Restriction Checking on OWL:2 Temporal Ontologies

Dissertation Thesis

Sepetas Nikolaos
Department of Electronic and Computer Engineering
Technical University of Crete (*TUC*)
sepe@intelligence.tuc.gr

Dissertation Thesis Committee
Eyripides G.M. Petrakis, Associate Professor (Supervisor)
Stavros Christodoulakis, Professor
Michail G. Lagoudakis, Assistant Professor

February 15, 2011

Abstract

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 (e.g. representing positions of a moving object, the price of a product that changes). However, dealing with information that changes in time in ontologies is a difficult problem to deal with. Many approaches have been proposed to deal with this problem including among others, N-ary relations, reification and, recently, the 4D-fluents or perdurantists approach with the later being the most efficient. Focusing on the 4D-fluents approach, we show how the underlying ontology can effectively be represented in OWL. Representing property restrictions (e.g., cardinality constraints) as well special properties of property elements (e.g., transitive, symmetric, inverse properties) is a distinctive feature of OWL that can be used to enhance the ontology with valuable semantics. Checking an ontology for consistency taking the above OWL features into account ensures integrity and correctness of the representation, a process which is usually implemented by a reasoner (e.g., Pellet). However, handling properties in a 4D fluent representation of a temporal ontology is a feature not yet implemented in existing reasoners. This is exactly the problem this work is dealing with. In particular, we implement an effective property checking method that handles property features of dynamic (e.g., temporal) properties in a 4D fluent implementation of a temporal ontology. It complements existing reasoning mechanisms and handles dynamic properties as static ones: users can declare property restrictions either static or dynamic as usual. Although general, the method is implemented and tested within the environment of TOQL, an SQL-like temporal ontology query language proposed and developed by the Intelligent Information Systems group of the Intelligent Systems Laboratory of TUC.

Contents

1 Introduction **8**
1.1 Problem Definition . . . 8
1.2 Proposed Solution . . . 9
1.3 Contributions of Present Work . . . 9
1.4 Thesis Outline . . . 10

2 Background and Related Work **11**
2.1 Ontologies-OWL . . . 11
2.2 Representation of Time in Ontologies . . . 11
2.3 TOQL . . . 15

3 Restrictions **16**
3.1 Property Restrictions . . . 16
3.1.1 Value Constraints . . . 17
3.1.1.1 OWL:allValuesFrom . . . 17
3.1.1.2 OWL:someValuesFrom . . . 18
3.1.1.3 OWL:hasValue . . . 19
3.1.2 Cardinality Constraints . . . 19
3.1.2.1 OWL:maxCardinality . . . 19
3.1.2.2 OWL:minCardinality . . . 20
3.1.2.3 OWL:Cardinality . . . 20
3.2 Global Cardinality Constraints on Properties . . . 21
3.2.1 OWL:functionalProperty . . . 21
3.2.2 OWL:inverseFunctionalProperty . . . 22

4 Implementation **24**
4.1 Reading the Ontology . . . 25
4.1.1 Loading Restrictions . . . 25
4.1.2 Loading timeIntervals . . . 26
4.1.3 Creating a List of TimeIntervals . . . 27
4.1.4 Loading Individuals . . . 27
4.2 Consistency Checking . . . 28
4.2.1 Cardinality Constraints . . . 28
4.2.2 Value Constraints . . . 29
4.2.3 Global Cardinality Constraints on Properties . . . 30
4.3 Experimental Results . . . 30

5 Conclusion and Future Work **33**

List of Figures

2.1 Schematic Representation of the Concept of Time Determined Ontology . 12
2.2 Static Enterprise Ontology 13
2.3 Dynamic Enterprise Ontology 14

4.1 Company 'C3' Products Over Time 24
4.2 Loading *timeIntervals* 26
4.3 *ToqlTriple* in case of an *OWL:objectProperty* 27
4.4 *ToqlTriple* in case of an *OWL:datatypeProperty* 28
4.5 Checking List after processing the list of *ToqlTriples* 29

List of Tables

3.1 General Form of Restrictions 17
3.2 OWL:allValuesFrom 18
3.3 OWL:someValuesFrom Restriction 19
3.4 OWL:hasValue Restriction 19
3.5 OWL:maxCardinality Restriction 20
3.6 OWL:minCardinalityRestriction 20
3.7 OWL:Cardinality 21
3.8 OWL:functionalProperty - variation 1 21
3.9 OWL:functionalProperty - variation 2 22
3.10 OWL:inverseFunctionalProperty 22

4.1 Classes declared on 'DEn2.owl' 31
4.2 Object Properties declared on DEn2.owl 31
4.3 Datatype Properties declared on 'DEn2.owl' 32
4.4 Experimental Results 32

1 Introduction

Ontologies represent a set of concepts and the relationships between these concepts. Over the past few years there has been an extensive use of ontologies on domains such as artificial intelligence, software engineering and semantic web as a form of knowledge representation, replacing in many cases relational databases. The evolution of the idea, is that knowledge progresses over time. Many projects that concern dynamic ontologies are based on this idea. Representing information that evolves over time in ontologies is still an open research problem.

1.1 Problem Definition

A critical issue in Knowledge Representation (KR) domain is dealing with information that changes over time. Many Knowledge Representation applications need, not only to provide well organised data describing the current state of a domain, but also to provide data describing the domain's evolution. State of the art information representation and reasoning methods have limited expressive power for describing real world changing processes. For example the (important and persistent) knowledge that a person will go through the stages of infant, adolescent and adult, or a company will be established, hire personnel and develop products which evolve as a result of time, cannot be adequately described using existing methods. Ontology representation languages such as OWL are based on binary relations (relations connecting two instances with no time dimension) making the representation of time a difficult matter to deal with as temporal relations become ternary with the addition of the temporal property.

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

Ontologies offer the means for representing high level concepts, their properties and interrelationships. Dynamic or temporal ontologies will in addition enable representation of time evolving information in ontologies through e.g., versioning or the 4-D perdurandist approach. According to this approach all entities are perdurants, making no distinction between endurants (physical objects such as cars, companies, people) and occurants (events such as buying a car). The idea is that each entity is considered to

be an event and has a start and an end point. An entity can be seen as a 'space-time worm', with the slices of the worm being temporal parts (*timeSlices*) of the entity.

The temporal ontology then become more complicated compared to static ontologies (where all properties are directly attached to the static classes) as any temporal properties are attached to temporal classes (i.e. the *timeSlice* and *temeInterval* classes) introduced by the representation [9] and are only indirectly associated with the dynamic classes that the property refers to. In turn, checking for property restrictions becomes complicated and is not supported by the existing reasoners. This is exactly the problem this work is dealing with.

A temporal restriction is associated with a dynamic property. In this case, the restriction must progress over time in the same way the property does. Then, consistency must be ascertained not only once, but at every instance of time (i.e., for every *timeSlice* the property is associated with). As an example, consider a company producing a given number of products. If the property *produces* is static, then reasoning is carried out by simply counting the number of products that are produced from this company. If the property is dynamic, then the reasoning becomes more complex as an individual of a company can produce more products than the restriction allows, but not during the same *timeInterval*. This means that a simple counter cannot determine consistency.

1.2 Proposed Solution

We introduce a mechanism for restriction checking in temporal ontologies. Our mechanism distinguishes between two types of restrictions. If a restriction is associated with a static property, in the following it is referred to as static restriction and it is handled by a reasoner (e.g. PELLET). If it is associated with a dynamic property, it is hereafter referred to as dynamic and calls for the consistency checking mechanism proposed in this work.

1.3 Contributions of Present Work

The contributions of this work are summarized below:

1. We propose a common declaration mechanism for both dynamic and static restrictions on temporal ontologies.
2. Building upon the 4D-fluents mechanism for the representation of temporal concepts, we propose a mechanism to perform restriction checking on temporal ontologies. The implementation of our mechanism is fully described.
3. To show proof of concept we demonstrate the functionality of our proposed mechanism within TOQL, a query language for temporal ontologies incorporating the 4D-fluents representation.

1.4 Thesis Outline

Background knowledge and related research are discussed in Chapter 2. A description of OWL (Web Ontology Language) and of the 4-D perdurantist [1] approach to handle time in ontologies is presented. Chapter 3 provides an overview of OWL and OWL restrictions. The implementation of our mechanism is discussed in Chapter 4. An ontology holding the essential information is presented as a use case. Examples of restrictions on this ontology, along with the results of constraint checking obtained from the proposed method are discussed. Finally, conclusions and issues for further research are discussed in Chapter 5.

2 Background and Related Work

2.1 Ontologies-OWL

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

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

1. OWL Lite that supports a classification hierarchy and simple constraints.
2. OWL DL that supports maximum expressiveness while retaining computational completeness and decidability.
3. OWL Full that supports maximum expressiveness but no computational guarantees.

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

2.2 Representation of Time in Ontologies

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 [28]. Existing methods are relying mostly on temporal Entity Relation (ER) models [29] 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 [30, 31] 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.

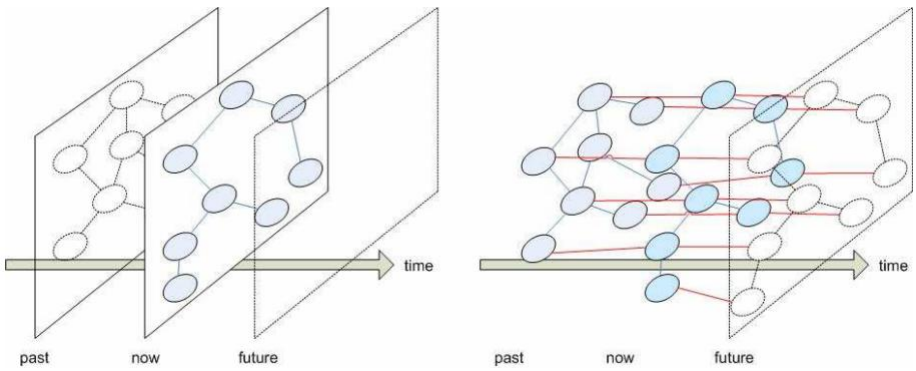

Figure 2.1: Schematic Representation of the Concept of Time Determined Ontology

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 [26].

Versioning [18] 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 [20] are based on binary relations (relations connecting two instances) with no time dimension.

The *4D* (perdurantist) approach [1] 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.

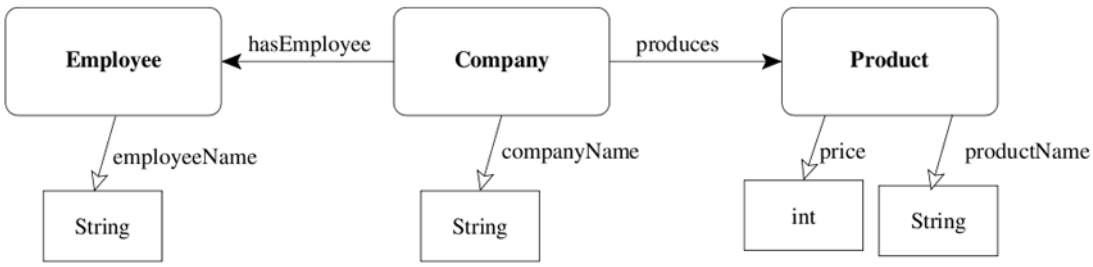

Figure 2.2: Static Enterprise Ontology

As illustrated in Figure 2.1[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

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

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. Following the approach by Welty and Fikes [25], 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*.

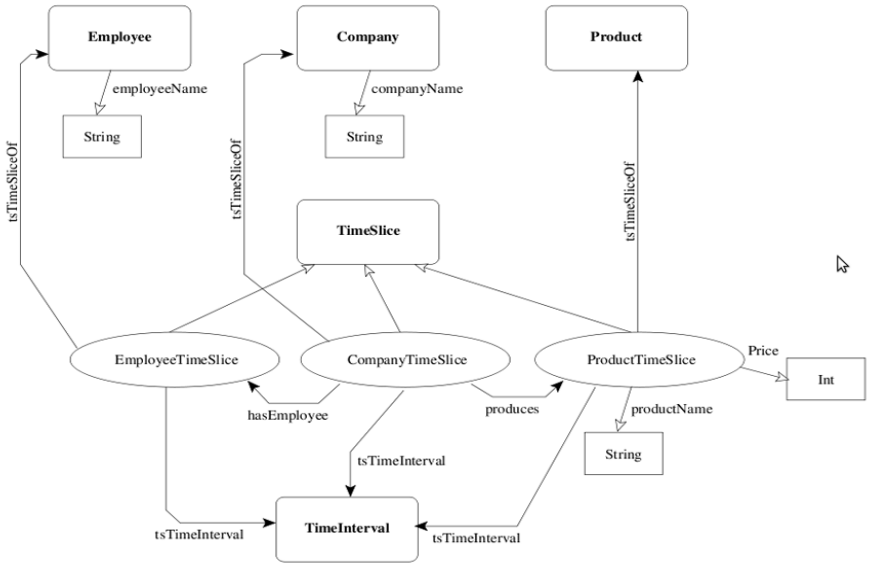

Figure 2.3: Dynamic Enterprise Ontology

Figure 2.3 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 [13] or can be downloaded from http://www.intelligence.tuc.gr/∼petrakis/downloads/TOQL.zip.

2.3 TOQL

TOQL [9] is a high-level query language for querying (time) information in ontologies. TOQL handles ontologies almost like relational databases. Queries in TOQL are issued as SQL statements involving time and high-level ontology concepts that vary in time. TOQL expands the capabilities of state-of-the-art ontology query languages such as SeRQL and SPARQL to handle real world changing information. Unlike SeRQL and SPARQL treats ontologies almost like relational databases, hiding from the user most of the peculiarities of an ontology. TOQL maintains the basic structure of an SQL language (SELECT, FROM,WHERE) and treats the classes and the properties of an ontology almost like tables and columns of a database. TOQL supports expressions relating concepts with time instances or time intervals. This prevents users from being familiar with representation of time in ontologies. TOQL supports queries not only on static information in the static part of the ontology (as conventional query languages do) but also queries on time evolving information instantiated to the ontology (dynamic part). TOQL also introduces the Allen operators (BEFORE, AFTER, MEETS, METBY, OVERLAPS, OVERLAPPEDBY, DURING, CONTAINS, STARTS, STARTEDBY, ENDS, ENDEDBY, EQUALS) that allow comparisons between time intervals, and the operator AT(time point) or AT(time point, time point) that allows comparisons between an interval and a specific time interval or time point.

3 Restrictions

Classes provide an abstraction mechanism for grouping resources with similar characteristics. Every OWL class is associated with a set of individuals, called the class extension. The individuals in the class extension are called the instances of the class. A class has an intensional meaning (the underlying concept) which is related but not equal to its class extension. Thus, two classes may have the same class extension, but still be different classes. OWL classes are described through 'class descriptions'.

A class description is the term used in this document (and in the OWL Semantics and Abstract Syntax) for the basic building blocks of class axioms (informally called class definitions). A class description describes an OWL class, either by a class name or by specifying the class extension of an unnamed anonymous class.

OWL distinguishes six types of class descriptions:

1. A class identifier (a URI reference).
2. An exhaustive enumeration of individuals that together form the instances of a class.
3. A property restriction.
4. The intersection of two or more class descriptions.
5. The union of two or more class descriptions.
6. The complement of a class description.

All information provided in this chapter is following the W3C recommendation [2].

3.1 Property Restrictions

A property restriction is a special kind of class description. It describes an anonymous class, namely a class of all individuals that satisfy the restriction. OWL distinguishes two kinds of property restrictions: value constraints and cardinality constraints.

A value constraint puts constraints on the range of the property when applied to this particular class description. For example, we might want to refer to those individuals whose value of the property *adjacentTo* should be some *Region*, and then use this within a class axiom, perhaps even a class axiom for *Region* itself. Note that this is different from *rdfs:range*, which is applied to all situations in which the property is used.

A cardinality constraint puts constraints on the number of values a property can take, in the context of this particular class description. For example, we might want to say that for a soccer team the *hasPlayer* property has 11 values. For a basketball team the same property would have only 5 values.

Property restrictions have the form shown in table 3.1.

```
<owl:Restriction>
  <owl:onProperty rdf:resource='(some property)' />
  (precisely one value or cardinality constraint, see below)
</owl:Restriction>
```

Table 3.1: General Form of Restrictions

The class *OWL:Restriction* is defined as a subclass of *OWL:Class*. A restriction class should have exactly one triple linking the restriction to a particular property, using the *OWL:onProperty* property. The restriction class should also have exactly one triple that represents the value constraint c.q. cardinality constraint on the property under consideration (e.g., that the cardinality of the property is exactly 1).

Property restrictions can be applied both to datatype properties (properties for which the value is a data literal) and object properties (properties for which the value is an individual).

3.1.1 Value Constraints

3.1.1.1 OWL:allValuesFrom

The value constraint *OWL:allValuesFrom* is a built-in OWL property that links a restriction class to either a class description or a data range. A restriction containing an owl:allValuesFrom constraint is used to describe a class of all individuals for which all values of the property under consideration are either members of the class extension of the class description or are data values within the specified data range. In other words, it defines a class of individuals x for which holds that if the pair (x,y) is an instance of P (the property concerned), then y should be an instance of the class description or a value in the data range, respectively.

A simple example is shown in table 3.2.

```
<owl:Restriction>
  <owl:onProperty rdf:resource='#hasParent' />
  <owl:allValuesFrom rdf:resource='#Human' />
</owl:Restriction>
```

Table 3.2: OWL:allValuesFrom

This example describes an anonymous OWL class of all individuals for which the *hasParent* property only has values of class *Human*. Note that this class description does not state that the property always has values of this class; just that this is true for individuals that belong to the class extension of the anonymous restriction class. Note that in OWL Lite the only type of class description allowed as object of *OWL:allValuesFrom* is a class name.

An *OWL:allValuesFrom* constraint is analogous to the universal (for-all) quantifier of Predicate logic - for each instance of the class that is being described, every value for P must fulfill the constraint. Also notice that the correspondence of *OWL:allValuesFrom* with the universal quantifier means that an *OWL:allValuesFrom* constraint for a property P is trivially satisfied for an individual that has no value for property P at all. To see why this is so, observe that the *OWL:allValuesFrom* constraint demands that all values of P should be of type T, and if no such values exist, the constraint is trivially true.

3.1.1.2 OWL:someValuesFrom

The value constraint *OWL:someValuesFrom* is a built-in OWL property that links a restriction class to a class description or a data range. A restriction containing an *OWL:someValuesFrom* constraint describes a class of all individuals for which at least one value of the property concerned is an instance of the class description or a data value in the data range. In other words, it defines a class of individuals x for which there is at least one y (either an instance of the class description or value of the data range) such that the pair (x,y) is an instance of P. This does not exclude that there are other instances (x,y') of P for which y' does not belong to the class description or data range.

The example shown in table 3.3 defines a class of individuals which have at least one parent who is a physician.

The *OWL:someValuesFrom* constraint is analogous to the existential quantifier of Predicate logic - for each instance of the class that is being defined, there exists at least one value for P that fulfills the constraint. Note that in OWL Lite the only type of class description allowed as object of *OWL:someValuesFrom* is a class name.

```
<owl:Restriction>
  <owl:onProperty rdf:resource='#hasParent' />
  <owl:someValuesFrom rdf:resource='#Physician' />
</owl:Restriction>
```

Table 3.3: OWL:someValuesFrom Restriction

3.1.1.3 OWL:hasValue

The value constraint *OWL:hasValue* is a built-in OWL property that links a restriction class to a value V, which can be either an individual or a data value. A restriction containing a *OWL:hasValue* constraint describes a class of all individuals for which the property concerned has at least one value semantically equal to V (it may have other values as well). For datatypes 'semantically equal' means that the lexical representation of the literals maps to the same value. For individuals it means that they either have the same URI reference or are defined as being the same individual.

The example on table 3.4 describes the class of individuals who have the individual referred to as Clinton as their parent.

```
<owl:Restriction>
  <owl:onProperty rdf:resource='#hasParent' />
  <owl:hasValue rdf:resource='#Clinton' />
</owl:Restriction>
```

Table 3.4: OWL:hasValue Restriction

3.1.2 Cardinality Constraints

In OWL, like in RDF, it is assumed that any instance of a class may have an arbitrary number (zero or more) of values for a particular property. To make a property required (at least one), to allow only a specific number of values for that property, or to insist that a property must not occur, cardinality constraints can be used. OWL provides three constructs for restricting the cardinality of properties locally within a class context. We must note that OWL Lite includes the use of all three types of cardinality constraints, but only when used with the values '0' or '1'.

3.1.2.1 OWL:maxCardinality

The cardinality constraint *OWL:maxCardinality* is a built-in OWL property that links a restriction class to a data value belonging to the value space of the XML Schema datatype *nonNegativeInteger*. A restriction containing an *OWL:maxCardinality* constraint describes a class of all individuals that have at most N semantically distinct

values (individuals or data values) for the property concerned, where N is the value of the cardinality constraint. Syntactically, the cardinality constraint is represented as an RDF property element with the corresponding rdf:datatype attribute.

The example on table 3.5 describes a class of individuals that have at most two parents.

```
<owl:Restriction>
  <owl:onProperty rdf:resource='#hasParent' />
  <owl:maxCardinality rdf:datatype='&xsd;nonNegativeInteger'>2</
      owl:maxCardinality>
</owl:Restriction>
```

Table 3.5: OWL:maxCardinality Restriction

3.1.2.2 OWL:minCardinality

The cardinality constraint *OWL:minCardinality* is a built-in OWL property that links a restriction class to a data value belonging to the value space of the XML Schema datatype *nonNegativeInteger*. A restriction containing an *OWL:minCardinality* constraint describes a class of all individuals that have at least N semantically distinct values (individuals or data values) for the property concerned, where N is the value of the cardinality constraint. Syntactically, the cardinality constraint is represented as an RDF property element with the corresponding *rdf:datatype* attribute.

The example of table 3.6 describes a class of individuals that have at least two parents.

```
<owl:Restriction>
  <owl:onProperty rdf:resource='#hasParent' />
  <owl:minCardinality rdf:datatype='&xsd;nonNegativeInteger'>2</
      owl:minCardinality>
</owl:Restriction>
```

Table 3.6: OWL:minCardinalityRestriction

Note that an *OWL:minCardinality* of one or more means that all instances of the class must have a value for the property.

3.1.2.3 OWL:Cardinality

The cardinality constraint *OWL:cardinality* is a built-in OWL property that links a restriction class to a data value belonging to the range of the XML Schema datatype *nonNegativeInteger*. A restriction containing an owl:cardinality constraint describes a

class of all individuals that have exactly N semantically distinct values (individuals or data values) for the property concerned, where N is the value of the cardinality constraint. Syntactically, the cardinality constraint is represented as an RDF property element with the corresponding *rdf:datatype* attribute.

This construct is in fact redundant as it can always be replaced by a pair of matching *OWL:minCardinality* and *OWL:maxCardinality* constraints with the same value. It is included as a convenient shorthand for the user.

The example of table 3.7 describes a class of individuals that have exactly two parents.

```
<owl:Restriction>
  <owl:onProperty rdf:resource='#hasParent' />
  <owl:cardinality rdf:datatype='&xsd;nonNegativeInteger'>2</owl:
      cardinality>
</owl:Restriction>
```

Table 3.7: OWL:Cardinality

3.2 Global Cardinality Constraints on Properties

OWL also supports a limited set of constructs for defining global property cardinality, namely *OWL:FunctionalProperty* and *OWL:InverseFunctionalProperty*.

3.2.1 OWL:functionalProperty

A functional property is a property that can have only one (unique) value y for each instance x, i.e. there cannot be two distinct values y1 and y2 such that the pairs (x,y1) and (x,y2) are both instances of this property. Both object properties and datatype properties can be declared as *functional*. For this purpose, OWL defines the built-in class *OWL:FunctionalProperty* as a special subclass of the RDF class *rdf:Property*.

```
<owl:ObjectProperty rdf:ID='husband'>
  <rdf:type      rdf:resource='&owl;FunctionalProperty' />
  <rdfs:domain rdf:resource='#Woman' />
  <rdfs:range   rdf:resource='#Man' />
</owl:ObjectProperty>
```

Table 3.8: OWL:functionalProperty - variation 1

The axiom of table 3.8 states that the husband property is functional (i.e., a woman can have at most one husband - a good example of culture dependence of ontologies).

As always, there are syntactic variations. The example of table 3.8 is semantically equivalent to the one of table 3.9.

```
<owl:ObjectProperty rdf:ID='husband'>
  <rdfs:domain rdf:resource='#Woman' />
  <rdfs:range  rdf:resource='#Man' />
</owl:ObjectProperty>
<owl:FunctionalProperty rdf:about='#husband' />
```

Table 3.9: OWL:functionalProperty - variation 2

3.2.2 OWL:inverseFunctionalProperty

If a property is declared to be inverse-functional, then the object of a property statement uniquely determines the subject (some individual). More formally, if we state that P is an *OWL:InverseFunctionalProperty*, then this asserts that a value y can only be the value of P for a single instance x, i.e. there cannot be two distinct instances x1 and x2 such that both pairs (x1,y) and (x2,y) are instances of P.

Syntactically, an inverse-functional property axiom is specified by declaring the property to be an instance of the built-in OWL class *OWL:InverseFunctionalProperty*, which is a subclass of the OWL class *OWL:ObjectProperty*.

Because in OWL Full datatype properties are a subclass of object properties, an inverse-functional property can be defined for datatype properties. In OWL DL object properties and datatype properties are disjoint, so an inverse-functional property cannot be defined for datatype properties.

A typical example of an inverse-functional property is shown in table 3.10.

```
<owl:InverseFunctionalProperty rdf:ID='biologicalMotherOf'>
  <rdfs:domain rdf:resource='#Woman'/>
  <rdfs:range rdf:resource='#Human'/>
</owl:InverseFunctionalProperty>
```

Table 3.10: OWL:inverseFunctionalProperty

This example states that for each object of *biologicalMotherOf* statements (some human) one should be able to uniquely identify a subject (some woman). Inverse-functional properties resemble the notion of a key in databases.

One difference with functional properties is that for inverse-functional properties no additional object-property or datatype-property axiom is required: inverse-functional properties are by definition object properties.

Notice that *OWL:FunctionalProperty* and *OWL:InverseFunctionalProperty* specify global cardinality constraints. That is, no matter which class the property is applied to, the cardinality constraints must hold. This is different from the cardinality constraints contained in property restrictions. The latter are class descriptions and are only enforced on the property when applied to that class.

4 Implementation

This work extends to TOQL to support restriction checking in the underlying 4D-fluents ontology. The extension is implemented in JAVA and uses JENA [4] (for loading and handling the ontology in the main memory) and PELLET (for reasoning on the non dynamic parts of the ontology). The restriction checking mechanism is based upon the following idea: counters between timestamps hold the number of individuals that are subjects to a restricted property. The reason that makes PELLET non applicable to dynamic parts is that it uses one counter and thus cannot separate individuals that refere to other time spaces.

Figure4.1 illustrates the main idea of our mechanism. *Company C3 produces* three different products (*P1, P2, P3*) during different periods of time. Our mechanism can determine that for each time period, only one product is produced and therefore, if a hypothetical *OWL:maxCardinalityRestriction* of value 2 was declared, ontology would be consistent. On the other hand, PELLET (using one counter), would produce as an answer that there are 3 products of *C3*. Obviously, PELLET's answer is incorrect.

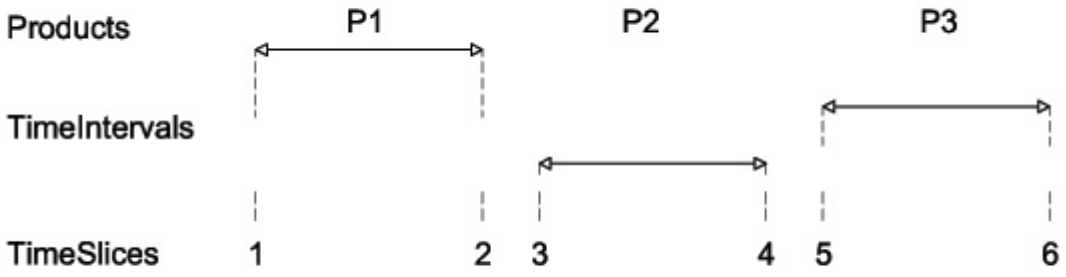

Figure 4.1: Company 'C3' Products Over Time

Our approach to the implementation of this reasoning system was, firstly, to create lists according to the type of restriction. Then, we sequentially access the ontology and create lists that hold all individuals that are subjects and objects to each restriction along with a *timeInterval* that this action is taking place. Having all information that we need, we can calculate the counters described above and check if the ontology is consistent.

Our mechanism works in two phases, described in Sec 4.1 and Sec 4.2 below. During the first phase, the mechanism loads the ontology into the main memory. During the second phase, the mechanism performs restriction checking on the ontology.

4.1 Reading the Ontology

The mechanism determines if the loaded ontology is in a valid 4D fluent representation. If it is not, the method terminates. More specifically, the method performs checks for the existence and syntax on object and datatype properties and determines whether their domain or range is *timeInterval* or *timeSlice* class. This functionality has been previously implemented by Evdoksios Baratis [8].

Our contribution to this work is the extension of this mechanism for supporting inheritance. PELLET reasoning can easily determine the domain of a property, as long as this property is not dynamic. To demonstrate the importance of this extension, we present the following example. If a static object Property, e.g. *hasStoresAt*, has domain of type *Company*, then PELLET autoatically adds all subclasses of class *Company* to domain's list of objectProperty *hasStoresAt*. On the other hand, if an objectProperty, e.g. *produces*, has a domain of type *'timeslice of only Company'* which means that it is dynamic, then we must manually add all subclasses to domain's list of objectProperty *produces* because PELLET cannot determine domain and range of a 4D fluent property.

Assuming that the ontology is based on the 4D perdurandist approach, the information that requires extraction is the restrictions, *timeIntervals* and individuals.

4.1.1 Loading Restrictions

Loading all the declared dynamic restrictions is the first stage for performing the consistency checking on the ontology. We do not load static restrictions, knowing that consistency checking to all static parts of our ontology can be achieved using a common reasoner. To store our (dynamic) restrictions, we created a new class, named *ToqlRestriction*. In this class we hold all the important information of a restriction, which is the domain and range along with some additional information, such as the restriction value, if exists. In addition, two optional fields hold information that is useful when we are dealing with *OWL:hasValueRestriction* and *OWL:allValuesFromRestriction* respectively.

When these fields are loaded in the main memory, the restrictions are stored in vectors with respect to their type. We end-up with five lists that contain the following types of *OWL:restrictions*. *OWL:minCardinalityRestriction*, *OWL:maxCardinalityRestriction*, *OWL:someValuesFromRestriction*, *OWL: allValuesFromRestriction* and *OWL: hasValueRestriction*.

Note that the restrictions of type *OWL:functionalProperty*, *OWL: inverseFunctionalProperty* and *OWL:cardinalityRestriction* are handled as two different ones, one of type *OWL: maxCardinalityRestriction* and one of type *OWL:minCardinalityRestriction*. In the first two cases, the field value is '1'.

4.1.2 Loading timeIntervals

Our main concern is the difficulty to retrieve all the essential information from the ontology because of the complexity of the 4-D perdurandist approach. If a property is dynamic, then the domain and range (if any) should be done with multiple accesses to the same individuals (*OWL:Individual*) that contain information about time, the *timeIntervals*. To speed-up access to *timeIntervals* we load them in a sorted list, using the name of the timeInterval as the sorting criterion. Note that *timeIntervals* are also stored in the ontology as *OWL:Individuals*.

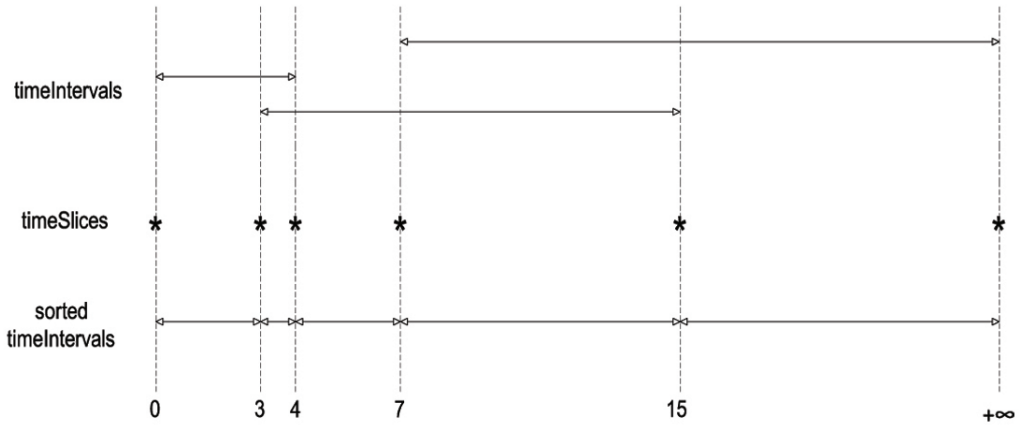

Figure 4.2: Loading *timeIntervals*

We created a new class, named *ToqlTI*, holding the *timeIntervals* of our ontology. This class has three fields: the name of the *timeInterval*, the start value and the end value. Then, we use an iterator through all the *timeIntervals* declared on the loaded ontology and this information forms a list. Now the access to this information will be much faster,

as we are going to perform multiple accesses to the same *timeIntervals* while we load the essential *OWL:Individuals*.

4.1.3 Creating a List of TimeIntervals

Consistency checking on restrictions, that refer to dynamic properties, is performed taking into account the known lifetime of each individual. This means that we do not apply our mechanism from start value 0 to infinity. We only care for the duration that an individual holds.

Our first concern was to create a list of counters in order to determine how many individuals refer to a specific *timeInterval*. This is achieved by using an iterator through *timeIntervals* and retrieving all time stamps. Afterwards, they are stored in a list, sorted by starting point. Using these time points in pairs, we created new *timeIntervals* that in addition have as extra field a counter. Figure 4.2 shows this mechanism.

4.1.4 Loading Individuals

The last step of preprocessing was to load all individuals that are subjects (and objects) to restricted properties. This allowed us to have all information needed easily accessible. Otherwise, we should perform multiple accesses to the ontology which is a time consuming procedure.

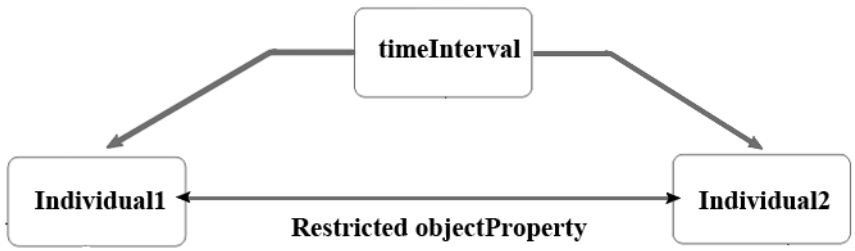

Figure 4.3: *ToqlTriple* in case of an *OWL:objectProperty*

We created class *ToqlTriple* in order to hold information about a restricted property, whether it is an *OWL:datatypeProperty* or an *OWL:objectProperty*. This class has three fields. The first filed is an individual that is always a member of the domain class of the property. The second one is either an individual, member of the range class of the property, or a literal, on respect of the type of the property. Finally, the third field,

holds information about the *timeInterval* that this relation is taking place. Figures 4.3 and 4.4 demonstrate the relations that are stored using the class *ToqlTriple*.

Preprocessing reults into lists holding *ToqlTriples* corresponding to *ToqlRestrictions*.

4.2 Consistency Checking

Because of the particularities of the different kinds of restrictions, consistency checking takes place using different procedures. As described above, *OWL:restricitions* can be of different types. In the following we handle restrictions depending on their type. Examples of these categories are fully described in section 4.3.

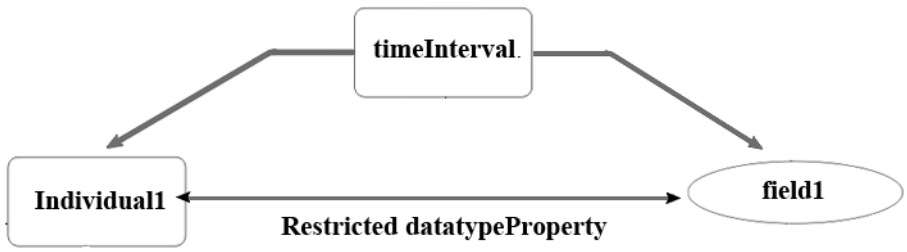

Figure 4.4: *ToqlTriple* in case of an *OWL:datatypeProperty*

4.2.1 Cardinality Constraints

The cardinality constraints is the simplest form of restriction and can be handled easily using a list of *ExtendedToqlTIs*, described in section 4.1.3. In order for a restriction to be consistent, we use the function *checkCardinalityRestrictions()*. This procedure uses an Iterator to examine all restrictions of this category. Then, for every individual of the *ToqlTriple* which is the domain of the restricted property, we increase the counter on the appropriate *ExtendedToqlTI*, examining first the overlaps between the *ToqlTI* that corresponds to the *ToqlTriple* and every *ExtendedToqlTI* that is member of the checking list.

When the checking list is empty, all counters are initialized to '-1', which means that we have no information about this individual during this period. When the filling procedure is over, we have a list that constitutes of several counters.If all of them are

greater/smaller/equal to the restrictions value but not '-1', then the min/max/exact cardinality restriction is consistent and our mechanism resets the checking list and repeats this procedure for the next individual.

As an example assume that we have an *OWL:maxCardinalityRestriction* that allows a *Company* to *produce* only 2 *Products*. During the check of individual *Company3*, our procedure ends up with the checking list of Figure 4.5. This means that for every *timeSlice* that our ontology holds information, the restriction for this specific individual is consistent.

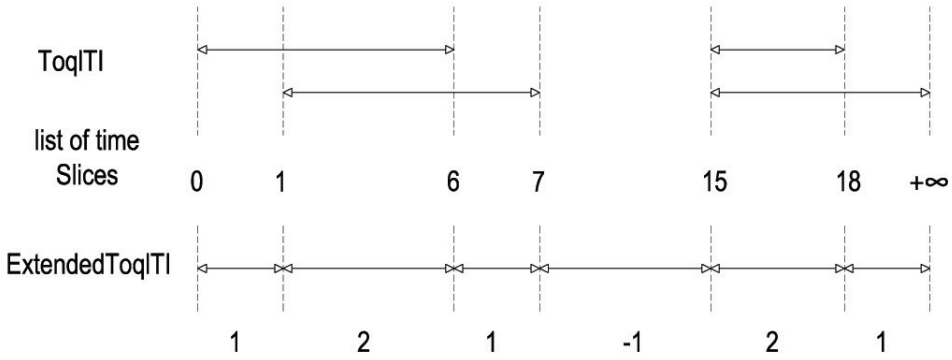

Figure 4.5: Checking List after processing the list of *ToqlTriples*

Lets now assume that we have an *OWL:minCardinalityRestriction* that allows a *Company* to *produce* at least 2 individuals of class *Product* and the generated checking list for the individual *Company3* is the same with the previous example. Then the restriction for this individual is inconsistent, because during *timeInterval* '6' to '7' *Company3 produces* only one product. Once again, the *timeInterval* '7' to '15' does not participate to the descision of the procedure.

4.2.2 Value Constraints

As described in section 3.1.1, there are three types of value constraints. *OWL: hasValueRestriction*, *OWL:allValuesFromRestriction* and *OWL:someValuesFromRestriction*. In this section we show how our mechanism handles these restrictions.

In case of an *OWL:allValuesFromRestriction*, restriction checking is performed much earlier. Although for most restrictions the mechanism remains almost the same, this case is much simpler because there is no need to check all the *timeIntervals*. The restriction

check can be performed during the loading of individuals. Lets assume that a *ComputerCompany hasEmployee* only *Technicians*. In order to be consistent, we load all members of class *ComputerCompany* and determine wether they are connected through property *hasEmployee* only with members of class *Technician*. If not, an exception is thrown. The same actions will take place if the restricted property is an *OWL:datatypeProperty*.

In case of an *OWL:someValuesFromRestriction*, our mechanism behaves almost the same way with *OWL:minCardinalityRestriction*. Min value in this case is '1' but it is not the only inspection that takes place. The individual, that is part of the triple, must be examined whether it is a member of the class that is subject to the restricted property. If these two conditions are not satisfied, an exception is thrown.

In case of an *OWL:hasValueRestriction*, our mechanism must determine whether the ontology has the essential value as a member of the object specified from the restricted property. For instace, a *Company* must always have an employee (*hasEmployee*) named 'John'. This means that for every *timeInterval* that we know that a *Company hasEmployees*, one of them must be 'John'.

4.2.3 Global Cardinality Constraints on Properties

OWL supports a limited set of constructs for defining global property cardinality, namely *OWL:FunctionalProperty* and *OWL:InverseFunctionalProperty*. Knowing that a functional property is a property that can have only one (unique) value y for each instance, we can reduce this special case to a common *OWL:maxCardinalityRestriction* with max value '1'. Listing all *OWL:FunctionalProperties* declared on the ontology and temporary storing them as *ToqlRestrictions* in the *maxCardinalityRestriction* list, allows us to handle them as common cardinalities.

The procedure to handle *OWL:InverseFunctionalProperty* is almost the same. We list all the *OWL:InverseFunctionalProperties* declared on the ontology. Then we store the inverse property in class *ToqlRestriction* as the restricted property and build the *ExtendedToqlTI* list with a small variation. This time we use the first individual as subject and the second as object. This way we can determine the consistency of the restriction using the already implemented functions.

4.3 Experimental Results

To show proof of concept and demonstrate the functionality of our proposed mechanism we implemented an example ontology illustrating all possible cases of property

Class	*Subclass/es*
Company	ComputerCompany HardwareComputerCompany SoftwareComputerCompany
Country	-
Employee	Technician Manager
ComputerCompany	HardwareComputerCompany SoftwareComputerCompany
HarwareComputerCompany	-
SoftwareComputerCompany	-
Technician	-
Manager	-

Table 4.1: Classes declared on 'DEn2.owl'

restrictions and possible violations of these restrictions. This ontology is implemented using Protege 4 [6] and therefore, follows OWL:2 semantics [3]. Classes declared on this ontology are described on table 4.1.

All Object Properties declared on 'DEn2.owl' are described in table 4.2 while Datatype Properties are described in table 4.3. These two tables provide information not only for the type, but also for the domain and range of the property as well.

Object Properties		*Domain*	*Range*
dynamic Properties	hasEmployee produces isEmployeeof	Company Company Employee	Employee Product Company
static Properties	hasStoresAt isnear	Company Company	Country Company

Table 4.2: Object Properties declared on DEn2.owl

Declared restrictions on this ontology are described in table 4.4. Examining table 4.4 and the lists of properties, we come to conclusion that not every restriction is dynamic. The static ones are handled by PELLET Reasoner [5], applied on the 'DEn2.owl' ontology during implementation, using the option provided by Protege editor [6]. The results, as expected, are correct.

Datatype Properties		*Domain*	*Range*
dynamic Properties	price	(tsTimesliceOf ONLY) Product	decimal
	productName	(tsTimesliceOf ONLY) Product	string
static Properties	site	ComputerCompany	string
	salary	Employee	decimal
	employeeName	Employee	string
	companyName	Company	string

Table 4.3: Datatype Properties declared on 'DEn2.owl'

As described above, dynamic restrictions are handled by our mechanism. We declare two types of dynamic restrictions knowing the actual results, one inconsistent and one consistent. Creating the entire ontology from the beginning allows us to choose the appropriate cases. Table 4.5. shows the full list of dynamic restrictions declared in our example ontology. The second column shows property's domain. The third column shows information concerning estimated consistency. Finally, the last column shows experimental results after applying our mechanism. We can see that experimental results match with expected ones.

Restriction	*Expected Results*	*Experimental Results*
hasEmployee *max 2* Employee	Consistent	Consistent
hasEmployee *min 4* Employee	Inconsistent	Inconsistent
Produces *max 3* Product	Inconsistent	Inconsistent
Produces *min 2* Product	Consistent	Consistent
productName *exact 1* 'String'	Consistent	Consistent
price *exact 2* 'decimal'	Inconsistent	Inconsistent
productName *hasValue* 'Amfetamine'	Inconsistent	Inconsistent
productName *hasValue* 'Aspirine'	Consistent	Consistent
Produces *hasValue* P1	Consistent	Consistent
hasStoresAt *some* Country	Consistent	Consistent
hasEmployee *allValuesFrom* Technician	Inconsistent	Inconsistent
hasEmployee *allValuesFrom* Employee	Consistent	Consistent

Table 4.4: Experimental Results

5 Conclusion and Future Work

We intoduce a mechanism for restriction checking in OWL:2 temporal ontologies,that is ontologies representing evolution of concepts in time. Extending our mechanism to perform restriction checking on ontologies that use qualitative *timeIntervals* along with the quantitative ones is an intresting issue for future work. Also, implementing our mechanism as a plug-in for Protege as well as making the method available to the interested practitioners would offer additional visibility to our work. Future work also includes extending our mechanism to support complete consistency checking on additional OWL: 2 temporal constructs such as *OWL:Restrictions*, *OWL:equivalentProperty*, *OWL:inverseOf*, *OWL:transitiveProperty* and *OWL:symmetricProperty*.

Bibliography

[1] Christopher Welty, Richard Fikes, and Selene Makarios. *A reusable ontology for fluents in owl.* IBM Research Report RC23755 (W0510- 142), October 2005. 10, 13

[2] *http://www.w3.org/TR/owl-ref* 16

[3] *http://www.w3.org/TR/owl2-overview* 31

[4] *http://jena.sourceforge.net* 24

[5] *http://clarkparsia.com/pellet* 31

[6] *http://protege.stanford.edu* 31

[7] *http://www.openrdf.org*

[8] Baratis Evdoxios *TOQL: Querying temporal information in ontologies.* Technical Report TR-TUC-ISL-02-2008. 25

[9] Evdoxios Baratis, Euripides G.M. Petrakis, Sotiris Batsakis, Nikolaos Maris and Nikolaos Papadakis, *TOQL: Temporal Ontology Querying Language*, 11th International Symposium on Spatial and Temporal Databases (SSTD 2009), July 8-10, 2009, Aalborg, Denmark 9, 15

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

[11] 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.

[12] J. F. Allen and G. Ferguson. Actions and Events in Interval Temporal Logic. *Journal of Logic and Computation*, 4(5):531–579, 1994.

[13] E. Baratis. TOQL: Querying Temporal Information in Ontologies. Master's thesis, Techn. Univ. of Crete (TUC), Dept. of Electronic and Comp. Engineering, July 2008. 15

[14] 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.

[15] D. Martin et.al. OWL-S: Semantic Markup for Web Services. W3C Recommendation, November 2004. http://www.w3.org/Submission/OWL-S.

[16] J.R. Hobbs and P. Fang. Time Ontology in OWL. W3C Recommendation, September 2006. http://www.w3.org/TR/owl-time/.

[17] 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.

[18] 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. 13

[19] N. Kline, R. T. Snodgrass, and T. Y. Cliff Leung. Aggregates. In *The TSQL2 Temporal Query Language*, pages 393–424. Kluwer, 1995.

[20] D. L. McGuinness and F. VanHarmelen. OWL Web Ontology Language Overview. W3C Recommendation, February 2004. http://www.w3.org/TR/owl-features. 13

[21] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, January 2008. http://www.w3.org/TR/rdf-sparql-query.

[22] Andy Seaborne. RDQL - A Query Language for RDF. W3C Recommendation, January 2004. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109.

[23] T. Sider. *Four-Dimensionalism: An Ontology of Persistence and Time*. Oxford University Press, USA, 2002.

[24] R. T. Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems (TODS)*, 12(2):247–298, 1987.

[25] C. Welty and R. Fikes. A Reusable Ontology for Fluents in OWL. *Fontiers in Artificial Intelligence and Applications*, 150:226–236, 2006. 14

[26] 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. 13

[27] Z. Zhang. Ontology Query Languages : A Performance Evaluation. Master's thesis, The University of Georgia, Comp. Science Dept., August 2005.

[28] G. Ozsoyglu, and R.T. Snodgrass. Temporal and Real-Time Databases: A Survey. Knowledge and Data Engineering 4 (1995) 513532. 12

[29] H. Gregersen,and C.S. Jensen. C.S.: Temporal Entity Relationship Models A Survey. IEEE Transactions on Knowledge and Data Engineering 3 (1999) 464497. 12

[30] A. Artale, and E. Franconi. A survey of temporal extensions of description logics. Annals of Mathematics and Artificial Intelligence, 30(1-4), 2001. 12

[31] C. Lutz,F. Wolter, and M. Zakharyaschev. Temporal description logics: A survey. In Proc. TIME08, IEEE Press, 2008. 12

[32] 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.