *Employee*. Class *Employee* is connected with a name (string value) with datatype property *employeeName*.

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. Table 3 summarizes the mapping between database relations and ontology concepts used by TOQL.

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:

[2] CO: comparison operator can be any of “=”, “!=”, “<”, “>”, “<=”, “>=”

Relational Database	**Ontology**
Table representing concept	Class
Table representing N:N relation	Object Property
1:N or 1:1 relation	Object Property
Attribute	Datatype Property

Table 3. Mapping between database relations and ontology concepts.

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 ontology of Figure 2:

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

3.2 Dealing with Time

TOQL is a high level language, hiding from the users the implementation of time at the ontology level. A temporal ontology consists of (a) the static part where application classes, properties and their instances are defined and (b) the dynamic part where the additional temporal classes (i.e., classes *TimeSlice*, *TimeInterval*), properties and instances of the above temporal classes and fluent properties are defined (i.e., *tsTimeSliceOf*, *tsTimeInterval*). TOQL automatically determines references to time related information.

To do this, TOQL:

– Retrieves 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 (i.e., a property that connects "static" classes or a "static" class with a datatype).
– Uses the ontology's dynamic part to answer the query, if a property specified by the query is a fluent one. In case the fluent property is a functional one (i.e. can have only one value at each instance of time) then the reasoner described in Sec. 3.7 is used to answer the query. The rationale behind this

choice is that functional properties have unique values, which may change at a later time as the result of events affecting them. For example, if the price of product changes, then the new value substitutes any previously known value, while non functional properties retain both older and newer values.
– Uses the ontology's static part to answer the query, if a property specified by the query is not a fluent one.

TOQL prevents users from being familiar with the representation of time in ontologies. As an example consider the *DEn* Ontology of Figure 2. Typically to retrieve companies that hired employees, one should be familiar with the 4D fluent mechanism and ask for all time slices (instances) of class *Company* and all time slices of class *Employee* and then query on the object property *hasEmployee* that connects those instances. In TOQL (without implementing the high level functionality described above), this is 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 a rather complicated expression and requires the user to 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 class *Company* and its range class *Employee.*

3.3 Abstract ontology View

TOQL is a high level language independent from the actual representation of time in an ontology. A user need only be aware of the so called "abstract ontology view". Classes and properties specific to the 4D fluent mechanism are excluded from the abstract view. The fluent properties that connect time slices are considered to connect directly the static classes. Figure 3 illustrates the abstract ontology view corresponding to the *DEn* ontology of Figure 2. Fluent properties are shown in blue color.

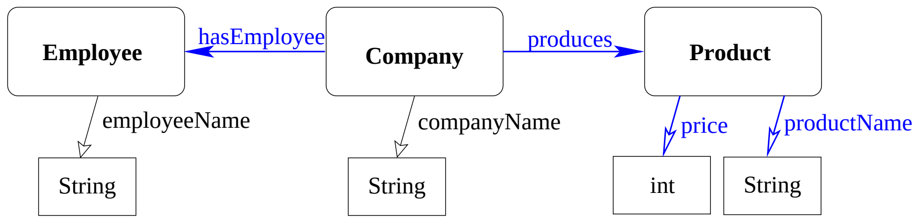

Fig. 3. Abstract ontology view corresponding to the *DEn* ontology of Figure 2.

3.4 Allen Operators

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, the implementation of Allen operators correspond to comparisons between time intervals. The following operators are supported in TOQL: BEFORE, AFTER, MEETS, METBY, OVERLAPS, OVERLAPPEDBY, DURING, CONTAINS, STARTS, STARTEDBY, ENDS, ENDEDBY and EQUALS, representing the corresponding relations holding between two time intervals.

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”

3.5 AT, TIME Operators

TOQL also introduces 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 the AT clause retrieves data explicitly defined in the knowledge base. As an example, 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 a reasonable answer would be 10 (the last known price in the KB). Answering such queries effectively is achieved by combining TOQL with the reasoner described in Sec. 3.7. In the current implementation:

– **AT(time point)** operation returns true if the time interval holds true at the time specified.

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

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”

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 for TOQL to return values of time, 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 holds true (the time interval in which the property is true). If no end point exists, it returns only its start point. 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”

3.6 Special Cases

This section describes TOQL special features. These are related to the way TOQL deals with Class keys, wildcards (*) and Scope.

Dealing with keys: In relational databases each 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 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

Dealing with wildcards (*): In TOQL, wildcards 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, 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). 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"

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, class and property usage and this introduces the need for the handling of scopes.

Queries combined by set operators have 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"* from 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 separate queries combined by the set operator UNION. Each subquery has a different scope: classes declared in the first subquery are not visible to the second one. Even if the same class is used by the second subquery, it must be redeclared.

In TOQL, a nested query inherits all the classes declared in the query it is nested into. A nested query can use these classes, but cannot (re)declare any of them. The following nested TOQL query (a second query follows clause ANY) retrieves products whose price is at least 10 and not smaller than than the price of any other product. Both subqueries use class *Product* but with different names (*P1* and *P2* respectively) otherwise a semantic error will be reported.

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

3.7 Reasoning in TOQL

TOQL can be used to access temporal information that is explicitly represented in a temporal ontology, but cannot provide answers on information that can be inferred from existing information. For example if the price of a product at time *t* is *p*, TOQL should be able to infer that the price of the product remains the same since the last time it was changed. This is exactly the problem the TOQL reasoner is dealing with. The reasoner implements an action theory based on Event Calculus [22]. Event calculus records the events that have taken place. It comprises of events (or actions), fluents and time points. Table 4 illustrates the predicates of Simple Event Calculus. Time points are natural numbers which means that time is ordered, discrete and unbounded. A fluent is a predicate of the form "fluentName1(objectID1)" and the same is an action "actionName1(objectID1,objectID2)".

Predicate	Meaning
Initiates(A, f, x, t)	if action A is executed at time t, then f will have value x at time point t
Terminates(A, f, x, t)	if action A is executed at time t, then f will not have value x after the time point t
HoldsAt(f, x, t)	fluent f has value x at time point t
Initially(f, x)	fluent f has value x in the beginning
HappensAt(A, t)	action A is executed at time point t
`t1<t2`	time point t1 is before time point t2

Table 4. Predicates of Simple Event Calculus.

The definition of the $HoldsAt$ and $HoldsBetween$ predicates for an arbitrary fluent f is presented below along with rules that state that a fluent retains the same value since the last time it was changed:

$$Started(t1, f, x, t2) \leftarrow \exists a : HappensAt(a, t1) \land Initiates(a, f, x, t1) \land (t1 < t2)$$

$$Releases(a, f, x, t) \leftarrow \exists a' : HappensAt(a', t) \wedge Initiates(a', f, y, t) \wedge (y \neq x)$$

$$Clipped(t1, f, x, t2) \leftarrow \exists a, t : HappensAt(a, t) \wedge (t1 < t < t2) \wedge Terminates(a, f, x, t)$$

$$HoldsAt(f, x, t) \leftarrow (Initially(f, x) \wedge (0 < t) \wedge \neg Clipped(0, f, x, t))$$
$$\vee (\exists t1 : Started(t1, f, x, t) \wedge \neg Clipped(t1, f, x, t))$$

$$HoldsBetween(f, x, t1, t2) \leftarrow (\exists t : Started(t, f, x, t1) \wedge \neg Clipped(t, f, x, t2))$$
$$\vee (Initially(f, x) \wedge (0 < t1) \wedge \neg Clipped(0, f, x, t2))$$

The reasoner applies when an object property is defined as temporal and functional (e.g. the price of a product, which can have only one value at a time point). For example if the price of the product *"Product4"* is set at 50 euro at time point 2 and 60 euro at time point 4 then the following query:

```
SELECT Product
FROM Product
WHERE Product.price LIKE "50" AT(9)
```

will return an empty list as a result, because the reasoner infers that setting the price at 60 euros at time point 4 implies that the price is not 50 euros after that time point. If the reasoner is not used then the query will return *"Product4"* as a result, which is not correct. Thus the AT operator is handled by the reasoner in case of functional fluent properties.

4 TOQL Implementation

To show proof of concept, a TOQL system has been implemented in Java[3]. The system supports query translation and execution of TOQL queries on temporal ontologies in OWL. The input is a query written in TOQL and an ontology in OWL (in RDF/XML or RDF/XML-ABBREV syntax).

Figure 4 illustrates the architecture of the proposed system. The TOQL system consists of several modules whose purpose is to translate the TOQL query into an equivalent SeRQL one (which is then executed on the knowledge base).Notice that SeRQL is independent from TOQL. Any other language supporting SQL syntax and comparison between date times (such as SPARQL) would do for this translation. Notice also that executing TOQL statements directly on the ontology is also feasible but the implementation would be more involved. TOQL and SeRQL have different syntax, however, queries are much easier to express in TOQL. SeRQL supports comparison between date times but not the full range of TOQL's time features. Therefore, even simple TOQL queries are translated to complicated SeRQL ones. The complete discussion of the TOQL implementation can be found in [3]. The application loads the ontology schema

[3] Available at http://www.intelligence.tuc.gr/∼petrakis/downloads/TOQL.zip

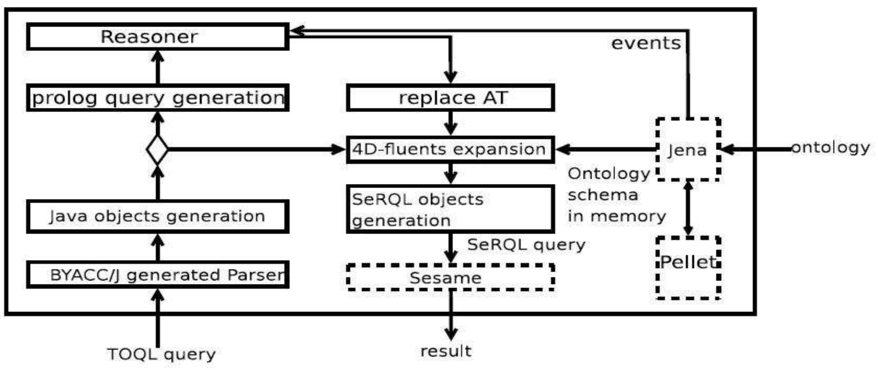

Fig. 4. TOQL system architecture.

in memory. TOQL queries are translated into equivalent SeRQL queries which are applied to the knowledge base using SESAME [4]. The TOQL query is parsed and if fluent properties are detected then the query is converted to an equivalent query addressing the underlying 4-D fluent representation, which in turn is translated into a SeRQL query. For example the following TOQL query is translated to the SeRQL query of page 17:

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

In case of queries over functional fluent properties fluents are represented as predicates of event calculus and the Prolog reasoner is applied, which transforms the query into an equivalent one that conforms to the event calculus axioms, before the translation to SeRQL occurs. Specifically, at the "java objects generation" phase, if the query uses the AT operator, it is replaced with an equivalent one where every expression that uses the AT operator is replaced with the reasoners answer.

The Pellet[5] reasoner applies to the initial ontology schema, thus the schema loaded in memory containes all infered facts using OWL semantics. For example if the class *ComputerCompany* is defined as a subclass of class *Company*, then a query regarding instances of class *Company* will also apply to instances of the class *ComputerCompany.*

[4] http://www.openrdf.org/
[5] http://clarkparsia.com/pellet

```
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#>
```

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. Temporal concepts are assumed to be represented in OWL (or RDF) using the 4D perdurantist approach [15], implementing events occurring at specific time points or time intervals and evolving in time. The language supports a powerful set of operations including Allen operators. An application supporting execution of TOQL queries on OWL temporal (or static) ontologies has been developed and is available on the Web. TOQL is combined with a reasoner based on event calculus to better support queries on temporal ontologies. Query optimization as well as adding new features in TOQL (such as INSERT, UPDATE, DELETE, ORDER BY, GROYP BY operations) are important issues for further research. Extending TOQL's syntax to handle queries on spatial data as well as queries on ontology structure (i.e., sub-classes and super-classes) and improving query performance by applying indexing on ontology information are also directions for further research.

Acknowledgement

This work was supported by project TOWL: "Time-determined ontology based information system for real time stock market analysis" (FP6-STREP, contract number 26896) of the European Union.

References

1. B. V. Aduna. The SeRQL query language. User Guide for Sesame 2.1, Chapter 9, 2002–2008. http://www.openrdf.org/doc/sesame2/2.1.2/users/ch09.html.
2. J. F. Allen and G. Ferguson. Actions and Events in Interval Temporal Logic. *Journal of Logic and Computation*, 4(5):531–579, 1994.

3. E. Baratis. TOQL: Querying Temporal Information in Ontologies. Master's thesis, Techn. Univ. of Crete (TUC), Dept. of Electronic and Comp. Engineering, July 2008.
4. M. H. Bohlen and C. S Jensen. Seamless Integration of Time into SQL. Technical Report R-96-49, Dept of Comp. Science, Aalborg University, 1996.
5. D. Martin et.al. OWL-S: Semantic Markup for Web Services. W3C Recommendation, November 2004. http://www.w3.org/Submission/OWL-S.
6. J.R. Hobbs and P. Fang. Time Ontology in OWL. W3C Recommendation, September 2006. http://www.w3.org/TR/owl-time/.
7. G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In *Intern. Conf. on World Wide Web (WWW2002)*, Honolulu, Hawaii, USA, May 2002.
8. M. Klein and D. Fensel. Ontology Versioning for the Semantic Web. In *International Semantic Web Working Symposium (SWWS'01)*, pages 75–92, California, USA, July–August 2001.
9. N. Kline, R. T. Snodgrass, and T. Y. Cliff Leung. Aggregates. In *The TSQL2 Temporal Query Language*, pages 393–424. Kluwer, 1995.
10. D. L. McGuinness and F. VanHarmelen. OWL Web Ontology Language Overview. W3C Recommendation, February 2004. http://www.w3.org/TR/owl-features.
11. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, January 2008. http://www.w3.org/TR/rdf-sparql-query.
12. Andy Seaborne. RDQL - A Query Language for RDF. W3C Recommendation, January 2004. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109.
13. T. Sider. *Four-Dimensionalism: An Ontology of Persistence and Time.* Oxford University Press, USA, 2002.
14. R. T. Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems (TODS)*, 12(2):247–298, 1987.
15. C. Welty and R. Fikes. A Reusable Ontology for Fluents in OWL. *Fontiers in Artificial Intelligence and Applications*, 150:226–236, 2006.
16. C. Welty, R. Fikes, and S. Makarios. A Reusable Ontology for Fluents in OWL. Technical Report RC23755 (Wo510-142), IBM Research Division, T. Watson Research Center, Yorktown Heights, NY, October 2005.
17. Z. Zhang. Ontology Query Languages : A Performance Evaluation. Master's thesis, The University of Georgia, Comp. Science Dept., August 2005.
18. G. Ozsoyglu, and R.T. Snodgrass. Temporal and Real-Time Databases: A Survey. Knowledge and Data Engineering 4 (1995) 513532.
19. H. Gregersen,and C.S. Jensen. C.S.: Temporal Entity Relationship Models A Survey. IEEE Transactions on Knowledge and Data Engineering 3 (1999) 464497.
20. A. Artale, and E. Franconi. A survey of temporal extensions of description logics. Annals of Mathematics and Artificial Intelligence, 30(1-4), 2001.
21. C. Lutz,F. Wolter, and M. Zakharyaschev. Temporal description logics: A survey. In Proc. TIME08, IEEE Press, 2008.
22. M. Shanahan. The event calculus explained. In Wooldridge, M., and Veloso, M.(Eds.), Artificial Intelligence Today, pp. 409430. Springer Lecture Notes in Artificial Intelligence no. 1600, 1999.