Technical University of Crete

Department of Electronic and Computer Engineering

# A reasoner for querying temporal ontologies

Dissertation Thesis

Nikos Maris

review committee:
Assoc. Prof. Euripides Petrakis (supervisor),
Dept. of Electronic and Comp. Engineering TUC
Prof. Stavros Christodoulakis,
Dept. of Electronic and Comp. Engineering, TUC
Assist. Prof. Nikos Papadakis,
Dept of Sciences, TEI of Crete

**Abstract**

Dealing with information that changes with time is a critical issue in Knowledge Representation. Existing research efforts limit to temporal information representation and querying in temporal databases. However, the advent of the semantic web over the last few years calls for the representation of temporal information based on temporal ontologies and for querying this information using temporal query languages such as TOQL [7]. However, even the most elaborate querying approaches (such as those referred to above) only support temporal queries that is explicitly represented in a temporal ontology. They cannot provide answers cencerning information that can be influenced from existing information. For example, if the price of a product at time 't' is 'p' a temporal query language should be able to inference that the price of the product (unless changed) will still be 'p' at later time. This is exactly the problem this work is dealing with. We extent "TOQL" (the Temporal Ontology Query Language developed at the Intelligence Systems Laboratory of TUC) with reasoning capillarities so that TOQL can answer queries for information that that can be inferenced from information represented in the underlying temporal ontology. The reasoner developed to support TOQL implements an action theory based on "Event Calculus" in Prolog.

**Keywords** : ontologies, logic programming, temporal reasoning

## Περίληψη

Η αναπαράσταση πληροφοριών που αλλάζουν με την πάροδο του χρόνου είναι ένα σημαντικό ζήτημα στον χώρο της Αναπαράστασης Γνώσης. Ενώ η έρευνα πάνω σε χρονικές βάσεις δεδομένων συνεχίζεται, πρόσφατες εξελίξεις στον χώρο του Σημασιολογικού Ιστού προτείνουν εναλλακτικές προσεγγίσεις όπως την TOQL, μια γλώσσα επερωτήσεων σε χρονικές οντολογίες. Η παρούσα εργασία επεκτείνει την TOQL για να εξάγει γνώση από τις υπάρχουσες πληροφορίες βασιζόμενη στην υπόθεση πώς οτι δεν αλλάζει, θα έχει την ίδια τιμή σε επόμενη χρονική στιγμή. Το νέο υποσύστημα της TOQL βασίζεται στο Event Calculus και είναι υλοποιημένο σε Prolog.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The semantic web is a vision of information that is understandable by computers, so that computers can perform more of the tedious work of finding and combining information on the web. Due to the complexity and variety of applications, experts on different technologies provide different definitions of the semantic web. The figure below illustrates the building blocks - in terms of technological resources - of what is referred to as the fourth version of the "semantic web stack"[1].

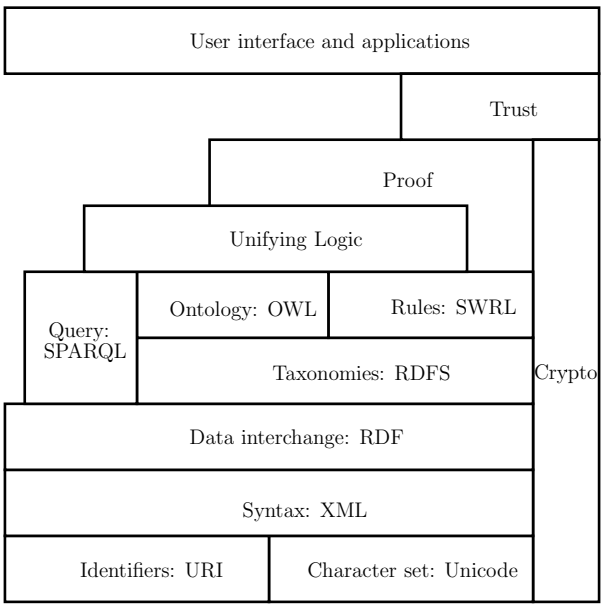



Figure 1.1: Semantic web stack

[1]Introduced by Tim Berners-Lee at 2006 on his speech "AI and the semantic web": http://www.w3.org/2006/Talks/0718-aaai-tbl/Overview.html#(14)

Each component exploits and uses the capabilities of the components below. However, the application developer is not limited to use only the top component, but can use the component that fits to his needs. The building blocks of the semantic web [2] are :

1. Uniform Resource Identifier (URI) is a compact string of characters used to identify or name a resource on the Internet. URIs are defined in schemes defining a specific syntax and associated protocols.

2. Unicode: an industry standard allowing computers to consistently represent and manipulate text expressed in most of the world's writing systems. UTF-8 is the standard character encoding for unicode.

3. Extensible Markup Language (XML) provides a uniform framework for interchange of data and meta-data between applications but not any means of talking about the semantics (meaning) of data. XML Schema is a language for providing and restricting the structure and content of elements contained within XML documents.

4. Resource Description Framework (RDF) is a simple language for expressing data models which refers to objects (called "resources") and their relationships.

5. RDF Schema is a vocabulary for describing properties and classes of RDF-based resources, with semantics for generalized-hierarchies of such properties and classes.

6. Web Ontology Language (OWL) is a family of knowledge representation languages that can express more complex domains than RDFS can.

7. Simple Protocol and RDF Query Language (SPARQL) is the standard language for querying RDF graphs.

8. The rules, unifying logic, proof and trust layers enable the writing of application-specific declarative knowledge, increase users' confidence in Semantic Web agents and enable secure activities between agents. For logic to be useful on the Web it should be machine-processable and usable in conjunction with other data. Therefore, there is ongoing work on representing logical knowledge and proofs in XML (like RuleML) and OWL (like SWRL).

9. An intuitive User Interface allows users to take advantage of the above technologies.

The identifiers, the character set and the syntax are already components of the (hypertext) web and the data interchange, the taxonomies and the ontologies are well established components of the semantic web. However, the query and the rules components were introduced only in the fourth version of the "semantic web stack". Although, there is a standard RDF query language (SPARQL), there is an increasing need for query languages that are aware of the OWL semantics (ontology query languages).

## 1.1 Motivation

In OWL there is no notion of space and time, which are basic concepts of common sense. Although some temporal OWL extensions have been proposed [23], current query systems do not return the value of an OWL property at an arbitrary time point unless it is explicitly represented in the underlying ontology representation. .

## 1.2 Problem Definition

The Temporal Ontology Query Language (TOQL) [7] can be used to query temporal ontologies based on a specific OWL extension [23], but it cannot infer knowledge from existing information. For example, in a sample ontology which states only that the price of product X is set to 50 euro at time 2004 and to 60 euro at time 2006, the price of product X at time 2008 remains at 60 euro. In some domains, it is preferable to allow overlapping events but return at most one value in queries like getting all prices of product X at 2008.

## 1.3 Contributions of this work

Reasoning in OWL is provided by a Description Logic reasoner. Temporal ontologies are usually based on a Temporal Description Logic which is the combination of a Description Logic with a temporal logic [16]. This work instead, translates the temporal part of an ontology to instances of a temporal logic. The temporal logic used is called "Event Calculus" and expresses the knowledge that everything that does not change, will have the same value at a later time. As a proof of concept, a reasoner is implemented to handle the AT operator of TOQL [7]. This work implements TOQL2 which is an extension of TOQL where the AT operator is handled by the reasoner (if only the temporal operant is owl:functional). Alongside, the TOQL syntax is extended

to support the AT operator within the SELECT clause. Additionally, our mapping enables a temporal ontology to use extensions of Event Calculus [20].

## 1.4 Limitations

As the reasoner is based on Event Calculus, the semantics of the AT operator are skeptical which means that the predicate has to be true at least in the whole time interval. Thus, there is no alternative operator with credulous semantics which means that the predicate has to be true only in a subset of this interval. Additionally, the reasoner ensures that "a product can not have many prices" and not that "a product can not have many prices set by the same company".

## 1.5 Outline

Chapter 2 provides an introduction to reasoning, analyses TOQL and the requirements for a TOQL-valid temporal ontology. Chapter 3 presents the reasoner and its underlying theory. TOQL2 is presented in chapter 4 and future work is discussed in chapter 5.

# Chapter 2

# Background

## 2.1 Knowledge Representation

A Knowledge Base differs from a database in that the extracted information is usually not explicitly stored in the database but is inferred from facts of a logical theory. Attempting to describe the richness of the natural world is the driving force behind an area in AI called Knowledge representation (KR)[9]. Formal logic is the usual framework as it provides semantics that are not open to the subjective intuition of a person nor to different interpretations by different people (or programs)[2]. Section 2.2 outlines the Description logics that have become a cornerstone of the Semantic Web for their use in the design of ontologies [5].

### 2.1.1 Ontologies

An ontology is a formal representation of a domain through concepts and relations between these concepts. A language for describing ontologies defines vocabulary terms (classes and properties) and the relations between them (relations between classes, relations between properties and relations between classes and properties). An instance of a class is called an *individual*.The Web Ontology Language (OWL) is a family of knowledge representation languages that have been proposed to deal with various aspects of KR :

1. OWL-Lite was designed to support those users primarily needing a classification hierarchy and simple constraints.

2. OWL-DL (where DL stands for "Description Logic") is supported by a DL reasoner.

3. The complete OWL language (called OWL-Full in order to distinguish it from the subsets) provides full compatibility with RDF Schema but adding support of reasoning services is like solving the halting problem.

In order for a query language to support the expressiveness of OWL-DL, the query evaluation procedure works closely with a Description Logic reasoner as shown in figure 2.1. For example, OWL-SAIQL [14] uses Pellet [21] to answer queries like returning all named subclasses of a class. The usual services of a DL reasoner include computing the subsumption hierarchy between classes (classification), answering queries, testing the consistency of class description and finding explanations for inconsistencies. Pellet and most DL reasoners are based on the tableaux algorithm which reduces reasoning services to satisfiability checking [5].

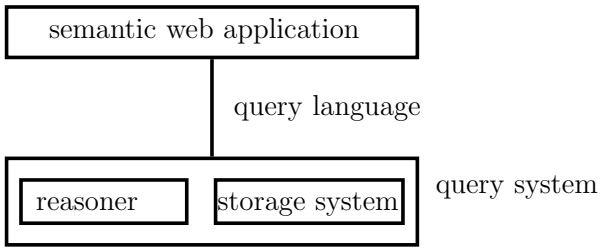



Figure 2.1: A system for querying ontologies

### 2.1.2 Temporal Ontologies

OWL is based on binary relations (relations connecting two instances with no time dimension) making the representation of time a difficult matter to deal with. For example, specifying "the price of Product1" in a dynamic domain where the price of a Product varies over time, is meaningless. 4D-fluents [23] is a formalism based on OWL-DL that can represent knowledge like "Company1 was producing Product1 from 2001 to 2005 and was selling it for 50 euro" as shown on figure 2.2. Each part of a temporal relation (action like "produces") is not an entity, but a temporal instance of an entity (a timeslice is an individual of the class Timeslice). For example, in the sentence "the price of Product1 at 2001 is 50 euro", "Product1 at 2001" is a timeslice of Product1 and price is a property of timeslices of Product (a fluent of Product).

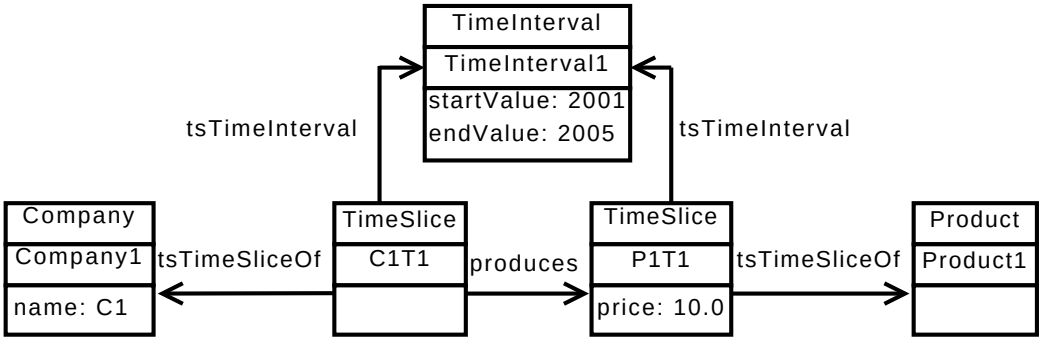
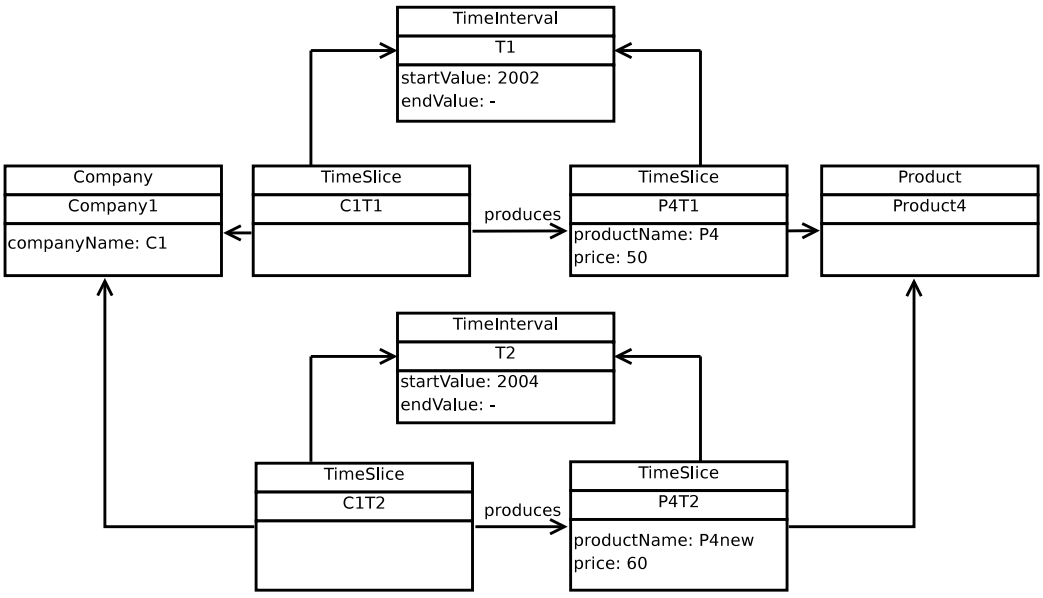



Figure 2.2: Sample individuals of the 4D-fluents schema

This work deals with overlapping events as shown below where a property (like the price of a product) should have at most one value at a time. For example, from the following events we want to derive that the price of Product4 at time 2005 is 60 euro. The reasoning process we use is introduced in section 2.2 and analysed in chapter 3.




TOQL notation [7] enables the representation of temporal ontologies without requiring knowledge of the 4D-fluents mechanism. In figure 2.3 small boxes represent primitive XML data types, squares represent OWL classes, simple arrows represent static properties and marked arrows represent temporal properties. This notation shows for example that 'productName' is a fluent and 'produces' is an action.

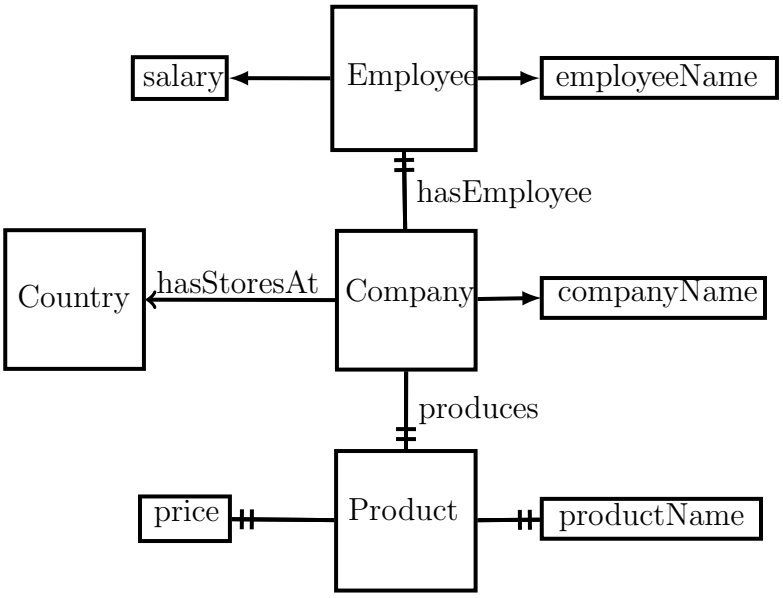



Figure 2.3: Sample temporal ontology in TOQL notation

This work has been tested over a sample temporal ontology presented in appendix B in the Manchester OWL syntax [12]. In our ontology, a TimeInterval has a startValue and an endValue. Each of these two integers corresponds to a number of seconds after a fixed time point (for example 1/1/1970 00:00). It is assumed that the conversion between this measurement of time and a human readable format is done in the application layer. If an individual has a startValue X and an endValue '-1', then the time interval is interpreted as $(X, +\infty)$. Time points can be viewed as incomplete information where the application that fills the ontology does not know when the action stopped having an effect. Additionally, in this work the 4D-fluents schema is subject to the following restrictions :

1. The class TimeSlice should not be domain or range of any property except for tsTimeSliceOf.

2. Class TimeInterval should not be domain or range of any property except for tsTimeInterval, startValue and endValue.

The major disadvantage of 4D-fluents is prolification of objects [23]. Another approach is to treat the ordinary individuals as timeslices [13]. However, in that case, the application developer can not find what is temporal based only on the ontlogy schema.

## 2.2 Reasoning

### 2.2.1 Temporal reasoning

Generally, each logical system comes with both a syntactic component, which among other things determines the notion of a formula, and a semantic component, which determines the notion of logical validity. The logically valid formulas of a system are sometimes called the theorems of the system. A logical system is decidable if there is an effective method for determining whether arbitrary formulas are theorems of the logical system. For example, propositional logic is decidable, because the truth-table method can be used for satisfiability checking, in other words, to determine whether an arbitrary propositional formula is logically valid.

Commonsense reasoning, the branch of artificial intelligence concerned with replicating basic cognitive processes, can contribute to the advancement of reasoning services of the semantic web. When designing an agent in artificial intelligence, the knowledge of the domain cannot always be represented statically. In a dynamic world, the effects of an action are associated not only with the action but also with dynamic parameters of the environment, like time. To formalize a dynamic world, one essential thing is, not only to specify what will change, but also what will not change due to an action. As there is a very large number of such axioms, it is very easy for the designer to leave out a necessary axiom, or to forget to modify all appropriate axioms, when the description of the world changes. This problem (known as the frame problem[18]) is considered to be equivalent to listing all preconditions of an action (known as the qualification problem) or to listing all effects of an action (known as the ramification problem).

Action theories aim at providing more general solutions to the frame problem (which poses major difficulties in commonsense reasoning) without always admitting decidability. Undecidability in action theories is usually resolved by letting the researcher solve the problem instead of providing automated reasoning. Two action-oriented programming languages are Golog[15] and Flux[22]. In contrast with action theories, Description logics, the logics behind OWL, aim at increasing their expressiveness while retaining decidability [5].

There are three main approaches to the representation of temporal information in the AI literature [17]:

1. the so-called “method of temporal arguments” like Situation Calculus [18] simply extends functions and predicates of First-Order-Logic to include time as the additional argument.

2. "modal temporal logics" like Linear Temporal Logic [19] are extensions of the propositional or predicate calculus with modal temporal operators. As there is a correspondence between simple DLs and modal logics, there is also a correspondence between expressive DLs and expressive modal temporal logics.

3. "reified temporal logics"[1] like Event Calculus [20] describe standard propositions of some initial language (e.g., the classical first-order) as objects denoting propositional terms [17]. Propositional terms are related to temporal objects or other propositional terms through an additional sort of "meta-predicates" (predicates about predicates).

Description logics (DLs) form a family of logics used for the representation of the knowledge of an application domain. All DLs are based on the same syntax and semantics[5]. Research on a DL usually involves the formal definition of its syntax, its semantics and its computational complexity as regards a reasoning service. In description logics, a distinction is drawn between the so-called TBox (terminological box) and the ABox (assertional box). In general, the TBox contains sentences describing concept hierarchies (concepts and relations between concepts) while the ABox contains "ground" sentences, stating relations between individuals and concepts and relations between individuals. For example, the statement "Every employee is a person" is part of the TBox, while the statement 'Bob is an employee" is part of the ABox.

Temporal extensions of DLs [6, 10, 4] are called Temporal Description Logics and differ on [4]:

- whether they support time-points or time intervals
- whether the temporal information is embedded thus forming sequences of events or it is stated explicitly (through temporal operators). In the last case :
  - the ontology may have a static and a temporal part or each entity may be a collection of temporal "parts"

Since Description Logics allow only unary and binary relations, in Temporal Description Logics times (when something happened) or "situations" are removed from the syntax and instead are part of the implicit quantification[2]

[1] For the importance of Reified Temporal Logics and for the domains they can represent, see [17]

[2] Most Temporal Description Logics correspond to modal temporal logics.

with only one ordering relation [23]. Our approach is to translate individuals of an OWL-DL ontology (a temporal ontology based on 4D-fluents) to facts of the action theory Event Calculus and provide query answering over a temporal Abox like:

1. getting the companies for which person A worked for over the time interval C.

2. getting the value of a datatype property X at the time point Z.

### 2.2.2 Event Calculus

Our reasoner is based on the action theory Event Calculus[3]. As there are multiple versions of Event calculus [20], the analysis below presents the predicates of Simple Event Calculus. Event calculus records the events that have taken place and it comprises of events (or actions), fluents and time points. Time points are natural numbers which means that time is ordered, discrete and unbounded. A fluent is a predicate like $fluentName1(objectID1)$ and an action is a predicate like $actionName1(objectID1, objectID2)$.

| **Predicate** | **Meaning** |
|---|---|
| $Initiates(A, f, x, t)$ | if action A is executed at time t,<br>then f will have value x at time point t |
| $Terminates(A, f, x, t)$ | if action A is executed at time t,<br>then f will not have value x after the time point t |
| $HoldsAt(f, x, t)$ | fluent f has value x<br>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 |
| $t_1 < t_2$ | time point $t_1$ is before time point $t_2$ |

Table 2.1: Predicates of Simple Event Calculus

The definition of the 'HoldsAt' "meta-predicate" for an arbitrary fluent $f$ is presented with the following rules which state that a fluent will have the same value unless it changes.

[3]For an analysis of Event Calculus from a logic programming point of view, read the white paper of Michiel Van Lambalgen : http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.12.4993&rep=rep1&type=pdf

$$
\begin{aligned}
Started(t_1, f, x, t_2) \leftarrow \exists a : & HappensAt(a, t_1) \wedge Initiates(a, f, x, t_1) \wedge (t_1 < t_2) \\
Clipped(t_1, f, x, t_2) \leftarrow \exists a, t : & HappensAt(a, t) \wedge (t_1 < t < t_2) \wedge \\
& Terminates(a, f, x, t) \\
HoldsAt(f, x, t) \leftarrow & (Initially(f, x) \wedge (0 < t) \wedge \neg Clipped(0, f, x, t)) \vee \\
& (\exists t_1 : Started(t_1, f, x, t) \wedge \neg Clipped(t_1, f, x, t))
\end{aligned}
$$

For example, from the following facts it is inferred that fluent $f$ has the value $x$ for the time interval $(t_1, t_2]$ :
$Initiates(A, f, x, t_1)$
$HappensAt(A, t_1)$
$Terminates(A, f, x, t_2)$

### 2.2.3 Prolog

Prolog was the chosen implementation language for rapid prototyping as it is a logic programming language. Alternative future implementations are discussed in chapter 5. In mathematical logic, a literal is an atomic formula (positive literal) or its negation (negative literal) and a clause is a disjunction of literals. A useful subset of First Order Logic is Horn clauses [11] which have at most one positive literal like $\neg a \vee \neg b \vee g$. This Horn clause can be interpreted as "to prove g, prove a and prove b" where $g$ is called the head of the clause (or the goal) and $a \wedge b$ is called the body of the clause (or the subgoals). A prolog program is a set of Horn clauses and execution is triggered by running a query over the head of a clause. For example, the prolog query $descendant(bob, X)$ returns all descendants of "bob".

Prolog is a declarative language as it attempts to minimize side effects by describing what the program should accomplish, rather than describing how to accomplish it. The prolog execution algorithm has to be sound, complete and effective which makes it rather complicated. Its basic concepts are :

1. Unification is the association (binding) of variables (which always start with an upper case letter) with values (which always start with a lower case letter). Variables that receive a value are said to be instantiated.

2. Resolution condenses the traditional syllogisms of logical inference down to the rule "If C1 and C2 are Horn clauses and the head of C1 matches one of the terms in the body of C2, then the term in C2 is replaced by the body of C1". Prolog uses this rule as a computational step called

SLD resolution for which there is a proof of soundness and completeness [3].

3. Backtracking is an algorithm for finding all (or some) solutions to some computational problem by incrementally building candidates to the solutions, and abandoning each partial candidate c ("backtracks") as soon as it determines that c cannot possibly be completed to a valid solution.

Prolog compilers translate prolog code to low-level code of the standard Prolog Virtual Machine called Warren Abstract Machine (WAM) [1]. WAM implementations map the abstract machine into a general purpose computer architecture or design specific architectures to execute more efficiently WAM programs. The WAM instruction set can be divided into control instructions related to backtracking and unify instructions. Goal expansion and resolution is done recursively and at least in the first step some variables are unified with the values of the given query (like "bob" in the query $descendant(bob, X)$). In each step, the resulting disjunctive conjunctions can be viewed as different nodes at the same level of a tree, called the SLD-tree[4]. As a result, the problem of query evaluation is reduced to finding all facts of the prolog Knowledge Base that satisfy a node in the SLD-tree. Two search strategies are possible:

1. Forward chaining, derive the goal from the axioms.

2. Backward chaining, start with the goal and attempt to resolve them working backwards.

Due to the inefficiency of forward chaining when the facts are much more than the rules, Prolog's search strategy is backward chaining which usually follows a depth-first backtracking algorithm for each solution.

## 2.3 Temporal Ontology Query Language

Temporal ontologies are based on the idea of temporal databases. Some of the suggestions for the design of a temporal database query language[5] are that it should be based on formal semantics, independent of the underlying structure (for example optimization techniques) and consistent with the

[4]It should be noted that the SLD-tree is not really a tree but a graph as it is not always acyclic.

[5]For details, consider the white paper of Richard Snodgrass written at 1992: http://www.cs.arizona.edu/ rts/initiatives/tsql2/tsqldesignapproach.pdf

widely used SQL. In the Semantic Web, query languages should increase their expressiveness while keeping their syntax simple to learn and consequently less error-prone. For example, on the one hand eRQL (easy RDF Query Language) has too limited expressiveness and on the other hand SeRQL (Sesame RDF Query Language) requires a good understanding of RDF in order for the user to write path expressions that match specific paths through the underlying RDF graph. TOQL (Temporal Ontology Query Language) achieves this goal by introducing an additional abstraction level on top of SeRQL in order to provide the expressiveness of SeRQL without requiring knowledge of RDF. In TOQL [7] we showed that TOQL queries are translated into SeRQL equivalent ones using the TOQL interpreter.

TOQL is an ontology query language with syntax resembling a subset of the 'SELECT' syntax of SQL. It supports the basic OWL concepts which are the classes, the object properties and the datatype properties. It requires knowledge of the OWL schema (no knowledge of RDF is required) and in case of a temporal ontology, notation like that of figure 2.3 can be used to ignore the details of the 4D-fluents KR.

The syntax of TOQL is a subset of the 'SELECT' syntax of SQL augmented to support temporal relations. 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. The table below summarizes the mapping between database relations and ontology concepts used by TOQL.

| **Relational Database** | **Ontology** |
|---|---|
| Table representing concept | Class |
| Table representing N/M relation | Object Property |
| 1/N, 1/1 relations | Object Property |
| Attribute | Datatype Property |

Table 2.2: Mapping between database relations and ontology concepts

### 2.3.1 TOQL over ontologies

In TOQL, all classes should be declared in the FROM clause separated with commas where they can be renamed with the AS operator, just like in SQL. To access a datatype property (a property connecting an individual with a value) we write its name or *X.PropertyName* where *PropertyName* is the name of the property and $X$ is the name of the class as defined in the FROM clause. To access object properties (properties connecting two

classes) we use the pattern *DomainClassName.objectPropertyName:RangeClassName*. The syntax is shown in appendix A where from the SELECT, FROM, WHERE, OFFSET and LIMIT clauses only SELECT and FROM are mandatory. The (unique) name of a class instance is accessed using the name of the class itself (without reference to a property) in the SELECT clause. To get a table where columns are all datatype properties of a class except for the unique name, the wildcard character '*' can be used (e.g. Company.*).

The following query returns "C1" and "C2" which are the names of companies with stores in Greece.

```
SELECT Company.companyName
FROM Company, Country
WHERE Company.hasStoresAt : Country AND Country LIKE "Greece"
```

Table 2.3: TOQL query with object and datatype properties

Result:

| **companyName** |
|---|
| C1 |
| C2 |

TOQL also supports inner and nested queries. For example, the result of the "UNION ALL" operator is the combination of the results of both queries even if it contains duplicate tuples.

| **Inner Queries** | | | |
|---|---|---|---|
| Query<br>MINUS<br>Query | Query<br>UNION<br>Query | Query<br>UNION ALL<br>Query | Query<br>INTERSECT<br>Query |
| **Nested Queries** | | | |
| SELECT ...<br>FROM ...<br>WHERE EXISTS<br>(Query) | SELECT ...<br>FROM ...<br>WHERE ... CO[1]<br>ALL (Query) | SELECT ...<br>FROM ...<br>WHERE ... CO[1]<br>ANY (Query) | SELECT ...<br>FROM ...<br>WHERE ...<br>IN (Query) |

Table 2.4: TOQL syntax for inner and nested queries

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

In an inner query, a class declared in the FROM clause of one subquery can not be used in the other subquery without declaring it again. In contrast, a nested query inherits the class declarations, so the subquery should not declare the same classes again. Another feature of TOQL is the LIKE operator that checks whether a value matches a specified pattern of characters surrounded with double quotes. Optionally, wild-card characters '*' can be used to match zero or more arbitrary characters. The '=' operator can compare numbers but not strings, whereas the LIKE operator supports both comparisons. Note that all keywords are case not sensitive.

### 2.3.2 TOQL over 4D-fluents

Typically to retrieve the companies John has worked for, one should be aware of the 4D-fluents mechanism and ask for all timeslices (instances) of class Company and all timeslices of Employee and then query on the object property hasEmployee that connects those instances :

```
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 "John"
```

Table 2.5: Query with timeslices

This is a rather complicated expression and it requires the user to be familiar with the implementation of time at the level of the ontology (the 4D-fluents KR in this work). However, TOQL is a high level language hiding the implementation of time at the ontology level. As such, the user does not need to be aware of the details of the 4D-fluents mechanism. An ontology implementing the 4D-fluents mechanism consists of two parts: the static part, which is the initial ontology (classes, properties, instances), and the dynamic part, consisting of the 4D-fluents schema and temporal relations. TOQL supports "high level functionality" in order to deal with both the static and the dynamic part without requiring from the user to be aware of the 4D-fluents mechanism. The previous query can be expressed as:

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

Table 2.6: TOQL query equivalent to that of table 2.5

This query is much easier to write than the first one and the answer based on our ontology is "C1" and "C2". Notice that TOQL does not support queries with timeslices and the object property hasEmployee is treated as its domain was the class Company and its range was the class Employee. In fact the object property hasEmployee is a temporal property and has domain the class TimeSlice (instances of the class TimeSlice that are timeslices of Company) and range the class TimeSlice (instances of the class TimeSlice that are timeslices of Employee).

**Allen operators :**

Moreover, TOQL supports Allen operators between time intervals.

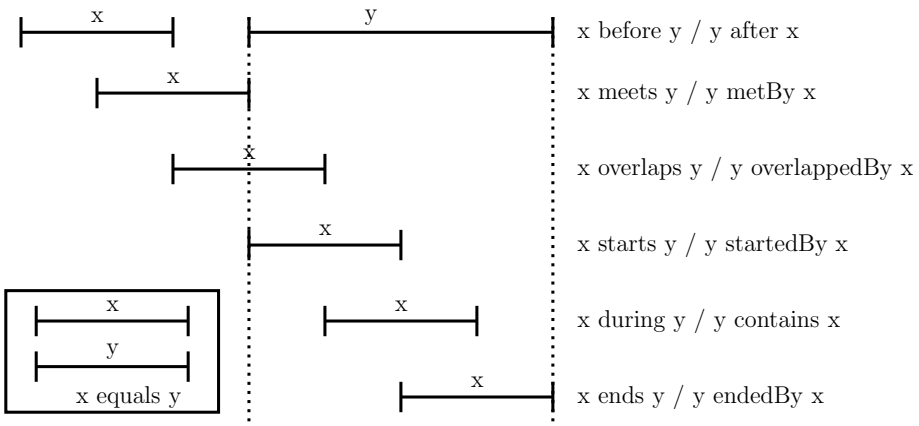



Figure 2.4: Allen operators

In our ontology, "Product3" is named "P3" for the time interval $(3, 7]$ and renamed to "P3x" for the time interval $(8, 13]$. The following query returns "Product3" as the Product which was renamed from "P3" to "P3x". Note that TOQL queries that use Allen operators do not consider events with no ending time point.

```
SELECT Product
FROM Product
WHERE Product.productName LIKE “P3”
BEFORE Product.productName LIKE “P3x”
```

Table 2.7: TOQL query with an Allen operator

**TIME operator :**

TOQL also provides the TIME operator that returns the asserted time interval (the starting and ending time points) of the returned timeslices.

```
SELECT Company.companyName, Company.hasEmployee.TIME
FROM Company, Employee
WHERE Company.hasEmployee:Employee
```

Table 2.8: TOQL query with the TIME operator

The result of the query above is the time interval each company had an employee.

| **companyName** | **hasEmployee_startValue** | **hasEmployee_endValue** |
|---|---|---|
| C2 | 3 | 7 |
| C1 | 6 | 10 |
| C1 | 1 | 5 |

Table 2.9: Result of the query of table 2.8

**AT operator :**

Finally, TOQL provides the AT operator in order to keep only the results that hold over a given time point or time interval. For example, the following query returns the Companies that produce Products for the time interval [2001,2005] and the corresponding Products (in this case “Company1” and “Product1”).

```
SELECT Company, Product
FROM Company, Product
WHERE Company.produces : Product AT(2002,2004)
```

Table 2.10: TOQL query with a temporal object property

TOQL evaluates this query by finding the right timeslices whose timeinterval has startValue equal or less than 2002 and endValue equal or greater than 2004. A similar process is taken for the evaluation of temporal datatype properties in the WHERE clause like the query below which returns the products whose price at time 9 is 50 euro.

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

Table 2.11: TOQL query with a temporal datatype property

Suppose that the price of a product varies over time. As shown in the following figure, Product4 is sold at 50 euro for the time interval $(2, +\infty)$ and at 60 euro for the time interval $(4, +\infty)$. Although TOQL returns "Product4", there is no product with price set at 50 euro at the time point '9'. In other words, the right answer is null (the empty table). In order to deal with this problem, this work implements a temporal reasoner. Generally, the problem is that TOQL knows nothing about instances at specific time or interval unless explicitely defined and stored in the ontology (values cannot be inferred from existing ones and this problem is solved by integrating a reasoner within TOQL).

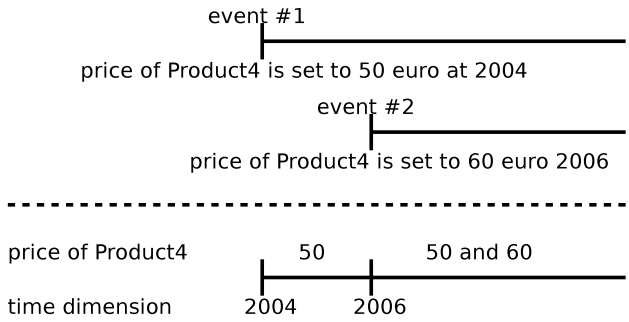



Figure 2.5: Overlapping events in TOQL

# Chapter 3

# Temporal reasoner

## 3.1 Mapping between 4D-fluents and Event Calculus

A known limitation of the 4D-fluents KR [23] is that it does not represent the knowledge that everything that does not change, will have the same value at a later time. Thus, the 4D-fluents KR can not represent property that have at most one value at a time. The following mapping between 4D-fluents and Event Calculus enriches the 4D-fluents KR with this restriction which is applied over fluents and actions. In order to use the reasoner to infer knowledge from information expressed in OWL-DL, the temporal part of the ontology is translated to prolog predicates. This also means that if an event is added to the ontology, this addition has to be reflected to the prolog predicates. As shown below, datatype properties of timeslices are mapped to fluents of Event Calculus, object properties connecting timeslices are mapped to actions, startValue is the time point of the predicate *initiates* and endValue is the time point of the predicate *terminates*.

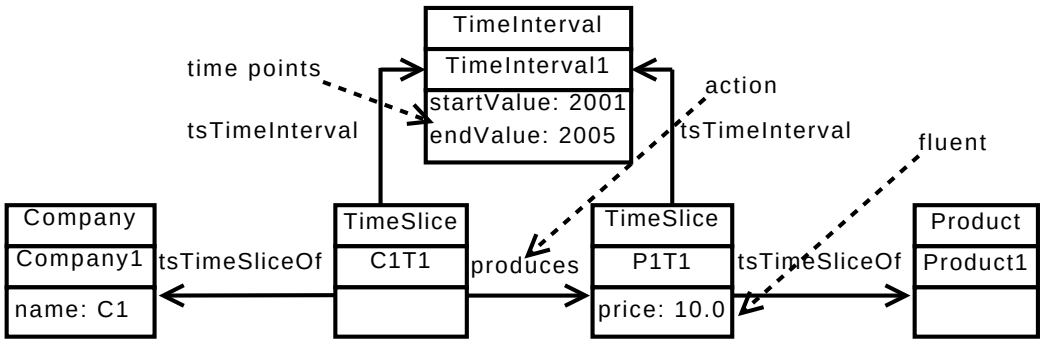



Figure 3.1: Mapping of instances between 4D-fluents and Event Calculus

## 3.2 Implementation

The reasoner implements the action theory Event Calculus where it removes the predicate *Initially* and adds the predicate *Releases* [20]. Note that the predicate *happensAt* comes with the predicate *Initiates* but is not related to the predicate *Terminates*. In other words, no action is needed to "disable" a fluent. Through Event Calculus, the reasoner ensures that each property has at most one value at a time by enforcing that a property will have the same value unless it changes.

$$
\begin{aligned}
Started(t_1, f, x, t_2) \leftarrow\ & \exists a : happensAt(a, t_1) \wedge \\
& Initiates(a, f, x, t_1) \wedge \\
& (t_1 < t_2) \\
Releases(a, f, x, t) \leftarrow\ & \exists a', y : happensAt(a', t) \wedge \\
& Initiates(a', f, y, t) \wedge \\
& y \neq x \\
Clipped(t_1, f, x, t_2) \leftarrow\ & (t_1 < t < t_2) \wedge \\
& \exists a, t : (Terminates(a, f, x, t) \vee \\
& Releases(a, f, x, t)) \\
HoldsAt(f, x, t) \leftarrow\ & \exists t_1 : Started(t_1, f, x, t) \wedge \\
& \neg\, Clipped(t_1, f, x, t) \\
HoldsBetween(f, x, t_1, t_2) \leftarrow\ & \exists t : Started(t, f, x, t_1) \wedge \\
& \neg\, Clipped(t, f, x, t_2)
\end{aligned}
$$

In addition, fluents are not considered as relational (where their values can be $true$ or $false$) but as functional (where for each time point, each

fluent has at most one value). Finally the predicate $HoldsAt$ can be used to find the value of a fluent at a specific time point. Accordingly, the predicate $HoldsBetween$ can be used to find the value of a fluent at a specific time interval. Its semantics are skeptical and not credulous which means that the predicate has to be true at least in the whole time interval and not only in a part of this interval.

This prototype serves as a proof of concept without dealing with optimization of performance, so the reasoner is implemented in SWI-Prolog and accessed through the java-to-prolog interface (JPL)[1]. In the output predicates, T1 is the starting time point and T2 is the ending time point which should be an integer. If T2 is not an integer (for example the string 'a') then the time interval (T1,T2] is interpreted as $(T1, +\infty)$. Each argument in Prolog can always be used as an input (notated as '+'), always as an output (notated as '-') or sometimes as an input and sometimes as an output (notated as '?').

In order for the reasoner to evaluate queries, the prolog Knowledge Base has to be instantiated. The prolog interface of the reasoner is composed of 3 predicates for instantiation and 3 for inference.

| **purpose** | **prolog predicates** |
|---|---|
| Instantiation | keepNewInitiates$(+A, +F, +V, +T_{init}, -CA, -CV)$<br>happensAt$(+A, +T)$<br>terminates$(+A, +F, +V, +T_{term})$ |
| Inference | holds$(?F, ?V, +T1, +T2)$<br>holds$(-F, +Comparator, +V, +T1, +T2)$<br>actionHolds$(-F, -V, +T1, +T2)$ |

**keepNewInitiates**$(+A, +F, +V, +T, -CA, -CV)$

The predicates $keepNewInitiates$ and $happensAt$ are used to add an event. An inconsistency is reported when a new event has the same fluent and starting time point with another one but different fluent value. In other words, the predicate $keepNewInitiates$ returns the action that conflicts with the new one and the value of the fluent according to that action. If it returns nothing then there is no inconsistency and the new event is added to the prolog Knowledge Base.

**happensAt**$(+A, +T)$

The predicates $keepNewInitiates$ and $happensAt$ are used to add an

[1]JPL is part of the SWI-Prolog distribution : http://www.swi-prolog.org/

event. Although the semantics of the predicate *happensAt* could be merged with those of *initiates*, all extensions of Event Calculus are based on these semantics.

**terminates**$(+A, +F, +V, +T)$

It is used optionally to specify when the event stops having an effect.

**holds**$(?F, ?V, +T1, +T2)$

It is true for the facts where fluent F has value V for the entire time interval (T1,T2] and not only during a part of this interval. If T2 is not an integer (like the letter 'a'), then it is true for the facts where fluent F has value V for the time point T2. If both fluent F and value V are outputs of the same prolog query, then the predicate *actionHolds* should be used.

**holds**$(-F, +Comparator, +V, +T1, +T2)$

This predicate compares the value of fluents $F$ at (T1,T2] with $V$ using the given *Comparator* as a comparison operator. It returns the fluents $F$ for whom the comparison is true. If V is a number, then the *Comparator* can be one of '=', '! =', '<', '>', '<=', '>='. This predicate can also be used to check if the value of fluent F at (T1,T2] matches a regular expression limited to the use of wildcards. The wildcards partition the regular expression to many strings which are represented as a list of strings in argument $V$. In this case, the *Comparator* declares whether there is a prefix or a suffix in the regular expression by taking one of the following values:
0 means that there is no prefix and no postfix
1 means that there is a prefix but no postfix
2 means that there is no prefix and there is a postfix
3 means that there is a prefix and a postfix

**actionHolds**$(-F, -V, +T1, +T2)$

Same as *holds* but it uses both fluent F and value V as output of the same prolog query.

For example, providing that currency is euro and that the name of a time point is the number of seconds passed after a common time point, the following prolog queries are examples of all possible uses of the output predicates:

1. holds('productName'('Product1'),V,'3',a) returns the name of the product with ID 'Product1' at the time point 3.

2. holds('price'(ID),'22','9',a) returns the product whose price at the time point 4 was 22.

3. holds('productName'(ID),1,['P','a'],'4',a) returns the products whose name follows the pattern 'P*a*' at the time point 4

4. actionHolds('produces'(ID1),ID2,'4',a) returns the IDs of companies that produce a product at the time point 4 and the IDs of these products. As a result, each company returned produces exactly one product.

The complexity of the predicate $Holds$ depends on the fluent used. The reasoner checks all timeslices of this fluent to find if there is a timeslice in the given time interval with a different fluent value. Thus the complexity of the predicate $Holds$ is $\Omega(n_f)$ where $n_f$ is the number of individuals of class X where "tsTimeSliceOf only X" is the domain of fluent $f$. The evaluation of the predicate $actionHolds$ depends on the action used as it starts by finding all individuals $id$ of class X where "tsTimeSliceOf only X" is the domain of action $a$. Then the goal for each individual found is : holds($a(id)$,V,$t_1$,$t_2$). Thus the complexity of the predicate $actionHolds$ is $\Omega(n_a^2)$ where $n_a$ is the number of individuals of class X where "tsTimeSliceOf only X" is the domain of action $a$.

# Chapter 4

# TOQL extensions

TOQL2 is an extension of TOQL where the AT operator is handled by the reasoner presented in the previous chapter. Additionally, the syntax has been extended to support the AT operator in the SELECT clause to simplify the composition of some queries. TOQL2 has been tested over a sample temporal ontology presented in appendix B in the Manchester OWL syntax [12]. The following queries explain how the new semantics of the AT operator affect the result of the query.

Query example #1:

In our ontology the price of "Product2" is '15' for the time interval (6,10) and '16' for the time interval (8,15). This query returns the price of "Product2" at the time interval [9,10] which is '16'. TOQL2 has changed the semantics of the AT operator from finding the price of "Product2" that holds over the given time interval or time point to enforcing that the price of "Product2" will have the same value unless it changes. The AT operator is handled by the reasoner described in the previous chapter which adds the restriction that a fluent can not have many values at the same time. This restriction is applied only over functional fluents (and functional actions), thus the ontology developer can choose if he wants this restriction. For example, in our ontology, although it is not conceptually right, a company can produce at most one product at a time.

```
SELECT Product.price AT(9,10) AS current_price
FROM Product
WHERE Product LIKE "Product2"
```

Table 4.1: TOQL2 query #1

Query example #2:

The result of the query states that company "C1" produces the product "Product1" at the time point '2'. Although, TOQL would return the products "Product1" and "Product4", TOQL2 returns only the product "Product1".

```
SELECT Product
FROM Company, Product
WHERE Company.produces:Product AT(2)
AND Company.companyName LIKE "C1"
```

Table 4.2: TOQL2 query #2

Query example #3:

Although all productNames start with the letter 'P', the property *productName* is temporal, so we use the AT operator. This query returns the names of the companies that produce products at the time point '7' whose name starts with 'P'. Also it returns the names of the corresponding products.

```
SELECT Company.companyName, Product.productName
FROM Company, Product
WHERE Company.produces:Product AT(7)
AND Product.productName LIKE "P*" AT(7)
```

Table 4.3: TOQL2 query #3

| **companyName** | **productName** |
|---|---|
| C1 | P2 |
| C2 | P3 |

As in SeRQL, the pattern is not a regular expression as the only metacharacter allowed is the wildcard character. However, in TOQL2 comparisons are always case sensitive.

Query example #4:

This query returns "Product3" whose price at time point 9 equals the salary of an employee. TOQL2 can compare two datatype properties (only

through nested queries) and a datatype property with a constant value. Before analyzing the implementation of the AT operator, the implementation of TOQL is presented.

```
SELECT Product
FROM Company, Product, Employee
WHERE Company.produces : Product
AND Product.price AT(9) =
ANY ( SELECT salary FROM Employee )
```

Table 4.4: TOQL2 query #4

## 4.1 TOQL implementation

Sesame is a framework for querying and analyzing RDF data which provides a Storage And Inference Layer (SAIL) API that separates the query language from the storage device used (in-memory storage, disk-based storage, RDBMS)[1]. In other words, a SeRQL or SPARQL query can be translated into an SQL query under certain circumstances. Additionally, although the middleware OpenLink Virtuoso[2] enables querying over a quad store (where each "tuple" has 4 columns), the query language that can be used is a subset of SPARQL. As SPARQL uses triple patterns (a whitespace-separated list of a subject, predicate and object), there is no added expressivity on using Virtuoso for temporal querying.

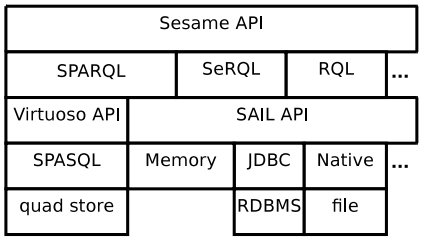



Figure 4.1: Sesame architecture

A TOQL query is translated to a SeRQL (Sesame RDF Query Language) query which is a query language provided by Sesame. The figure below shows all components needed for the evaluation of a TOQL query.

[1]The architecture of sesame can be found here : http://www.openrdf.org/doc/sesame/users/userguide.html#d0e129

[2]The architecture of Virtuoso can be found here : http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/VirtSesame2Provider

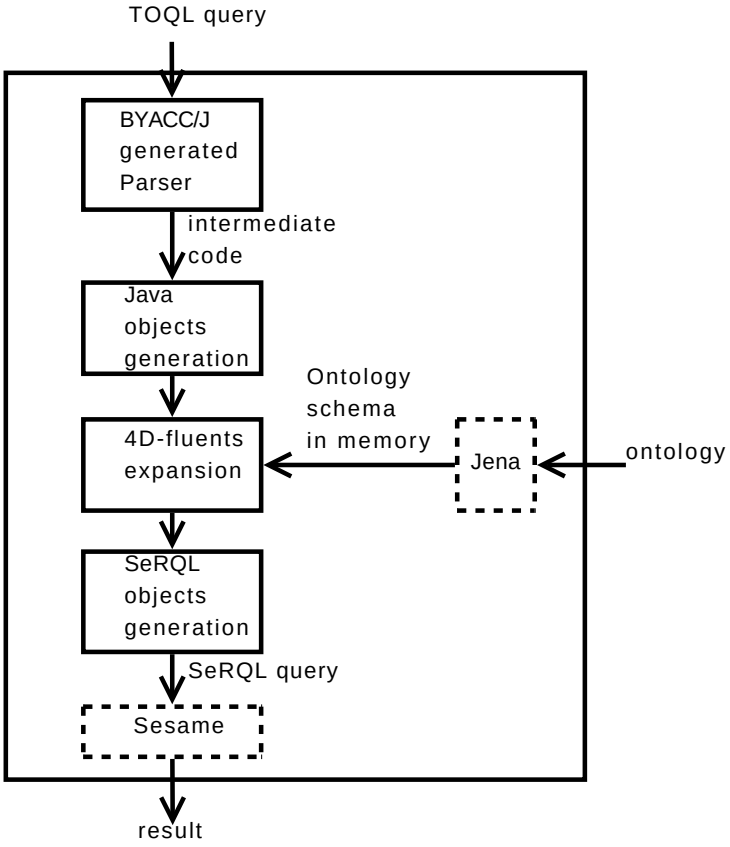



Figure 4.2: Internal structure of TOQL

SeRQL is an SQL-like language that 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. The FROM clause specifies a path expression. Path expressions are expressions that match specific paths through an RDF graph like :
{Person} ex:worksFor {Company} rdf:type {ex:ITCompany}

The words surrounded by curly brackets represent the nodes in the RDF graph, the rest represents the edges in the graph. The nodes and edges in the path expressions can be variables, URIs and literals. In SeRQL queries, multiple path expressions can be specified by separating them with commas.

For example, the path expression shown before can also be written down as two smaller path expressions:
{Person} ex:worksFor {Company},
{Company} rdf:type {ex:ITCompany}

Every class and every property in a TOQL query is represented by a path expression in SeRQL.

| **Resource** | **Generated Path Expression** |
|---|---|
| Class | {Class name} rdf:type {namespace :Class} |
| Object Property | {DomainClassName}<br>namespace: ObjectPropertyName {RangeClassName} |
| Datatype Property | {DomainClassName}<br>namespace: DatatypePropertyName<br>{DatatypePropertyName} |

Table 4.5: OWL basic semantics to SeRQL path expressions

TOQL knows the ontology schema in order to distinguish temporal from non-temporal properties. The ontology schema is loaded through Jena[1] which is also a storage system like Sesame but provides an API (Application Programming Interface) for loading and manipulating an OWL ontology.

As SeRQL does not support TOQL's temporal features (the Allen operators, the AT operator and the TIME operator), it is not aware of the 4D-fluents mechanism. Thus the interpreter translates TOQL queries that use temporal operators to rather complicated SeRQL queries that does not use temporal operators. TOQL is implemented in Java as most tools for RDF and OWL. The building blocks of TOQL[1] are :

1. **Parser** : Jflex generated the lexical analyzer and BYACC/J generated the parser which returns parsed code as a list. This list is an alternative structure of the TOQL query, called intermediate code, which separates parsing from code generation.

2. **Java objects generation** : Intermediate code nodes map to object of various java classes. An object called "upperQuery" of the class *Query* holds all these objects. As shown in figure 4.3, a *Query* has

[1]Jena is an RDF and OWL framework : http://jena.sourceforge.net/

[1]A description of the intermediate code types and of the class hierarchy of java objects can be found at [7]

many *TupleQueries* which has a list for each part of a TOQL query (the SELECT part, the FROM part and the WHERE part).

3. **4D-fluents expansion** : Expressions with temporal operators are replaced with objects through adding timeslices and time intervals to the query (to fields of the class *Query*). In this phase, an object of the class *FromNode* corresponds to a path expression. Every class and every property in a TOQL query is represented by a path expression in SeRQL.

4. **SeRQL objects generation** : Objects of the class *Query* are copied to a new object of the class *FinalQuery* which deals with fixing inconsistencies between some Java objects and the syntax of SeRQL. Then the object of the class *FinalQuery* traverses all of its objects to generate the SeRQL query.

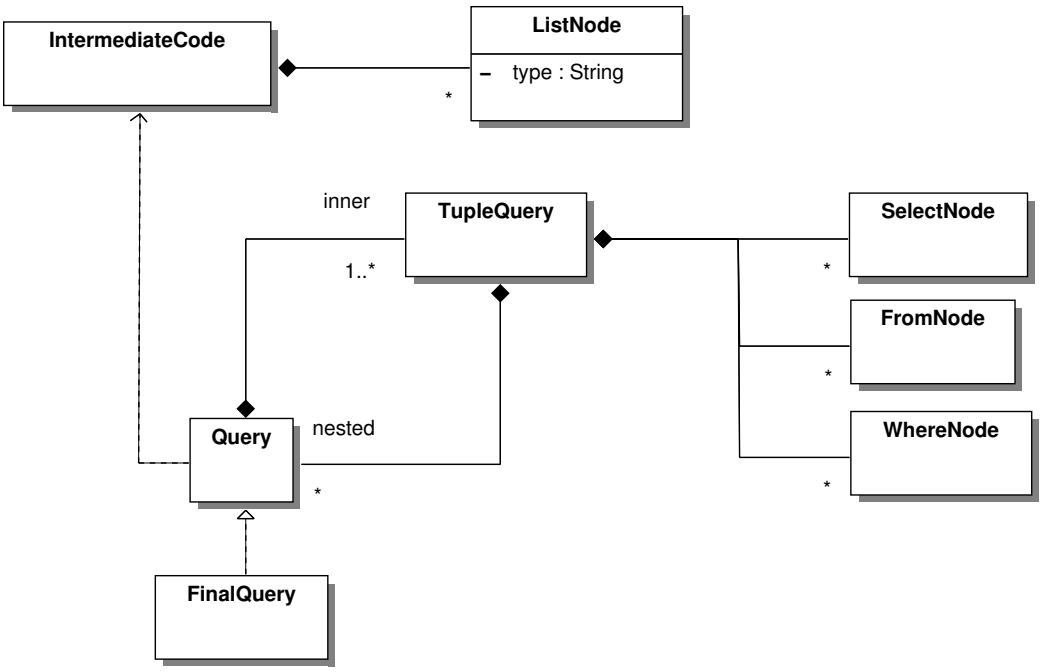
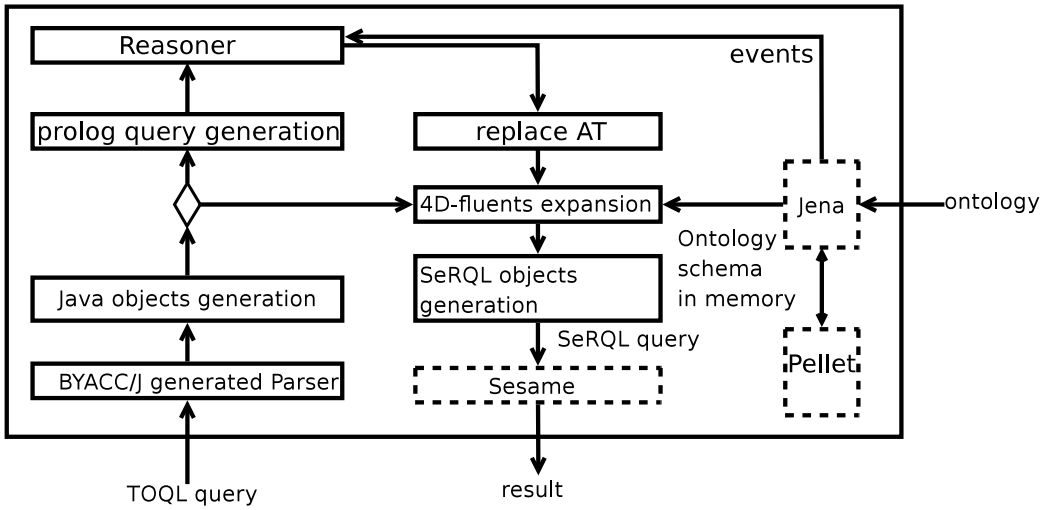



Figure 4.3: Important classes for code generation

In case of an inner query like "Query1 UNION Query2", each subquery is saved as a *Query*. And in case of a nested query, the subquery is saved as a *Tuplequery*, except for the simple case (with no inner or nested operators) where the query is held both by a *Query* and a *Tuplequery*.

## 4.2 TOQL2 implementation

This work uses Pellet [21] to classify the ontology. For example, if the temporal relation "Man is_married_with Woman" is symmetric, a reasoner is needed to infer the temporal relation "Woman is_married_with Man". In other words, TOQL2 can answer queries like getting all women that are married with men (and their men) at 2007, although the temporal relation "Woman is_married_with Man" is not explicitly stated in the ontology. Additionally, the events of the ontology are loaded to the reasoner presented on chapter 3 which handles the AT operator if only the operant is temporal and owl:functional.




TOQL2 has changed the first two phases of TOQL which involve parsing and java objects generation.

**Parser** :

1. Parsing of an expression that includes the AT operator (in the SELECT or the WHERE clause) generates a *ListNode* of type "callReasoner", if only the expression has a functional fluent or action.

**Java objects generation** :

1. The class *TupleQuery* processes the intermediate code. When a *ListNode* of type "callReasoner" is reached, a prolog query is generated and sent to the reasoner.

2. The reasoner answers and the results are expressed in the WHERE clause.

As an example, consider the following condition in the WHERE clause :

```
Company.produces : Product AT(4)
```

Generated prolog query : $actionHolds(produces(Company), Product, 4, -)$.

Reasoner's answer :

| Company | Product |
|---|---|
| Company1 | Product4 |
| Company2 | Product3 |

Condition in place of the previous one :

```
(
      ( Company Like "Company1" AND Product Like "Product4" )
OR
      ( Company Like "Company2" AND Product Like "Product3" )
)
```

As stated in the beginning of this chapter, the TOQL's syntax has been extended to support the AT operator in the SELECT clause to simplify the composition of some queries. The evaluation of a TOQL2 query that uses the AT operator in the SELECT clause is split in the following steps :

1. SELECT clauses that use the AT operator are saved in a list L and replaced with their identifier.
2. The SeRQL query is generated and executed.
3. The reasoner is called for each element of list L and the results of the prolog queries are expressed as WHERE clauses.
4. The modified SELECT clauses are replaced with their initial form except for the AT operator.

As an example, consider the following TOQL2 query :

```
SELECT Product.productName AT(2)
FROM Product
```

Table 4.6: TOQL2 query

1. The SELECT clause "SELECT Product.productName AT(2)" is replaced with the clause "SELECT Product".

2. The result of the generated SeRQL query is "Product1".

3. The prolog query $holds('productName'('Product1'), V, '2', a)$ returns the value "P1".

4. The SELECT clause "SELECT Product" is changed to "SELECT Product.productName".

```
SELECT productName_ProductSlice_1
FROM {ProductSlice_1} ex1:productName {productName_ProductSlice_1},
{Product} rdf:type {ex1:Product},
{ProductSlice_1} rdf:type {ex1:TimeSlice},
{ProductSlice_1} ex1:tsTimeSliceOf {Product}
WHERE ( productName_ProductSlice_1 Like "P1" )
USING NAMESPACE ex1= <http://www.semanticweb.org/ontologies/
2008/4/Ontology1211440295085.owl#>
```

Table 4.7: SeRQL query equivalent to TOQL2 query of table 4.6

In order for TOQL2 to provide Abox reasoning (i.e. reasoning based only on relations between individuals and not between concepts) for expressions that use the AT operator , the individuals of the temporal part of the ontology are translated to into prolog predicates as stated in section 3.1. The notation below is used to provide an example of this translation.

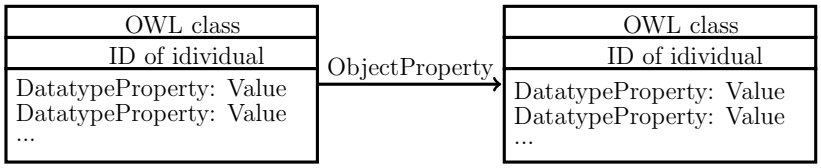



Figure 4.4: Notation of OWL individuals

TOQL2 uses Jena to get all fluentProperties and TOQL queries (that do not use the AT operator) to get all individuals of the OWL classes TimeSlice and TimeInterval. A sample domain is one where a company named C1 produces Product1 and sells it for 10.0 euro from 2001 to 2005.

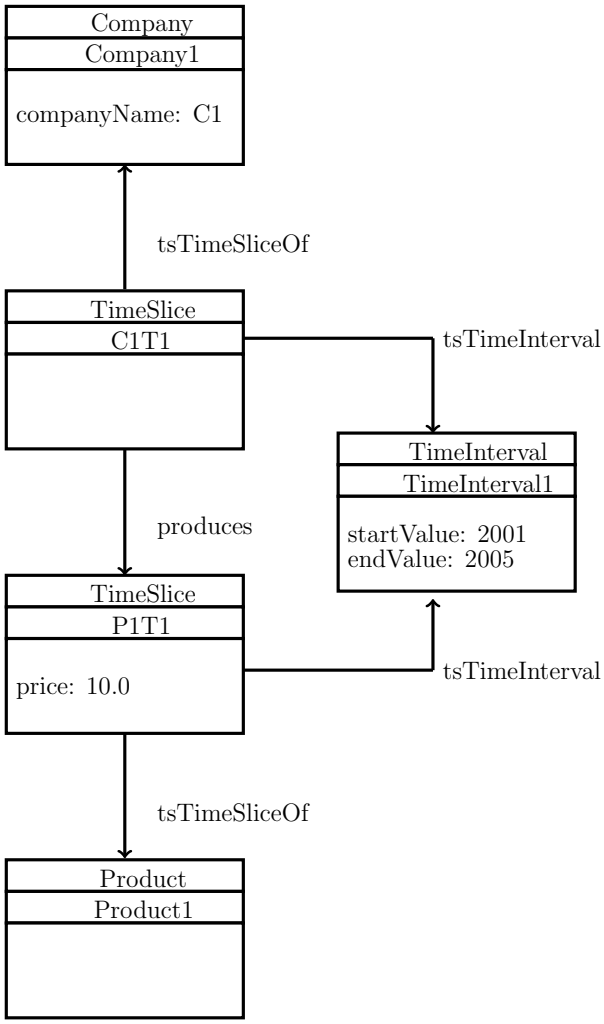



Figure 4.5: A temporal relation in the 4D-fluents schema

Note that the companyName is not on the temporal part of the ontology. "Price" is a fluent and "produces" is an action. The TOQL2 interpreter generates the following prolog predicates which ensure that "A Product has the same name until is changes, thus it can not have many names at a time":

1. keepNewInitiates('produces'('C1','Product1'), 'price'('Product1'), '10.0', '2001', 'produces'(ID1,ID2), V).
2. assert(happensAt('produces'('C1','Product1'),'2001')).
3. assert(terminates('produces'('C1','Product1'),'price'('Product1'), '10.0','2005')).

Also, the TOQL2 interpreter generates the following prolog predicates

which ensure that "A Company produces the same product until it change product, thus it can not produce many Products at a time":

1. keepNewInitiates('produces'('C1','Product1'), 'produces'('C1'), 'Product1', '2001', 'produces'(ID1,ID2), V).
2. assert(happensAt('produces'('C1','Product1'),'2001')).
3. assert(terminates('produces'('C1','Product1'),'produces'('C1'), 'Product1','2005')).

# Chapter 5

# Conclusion and future work

Since OWL allows only unary and binary relations, information that changes over time can not be easily represented. In the 4D-fluents KR (Knowledge Representation) [23] the ontology has a static and a temporal part (a static and a temporal Tbox as well as a static and a temporal Abox). A known [23] limitation of the 4D-fluents KR is that it can not represent temporal cardinality constraints like restricting each product to have at most one value at a time. As the Temporal Ontology Query Language (TOQL)[7] answers temporal queries by finding all events that occured on an interval that includes time Y, it cannot provide the needed functionality.

Although, our approach is not to prevent the insertion of overlapping events, on queries like "Which are the prices of product X at time Y?" we return at most one result as we find the last event that occured on an interval that includes time Y. Queries like the above are written in TOQL2 which extends the semantics of the AT operator of TOQL. First we translate the temporal Abox to instances of a prolog Knowledge Base. Then in queries where the temporal property that uses the AT operator is functional, the AT operator is handled by our Prolog reasoner. Our reasoner is based on Event Calculus which represents the knowledge that everything that does not change, will have the same value at a later time. In queries like getting the price of a product, the time complexity of the reasoner is linear to how many times the price of this product has changed.

Future work includes a better reasoning process and a better query language :

- **Reasoner** : faster implementation and extensions of Event Calculus
- **TOQL2** : Allen operators over time points, removal of ontology schema loading, refactoring of error handling, schema awareness, update mechanism

**Faster implementation :**

TOQL2 replaces a query that used the AT operator with a query that does not use the AT operator by partially evaluating the initial query and then letting sesame evaluate the generated query. An alternative approach is to express the Event Calculus rules as conditions in the WHERE clause as shown in table 5.2, thus avoiding partial evaluation which is now done by the prolog compiler. However, this requires two new operators instead of the TIME operator. Additionally, translating the AT operator from TOQL to TOQL and finally to SeRQL enables the combination of the AT operator with schema aware queries.

```
SELECT fixedValue
FROM Product
WHERE fixedValue2 = '5'
AND productName AT(3) LIKE "P*"
```

Table 5.1: TOQL2 query

```
SELECT X.fixedValue
FROM Product AS X
WHERE (X.t1 < 3 AND
(X.t2 > 3 OR
NOT EXISTS(
   SELECT productName
   FROM Product
   WHERE X.t1 < t1 AND t1 < 3
   AND productName != X.productName)))
AND X.productName LIKE "P*"
AND X.fixedValue2 = '5'
```

Table 5.2: "TOQL3" query equivalent to TOQL2 query of table 5.1

**Extensions of Event Calculus :**

Our mapping enables a temporal ontology to use extensions of Event Calculus [20]. Additionally, TOQL2 can be extended to support qualitative spatio-temporal reasoning like answering the query "which Book is on Table X at the time point T". The knowledge that has to be represented is when the positions of Books and Tables changed and which are the coordinates of the Book or Table that moved. The coordinates can be expressed as fluents

whose values are measured in relation with a fixed point in a 2 dimensional space. Although the movement of a Book or a Table is assumed to occur in zero-time, there are extensions of Event Calculus [20] that can cope with the so called "actions with continuous change".

**Allen operators over time points :**

If an individual has a startValue X and an endValue '-1', then the time interval is considered as $(X, +\infty)$. TOQL2 replaces Allen operators with inequalities between the startValue and the endValue of the corresponding time intervals but it does not take into account that one of the time intervals or both may have no ending time point. In other words, TOQL and TOQL2 do not support queries with Allen operators over time intervals whose endValue is infinite.

**Refactoring of error handling :**

The use of a static property X with the TIME operator raises the error "Property is not a fluent" which should be more general like "The value of property X does not depend on time". Additionally, the same error should be raised when a static property is used as an argument of an Allen operator.

However, restricting a static property X through the AT operator should not raise the error "Property is not a fluent" but a warning like "The AT operator over the property X is useless". If two temporal properties have a common OWL class in their domain or range, then we define them as *connected* and many *connected* properties form a *set*. If no property (in the SELECT or the WHERE clause) of a *set* uses a temporal operator, then the warning "Expected a temporal operator (the TIME, the AT or an Allen operator)" should be raised.

**Removal of ontology schema loading :**

TOQL2 needs to distinguish between temporal and static OWL properties to treat them accordingly. To deal with this problem, TOQL2 loads the whole ontology schema to memory via Jena. A change in the ontology schema requires reloading the schema into memory in order for TOQL2 to return the right results. Alternatively, the classes and properties of a query can be checked if they conform to the ontology schema when needed. In other words, Jena can be used at each TOQL2 query (at the phase of java objects generation) without saving the ontology schema in memory.

**Schema awareness :**

As stated in [8] "a query language should be aware of the structure it is querying and capable of exploiting this structure for type checking, optimization, inheritance, etc". SeRQL supports RDF Schema semantics through the following operators in the WHERE clause :

1. *X subClassOf Y* : class X is a sub-class of Y

2. *X subPropertyOf Y* : property X is a sub-property of Y

3. *X instanceOf Y* : X is an individual of class Y

Additionally, the application developer can choose to get only the direct subclasses (and not the whole class subtree) through the operator *directSubClassOf*. Similarly, SeRQL provides the operators *directSubPropertyOf* and *directInstanceOf*. Likewise, TOQL2 can provide queries based on the knowledge that already has, like:

1. Does property X depend on time?

2. Is property X temporal and functional (restricted to have at most one value at a time)?

3. Which are the properties of class X?

**Update mechanism :**

TOQL2 can be enhanced to support not only queries for information retrieval but also update operations on OWL individuals. However, sesame supports only queries for information retrieval and as of 2008 most semantic web applications update data directly via APIs provided by specific storage systems. Thus, TOQL2 can provide INSERT/UPDATE/DELETE statements through the "Repository API" of sesame.

Query:

```
UPDATE Company, Product
SET Company.companyName = "C7"
WHERE Company.produces:Product AT(3)
AND productName AT(2003) LIKE "P1"
```

A sample process for the above query is :

1. The UPDATE query is translated to the following SELECT query which returns the value “Company1”.

```
SELECT Company
FROM Company, Product
WHERE Company.produces:Product AT(3)
AND productName AT(2003) LIKE “P1”
```

2. The “Repository API” is used to change the RDF triple (Company1, companyName,C1) to (Company1,companyName,C7).

Obviously, the use of the AT operator in clauses other than the WHERE clause needs special treatment. Additionally, if Event Calculus is not embedded in TOQL2 (which is the approach of a faster implementation discussed above), the prolog Knowledge Base will have to be updated.

# Appendix A

# TOQL2 grammar in BNF

The compiler-compiler used is BYACC/J, a variant of Berkeley YACC that supports actions expressed in java. Note that the symbols with only upper case letters are terminal symbols and obviously the empty rules have useful actions.

```
ParseUnit:
  Query
  ;
Query:
  TupleQuerySet
  ;
TupleQuerySet:
  beforeTupleQueries TupleQueries afterTupleQueries
  ;
beforeTupleQueries:/* empty */
  ;
afterTupleQueries:/* empty */
  ;
TupleQueries:
  TupleQuery TupleQuerySetCont
  ;
TupleQuerySetCont:
  /* empty */
| SetOperator TupleQueries
  ;
TupleQuery:
  '(' TupleQuerySet ')'
| SelectQuery
```

```
  ;
SetOperator:
  UNION
| UNION ALL
| MINUS
| INTERSECT
  ;
SelectQuery:
  SELECT SelectQueryBody
| SELECT DISTINCT SelectQueryBody
| error SelectQueryBody
  ;
SelectQueryBody:
  Projection
| Projection QueryBody
  ;
Projection:
  /* empty */
| STAR
| ProjectionElemList
  ;
ProjectionElemList:
  ProjectionElem1 mayhave1
  ;
mayhave1:
  /* empty */
| ',' ProjectionElemList
  ;
ProjectionElem1:
  ValueExpr2
| ValueExpr2 AtOperators
| ValueExpr2 AtOperators AS NAME
| ValueExpr2 AS NAME
  ;
QueryBody:
  FROM FromPathExpr
| error FromPathExpr
  ;
FromPathExpr:
  PathExprList
| PathExprList WHERE WhereBoolExpr
```

```
| PathExprList LIMIT INTNUM
| PathExprList LIMIT INTNUM OFFSET INTNUM
| PathExprList OFFSET INTNUM
  ;
WhereBoolExpr:
  BooleanExpr
| BooleanExpr LIMIT INTNUM
| BooleanExpr LIMIT INTNUM OFFSET INTNUM
| BooleanExpr OFFSET INTNUM
| error
  ;
PathExprList:
  PathExpr
| PathExpr ',' PathExprList
| error
  ;
PathExpr:
  NAME
| NAME AS NAME
  ;
ValueExpr:
  Var
  ;
ValueExpr2:
  SelectVar
  ;
ValueExpr3:
  NAME
| NAME '.' NAME
| FLOATNUM
| INTNUM
  ;
BooleanExpr:
  OrExpr
  ;
OrExpr:
  AndExpr
| AndExpr OR BooleanExpr
  ;
AndExpr:
  BooleanElem
```

```
| BooleanElem AND AndExpr
| BooleanElem2 AtOperators
| BooleanElem2 AtOperators AND AndExpr
| atExpr AtOperators COMP atExpr AtOperators
| atExpr AtOperators COMP atExpr AtOperators AND AndExpr
| atExpr AtOperators COMP ValueExpr3
| atExpr AtOperators COMP ValueExpr3 AND AndExpr
| nested
| BooleanElem2 AllenOperators BooleanElem2
| BooleanElem2 AllenOperators BooleanElem2 AND AndExpr
  ;
Cont:
  '(' TupleQuerySet ')'
| '(' TupleQuerySet ')' AND AndExpr
  ;
nested:   atExpr AtOperators COMP ALL Cont afterNestedQuery
| atExpr AtOperators COMP ANY Cont afterNestedQuery
| atExpr AtOperators IN Cont afterNestedQuery
  ;
AtOperators:
  AT '(' INTNUM ')'
| AT '(' INTNUM ',' INTNUM ')'
| AT '(' DATE ')'
| AT '(' DATE ',' DATE ')'
  ;
AllenOperators:
  BEFORE
| AFTER
| EQUALS
| MEETS
| METBY
| OVERLAPS
| OVERLAPPEDBY
| DURING
| CONTAINS
| STARTS
| STARTEDBY
| ENDS
| ENDEDBY
  ;
BooleanElem:
```

```
  '(' BooleanExpr ')'
| TRUE
| FALSE
| NOT BooleanElem
| ValueExpr CompOp ANY '(' TupleQuerySet ')'
| EXISTS '(' TupleQuerySet ')'
| ValueExpr CompOp ALL '(' TupleQuerySet ')'
| ValueExpr IN '(' TupleQuerySet ')'
| ValueExpr
| BooleanElem2
  ;
BooleanElem2:
  ValueExpr LIKE STATICSTRING
| ValueExpr LIKE STATICSTRING IGNORECASE
| NAME '.' NAME ':' NAME
| ValueExpr CompOp ValueExpr3
  ;
atExpr:
  Var
  ;
Var:
  NAME
| NAME '.' NAME
  ;
SelectVar:
  NAME '.' TIME
| NAME '.' NAME '.' TIME
| NAME
  ;
COMP:
  CompOp
  ;
CompOp:
  E
| NE
| L
| LE
| G
| GE
  ;
```

# Appendix B

# Sample temporal ontology

## B.1 Static part of schema

Class: Company
Class: Product
Class: Employee
Class: Country
DataProperty: companyName
    Domain: Company
    Range: string
DataProperty: employeeName
    Domain: Employee
    Range: string
DataProperty: salary
    Domain: Employee
    Range: decimal
ObjectProperty: hasStoresAt
    Domain: Company
    Range: Country

## B.2 Temporal part of schema: basic

Class: TimeSlice
Class: TimeInterval
DataProperty: startValue
    Domain: TimeInterval
    Range: int
DataProperty: endValue

Domain: TimeInterval
Range: int
ObjectProperty: tsTimeInterval
Domain: TimeSlice
Range: TimeInterval
ObjectProperty: tsTimeSliceOf
Domain: TimeSlice
Range: TimeSlice

## B.3 Temporal part of schema: actions

ObjectProperty: hasEmployee
Characteristics: Functional
Domain: tsTimeSliceOf only Company
Range: tsTimeSliceOf only Employee
ObjectProperty: produces
Characteristics: Functional
Domain: tsTimeSliceOf only Company
Range: tsTimeSliceOf only Product

## B.4 Temporal part of schema: fluents

DataProperty: price
Characteristics: Functional
Domain: tsTimeSliceOf only Product
Range: decimal
DataProperty: productName
Characteristics: Functional
Domain: tsTimeSliceOf only Product
Range: string

## B.5 All individuals

```
Individual: TimePoint1
 Types: TimeInterval
 Facts:
endValue  "-1"^^xsd:int,
startValue  "2"^^xsd:int
Individual: Company1
```

```
 Types: Company
 Facts:
hasStoresAt  Greece,
companyName  "C1"^^xsd:string
Individual: C1T3
 Types: TimeSlice
 Facts:
tsTimeInterval  TimePoint1,
tsTimeSliceOf  Company1,
produces  Product4TimeSlice1
Individual: C2T1
 Types: TimeSlice
 Facts:
hasEmployee  E3T1,
tsTimeInterval  TimeInterval3,
tsTimeSliceOf  Company2,
produces  Product3TimeSlice1
Individual: E1T1
 Types: TimeSlice
 Facts:
tsTimeInterval  TimeInterval1,
tsTimeSliceOf  Employee1
Individual: TimeInterval3
 Types: TimeInterval
 Facts:
endValue  "7"^^xsd:int,
startValue  "3"^^xsd:int
Individual: Product4TimeSlice1
 Types: TimeSlice
 Facts:
tsTimeInterval  TimePoint1,
tsTimeSliceOf  Product4,
price  "50"^^xsd:decimal,
productName  "P4"
Individual: TimeInterval2
 Types: TimeInterval
 Facts:
endValue  "10"^^xsd:int,
startValue  "6"^^xsd:int
Individual: Employee1
 Types: Employee
```

```
 Facts:
salary  "22.0"^^xsd:decimal,
employeeName  "John"^^xsd:string
Individual: C1T4
 Types: TimeSlice
 Facts:
tsTimeInterval  TimePoint2,
tsTimeSliceOf  Company1,
produces  Product4TimeSlice2
Individual: Product4TimeSlice2
 Types: TimeSlice
 Facts:
tsTimeInterval  TimePoint2,
tsTimeSliceOf  Product4,
price  "60"^^xsd:decimal,
productName  "P4new"
Individual: Product3TimeSlice2
 Types: TimeSlice
 Facts:
tsTimeInterval  TimeInterval4,
tsTimeSliceOf  Product3,
price  "22.0"^^xsd:decimal,
productName  "P3x"^^xsd:string
Individual: Product3TimeSlice1
 Types: TimeSlice
 Facts:
tsTimeInterval  TimeInterval3,
tsTimeSliceOf  Product3,
price  "21"^^xsd:decimal,
productName  "P3"
Individual: TimePoint2
 Types: TimeInterval
 Facts:
endValue  "-1"^^xsd:int,
startValue  "4"^^xsd:int
Individual: TimeInterval4
 Types: TimeInterval
 Facts:
endValue  "13"^^xsd:int,
startValue  "8"^^xsd:int
Individual: C2T2
```

```
 Types: TimeSlice
 Facts:
tsTimeInterval  TimeInterval4,
tsTimeSliceOf  Company2,
produces  Product3TimeSlice2
Individual: Product2TimeSlice1
 Types: TimeSlice
 Facts:
tsTimeInterval  TimeInterval2,
tsTimeSliceOf  Product2,
price  "15.0"^^xsd:decimal,
productName  "P2"^^xsd:string
Individual: OverlappingTimeInterval
 Types: TimeInterval
 Facts:
endValue  "15"^^xsd:int,
startValue  "8"^^xsd:int
Individual: Employee3
 Types: Employee
 Facts:
salary  "30"^^xsd:decimal,
employeeName  "John"^^xsd:string
Individual: Product2
 Types: Product
Individual: E3T1
 Types: TimeSlice
 Facts:
tsTimeInterval  TimeInterval3,
tsTimeSliceOf  Employee3
Individual: C1T2
 Types: TimeSlice
 Facts:
hasEmployee  E2T2,
tsTimeInterval  TimeInterval2,
tsTimeSliceOf  Company1,
produces  Product2TimeSlice1
Individual: Product2OverlappingSlice
 Types: TimeSlice
 Facts:
tsTimeInterval  OverlappingTimeInterval,
tsTimeSliceOf  Product2,
```

```
price  "16"^^xsd:decimal,
productName  "P2new"^^xsd:string
Individual: C1T1
 Types: TimeSlice
 Facts:
hasEmployee  E1T1,
tsTimeInterval  TimeInterval1,
tsTimeSliceOf  Company1,
produces  Product1TimeSlice1
Individual: C1T5
 Types: TimeSlice
 Facts:
tsTimeInterval  OverlappingTimeInterval,
tsTimeSliceOf  Company1,
produces  Product2OverlappingSlice
Individual: Product1TimeSlice1
 Types: TimeSlice
 Facts:
tsTimeInterval  TimeInterval1,
tsTimeSliceOf  Product1,
price  "10.0"^^xsd:decimal,
productName  "P1"^^xsd:string
Individual: TimeInterval1
 Types: TimeInterval
 Facts:
endValue  "5"^^xsd:int,
startValue  "1"^^xsd:int
Individual: E2T2
 Types: TimeSlice
 Facts:
tsTimeInterval  TimeInterval2,
tsTimeSliceOf  Employee2
Individual: Product4
 Types: Product
Individual: Product1
 Types: Product
Individual: Employee2
 Types: Employee
 Facts:
salary  "15.0"^^xsd:decimal,
employeeName  "Mark"^^xsd:string
```

```
Individual: Product3
 Types: Product
Individual: Company2
 Types: Company
 Facts:
hasStoresAt  Greece,
companyName  "C2"^^xsd:string
Individual: Greece
    Types: Country
```

# Bibliography


[1] Hassan Aït-Kaci. *Warren's abstract machine: a tutorial reconstruction*. MIT Press, 1991. 21

[2] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. MIT Press, 2nd edition, 2008. 10, 13

[3] Krzysztof R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, 1982. 21

[4] Alessandro Artale and Enrico Franconi. *Handbook of time and temporal reasoning in artificial intelligence*, chapter Temporal Description Logics. MIT Press, 2000. 18

[5] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003. 13, 14, 17, 18

[6] Franz Baader, Ralf Küsters, and Frank Wolter. Extensions to description logics. pages 219–261, 2003. 18

[7] Evdoxios Baratis, Euripides G.M. Petrakis, and Nikolaos Papadakis. TOQL: Querying temporal information in ontologies. Technical Report TR-TUC-ISL-02-2008, Department of Electronic and Computer Engineering, Technical University of Crete, Greece, 2008. 4, 11, 15, 22, 37, 44

[8] Jeen Broekstra and Arjohn Kampman. SeRQL: A second generation RDF query language. In *In Proc. SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, pages 13–14, 2003. 47

[9] Randall Davis, Howard Shrobe, and Peter Szolovits. What is a knowledge representation. *AI Magazine*, 14:17–33, 1993. 13

[10] Dov Gabbay, Agi Kurucz, Frank Wolter, and Michael Zakharyaschev. *Many-dimensional modal logics: theory and applications*. Studies in Logic, 148. Elsevier Science, 2003. 18

[11] Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16(1):14–21, 1951. 20

[12] Matthew Horridge, Nick Drummond, John Goodwin, Alan Rector, Robert Stevens, and Hai Wang. The Manchester OWL syntax. In *OWLED2006 Second Workshop on OWL Experiences and Directions*, Athens, GA, USA, 2006. 16, 33

[13] Hans-Ulrich Krieger. Where temporal description logics fail: Representing temporally-changing relationships. In Andreas Dengel, Karsten Berns, Thomas M. Breuel, Frank Bomarius, and Thomas Roth-Berghofer, editors, *KI*, volume 5243 of *Lecture Notes in Computer Science*, pages 249–257. Springer, 2008. 16

[14] Alexander Kubias, Simon Schenk, Steffen Staab, and Jeff Z. Pan. OWL SAIQL - An OWL-DL query language for ontology extraction. In Christine Golbreich, Aditya Kalyanpur, and Bijan Parsia, editors, *OWLED*, volume 258 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007. 14

[15] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83, 1997. 17

[16] Carsten Lutz, Frank Wolter, and Michael Zakharyaschev. Temporal Description Logics: A survey. In *TIME*, pages 3–14. IEEE Computer Society, 2008. 11

[17] Jixin Ma and Brian Knight. Reified temporal logics: An overview. *Artif. Intell. Rev.*, 15(3):189–217, 2001. 17, 18

[18] J. Mccarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969. 17

[19] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977. 18

[20] Murray Shanahan. The event calculus explained. In *Artificial Intelligence Today*, pages 409–430. 1999. 12, 18, 19, 29, 45, 46

[21] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007. 14, 39

[22] Michael Thielscher. FLUX: A logic programming method for reasoning agents. *TPLP*, 5(4-5):533–565, 2005. 17

[23] Christopher Welty, Richard Fikes, and Selene Makarios. A reusable ontology for fluents in OWL. *Frontiers in Artificial Intelligence and Applications*, 150:226–236, 2006. 11, 14, 16, 19, 28, 44