a tool for temporary reasoning with pellet in rdf/xml based on owl

by

KOURTIKAKIS EMMANOUIL

B.A., Computer Science Departement, University of Crete, 2006

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

DEPARTMENT OF APPLIED INFORMATICS
AND MULTIMEDIA

SCHOOL OF APPLIED TECHNOLOGY

TECHNOLOGICAL EDUCATIONAL INSTITUTE OF CRETE

2015

Approved by:

Major Professor
Papadakis Nikolaos

Abstract

In this paper, we describe the implementation of a solution for the ramification problem in the setting of OWL data based on RDF/XML. The ramification problem is a hard and ever existing problem and it arises in robotics, databases, and in general in all systems exhibiting a dynamic behavior. In this essay, we also present an algorithm that we derived from the solution of a prior work that is presented in the literature [27]. Our project is based on this algorithm's theoretical solution, which is meant to produce the appropriate rules in a OWL schema in order to address the ramification problem. The purpose of this study is the development of a tool in Java that is connected onto any ontology of am OWL database and provides new rules back to the schema. These new rules, which are produced by our algorithm [28], update the time intervals that limit each object's actions and its participation in the relational rules. Thus, our code affects the functions of the reasoner in any database schema that is added on and specifically our program is developed in order to update the Pellet reasoner's rules of an OWL database schema based on RDF/XML. At this point, we should highlight the proper use of reasoning and knowledge representation that we used in order to describe and manipulate the ontology's data. The tool that we designed is in charge to export, from a given database, the classes and the individuals in order to build the new rules based on the time slots. The algorithm produces the new property data for the individuals and our program proceeds with the import and the update of this information into the schema again. The final step in our project, in terms of evaluating our code, is the creation of a database in the protégé tool and then the description of a motivational example.

Keywords: Ramification problem Implementation, Knowledge representation and reasoning, OWL file, Software engineering, Pellet query engine.

Table of Contents

1. Introduction 6
2. Technical Preliminaries 13
Knowledge representation and reasoning 13
Static Rules 15
DESIGN 25
Our tools 31
Eclipse 31
Protégé 32
Our Programing Architecture – Design 32
Classes 34
Static Rules 34
Variables 36
Sections 38
Properties of Classes 40
Static Rules 40
Arrays 40
Methods 42
Fluents 51
Sections 53
Motivational Example in details 55
3. CONCLUSIONS AND FUTURE WORK 65
4. REFERENCES 66

List of Figures

Figure 1. Example 8
Figure 2.1 Libraries 14
Figure 3. Variables 36
Figure 4. Fluent class 37
Figure 5. sections class 39
Figure 6. mainPellet class 40
Figure 7. Methods 48
Figure 8. Fifth step Methods 50
Figure 9. motivational example 1 database 56
Figure 10. motivational example export print 57
Figure 11. motivational example export print 2 57
Figure 12. motivational example export code 58
Figure 13. motivational example export code 2 58
Figure 14 motivational example create instances 59
Figure 15. motivational example print table Fk 60
Figure 16. motivational example final Gi union 60
Figure 17. motivational example insert trurtle 61
Figure 18. motivational example turtle call 61
Figure 19. motivational example database before update 62
Figure 20. motivational example database after update 63
Figure 21. motivational example before update 2 63
Figure 22. motivational example after update 2 64
Figure 23. motivational example before update 3 64
Figure 24. motivational example after update 3 65

// Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. Papadakis for the continuous support of my Msc study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

Besides my advisor, I would like to thank the rest of my thesis committee, for their insightful comments and encouragement, but also for the hard question which incented me to widen my research from various perspectives. I thank my fellow labmates Popi Kanaki, Harris Kondilakis and Yannis Roussakis, for enlightening me and our project.

Last but not the least, I would like to thank my family: my parents, my granny and to my brothers John, Larry, Statham, Guti, Albert, Mononome, Fleck, Fiogkakis for supporting me spiritually throughout writing this thesis.

1. Introduction

The maintenance of the consistency of a database and its data stored in it is a matter of great importance for the designers. In order to guarantee the consistency of data, our data base has to satisfy the integrity constraints in each and every of its states (situations), which is a difficult problem. A database state is considered consistent (thus valid) when every of its integrity constraints are fulfilled. When a transaction occurs, the direct effects create a new situation in the database, which might be inconsistent because some integrity constraints are not satisfied by means of indirect effects of transactions. It is really hard for a designer to foresee all the indirect effects that could arise in a data base system of thousands of transactions and integrity constraints. Thus, it is necessary to produce each and every of the effects (direct and indirect) in order to determine the satisfaction of the integrity constraints [17] . We developed a tool that will facilitate this procedure by producing all the direct and indirect effects.
This kind of problems which are related with the indirect consequences of some actions are called **ramification problems.** The ramification problem in computer science is a well-known one that can be found in AI. Thus, It is a hard and ever existing problem and it arises in robotics, databases , and in general in all systems exhibiting a dynamic behavior. Databases are considered dynamic worlds because they represent a changing world. As we mentioned before, the transactions of an ontology occur changes which are applied in the data and its values of the system. The consequent direct effects can be controlled by the presence of the integridy constraints, but we have to control the indirect effects too.

In order to describe further the ramification problem, we must first outline the struct and the way a **Database is composed**. Ontologies are consisted of functional fluents and fluents, also called as functions and predicates respectively, whose values are either false or true depending the state of the system. Each evolution of these fluents is a sequence of actions direct and indirect and we call it a situation (anaphora). Different situation contain different values (true or false) for every fluent of the ontology. The Database changes its situation when a transaction occurs. The initial situation at which no transaction has taken place yet is called the initial situation (S0). A function

act(α,S1) yields the new situation. An upcoming situation S1 that s from the execution of action α [17] . As we pinpointed earlier, also the functional fluents have values that are situation-dependent.

The ramification problem refers to the concise description of the indirect effects of an action in the presence of constraints. A Solution of the specific problem that we came by is: the *minimal change approach* [38] where for any an action that occurs in a situation S, we try to find the consistent situation S' which has the fewer changes from S [1] .

Another solution is the **categorizing fluents** method [14] that suggests each and every of the fluents to be separated in two groups. One group contains the fluents that can change only as a direct effect of an action. The other group contains the fluents, which can be changed as direct or indirect effect of an action. Those groups of fluents are called "*primary*" and "secondary" respectively. This algorithm, though, solves the ramification problem effectively only in the occasion that all the fluents can be categorized in one of those two groups. Consequently, a solution is adequate, when there are fluents in our database that for some actions are *secondary* and for a group of different actions are categorized as *primary*.

On the contrary, both of the solutions that we presented are quite simple and ineffective in some occasions. The most effective solutions are based on *causal relationships* [8] . The casual relationships motive is shown above:

$$\varepsilon \text{ causes } \rho \text{ if } \Phi$$

Where ε is an action, ρ is the indirect effect and Φ is the context, i.e., a set of fluents that describes the conditions under which the execution of ε leads to ρ.

In order to give a more comprehensive explanation, we introduce the problem by means of an example. Assume we are interested in creating a database that describes a real simple circuit, which includes only two switches and one lamp (figure 1 (A)). The circuit's behavior is

described by a set C comprising the integrity constraints as follows. First, in case that the two switches are up, the lamp should be lit. Second, the lamp must not be lit if one of the two switches is down. The integrity constraints are described as the following formulae employing predicates *up* and *light*: $C = \{up(s_1) \wedge up(s_2) \text{ -> } light, \text{ -}up(s_1) \text{ -> } \text{-}light, \text{-}up(s_2) \text{ -> } \text{-}light\}$.

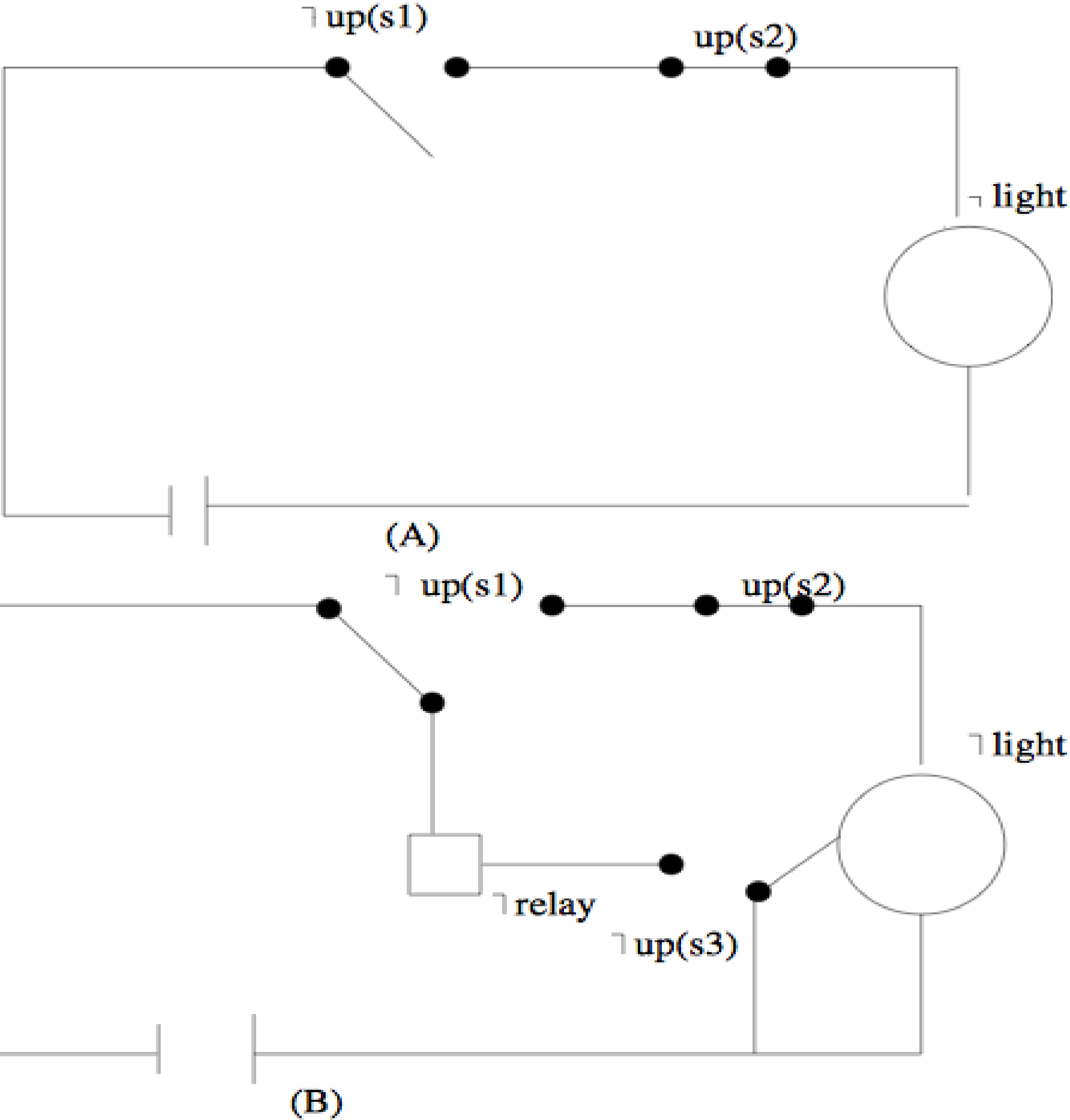


Figure 1. Example

Action *toggle_switch* changes the state of a switch as the following set E of propositions describing the direct effect of the action specify:

E = {toggle_switch(s) > up(s) if –up(s) toggle_switch(s) > -up(s) if up(s) }

We call a situation as consistent when all the integrity constraints are satisfied. Assume, for instance, that the circuit is in situation S = {$-up(s_1)$, $up(s_2)$, *-light* }, then the situation S is called consistent. The next step is to execute the action $toggle_switch(s_1)$. By this action, we have a direct effect that changes the state of switch s_1 from $-up(s_1)$ to $up(s_1)$ leading to the situation S_1 = {$up(s_1)$, $up(s_2)$, *-light* }. Consequently, the new situation is inconsistent. Situation S_1 violates our first integrity constraint. The reasonable conclusion, that we expected, is that the lamp should be lit. So, the final situation is S_2 = {$up(s_1)$, $up(s_2)$, *light* }. The action $toggle_switch(s_1)$ has as an indirect effect the change of the condition of the lamp. Notice that the presence of integrity constraints cause the indirect effects in our example. The ramification problem refers to the concise description of the indirect effects of an action in the presence of constraints [17].

In our example, we can pinpoint four of the *casual relationships* that we introduced earlier as a solution to the ramification problem.

$toggle_switch(s_1)$ *causes light if* $-up(s_1) \wedge up(s_2)$

$toggle_switch(s_2)$ *causes light if* $-up(s_2) \wedge up(s_1)$

$toggle_switch(s_1)$ *causes –light if* $up(s_1)$

$toggle_switch(s_2)$ *causes –light if* $up(s_2)$

As we can highlight from the above casual relationships of the example, the change of fluent f's truth value potentially affects the truth- value of some other fluents, while it does not affect others. All the proposed solutions determine the direct and indirect effects of an action that refer to the next consistent situation. We define a binary relation I between fluents as follows: if (f, f') $\in$ I then a change in fluent f's value may affect the value of f'. Reviewing the example we given earlier we find that the (up(s1), light) $\in$ I , whereas (up(s1) , up(s2)) $\in$ I. We have casual relationships that are defined for pairs of fluents belonging in I only. In other words, in our case

shown above with the single switch, there is a casual relationship connecting the toggle_switch(s1) and the light, but on the contrary the up(s2) and the toggle_switch(s1) have no such relationship at all. This problem was solved by the algorithm developed by N. Papadakis and D. Plexousakis and described in «Actions with Duration and Constraints: the Ramification Problem in Temporal Databases».

The development and maintenance of ontologies is a high importance subject in modern computer science. The need of ontologies and, in general, of a more structed way to present the information appeared in web documents. The problem we faced at first was that we were not able to distinguish between the information content and the presentation. The first obstacle was due to the HTML language, which was "solved" by the current technology style sheets, known as XML language too. XML stands for Extensible Markup Language. The style sheets allow for formatting and separating attributes from the information that we present. The user who writes the XML documents, also defines a set of tags. These tags are used to describe and express the "semantics" of the various different kinds of information and, also, have a set of rules in order to encode documents in machine-readable form. XML's design goals emphasize upon simplicity, generality, and usability over the Internet. It is a textual data format, with strong support via Unicode for the languages of the world. Although XML's design focuses on documents, it is widely used for the representation of arbitrary data structures, for example in web services [2].

Another obstruction the semantic scientists faced was the case of using the same tag or definition with a different meaning. We needed a common framework to share data on the Internet across application boundaries and "a formal representation of the knowledge by a set of concepts within a domain and the relationships between those concepts" [4] **Ontologies** provide a vocabulary of terms and a vocabulary for annotations. Thus, ontologies allow us to organize, specify the meaning of annotations and navigate easily in the web. Combining existing terms can form new ones and in every term the meaning (semantics) can be formally specified. We can even specify the relationships between those terms different ontologies.

In order to provide a more flexible mechanism for our problem, an alternative language has been developed, the **RDF** that stands for Resource Description Framework. It is a simple way of describing the rules of language: its underlying data structure is a directed graph and the only syntactic form is the *triple* one. This syntactic form consists of three components, as its name

suggests. Those components are the subject, the predicate and the object and they are represented as a single edge between the two nodes, the subject and the object respectively. In other words, it describes a binary relationship that connects the subject and the object via the predicate. For instance, we can describe the connection between Frodo and the ring using the triple form of: Frodo has Ring. RDF may have syntax similar to the XML's one, but on the contrary it is not based at all the XML its own. RDF schema is based on a hierarchy of classes and subclasses and a hierarchy of properties, defining fields and values for those properties [12]. Nevertheless, this language is real simple and a new one more expressive was in need, so the DAML + OIL language took place, which is much richer.

While RDF is used to define the structure of the data, we use OWL in order to describe the semantic relationships in a normal programming way, such as a C struct. There are three different species of OWL. The OWL full which is an OWL syntax and RDF combination. Second is the OWL DL, which is restricted to FOL (first-order logic) fragment, or in other words the DAML + OIL language. At this point, we have to mention also the use of the OWL lite, which is "simpler" subset of OWL DL and the OWL DL based on SHIQ Description Logic, that is, in fact, equivalent to the SHOIN DL [28]. Among the benefits we gain from the use of OWL DL are the well defined semantics, the reasoning algorithms, the implemented highly optimized systems and the formal well defined properties, such as the complexity and the decidability.

Taking under consideration, the importance of the ramification problem and the significance of the semantics in contemporary science, **we tried to implement** an application based on an algorithm by N. Papadakis, D. Plexousakis and Y. Christodoulou and described in « The ramification problem in temporal databases: a solution implemented in SQL». In this paper, a review of the previous work and the technical preliminaries are presented in details. The combination of those with the use of a new time-factor, conclude to a five-step algorithm giving the solution for the ramification obstacle. It is a time interval intersection-computing approach and its structure is described thoroughly in terms of pseudo-code and Java but it is set for PS/SQL databases. Nevertheless, we decided to implement this solution in Java and use the OWL data base with the Pellet reasoner. Pellet is supposed to be the reasoner most used in the development of new ideas.

The implementation of this theoretical approach is a high importance matter for the manipulation of the data in databases and in artificial intelligence. The creation of a real robust program was a future goal of the algorithm described in the N. Papadakis, D. Plexousakis and Y. Christodoulou and described in « The ramification problem in temporal databases: a solution implemented in SQL», which is also an extension of McCain's and Truner's work. We **developed a system of code automatic production,** which can be applied on any OWL database based on RDF/XML. Our program is based on recovery and timeslot information and its target is to design actual time transactions. When a database renewal (query of deletion or insertion for instance) takes place, the database user has to run, also, the main procedure. This procedure will trigger the application to take over the implementation and checks for the integrity of database as well as the input of the necessary elements and properties in it. Our implementation, in fact is a reasoner. It derives information from the Data base file, such as properties, relationships, classes and objects it processes them and updates the Data base schema with the new ones. In this way, the program guarantees the consistency of the database, thus it eliminates the ramification problem, often appeared in many schemas.

The application we implemented is an add-on extension to an existing OWL database system as we described above. Thus, the way our approach improves the running time of the transactions is by applying slight changes, or even adding new properties, to the entities of a database. At first our program collects and derives information form the existing OWL system. Then we process those data and our program generates the time intervals that each object can take place in a query. These kind of information, support and facilitate our procedure to avoid the appearance of the ramification problem. In order to import these information in each and every of the database objects we need to create a new property for them under the name "timeslot" and save the outcome of our algorithm in it. To summarize, the contribution of our program is to implement the theoretical solution of the ramification problem, by importing new time interval information in an already existing OWL database system, in order to prevent the use of two different objects, that could cause the ramification dead-end, in the same timeslot.

2. Technical Preliminaries

Knowledge representation and reasoning

In this section we are going to present in details the way we represent the knowledge of our database schema in our integrated program. The representation of the information, or in other words the reasoning, is a very crucial point of our implementation, because it also affects the way we interpret, edit and use our data. This step can make our code even more robust and efficient. As represented Knowledge we mean all the concepts, definitions, relationships and descriptions of our schema. Reasoning is quite similar, but it applies only in the relationships and the answers of questions.

Our first step, as far as it concerns the knowledge representation, is the derivation of each and every of the object of our database. We use the existing Java libraries created for reasoning the OWL databases, either for Jena or Pellet. Trying to brief, we import the database file named “myOwlDatabase.owl” ,for example, in our function, which is mining the information. We, then, use an OWL manager in this function which loads the ontology from the file. Then we create an OWLDataFactory and we get all the ontologies and the classes with their properties from the OWL manager.

```
import com.clarkparsia.owlapiv3.OWL;
import org.semanticweb.owlapi.apibinding.OWLManager;
import org.semanticweb.owlapi.io.*;
import org.semanticweb.owlapi.model.*;
import org.semanticweb.owlapi.util.SimpleIRIMapper;
import org.semanticweb.owlapi.reasoner.OWLReasonerFactory;
import org.semanticweb.owlapi.util.DefaultPrefixManager;
import org.semanticweb.owlapi.reasoner.*;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.net.URI;
import java.net.UnknownHostException;
import java.util.Map;

//import org.semanticweb.owl.inference.OWLReasonerFactory;
import org.semanticweb.owlapi.vocab.PrefixOWLOntologyFormat;
import com.clarkparsia.pellet.owlapiv3.PelletReasonerFactory;
import org.semanticweb.owlapi.io.OWLObjectRenderer;
import uk.ac.manchester.cs.owl.owlapi.OWLNamedIndividualImpl;
import uk.ac.manchester.cs.owlapi.dlsyntax.DLSyntaxObjectRendere
import java.util.Iterator;
import java.util.Set;
```

Figure 2.1 Libraries

The second step is about organizing the information. This action is also a part of the algorithm we are based on for our implementation. We develop a double array, the raws are fluents from our schema, that means they are connected using disjunctions with each other. The columns means that the raws are connected to each other using conjunctions. This is the way we transform the integrity constraint in its CNF form.

$C_1 \wedge C2 \wedge C3 \cdots \wedge Cn$, where each Ci is a disjunction of all fluents.

The next steps of the algorithm, which are described in details in the next section, are transforming the CNF information in two similar two-dimension tables. The elements of these Tables are, in fact, a union of the timeslots that each fluent is allowed to be executed. The first table indicates the timeslot for the fluents when their value is true, f, while the second one is with the values of the false fluents, ¬f. This information, in fact, is going to be a new property

insertion in our OWL database ontology, in order to change the reasoner's actions and rules. The reasoner that we use in our OWL database is the Pellet.

Our last method, inserts in the OWL ontology file "myOwlDatabase.owl" the results of our algorithm. For the insertion of the information, new properties are created in the objects of the ontology, in the same file that we used to export fluents for the rules of our program. Then we derive the results from the resulting tables that we mentioned in the previous paragraph and we import them in the respective property. We must highlight that for each object three new properties are created, in order to describe the triplet and the rule. First property is triplet ID, which describes the ID - timeslot of the triplet. The second one is the lower time bound and the third one is the higher time bound, giving an exact description on the time interval when the object – fluent is allowed to be used. We used once more many libraries of the Java code, creating again a manager that loads and edits the ontology with an incoming string we integrate dynamically.

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;


import org.apache.jena.iri.impl.Main;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.shared.Lock;
import com.hp.hpl.jena.tdb.TDB;
import com.hp.hpl.jena.update.UpdateAction;
import com.hp.hpl.jena.update.UpdateFactory;
import com.hp.hpl.jena.update.UpdateRequest;
import com.hp.hpl.jena.util.FileManager;
import com.hp.hpl.jena.util.FileUtils;
```

Figure 2.2 Libraries 2

Static Rules

The indirect effects of the execution of each action that we described above are encapsulated at the **static rules**. The presence of integrity constraints causes in some occasions the indirect effects. Hence, A closer look at the data indicates that it is reasonable to produce the static rules from the integrity constraint. We present the procedure of the static rules production as follows:

The underlying argument in favor of the ideas of McCain and Turner (McCain N, Turner H (1995) A causal theory of ramifications and qualifications. In: Proceedings of IJCAI-95 pp 1978–1984) , where our work is mainly based on, who propose the production of the static rules in order to capture and evaluate the indirect effects of the database actions (based on the integrity constraints of the particular domain), and dynamic rules to represent the direct effects of actions. In our approach, for each action A there is a dynamic rule of the form A → Fi (L′i), where each Fi(L′i) is fi(L′i) or ¬fi(L′i) for a given fluent f. On logical grounds, there is no compelling reason to argue that he above rules describe the direct effects of an action. In addition, the available evidence seems to suggest that for each fluent f, we define two rules, G(L) → f(L) and B(L) → ¬f(L). The G(L) is a fluent formula which, when true (at list L), causes fluent f to become true at the time intervals which are contained in the list L (respectively for B(L)). These rules encapsulate the indirect effects of an action. The fact that the former rules are evaluated after the execution of an action is giving them also the dynamic characterization.

Assume an **example** with a public worker in order to understand the static and dynamic rules, how they are produced and the direct and indirect effects. Consider that a public employee commits a misdemeanor, then for the next five months he/she is considered illegal, unless he/she receives a pardon. In the case that a public employee is illegal, then he/she must be suspended and cannot be promoted for the time interval over which he/she is considered illegal. Moreover, if a public employee is suspended, he/she cannot receive a salary until the end of the suspension period. Each and every public employee is evaluated for his/her work. In the case that he/she receives a bad evaluation then he/she is considered to be a bad employee and he/she cannot receive his/her salary. If he/she receives a good evaluation then he/she is considered to be a good employee and he/she may receive a bonus if he/she is not suspended. Every public employee receives a promotion and an increase every five and two years, respectively, only if he/she is not illegal.

The data appears to suggest that we can identify six actions : misdemeanor, receive_ pardon,

good_grade, bad_grade, take_promotion and take_ increasem, and seven fluents : good_employee, illegal, receive_salary, receive_bonus, position, suspended and salary. The fluent position (p,l,t1) means that the public worker is in position l for the last t1 months while salary(p, s, t1) means that the public worker has been receiving salary s for the last t1 months. The direct effects of the six actions are expressed in propositional form by the following rules:

occur(misdemeanor(p),t) ∧ t1 ∈ [t,t +5] →illegal(p,t1) (1)

occur(receive_pardon(p),t) ∧t1 ∈ [t,∞) → ¬illegal(p, t1) (2)

occur(bad_grade(p),t) ∧t1 ∈ [t,∞) → ¬good_employee(p, t1) (3)

occur(good_grade(p),t ∧t1 ∈ [t,∞) → good_employee(p, t1) (4)

occur(take_increase(p),t) ∧ salary(p,s,24) → salary(p, s + 1, 0) (5)

occur(take_promotion(p),t) ∧position(p,l,60) → position(p, l + 1, 0) (6)

where *t* is a temporal variable and the predicate occur (misdemeanor(p), t) denotes that the action misdemeanor(p) is executed at time t. The former four rules are dynamic and they are executed every time that the corresponding actions take place. The remaining two rules are also dynamic and are executed periodically because the corresponding actions take place in a periodic manner. The preconditions of the actions take_increase(p) and take_promotion(p) are

Poss(take_increase(p),t) ≡salary(p,s,24) ∧¬illegal(p,t) ∧ good_employee(p)

Poss(take_promotion(p), t) ≡position(p,l,60) ∧¬illegal(p,t)

∧ good_employee(p, t)

We also consider the following integrity constraints which give rise to indirect effects of the six actions:

illegal(p, t1) ∧suspended(p, t1) (7)

suspended(p, t1) ∧¬receive_salary(p, t1) (8)

¬suspended(p,t) ∧ good_employee(p,t) ∧ receive_bonus(p, t) (9)

¬good_employee(p, t1) ∧ ¬receive_bonus(p, t1) (10)

¬suspended(p, t1) ∧receive_salary(p, t1) (11)

good_employee(p, t) ∧ receive_salary(p, t) (12)

On these grounds, we can argue that the rules (7–12) are static and they are executed every time. This happens because the effects of the actions hold for a specific time interval (e.g. the effect illegal of the action misdemeanor holds out for 5 time points after the execution of the action). After the end of the time intervals, the effects automatically terminate. The aim of our next sections is to generalize beyond the data and describe how we extended the solution of McCain and Turner (McCain N, Turner H (1995) A causal theory of ramifications and qualifications. In: Proceedings of IJCAI-95 pp 1978–1984) in order to address the problem in temporal databases.

Taking under consideration the example we presented, it suggest that we have the following dynamic rules:

occur(*misdemeanor*(p),t) → *illegal* (p,[[t,t + 5]])) *occur*(*receive_pardon*(p),t) → ¬*illegal* (p,[[t,∞]]) *occur*(*bad_grade*(p),t) → ¬*good_employee* (p,[[t,∞]]) *occur*(good_grade(p),t) → *good_employee*(p,[[t,∞]])

Assume that the initial situation is:

S_0 ={¬*receive_bonus*(p,[[0,∞]]), *receive_salary*(p, [[0, ∞]]), ¬*suspended*(p, [[0, ∞]]), ¬*good_employee*(p, [[0, ∞]]), ¬*illegal*(p, [[0, ∞]])}

Assume that the following actions occur at the following time points, assuming that time starts at 0 and time granularity is that of months. Now assume the execution of the action

occur(*misdemeanor*(p), 2)

The new situation is S_1 ={¬*receive_bonus*(p,[[0,∞]]),

receive_salary(p, [[0, ∞]]), ¬*suspended*(p, [[0, ∞]]), ¬*good_employee*(p, [[0, ∞]]), *illegal*(p, [[2, 7]]), ¬*illegal*(p, [[7, ∞]])}

The following integrity constraint is not satisfied in S_1 (at time points in [2, 7]):

illegal(p, t1) → *suspended*(p, t1)

However, there is a set of static rules, which ensure the integrity constraint. We describe the set of static rules that is produced by using the algorithm described In the forthcoming section. Among them there is the static rule

illegal(p, [[2, 7]]) → *suspended*(p, [[2, 7]])

which is executable. After the execution, the situation changes to include the following information:

suspended(p, [[2, 7]]), ¬*suspended*(p, [[7, ∞]]) .

Meanwhile, in the new situation the following integrity constraint is not satisfied:

suspended(p, t1) → ¬*receive_salary*(p, t1).

On the other hand, due to the static rule :

suspended(p, [[2, 7]] → ¬*receive_salary*(p, [[2, 7]])

and as a result the final situation is :

S_2 ={¬*receive_bonus*(p,[[0,∞]]),

¬*receive_salary*(p, [[2, 7]]),

receive_salary(p, [[7, ∞]]),

suspended(p, [[2, 7]]),

¬*suspended*(p, [[7, ∞]]),

¬*good_employee*(p, [[0, ∞]]),

illegal(p, [[2, 7]]),

¬*illegal*(p, [[7, ∞]])}

A corner stone of our work is the **production of the static** rules from integrity constraints, according to the above ideas. We make use of a binary relation I, which is produced from the integrity constraints and encodes the dependences between fluents. The binary relation I is based on the idea that there are two kinds of integrity constraints,

(a) $G_f \rightarrow Kf$ and (b) $Gf \equiv Kf$,

where G_f and Kf are fluent propositions. A closer look at the rules and situations, indicates that the difference between the two is that, for the second type, when ¬Gf holds then ¬Kf also holds, whereas this is not necessarily the case for the first one. For the first type of constraints, for each $f \in Gf$ and $f' \in Kf$, the pair $(f,f')$is added to I. For the second, for each $f \in Gf$, $f' \in Kf$, if f can change its true value as the direct effect of an action, then (f,f') is added to I . If f' can change its true value as direct effect of an action, then (f', f) is added to I.

Theorem 1 *Let* A→B *be a constraint and* $C1 \wedge \cdots \wedge Cn$ *its CNF form. The algorithm produces* I *in such a way that, for each* Ci ,*'there is at least one pair* (f1, f2) ∈ I *and* $Ci = f1 \vee f2 \vee Ci$.

The **algorithm** used for the production of the static rules is the following.

Algorithm 1

1. Transform each integrity constraint in its CNF form. Now each integrity constraint has the form $C1 \wedge C2 \wedge C3 \cdots \wedge Cn$, where each Ci is a disjunction of all fluents.

2. Set R = {*False* → f, *False* → ¬f : *for each fluent* f }

3. For each I from 1 to n do: assume C_i=f1∨···∨fm For each j from 1 to m do For each k from 1 to m, and k≠ j , do if (fj,fk) ∈ I then R=R∪(¬fj *causes* fk if ¬fl) ,l≠j,k.

4. For each fluent f_k the rules in R have the following form

$\wedge$ fi *causes* fk *if* Φ, $\wedge$ fi′ *causes* ¬fk *if* Φ′.

The static rules change from G→f_k, K→¬fk

to (G∨(∧ fi ∧ Φ))→fk, (K∨(∧ fi ∧ Φ’))→¬fk.

5. At time moment t, for each static rule G^S(t,t1) → f do

(a) let G^S=G1∨···∨Gs_n

(b) For each j from 1 to s_n do

i. let G_j = f1([..]) ∧ .... ∧ fn([..])

A. for each fluent f_i(L) (s.t fi(L)∈Gj) take

the,first element [t′,t″] of the list L.

B. if t > t then G is false and terminates.

C. else t_i=t″−t and remove [t′,t″] from L.

ii. let t_{min} =min(t1,...,tsn)

iii. replace Gi with Gi(t,t + t*min*)

On the basis of the first 4 steps we currently mentioned, it seems fair to notice that they are static and they are executed once in the beginning only. The 5th step is executed in each time point

where the static rule must be evaluated at. This is due to the formula Gfp (t , L′) and may be true, depending the different values of t and L′.

Consider the example with the public worker. The above algorithm works as follows:

The transformation of the integrity constraints into their CNF form yields

¬*illegal*(p, t1) ∨ *suspended*(p, t1)

¬*suspended*(p, t1) ∨ *receive_salary*(p, t1)

¬*good_employee*(p, t1) ∨ *receive_salary*(p, t1)

suspended(p, t1) ∨ ¬*good_employee*(p, t2)

∨*receive_bonus*(p, t3)

good_employee(p, t1) ∨ ¬*receive_bonus*(p, t1)

suspended(p, t1) ∨ *receive_salary*(p, t1)

This is step 1 of the algorithm. In step 2 we have the following:

R = {*illegal*(p, t1) *causes suspended*(p, t1) if T ,

suspended(p, t1) *causes* ¬*receive_salary*(p, t1) if T ,

¬*good_employee*(p, t1) *causes* ¬*receive_salary*(p,t1) if T,

good_employee(p, t1) *causes receive_bonus*(p, t1) *if* ¬*suspended*(p, t1),

¬*suspended*(p, t1) *causes receive_bonus*(p, t1) *if good_employee*(p, t1),

¬*good_employee*(p, t1) *causes* ¬*receive_bonus*(p,t1)*if* T,

¬*suspended*(p, t1) *causes receive_salary*(p, t1) *if* T }

In step 3 we estimate the causal relationships based on the binary relationship I . In step 4 we have the following:

R = {*illegal*(p, t1) → *suspended*(p, t1),

¬*good_employee*(p, t1) ∨ *suspended*(p, t1) → ¬*receive_salary*(p, t1),

¬*suspended*(p, t1) ∧ *good_employee*(p, t1) → *receive_bonus*(p, t1),

¬*good_employee*(p, t1) → ¬*receive_bonus*(p, t1),

¬*suspended*(p, t1) → *receive_salary*(p, t1)}

In step 4, for each fluent, we construct the formula that makes the fluent true. Notice that, though, there are many causal relationships that affect the same fluents, we integrate these causal relationships into this step. On these grounds, we can argue that in the final step, step 5, we get the rules:

R = {*illegal*(p, L) → *suspended*(p, L),

suspended(p, L) → ¬*receive_salary*(p, L),

¬*suspended*(p, L1) ∧ *good_employee*(p, L2)→ *receive_bonus*(p, L1 ∪ L),

¬*good_employee*(p, L) → ¬*receive_bonus*(p, L),

¬*suspended*(p, L) → *receive_salary*(p, L)}

As we observe, the production of static rules is based on the binary relation I. There is a rapidly growing literature on static and dynamic rules, which indicates the idea that, when Ci is false, there must be at least one static rule that is executable. For this to happen, there must be at least one pair (fj,fw) ∈ I so as $C_i = f_j \vee f_w \vee C_i'$.

In previous documentations that we studied (Papadakis N, PlexousakisD (2002) Action with Duration and constraints: the ramification problem in temporal databases. In: 14th IEEE ICTAI, Washington, DC) we used an algorithm for the evaluation of static and dynamic rules for the sequential execution of actions when the effects of actions refer only to the future.

Algorithm 2

1. After the execution of an action the dynamic rule, referring to this action, is evaluated.

2. Evaluate all static rules until no change occurs.

When a static rule G(t , L) → f (L) evaluates the element [t,t′], then this element is added to the list L and removed from the L′, where ¬f (L′). Another key assumption is that the set of the integrity constraints in the domain can be satisfied. There seems to be no compelling reason to argue that with previous work (Nikos Papadakis · Dimitris Plexousakis · Yannis Christodolou (2012) The ramification problem in temporal databases: a solution implemented in SQL) that shows the following theorems, which have been proved.

Theorem 2 When a static rule is executable at least one integrity constraint is violated.

Theorem 3 Consider a set of static rules G. Then if for a fluent f1 there is a sequence of

$G_{f2}(...) \rightarrow f2(...)$ where $Gf2 \equiv f1 \vee G'f2$

$G_{f3}(...) \rightarrow f3(...)$ where $Gf3 \equiv f2 \vee G'f3$

...

$G_{fn}(...) \rightarrow fn(...)$ where $G_{fn} \equiv f_{n-1} \vee G'_{fn}$

$G_{\neg f1}(...) \rightarrow \neg f1(...)$ where $G_{\neg f1} \equiv f_{n} \vee G'_{\neg f1}$

$G_{fn+1}(...) \rightarrow fn+1(...)$ where . $G_{fn+1} \equiv \neg f_{1} \vee G'_{fn+1}$

…

$G_{fn+m}(...) \rightarrow fn+m(...)$ *where* $G_{fn+m} \equiv f_{n+m-1} \vee G'_{fn+m}$

$G_{f1}(...) \rightarrow f1(...)$ *where* $Gf1 \equiv fn{+}m \vee G'f1$

then the set of rules is not satisfied.

DESIGN

In this section, we will proceed in the explanation of the technics and methods we used on how we designed and developed our project. At first we will present you the pseudo-code we were based on, in order to build our program. The code consists of the parts analyzed here, with a sample code as a paradigm. Lets assume we have a table named X with 3 fields, KEY, START and END. Then, for instance, we assume we have only one integrity rule for the database and is the following: $1^{\wedge}2 \wedge 3 \rightarrow 4$.

```
CREATE GLOBAL TEMPORARY TABLE
  X_temporary_table AS
(

SELECT KEY, START, END
FROM X
);
DELETE FROM X_temporary_table;
```

In the first step, we build a table titled X_*temporary_table* which is will be used to store the temporary results between the calculation of a section of more than one fluents.

```
CREATE GLOBAL TEMPORARY TABLE
   X_id_temp_table AS
(

SELECT KEY

FROM X );

DELETE FROM X_id_temp_table;
```

In the second step, we create a table to store the keys involved in the left side of a given rule, if the rule has more than 2 keys in the left side. WE assign to this table the title X_id_temp_table which contains only one field of type same to the key field of the given table.

```
CREATE OR REPLACE PROCEDURE proc0 AS
TYPE cur_t IS REF CURSOR;
cursor1 cur_t;
cursor2 cur_t;
CURSOR id_cursor IS SELECT KEY
from X_id_temp_table;
row1     X%ROWTYPE;

row2     X%ROWTYPE;

s1       X.START%TYPE;

s2       X.START%TYPE;

e1       X.END%TYPE;


e2       X.END%TYPE;
```

```
resSt   X.START%TYPE;
resEnd  X.END%TYPE;
currTempTableId X.KEY%TYPE := 1;
currTableId   X.KEY%TYPE := 2;
changes  BOOLEAN;


BEGIN

INSERT INTO X_id_temp_table
SELECT KEY FROM X WHERE KEY = 3;


OPEN cursor1 FOR SELECT * FROM X
   WHERE KEY = currTempTableId ;

LOOP
FETCH cursor1 INTO row1;
EXIT WHEN cursor1%NOTFOUND;
s1 := row1.START;
e1 := row1.END;
OPEN cursor2 FOR SELECT * FROM X WHERE KEY = currTableId ;
LOOP
        FETCH cursor2 INTO row2;
        EXIT WHEN cursor2%NOTFOUND;
        s2 := row2.START;
        e2 := row2.END;
        changes := FALSE;
```

At the next step, after the word BEGIN, is the main part of the comparing algorithm.

```
IFs1< s2ANDe1>=s2ANDe1<=e2THEN resSt := s2;

resEnd := e1;

            changes := TRUE;
        ELSIF s1 >= s2 AND s1 <= e2 AND e1 >= s2 AND e1 <= e2 THEN
            resSt := s1;
            resEnd := e1;
            changes := TRUE;
        ELSIF s1 <= s2 AND e1 >= e2 THEN
            resSt := s2;
resEnd := e2;

            changes := TRUE;
        ELSIF s1 >= s2 AND s1 <= e2 AND e1 > e2 THEN
            resSt := s1;
            resEnd := e2;
            changes := TRUE;
        END IF;
        IF changes = TRUE THEN

LOOP

END IF;
    END LOOP;
    close cursor2;
END LOOP;
  close cursor1;
  DELETE FROM X_temporary_table
```

```
                    WHERE KEY = currTempTableId - 1;
    currTempTableId := currTempTableId + 1;
    open id_cursor;
```

The 5th step implements the comparing algorithm.

```
INSERT INTO X_temporary_table VALUES (currTempTableId,
                        resSt, resEnd ) ;
FETCH id_cursor INTO currTableId;
EXIT WHEN id_cursor%NOTFOUND;
OPEN cursor1 FOR SELECT *
LOOP

        FROM X_temporary_table
        WHERE KEY = currTempTableId - 1;
FETCH cursor1 INTO row1 ;
EXIT WHEN cursor1%NOTFOUND;
s1 := row1.START;
e1 := row1.END;
OPEN cursor2 FOR SELECT * FROM X WHERE
        KEY = currTableId;
LOOP

        FETCH cursor2 INTO row2;
        EXIT WHEN cursor2%NOTFOUND;
s2 := row2.START;
e2 := row2.END;
changes := FALSE;
        END IF;

END LOOP;
```

```
close cursor2;
IF s1 < s2 AND e1 >= s2 AND e1 <= e2 THEN
        resSt := s2;
resEnd := e1;

        changes := TRUE;
ELSIF s1 >= s2 AND s1 <= e2 AND e1 >= s2 AND e1 <= e2 THEN
        resSt := s1;
        resEnd := e1;
        changes := TRUE;
ELSIF s1 <= s2 AND e1 >= e2 THEN
        resSt := s2;
        resEnd := e2;
        changes := TRUE;
        ELSIF s1 >= s2 AND s1 <= e2 AND e1 > e2 THEN
        resSt := s1;
        resEnd := e2;
        changes := TRUE;
END IF;
IF changes = TRUE THEN
INSERT INTO X_temporary_table VALUES
(currTempTableId,
                    resSt, resEnd );
        END LOOP;
        close cursor1;
        DELETE FROM X_temporary_table WHERE KEY=currTempTableId - 1;
        currTempTableId := currTempTableId + 1;
        END LOOP;
        CLOSE id_cursor;
```

```
    OPEN cursor1 FOR SELECT * FROM X_temporary_table WHERE KEY =
    currTempTableId - 1;
    LOOP
        FETCH cursor1 INTO row1;
        EXIT WHEN cursor1%NOTFOUND;
        INSERT INTO X( KEY, START, END ) VALUES ( 4, row1.START,
        row1.END );
    END LOOP;
    CLOSE cursor1;
    DELETE FROM X_temporary_table ;
    DELETE FROM X_id_temp_table;
END;

run;

 CREATE OR REPLACE PROCEDURE main_proc
AS
BEGIN
     proc0();
END; . run;
```

Our tools

Eclipse

We connected our OLW data base schema and ontologies with our JavaTM created add-on program in order to implement the algorithm of our project, so what we needed was a platform which could combine different technologies together.
The Eclipse Platform was created to address to the issue of providing a common platform for diverse IDE- based products and facilitate their development.
Modern programs are intergraded using different parts implemented in many different languages and technologies, such as PagesTM, connectors, HTML, JSPTM, Enterprise JavaBeansTM, JavaTM, JavaServer ,COBOL or PL/1 programs, and, of courde, relational database schemas. In our thesis mainly we used the JavaTM language, creating an effective integrated development environment (IDE) for use in programming. Our applications presents some special challenges because a large number of different tool technologies have to be tightly integrated in support of development task flows.

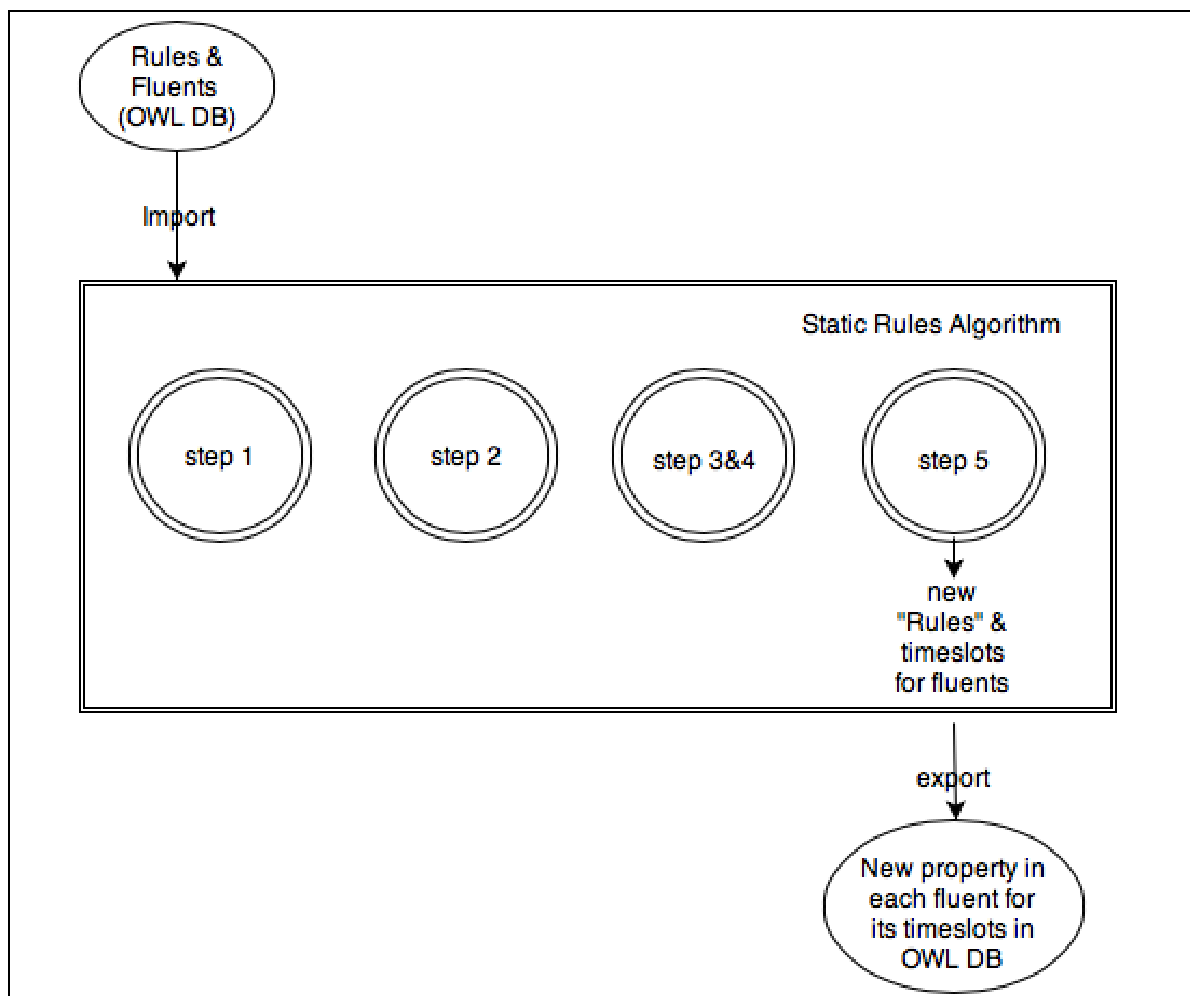
Protégé

In order to create a well - defined OWL data base – schema we decided the use of Protégé. Protégé-2000, or also called Protégé, is an OKBC-compatible knowledge-base–editing environment. Protégé-2000 is a Data Base platform, which is really easy-to-use and provides a configurable interface. Protégé-2000 also has a flexible metaclass architecture, which provides configurable templates for new classes in the knowledge base. We describe its OKBC-compatible knowledge model that makes the import and export of knowledge bases from and to other knowledge-base servers easy. The advantage that Protégé-2000 uses the architecture of the metaclasses, makes it easily extensible and enables its use with other knowledge models and applications such as our projects.

Our Programing Architecture – Design

As far as it concerns the architecture, our target was to implement an application based on the algorithms we explained in the previous sections, while using the most relevant tools, in order to intergrade a robust, simple and fast program. The main idea and the first step are simple, while

having the OWL database exported from the protégé-2000 platform, we import the ontologies and the rules in our add-on program. Proceeding to the next tribune, we implement the five steps of the algorithm, with the only change that the third and the forth ones are almost conjoined in one step created in some methods. We described those five steps in the previous sections of our thesis in details. The last stages of our design in our project are the export of all the results and import them in our previous ontology, but in the same time updating the ontology too. The update is once more simple as we add the time intervals that we created from the algorithm, in the form of properties of each fluent and subject in our database. The next figure shows us a general point of view of our program's flow chart.



We now move on at the description of all the classes, the variables and the functions we implemented in our project.

Classes

Static Rules

The staticRules Class is the one that contains the most of the properties, variables and methods of our Java code. At first, we import the packages of the other PelletPackage classes (i.e. the PelletPackage.fluent and the PelletPackage.sections), the one with the arrayLists and the packages for the input/output and the java language system. In this class the five steps of the Algorithm that we are based on are mainly implemented. It is, also, quotable to mention that at this point we also have methods that print some results and tables in order to test it and try a beta edition. Finally, this class has also the *main* function of our program that calls all the functions needed to be compiled and run, as an integraded make-all file.
In this Class we find the single-array variables :

- Static fluent [] *nameOftheVariable

We also find the double-array variables :

- Public static fluent [] [] conjunctionTable
- Public static fluent [] [] tableFfunctors
- Public static fluent [] [] tableFconditions
- Public static fluent [] [] tableFtfunctors
- Public static fluent [] [] tableFtconditions

Other futures that we find are the properties – methods of these Class :

- public static void printConjunctionTable
- public static void printTableF
- public static String[] initializeFluentNames
- public static fluent[][] initializeExample
- public static fluent[][] step2table

- public static void checkDoubles

- public static fluent [][] step4tableFfunctors
- public static fluent [][] step4tableFconditions
- public static fluent [][] step4tableFtfunctors
- public static fluent [][] step4tableFtconditions
- public static fluent[][] step4tableG
- public static fluent[][] step4tableK
- public static fluent[][] copyFluentTable
- public static fluent copyFluentElement
- public static sections setGipair
- public static sections [] setGi
- public static sections setUnionPair
- public static sections setUnionG
- public static rulesIOalgorithm
- public static void main(String[] args

All the variables, the properties and the methods of the staticClass will be described in details at the upcoming subsection and the upcoming figure presents our staticRules class' public global variables.

```java
package PelletPackage;

import java.io.*;
import java.lang.System;
import PelletPackage.fluent;
import java.util.ArrayList;
import PelletPackage.sections;



public class staticRules {



  static fluent[] illegal;
  static fluent[] suspended;
  static fluent[] receiveSalary;
  static fluent[] goodEmployee;
  static fluent[] receiveBonus;

  public static fluent[][] conjunctionTable; //conjunction between raws , disjunctions between elements of each raw
  public static fluent[][] tableFfunctors;
  public static fluent[][] tableFconditions;
  public static fluent[][] tableFtfunctors;
  public static fluent[][] tableFtconditions;



  static String allFluents[];
```

Figure 3. Variables

Variables

In this class we implement an interesting point of our framework development. In fact, it is an ontology that represent all the atrributes and the functions needed of any fluent that will be imported or exported from our database schema. There are many different Contructors and variables :

- boolean bln
- int startTime
- int endTime

- int pairFlag
- String name

This class generates also two different methods when called :

- public void copyFluent
- public boolean compareFluent

All the variables, the properties and the methods of the *fluent* will be described in details at the upcoming subsection and the upcoming figure presents our fluent class' public global variables..

```
package PelletPackage;

public class fluent {

  boolean bln = true;
  int startTime = 0;
  int endTime = 0;
  int pairFlag = 0;   // for table G if the fluent is in conjunction with another fluent
  String name = "nobodyYet" ;

  public fluent(String newName, int start, int end, int flag)

  public fluent(String newName)


  public fluent()
  {
    this.bln = true;
    this.startTime = 0;
    this.endTime = 0;
    this.name = null;
    this.pairFlag = 0;


  }

  public void copyFluent(fluent Incoming)
  public boolean compareFluent(fluent f)
}
```

Figure 4. Fluent class

Sections

The sections is the third major class of our project. It implements arrays for creating the proper timeslots for each and every fluent. It supports a quick one time slot entity with the variables given, either a multi-timeslot one with the arrays. Variables found in this class are :

- Int startTime
- Int endTime
- Int multipletimeSectionsFlag

Except the variables, the class utilizes also one constructor and the methods :

- Public void copySection
- Public void appendSection

All the variables, the properties and the methods of the *sections* will be described in details at the upcoming subsection and the upcoming figure presents our sections class' public global variables.

```
package PelletPackage;
import java.util.ArrayList;


public class sections {

  int startTime = 0;
  int endTime = 0 ;
  int multipleTimeSectionsFlag = 0 ;
  ArrayList<String> sectionNames = new ArrayList<String>( );
  ArrayList<Integer> startingTimeSections = new ArrayList<Integer>( );
  ArrayList<Integer> endingTimeSections = new ArrayList<Integer>( );



  public sections(String sectionName1 , String sectionName2, int start, int end)

  public sections()
  public void copySection(sections Incoming)
  public void appendSection(sections Incoming, sections old)



}
```

Figure 5. sections class

We have already described the three important classes that are called in order to compile the code for algorithm. On the other hand the mainPellet Class is the one that is used for the import and export of the data, which will be the input and output respectively of our main algorithm.

There is only one variable :

- Public static finale string

And only three functions

- Public static void main
- Void sparqInsertTurtle
- Void sparqDeleteTurtle

```
package PelletPackage;

import com.clarkparsia.owlapiv3.OWL;

public class MainPellet {

   public static final String DOCUMENT_IRI = "file:test1.owl";

    public static void main(String[] args) throws OWLOntologyCreationException {
```

Figure 6. mainPellet class

Properties of Classes

Static Rules

Arrays

illegal

This is an array of variables that was created for the test of the algorithm. In this array we imported the values of database schema under the entity "illegal".

suspended

This is an array of variables that was created for the test of the algorithm. In this array we imported the values of database schema under the entity "suspended".

receiveSalary

This is an array of variables that was created for the test of the algorithm. In this array we imported the values of database schema under the entity "receiveSalary".

goodEmploy

This is an array of variables that was created for the test of the algorithm. In this array we imported the values of database schema under the entity “goodEmploy”.

receiveBonus

This is an array of variables that was created for the test of the algorithm. In this array we imported the values of database schema under the entity “receiveBonus”.

double arrays

allFluents[];

This is an array of variables where we imported the names of all the fluents from the database schema. The names of the fluents that we need for the new time intervals.

conjunctionTable

This is an array of variables which is implemented in order to creat a series of conjunctions. It means that the elements of the array are in a raw connected with a conjunction between them.

tableFfunctors

This array implements a table of the elements that are the factors of the Φ of the fourth step of the algorithm that we were based on during the integration of our algorithm. Factors of the Φ is each and every of the elements that are a part of the Φ.

tableFconditions

This array implements a table of the elements that are the conditions of the Φ of the fourth step of the algorithm that we were based on during the integration of our program. Conditions of the Φ is all the conditions that must exist in order the statement of the Φ to be a true value.

tableFtfunctors

This array implements a table of the elements that are the factors of the Φ’ of the fourth step of the algorithm that we were based on during the integration of our code. Factors of the Φ’ is all the elements that create the Φ’.

tableFtconditions

This array implements a table of the elements that are the conditions of the Φ’ of the fourth step of the algorithm that we were based on during the integration of our program. Conditions of the Φ’ is all the conditions that must exist in order the statement of the Φ’ to be true.

Methods

public static void **printConjunctionTable** (int numberOFConjunctions, int maxNumberOfDisjunctions)

This method is implemented in order to print the global table of conjunctions that we use in our project . The printConjunctionTable has two different parameters. The first one is an integer, the numberOFConjunctions, which is the number of the elements in the global conjunction table that we use in our implementation. The second one is the maxNumberOfDisjunctions, which is also an integer and describes the maximum number of the disjunctions that might occur. Its return type is void, which means that this method does not return anything.

public static void **printTableF** (int numberOFConjunctions, int maxNumberOfDisjunctions)

The printTableF method is ceated in order to print the table Φ of that is described in the algorithm we were based on. This methos has two parameters. The first one is an integer, the numberOFConjunctions, which is the number of the elements in the global conjunction table that we use in our implementation. The second one is the maxNumberOfDisjunctions, which is also

an integer and describes the maximum number of the disjunctions that might occur. Its return type is void, which means that this method does not return anything.

public static String[] **initializeFluentNames** (int numberOfFluents)

This function is integrated in order to initialize a table with all the names of the fluents that we are going to use in every occasion, taken by the database schema. The initializeFluentNames has only one parameter, which is the number of the fluents exported. The parameter is an integer and it is called the numberOfFluents.. Ous method returns an array with all the names in a raw of the fluents we use.

static void **sparqlInsertTurtle** () throws Ioexception

The turtle functions are the ones that insert the information given from our algorithm as an outcome back to the RDF/XML based in OWL database file. There is a variable string in the function, which is the name of the file that we are going to write the new data down. Then we create dynamically a string -stream of bytes- which insert in the file the time interval boundaries in the correct form, in order the reasoner can manage them.

static void **sparqlDeleteTurtle** () throws Ioexception

The turtle functions are the ones that deletes the information given from the RDF/XML based in OWL database file. There is a variable string in the function, which is the name of the file that we are going to write the new data down. Then we create dynamically a string -stream of bytes- which includes the information that are going to be erased from the file. It is a method that helps us to manage the insertion and the edit of the new time boundaries.

public static fluent[][] **initializeExample** (int numberOFConjunctions, int maxNumberOfDisjunctions, int numberOfFluents)

This method is initializes our first example/step that we used to test our code. The initializeExample has three parameters, all of which are integers. The first one, the numberOFConjunctions, which is the number of the elements in the global conjunction table that we use in our implementation. The second one is the maxNumberOfDisjunctions, which describes the maximum number of the disjunctions that might occur. The third one is the numberOfFluents which is also the number of fluents that we exported from the database. Its return type is a double array of fluent, which means that this function gives back a double dimensioned array of fluents which are the rules, triplets of fluents, that we are going to begin with.

public static fluent[][] **step2table** (fluent [][] step1table, int numberOfConjunctions)

This method is implements the second step of the algorithm that we are based on . The step2table has two different parameters. The first one is a double dimension table, the step1table, which is the table with the fluents that we derived from the first step of our implementaion. The second one is an integer, the numberOFConjunctions, which is the number of the elements in the global conjunction table that we use in our implementation. Its return type is a double array of fluent, which means that this function gives back a double dimensioned array of fluents which are the new updated rules, triplets of fluents, that we are going to continue with.

step3

public static void **checkDoubles** (int numberOFConjunctions)

This method is implemented in order to check and eliminate rules, triplets of fluents, that are created two or more times, while we proceed to the next step of our project. We need each rule just one time to exist in our new table of the third step. The checkDoubles has only one parameter. The numberOFConjunctions, which is the number of the elements in the global conjunction table that we use in our implementation. We use this method in the same step as the

fourth one, so we do not need to return anything, thus its return type is void, which means that this method does not return anything.

public static fluent [][] **step4tableFfunctors** (int numberOfFluents)

This method is integrated in order to implement the table Φ with its factors of the fourth step. The step4tableFfunctors has one parameter, which is an integer, the numberOFConjunctions,. It is the number of the elements in the global conjunction table that we use in our implementation. Its return type is a double array of fluent, which means that this function gives back a double dimensioned array of fluents which are the factors of table Φ, of the fourth step, that we are going to continue with.

public static fluent [][] **step4tableFconditions** (int numberOfFluents)

This method is integrated in order to implement the table Φ with its conditions of the fourth step. The step4tableFconditions has one parameter, which is an integer, the numberOFConjunctions,. It is the number of the elements in the global conjunction table that we use in our implementation. Its return type is a double array of fluent, which means that this function gives back a double dimensioned array of fluents which are the conditions of table Φ, of the fourth step, that we are going to continue with.

public static fluent [][] **step4tableFtfunctors** (int numberOfFluents)

This method is integrated in order to implement the table Φ' with its factors of the fourth step. The step4tableFfunctors has one parameter, which is an integer, the numberOFConjunctions,. It is the number of the elements in the global conjunction table that we use in our implementation. Its return type is a double array of fluent, which means that this function gives back a double dimensioned array of fluents which are the factors of table Φ', of the fourth step, that we are going to continue with.

public static fluent [][] **step4tableFtconditions** (int numberOfFluents)

This method is integrated in order to implement the table Φ' with its conditions of the fourth step. The step4tableFconditions has one parameter, which is an integer, the numberOFConjunctions,. It is the number of the elements in the global conjunction table that we use in our implementation. Its return type is a double array of fluent, which means that this function gives back a double dimensioned array of fluents which are the conditions of table Φ', of the fourth step, that we are going to continue with.

public static fluent[][] **step4tableG** (fluent [][] step3table, int numberOfFluents, int numberOfrules)

This method is implementing the G table of the forth step of the algorithm that we are based on. The table G is a table with all the fluents that are crossed with an algorithm and have a triplet – rule with each of the rest fluents. Then a raw is created for every fluent and includes all the fluents that are matched in a triplet and their value is true. For instance, A V B → D will be on a raw of our table, but A V B → ¬D will not . Step3table is the first parameter, which is the table with all the triplets – rules we exported from our database scehma. The second parameter numbers the quantity of the different fluents that we use in each occasion and the third one is the number of the rules we have derived. It returns the new table with the type of a double array fluent.

public static fluent[][] **step4tableK** (fluent [][] step3table, int numberOfFluents)

This function creates the K table of the forth step of the algorithm that we are based on. The table K is a table with all the fluents that are crossed with an algorithm and have a triplet – rule with each of the rest fluents, but only with a "not" value. Then a raw is created for every fluent and includes all the fluents that are matched in a triplet and their value is true. For instance, A V B →

¬D will be on a raw of our table, but A V B → D will not. Step3table is the first parameter, which is the table with all the triplets – rules we exported from our database scehma. The second parameter numbers the quantity of the different fluents that we use in each occasion. It returns the new table with the type of a double array fluent.

public static fluent[][] **copyFluentTable** (fluent [][] step3table, int numberOfFluents)

This method is implemented in order to copy table of fluents to a new one. The copyFluentTable has two different parameters. The first one is a double array of fluents, the step3table, which is a double array of fluents taken from the third step of our algorithm. The second one is the numberOfFluents, which is an integer and describes the number of the fluents of the table given in tha first parameter. Its return type is also a double dimension table, which means that this method return the new array copied exactly as the one given in the parameter.

public static fluent **copyFluentElement** (fluent Incoming)

This method creates a new fluent copy, exactly the same as the one given in the parameter. This method has only one parameter which is a fluent which is called incoming. It copies the incoming fluent and returns the new fluent back. The next figure presents the methods included in the staticRules class divided in the first four steps.

```
//step1
public static String[] initializeFluentNames(int numberOfFluents)

public static fluent[][] initializeExample(int numberOFConjunctions, int maxNumberOfDisjunctions, int numberOfFluents) {

//step2
public static fluent[][] step2table(fluent [][] step1table, int numberOfConjunctions)

//step3
public static void checkDoubles(int numberOFConjunctions){

//step4 4 sunartiseis mia gia kathe pinaka F , F', K, G
//dimiourgei ta functors tou pinaka F

public static fluent [][] step4tableFfunctors(int numberOfFluents)


public static fluent [][] step4tableFconditions(int numberOfFluents)

//step4 - dimiourgei ton pinaka tvn functors gia ton F tonos

public static fluent [][]  step4tableFtfunctors(int numberOfFluents)

//step 4 - dimiourgei ton pinaka tvn conditions gia ton F tonos

public static fluent [][] step4tableFtconditions(int numberOfFluents)

//Epistrefei ton pinaka G

public static fluent[][] step4tableG(fluent [][] step3table, int numberOfFluents, int numberOfrules)


public static fluent[][] step4tableK(fluent [][] step3table, int numberOfFluents)
```

Figure 7. Methods

step 5

public static sections **setGipair** (fluent f1, fluent f2)

This function is implemented in order to set a Gi pair between two different fluents, so with the setGi method to create the whole complete Gi set. The setGipair has two parameters, both of them are fluent. Its return type is sections, which means that it is a new object we created to describe the relation between the two different fluents, combining their time intervals. In the sections object we can save the names of the fluents that are related together and the time intervals that this pair can be true.

public static sections [] **setGi** (fluent [][] IncomingFluentTable, int numberOfFluents)

This function is created in order to set the whole, complete Gi pair between many different pairs of fluents, pair of fluents that are given from the setGipair method. The setGi has two different parameters. The first one is a double array with the table of all the fluents that we are going to find their relation out and one integer that indicates the number of the different fluents existing in that table. Its return type is table of sections. Sections is a new object we created to describe the relation between the two different fluents, combining their time intervals. In the sections object we can save the names of the fluents that are related together and the time intervals that this pair can be true.

public static sections **setUnionPair** (sections G1, sections G2, int lastSectionAppended, int lastMultipleTimeSectionsFlag)

This method is implemented in order to set a Union pair between two different sections, so with the setUnionG method can integrate the whole complete Union G set, which is also the main target of our fourth step. Sections is a new object we created to describe the relation between the two different fluents, combining their time intervals. The setUnionpair has four parameters. The first two are called G1 and G2 and both of them are sections type variables. The third and the fourth ones are variables that help us in order to append the time slots if needed, in the case tha we have many different time intervals for these two sections. Its return type is sections, which means that it is a new object we created to describe the relation between the two different fluents. In the sections object we can save the names of the fluents that are related together and the time intervals that this pair can be true.

public static sections **setUnionG** (sections [] Gi, int numberOfFluents)

This method is implemented in order to reach our final point at the fourth step and create a complete Union with all the fluents from the Gi we produced before with the setGi method. In this method the previous function of creating Union pairs is used while building the table up.

Sections is a new object we created to describe the relation between the two different fluents, combining their time intervals. The setUnionpair has two parameters. The first one is called Gi and it is a sections type variable. The second one is an integer that indicates the number of the different fluents existing in that table. Its return type is sections, which means that it is a new object we created to describe the relation between the two different fluents. In the sections object we can save the names of the fluents that are related together and the time intervals that this pair can be true.

Public static voids **RulesIOalgorithm**

This is the packets last method. This function sums up in the right order all the previous methods and calls them in the right order and with the specific factors and variables in order to run our algorithm as it meant to be. The next figure presents the methods included in the staticRules class in the fifth step.

```
public static sections setGipair(fluent f1, fluent f2)

//  upologizei in tomi Gi = f1 Λ f2 Λ ... Λ fn
public static sections [] setGi(fluent [][] IncomingFluentTable, int numberOfFluents)

//upologizei to union to xroniko gia 2 Gi ... Union = G1 V G2

public static sections setUnionPair(sections G1, sections G2, int lastSectionAppended, int lastMultipleTimeSectionsFlag)

//upologizei to union to xroniko gia ola ta Gi ... Union = G1 V G2 V G3 V ...

public static sections setUnionG(sections [] Gi, int numberOfFluents)
```

Figure 8. Fifth step Methods

public static void **main** (String[] args)

This method is the main one, which is the method that calls all the functions and methods needed in order our algorithm to run in the right order. Its return type is void, which means that this method does not return anything.

Fluents

Boolean bln

This variable indicates if the fluent is true or false in a triplet of our rules. For instance, A V B → ¬D, the ¬D has a value of true in this specific variable.

int startTime

This variable indicates when a timeslot interval begins for this fluent. It is initialized with the value of zero.

int endTime

This variable indicates when a timeslot interval ends for this fluent. It is initialized with the value of zero.

Boolean pairFlag

This variable has a value of zero of true when fluent is in conjunction with another fluent.

String name

This variable is the name of the fluent. It is initialized as "noboryYet" but it depends on the constructor we call if the variable will be initialized with another different name.

public **fluent** (String newName, int start, int end, int flag)

This is a constructor of our object fluent we created. Its parameters are the newName, start, end and flag. The parameter newName is an string and in fact it is the name of the fluent. The three

other variables are integers and they are the starting time of the specific fluent, the ending time of the specific time and a flag respectively. The Boolean variable is always initialized with a true value.

public **fluent** (String newName)

This is a constructor of our object fluent we created. Its parameter is only the newName. The parameter newName is an string and in fact it is the name of the fluent. The other prices that are not given for the rest of the variables are all initialized to zero. The Boolean variable is always initialized with a true value.

public **fluent** ()

This is a constructor of our object fluent we created. It has no parameters. Thus all the other prices that are not given for the variables are all initialized to zero and the name of the fluent is "noboyYet" by default. The Boolean variable is always initialized with a true value.

public void **copyFluent** (fluent Incoming)

This method is implemented in order to make a copy of the incoming fluent and returns an exact copy of a fluent object .

public boolean **compareFluent** (fluent f1, fluent f2)

The compareFluent function is implemented in order to make a comparison between the two fluents. The return type of the method is a Boolean. It is true if the two elements are the same in every aspect, or a false if there is even one only difference between them.

mainpellet

public static final String DOCUMENT_IRI = "file:test1.owl"

This variable is the a string which indicates which is the file of the database schema that we are going to use in order to derive and export the entities, the fluents and the rules for our program.

static void sparqlInsertTurtle() throws IOException

It is a void method that Inserts back in the ontology information. Actually, it inserts the results of our application in new property slots for each object respectively. It creates a manager through whom it edits the ontology with an incoming string giving the stream of the data. The string is dynamically created for each case.

static void sparqlDeleteTurtle() throws IOException

It is a void method that deletes from the ontology information. It creates a manager through whom it edits the ontology with an incoming string indicating which information will be deleted. The string is dynamically created for each case.

public static void **main** (String[] args) throws OWLOntologyCreationException

In the main function of our class we call as the method from the API for exporting and importing the information from and to, respectively, our OWL database.

Sections

int startTime

This variable indicates when a timeslot interval begins for this section. It is used in the case of having only one time interval slot, so the use of it to easier and make our ontology more robust. It is initialized with the value of zero.

int endTime

This variable indicates when a timeslot interval ends for this section. It is used in the case of having only one time interval slot, so the use of it to easier and make our ontology more robust. It is initialized with the value of zero.

int multipleTimeSectionsFlag

The multipleTimeSectionsFlag variable is a flag. When the flag value is less than two then we have multiple time slots, thus we use the array of our ontology to describe the intervals, otherwise we use the integer variables that we mentioned above.

ArrayList<String> sectionNames

This arrayList of strings is a raw with the names of the fluents that are connected in a specific section – rule and their time intervals, where all of them are true, are described with the arrays that follow.

ArrayList<Integer> startingTimeSections

This array of integers describes the starting points in the time where the fluents that are connected in a specific section, are all of them of true value. This list is triggered by the flag, multipleTimeSectionsFlag, only if its value indicates that there are more than two fluents. This means that the integer variables of the sections are not used any more.

ArrayList<Integer> endingTimeSections

This list of integers describes the ending points in the time where the fluents, which are connected in a specific section, are all of them of true value. This list is triggered by the flag, multipleTimeSectionsFlag, only if its value is true. This means that the integer variables of the sections are not used any more.

public **sections** (String sectionName1 , String sectionName2, int start, int end)
This is a constructor of our object sections that we created. Its parameters are the sectionName1, start, end and sectionName2. The parameters sectionName1 and sectionName2 are strings and in fact it is the name of the elements that we use to create a pair of fluents or a pair of pairs etc. The two other variables are integers and they are the starting time of the specific sections, the ending time of the specific time. The Boolean variable is always initialized with a false value, if the fluents are more than two then we have to change the flag to true and use the arrays.

public **sections** ()
This is a simpler constructor of our object sections that we created. It has no parameters, thus the initialization of our arrays and variables are the default ones or we have to initialize them one at a time.

public void copySection(sections Incoming)
This method is implemented in order to make a copy of the incoming sections and returns an exact copy of a new object .

public void appendSection(sections Incoming, sections old)

The appendSection function is implemented in order to append the sections with more fluents. The two parameters are both of sections type. The second one is the old sections that will be appended and with the new information and the first one is the data that we will use for the update of our sections object. The return type of the method is void, meaning there is nothing to return.

Motivational Example in details

Our algorithm, in fact is a reasoner. It derives information from the Data base file, such as properties, relationships, classes and objects it processes them and updates the Data base schema with the new ones. In this way, the program guarantees the consistency of the database, thus it eliminates the ramification problem, often appeared in many schemas.

In this section we will describe a demonstration of an example that we created. We used the protégé tool to create a new RDF based on OWL database. There are six classes named as "illegal", "goodEmployee", "receiveSalary", "receiveBonus", "suspended", "triplet" and for each class we created one instance under the same name. We also inserted some typical relations and properties between the objects. The next picture shows us the protégé program and the database developed and its individuals.

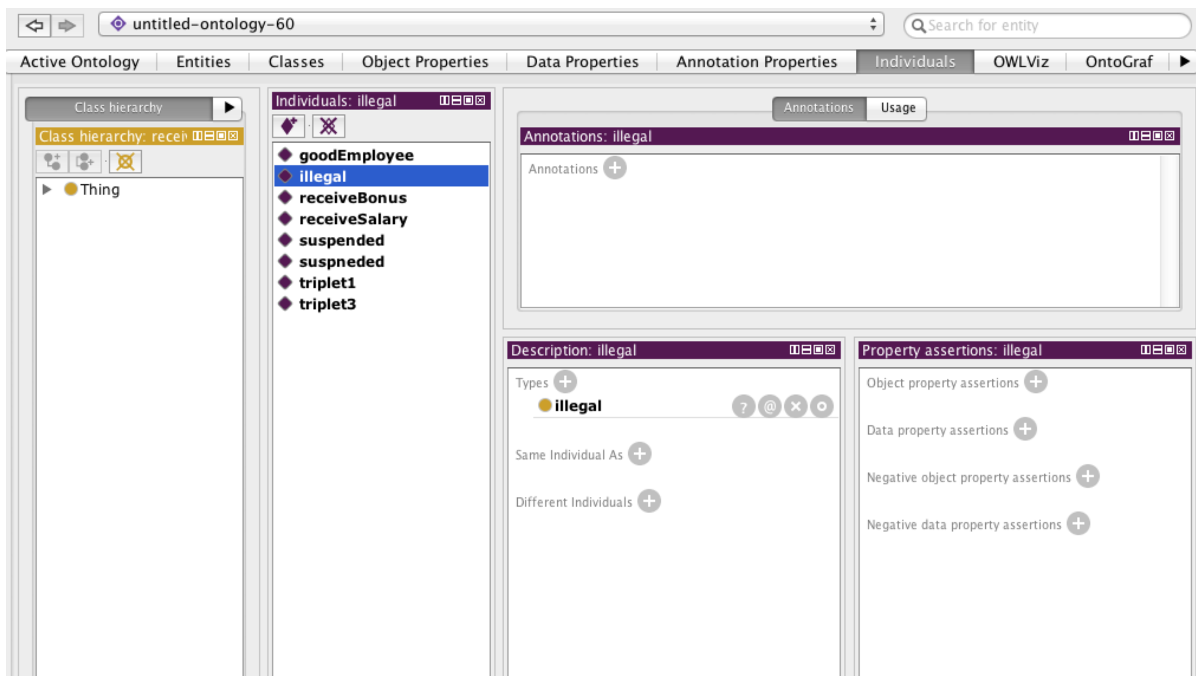


Figure 9. motivational example 1 database

The upcoming step is to run our program. Our program, firstly, export the information. Two more pictures follow (figure 10 and figure 11) , showing a System print we made with our code to show the data we export. There are the classes, the individuals , the properties and the disjunctions of the class illegal. We just printed out the disjunctions and the properties of just one class, in order our example not to be tiring.

```
*******************************************DATA FROM OWL FILE*****************************************************************************************
******************************************************************************************************************************************************
******************************************************************************************************************************************************
Loaded ontology: Ontology(OntologyID(OntologyIRI(<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60>))) [Axioms: 36 Logical Axioms: 20]
 from: file:/Users/manos/Documents/workspace/triplet.owl
--- Declared Classes ---
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#triplet>
triplet
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#receiveSalary>
receiveSalary
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#receiveBonus>
receiveBonus
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#goodEmployee>
goodEmployee
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#illegal>
illegal
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#suspended>
suspended
----
--- Class Assertion Axioms ---
----
--- Individuals ---
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#receiveSalary>
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#illegal>
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#goodEmployee>
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#triplet1>
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#receiveBonus>
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#triplet3>
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#suspneded>
----
******************************************************************************************************************************************************
Names taken from DATABASE :
[triplet, receiveSalary, receiveBonus, goodEmployee, illegal, suspended]
******************************************************************************************************************************************************
*******************************************END DATA FROM OWL FILE*************************************************************************************
******************************************************************************************************************************************************
```

Figure 10. motivational example export print

```
--- Properties ---
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#upperBound>
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#lowerBound>
<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#tripletID>
----
--- Disjoints ---
--- Class Illegal ---
DisjointClasses(<http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#goodEmployee> <http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#
```

Figure 11. motivational example export print 2

The two following picture (figure 12 and figure 13) present the method, code of our program, which we used in order to export and print the above information.

```
public static void mainPelletExport() throws OWLOntologyCreationException
{
  int i=0;
  int j=0;
  int stringLength=0;
  String newString;

  PrefixManager pm = new DefaultPrefixManager("http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#");
  OWLOntologyManager manager = OWLManager.createOWLOntologyManager();
  File file = new File("/Users/manos/Documents/workspace/triplet.owl");
  OWLOntology ontology = manager.loadOntologyFromOntologyDocument(file);
  System.out.println("Loaded ontology: " + ontology);
  ////
  // We can always obtain the location where an ontology was loaded from
  IRI documentIRI = manager.getOntologyDocumentIRI(ontology);
  System.out.println(" from:
  /////
  // In order to get access
  // Get a reference to a da
  OWLDataFactory factory = m
  System.out.println("--- De
  for (OWLClass c : ontology.getClassesInSignature()) {
      System.out.println(c.toString());

      newString = c.toString().substring ( 72 , c.toString().length ( ) );  //!!!!! MSUT CHANGE STRING LENGTH - fisrt par
      stringLength = newString.length();
      newString = newString.substring ( 0 , stringLength - 1 );
      System.out.println(newString);
      fluentNames.add(i,newString);
      i++;
  }
  System.out.println("----");
  ////
  OWLClass namedInd = factory.getOWLClass(":participant1", pm);
  System.out.println("--- Class Assertion Axioms ---");
```

```
IRI org.semanticweb.owlapi.model.OWLOntologyManager.getOntologyDocumentIRI(OWLOntology arg0)

Note: This element neither has attached source nor attached Javadoc and hence no Javadoc could be found.
Press 'F2' for focus
```

Figure 12. motivational example export code

```
System.out.println("--- Properties ---");

for (OWLDataProperty c : ontology.getDataPropertiesInSignature()) {
  System.out.println(c.toString());
}

System.out.println("----");

System.out.println("--- Disjoints ---");

System.out.println("--- Class Illegal ---");

for (OWLDisjointClassesAxiom c : ontology.getDisjointClassesAxioms(namedInd)) {
  System.out.println(c.toString());
}
```

Figure 13. motivational example export code 2

At this point, we can also highlight the fact that we insert the information we need in arrays, such as the names of the fluents are saved in the array called “fluentNames”. The next step in our code is to use the information of the array and create instances and tables describing the logical

relations in our program for those fluents. The upcoming figure shows us an instant of our code implementing this step and moreover we can observe that we insert some values in order to test our algorithm later.

```
conjunctionTable[0][0].bln = false;
conjunctionTable[0][0].name = fluentNames.get(4);
conjunctionTable[0][0].startTime = 0 ; // test gia to true.suspended
conjunctionTable[0][0].endTime = 3 ;// test gia to true.suspended
conjunctionTable[0][1].bln = true;
conjunctionTable[0][1].name = fluentNames.get(5);
conjunctionTable[0][1].startTime = 0 ; // test gia to true.suspended
conjunctionTable[0][1].endTime = 3 ;

conjunctionTable[1][0].bln = false;
conjunctionTable[1][0].name = fluentNames.get(5);
conjunctionTable[1][1].bln = true;
conjunctionTable[1][1].name = fluentNames.get(1);

conjunctionTable[2][0].bln = false;
conjunctionTable[2][0].name = fluentNames.get(3);
conjunctionTable[2][1].bln = true;
conjunctionTable[2][1].name = fluentNames.get(1);

conjunctionTable[3][0].bln = true;
conjunctionTable[3][0].name = fluentNames.get(5);
conjunctionTable[3][1].startTime = 0 ; // test gia to true.suspended
conjunctionTable[3][1].endTime = 4 ; // test gia to true.suspended
conjunctionTable[3][1].bln = false;
conjunctionTable[3][1].name = fluentNames.get(3);
conjunctionTable[3][1].startTime = 0 ; // test gia to true.suspended
conjunctionTable[3][1].endTime = 3 ; // test gia to true.suspended
conjunctionTable[3][2].bln = true;
conjunctionTable[3][2].name = fluentNames.get(2);
conjunctionTable[3][2].startTime = 0 ; // test gia to true.suspended
conjunctionTable[3][2].endTime = 2 ; // test gia to true.suspended
```

Figure 14 motivational example create instances

In order to explain the fluentNames array, we need to know that the row of the names that we "get" and use are "receiveBonus, receiveSalary, goodEmployee, illegal, suspended". It is in the same line that our "turtle" method derived them from the OWL file.

Our algorithm is executed, when the preparation function have finished, and processes the data given. The next figure is a print from our program that show the table G that we create in the fourth step of Mr. Papadakis algorithm. At this step, we could pinpoint that for each fluent we print each of the fluents that the Fk table is created for with a relationship of "AND" between them. In fornt of each fluent there are information concerning its time intervals and its true/false value. There is also a similar table with the exact true/false values of the fluents.

```
******** illegal *** true.0.0.null AND true.0.0.null AND true.0.0.null AND true.0.0.null AND true.0.0.null AND true.0.0.null A
*******
******** suspended *** false.0.3.illegal AND false.0.3.goodEmployee AND true.0.2.receiveBonus AND true.0.0.null AND true.0.0.n
*******
******** receiveSalary *** false.0.0.suspended AND false.0.0.goodEmployee AND true.0.0.null AND true.0.0.null AND true.0.0.nul
*******
******** goodEmployee *** false.0.0.receiveBonus AND true.0.0.null AND true.0.0.null AND true.0.0.null AND true.0.0.null AND t
*******
******** receiveBonus *** true.0.0.suspended AND false.0.3.goodEmployee AND true.0.0.null AND true.0.0.null AND true.0.0.null
```

Figure 15. motivational example print table Fk

The upcoming step of our code is the results that primarily are saved in a table. We call it the Gi table, as in the theoretical approach that we were based on. We printed, only the Gi row of the fluent "suspended" with the final timeslots and the names that has a relation with.

```
Gi.suspended.start=====0 und Gi.suspended.end=====3 und Gi.suspended.Names=====[illegal, goodEmployee]
```

Figure 16. motivational example final Gi union

The final functions are used to import the new data back in the database. We call them "trurtles". The "insertTurtle" generates dynamically the form of the information needed in order to insert the correctly in the OWL schema. We also have developed the "deleteTurtle" in order to delete the previous information, while we import the new ones. The next two figures (figure 17 and figure 18) show us the code of the "insertTurtle" function and the way we call it in our main.

```
static void sparqlInsertTurtle(int lowerBound, int upperBound, int tripletID, String fluentObjectName, String fluentClass


  FileManager.get().addLocatorClassLoader(Main.class.getClassLoader());
  Model model = FileManager.get().loadModel("/Users/manos/Documents/workspace/triplet.owl");

  String setValuesString =

  "PREFIX uni: <http://www.semanticweb.org/popi/ontologies/2015/7/untitled-ontology-60#>\n"+
  "\n" +
  "INSERT DATA \n"+
  "{\n"+
  "uni:" + fluentObjectName + " a uni:" + fluentClass + ";\n"+
  "uni:tripletID " + tripletID +";\n"+
  "uni:lowerBound "+ lowerBound  +";\n"+
  "uni:upperBound "+ upperBound  +".\n"+
  "}";

  model.enterCriticalSection(Lock.WRITE);
  try{
  UpdateRequest updateRequest = UpdateFactory.create(setValuesString);
  UpdateAction.execute(updateRequest, model);
   } finally {
     FileWriter fstream = new FileWriter("/Users/manos/Documents/workspace/triplet.owl");
     BufferedWriter out = new BufferedWriter(fstream);
     String lang = FileUtils.guessLang("/Users/manos/Documents/workspace/triplet.owl");
     model.write(out,lang);
     TDB.sync(model);
     model.leaveCriticalSection();
  }
}
```

Figure 17. motivational example insert trurtle

```
    try {
      sparqlInsertTurtle(finalSection.startTime,finalSection.endTime,1,fluentNames.get(3),fluentNames.get(3));
      sparqlInsertTurtle(finalSection.startTime,finalSection.endTime,2,fluentNames.get(4),fluentNames.get(4));
      sparqlInsertTurtle(finalSection.startTime,finalSection.endTime,3,fluentNames.get(5),fluentNames.get(5));

  //    sparqlDeleteTurtle();
    }
    catch (IOException e) {

      e.printStackTrace();
    }


    }
```

Figure 18. motivational example turtle call

After having finished with the description of a short test, we present the last figures taken from the protégé program before and after the use of our program. We printed pairs of screenshots in,

hence the first picture of the pair shows the individual that have not got time stamps at all, wether the second figure presents the updated individual that is updated with the time limits in order no ramification collision appear in our database.

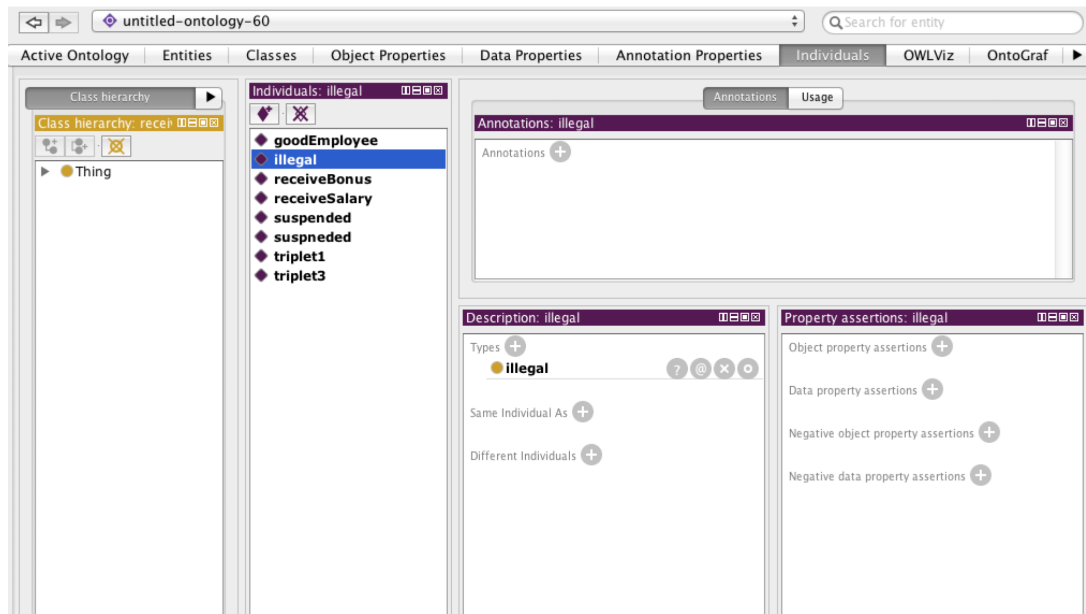


Figure 19. motivational example database before update

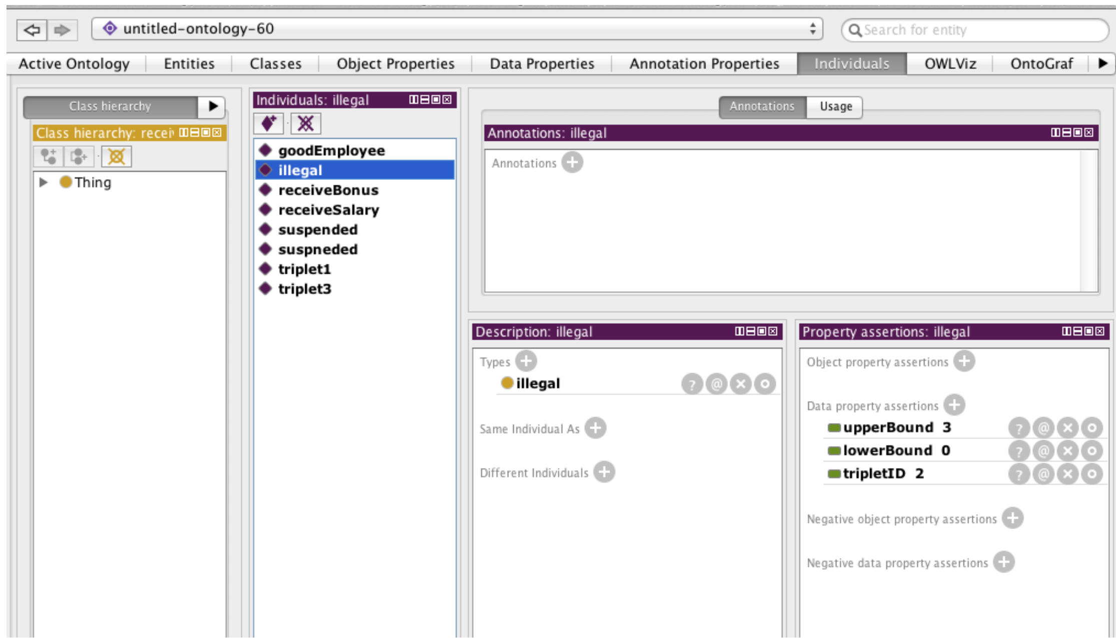


Figure 20. motivational example database after update

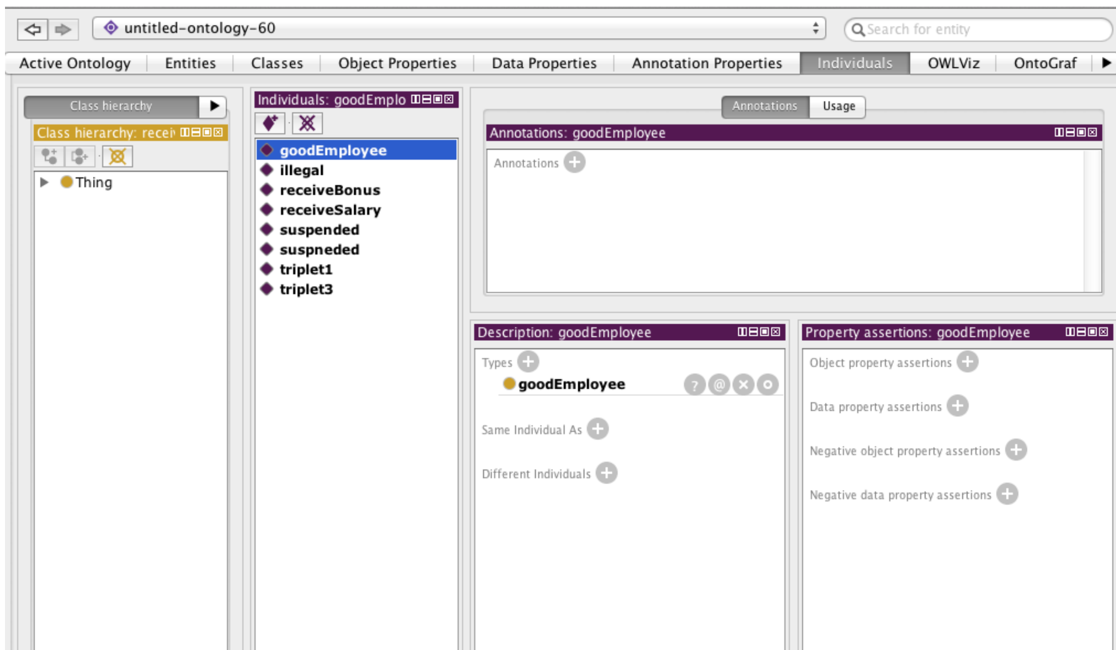


Figure 21. motivational example before update 2

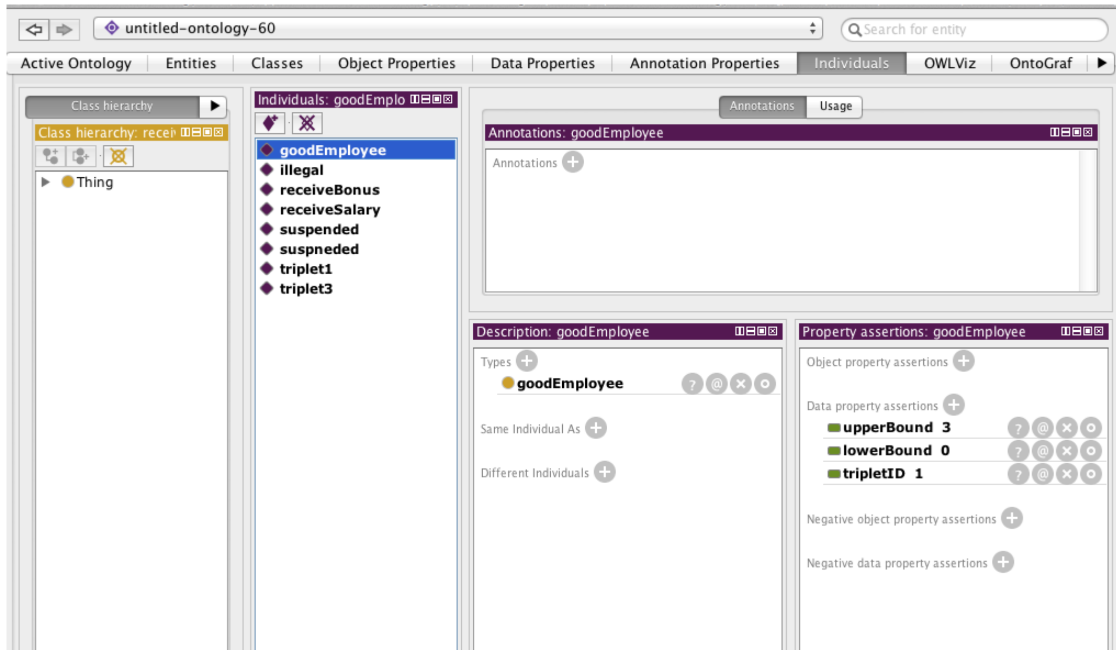


Figure 22. motivational example after update 2

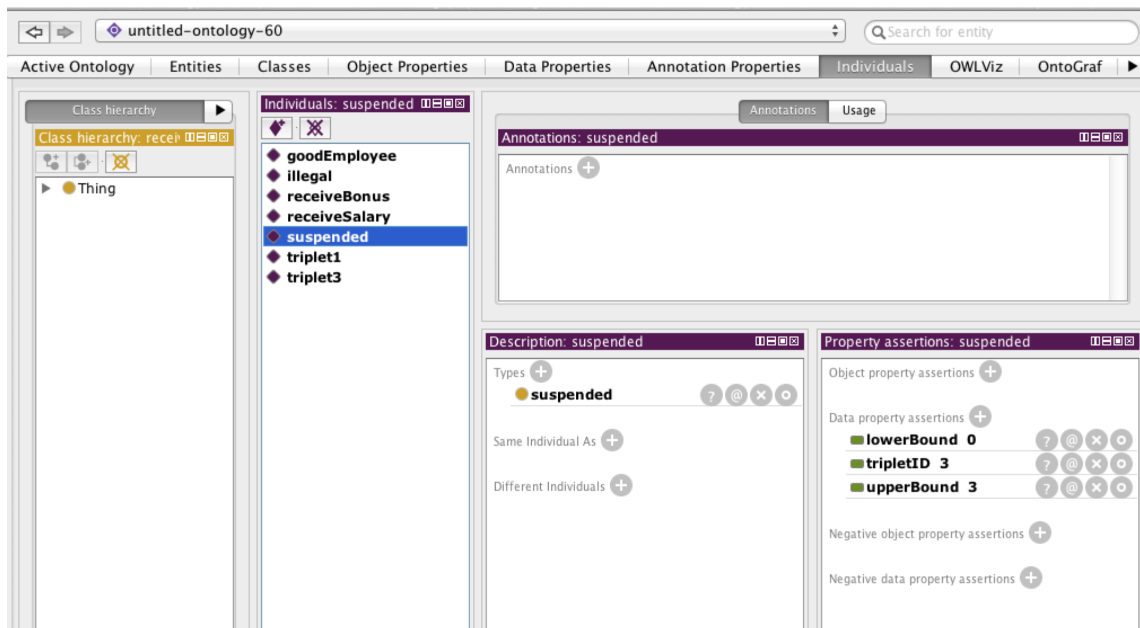


Figure 23. motivational example before update 3

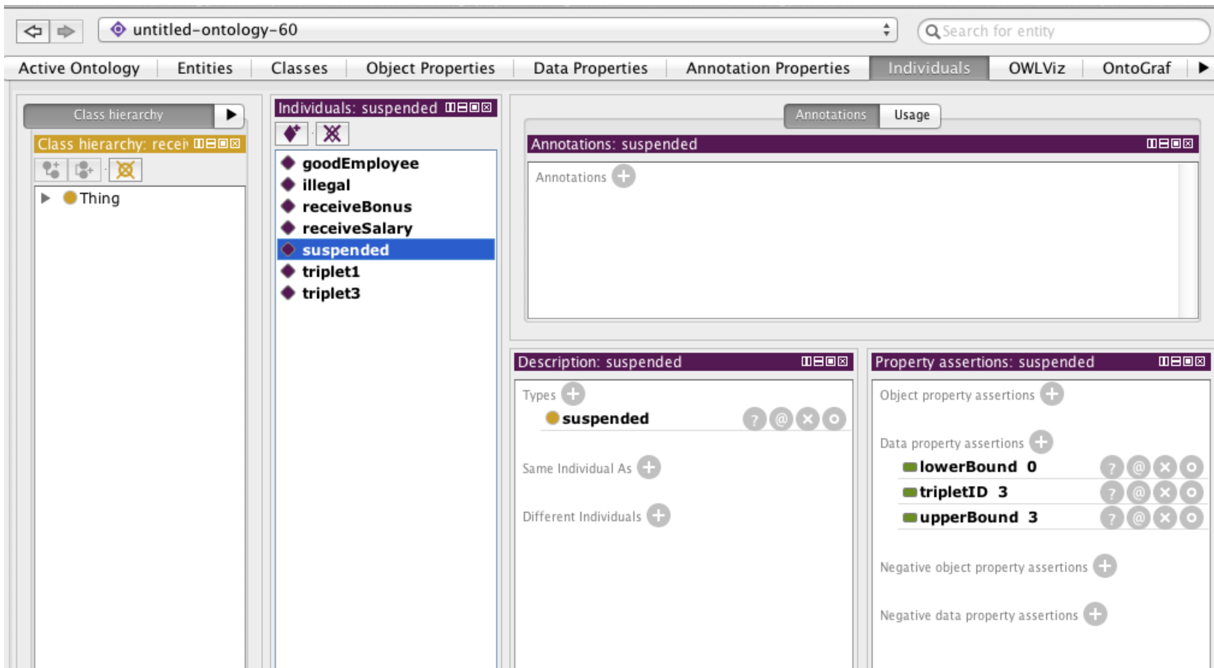


Figure 24. motivational example after update 3

3. CONCLUSIONS AND FUTURE WORK

For the needs of our project, we studied a lot of thesis and previous work done around our subject. We were based on that prior theoretical studies and mainly on the algorithm of Papadakis, Plexousakis and Christodolou in The ramification problem in temporal databases: a solution implemented in SQL. We, then, worked on the both the semantic aspect of our problem and the creation of a program manipulating and updating our data derived from an OWL database schema. This project aims to give a **solution in the ramification problem** through a scientific approach.

The ramification problem is one of the most important obstacles in modern databases, thus its solution is crucial in our opinion. At first, we tried to approach our problem with the use of

reasoning while taking under consideration the time as a factor**. Reasoning with time intervals** means that we introduce time slots for each of the schema's fluents. The reasoner that we ended up with is the Pellet of the OWL database, because it is the most commonly used one in the scientific community.

Of course, we needed to make a **representation of the information** that we exported and imported from and in our OWL database respectively, a quite important step, in order to achieve the time reasoning section. We decided the use of arrays that represent the disjunctions and the conjunctions between the fluents. Also, new entities, objects and arrays were created in JAVA tm language, to save the timeslots and the tables of the algorithm that we were based on. By this way we made our code robust, fast and easy to manipulate and edit. Our program is based on functions in JAVA language in order export data and import the new ones, thus it is easily connected with the algorithm that processed them. Summarizing overall, we implemented a program that **exports conclusions with time intervals and updates the OWL database schema.**

For **future work**, we propose measurements on how fast our code works. Many tests can be run in different database schemas, with a variety of fluents and how they interact with each other. More complicated rules can be tested in our program to reach results about how rapid our program exports its conclusions and if with slight editing on our import and export function or main even the main algorithm, we could improve it. Our tests were limited in a small database such as he example one we given in our previous section of this thesis. We, also, recommend the extension and the connection of this program with other reasoners, Jenna for instance, or even for different databases apart from the RDF/XML based on OWL one. In this case new methods for importing and exporting data should be developed and in some cases future reasoning and representation must be held.

4. REFERENCES

1. Anand S. Rao, Norman Y. Foo (2006) the Minimal Change and Maximal Coherence: A Basis for Belief Revision and Reasoning about Actions.

2. Antoniou and F. van Harmelen (2004) A Semantic Web Primer, MIT Press.

3. Baader, F. & Nutt, W. (2003) Basic Description Logics. , pp 43—95.

4. Cruz-Cunha Maria etc, (2013) : Handbook of Research on ICTs for Human-Centered Healthcare and Social Care. DOI: 10.4018/978-1-4666-3986-7.ch019

5. Denecker M, Ternovska E (2007) Inductive situation calculus. Artif Intell Arch 171(5–6): 332–360.

6. DebenhamJ(2006)Maintainingknowledgewithaformalmodel. Int J Appl Intell 24(3):205–218.

7. Drescher C, Thielscher M (2007) Integrating action calculi and description logics. In: KI 2007: advances in artificial intelligence, August, pp 68–83.

8. C. Elkan. (1992) Reasoning about action in first order logic. Proceedings of the Conference of the Canadian Society for Computational Studies in Intelligence, (CSCSI-92), Vancouver, BC pp. 221–227. [SD-008].

9. Franconi, E. & Toman, D. (2003) Fixpoint Extensions of Temporal Description Logics. In Description Logics.

10. Giordano L, Martellib A, Schwindc C (2005) Specialization of interaction protocols in a temporal action logic. In: LCMAS, pp 3– 22.

11. Goldin D, Srinivasa S, Srikanti V (2004) Active databases as in- formation systems. In: Proceedings of the international database engineering and applications symposium, pp 123–130.

12. Heflin and Z. Pan. (2004) In A Model Theoretic Semantics for Ontology Versioning. Lehigh University Technical Report LU-CSE-06-026

13. Kakas AC, Miller RS, Toni F (2001) E-RES: reasoning about ac- tions, events and observations. In: Proceedings of LPNMR2001. Springer, Berlin, pp 254–266.

14. Kakas A, Miller R (1997) A simple declarative language for de- scribing narratives with actions. J Log Program 31(1–3):157–200 (Special Issue on Reasoning about Action and Change)

15. Kerschberg L, Weishar DJ (2000) Conceptual models and ar- chitectures for advanced information systems. Int J Appl Intell 13(2):149–164

16. Koubarakis M (2002) Querying temporal constraint networks: a unifying approach. Int J Appl Intell 17(3):297–311

17. Kowalski RA (1992) Database updates in the event calculus. J Logic Program 12:121–146

18. V. Lifshitz (1990) Frames in the space of situations, Artificial Intel- ligence.

19. LifshitzV(1991)Towardsametatheoryofaction.In:Proceedings of the international conference on principles of knowledge repre- sentation and reasoning, pp 376–386

20. Lutz, C. (2001) Interval-based Temporal Reasoning with General TBoxes. In IJCAI pp. 89—96.

21. McCain N, Turner H (1995) A causal theory of ramifications and qualifications. In: Proceedings of IJCAI-95, pp 1978–1984

22. McCarthy J, Hayes PJ (1969) Some philosophical problem from the standpoint of artificial intelligence. In: Machine intelligence, vol 4, pp 463–502

23. Miller R, Shanahan M (1999) The event calculus in classi- cal logic—alternative axiomatisations. Linkping Electron Articles Comput Inf Sci 4(16)

24. PatonNW,DiazO(1999)Activedatabasesystems.ACMComput Surv 31(1):63–103

25. Papadakis N, Plexousakis D (2002) Actions with Duration and Constraints: the Ramification Problem in Temporal Databases, Department of Computer Science, University of Crete, and Institute of Computer Science, FORTH, Greece.

26. Papadakis N, Petrakis P, Plexousakis D, Kondylakis H (2008) A Solution to the Ramification Problem Expressed in Temporal Description Logics, Science Department, Technological Educational Institute of Crete
2: Computer Science Department University of Crete & Institute of Computer Science, FORTH,Crete.

27. Papadakis, Plexousakis and Christodolou (2012) The ramification problem in temporal databases: a solution implemented in SQL , Appl Intell 36:749–767

28. Papadakis N, Plexousakis D , Antoniou Gr, Daskalakis M, Christodoulou Y (2007) the ramification problem in Temporal Databases: a solution Implemented in SQL.

29. Pinto J (1994) Temporal reasoning in the situation calculus. PhD Thesis, Dept of Computer Science, Univ of Toronto, Jan

30. Pinto J, Reiter R (1993) Temporal reasoning in logic program- ming: a case for the situation calculus. In: Proceedings of 10th int conf on logic programming, Budapest, Hungary, June 21–24

31. Plexousakis D, Mylopoulos J (1996) Accommodating integrity constraints during database design. In: Proceedings of EDBT 1996, Avignon, France, pp 497–513

32. Reiter R (2001) Knowledge in action logical foundations for spec- ifying and implementing dynamical systems. MIT Press, Cam- bridge

33. Reiter R (2001) Knowledge in action logical foundations for specifying and implementing dynamical systems. MIT Press, Cam- bridge.

34. Shanahan M (2001) The Ramification Problem in the Event Calculus, Department of Electrical and Electronic Engineering, Imperial College.

35. Sandewall E (2005) Actions as a basic concept in the Leonardo computation system. In: Proc of workshop on nonmonotonic rea- soning, actions and change, IJCAI.

36. Smith, C. Welty, and D. McGuinness, eds (2004) OWL Web Ontology Language Guide, W3C Recommendation

37. Thielscher M (1997) Ramification and causality. Artif Intell 89(1– 2):317–364

38. Thielscher M (1988) Reasoning about actions: steady versus sta- bilizing state constraints. Artif Intell 104:339–355

39. M.Winslett. (1988) Reasoning about action using a possible models approach, Computer Science Department University of Illinois AAAI-88 Proceedings.

40. Zhang, J. (2006) Description Logics and Time