PROTON: A Prolog Reasoner for Temporal ONtologies in OWL

Nikos Papadakis[1] ,Kostas Stravoskoufos[2],Evdoxios Baratis[2],Euripides Petrakis[1],Dimitris Plexousakis[3]
1: Department of Sciences, Techological Educational Institute of Crete,npapadak@cs.teicrete.gr
2:Department of Electronic and Computer Engineering, Technical University of Crete,(kstravo,dakis,petrakis)@intelligence.tuc.gr
3: Department of Computer Science, University of Crete, dp@csd.uoc.gr

Abstract

We present PROTON, a reasoner for managing temporal information over OWL ontologies. We adopt the so called 4d-fluent or 4-dimensionalist approach for representing temporal information in ontologies ie. for time points or intervals and for events that occur in time points or intervals. Also, we propose an extension to the situation calculus in order to encapsulate time. PROTON is implemented using this extension.

Key words: Ramification problem; Temporal Ontologies; Knowledge representation and reasoning;

1. Introduction

The Semantic Web, information is given a well-defined meaning, aiming at improving the communication between humans and computers. The first steps in weaving the Semantic Web onto the structure of the existing Web are already under way. In the near future, developers will use this new functionability as machines will be able to proccess and "understand" the data that they merely display at present. Two important techologies for developing the Semantic Web are already in place: XML and RDF. XML lets everyone create his their tags - hidden labels or to annotate Web pages.

Dealing with time and with the way information changes as time progresses is a well known problem in knowledge representation. Ontology representation languages like OWL and RDF are typically based on binary relations. Although temporal information (e.g. being an employee of company) can be directly represented in termporal ontologies, the fact that such relationships may change in time cannot directly be represented in OWL or RDF. Consequently, reasoning with ontology information represented in OWL or RDF cannot take temporal information into account. Assume for example a company share that has a price. The main problem is the representation of information that changes over time and that we need to represent the time and the value of the share at all time instances. Subsequently, we need to represent information changes as a result of time and to reason on such changes. This is exactly the problem this work is dealing with.

We introduce PROTON, a reasoning system for temporal ontologies in OWL. Answering queries about events that change in time is a distinguishing feature of PROTON. The system takes as input a temporal ontology in OWL and transforms it to triplets of the form (subject predicate object) using SWI-Prolog. Then the triplets are transformed into Prolog clauses. Proton is implemented using temporal situation calculus [21]. Subsequently, PROTON takes advantage of mechanisms inherent in Prolog for implementing the reasoner.

For existing OWL reasoners (like [8, 9, 10]) to deal with information about time, the information must explicitly be represented in the knowledge base. They cannot deal with temporal information and, subsequently, they cannot answer queries on temporal information that can be inferred from existing information. Specifically, an event record describing the event at the time specified by the query must exist in the knowledge base. The reasoner cannot deal with queries specifying the same event in a future time even though its value remains the same (the reasoner cannot infer an event value from its existing value). For instance, the question"what is the value of share X at time t1 can be answered only if the value of X at time t1 exists in the database. Existing (non temporal reasoners) cannot handle such common sense knowledge. PROTON handles all these problems.

The rest of the paper is organized as follows: In section 2 we describe how time is represented at the ontology layer in OWL. Also we present the formalisms for reasoning using an extension to Event Calculus for supporting temporal reasoning. Section 3 describes the main components of the proposed reasoner and its supported functionability followed by conclusions and issues for futher work in Section 4.

2. Related Work and Background

2.1. Time Representation in Ontologies

There are two main approaches for representing changes in information as a result of time in ontologies: Versioning [11] (the classic approach) and the more recent "perduranlist" approach [12]. Approaches such as OWL-TIME are only capable for representing temporal concepts and temporal relations rather than events and how they evolve in time. At the same time, they serve as "low-end" representations for other high semantic level representation approaches based on "concrete-domains" for the definition of new temporal languages such as TOWL [5, 6, 7].

Versioning suffers from information redundancy as it is based on information repetition. Also, changes in time can only be inferred by comparing the present and past states and cannot be directly represented in the ontology. The 4-dimensionalist (perdurandist) approach [12] solves both these problems. This approach distinguishes the world into two basic categories of entities: the endurants (physical objects such as cars, companies and people) and the occurants (events such as buying a car). The endurants represent time independent information (information that exists at all times) while the occurants have temporal parts. Endurants are represented by a set of properties that do not change over time (e.g. someone's DNA) and by a set of properties with values that depend on time. Concepts in time are represented as 4-dimensional objects with the 4th dimension being time. Time instances and time intervals are represented as instances of a time interval class which in turn is related with time concepts varying in time. Changes occur on the properties of the temporal part keeping the entities unchanged. Fig.1 illustrates a share management ontology with investors and shares. There are two actions (events): InvestorSellShare and InvestorBuyShare. The time slice instances of a share have (at any instance of time) the value of the share as property.

This world can be described by the ontology that has an Investor class, a Share class, an Event class (with the two actions described above as subclasses) and a FourDFluents class that has the TimeSlice and TimeInterval subclasses. The TimeSlice class holds all the time slices and the TimeInterval class holds the time intervals. When a new share is created, a new instance of the Share class is created as well. Whenever the price of this share changes, a new instance of the TimeSlice class is created and it is connected to the instance of the Share class. The datatype property value holds the new share value and it is connected to the new instance of the TimeSlice class. Finally,

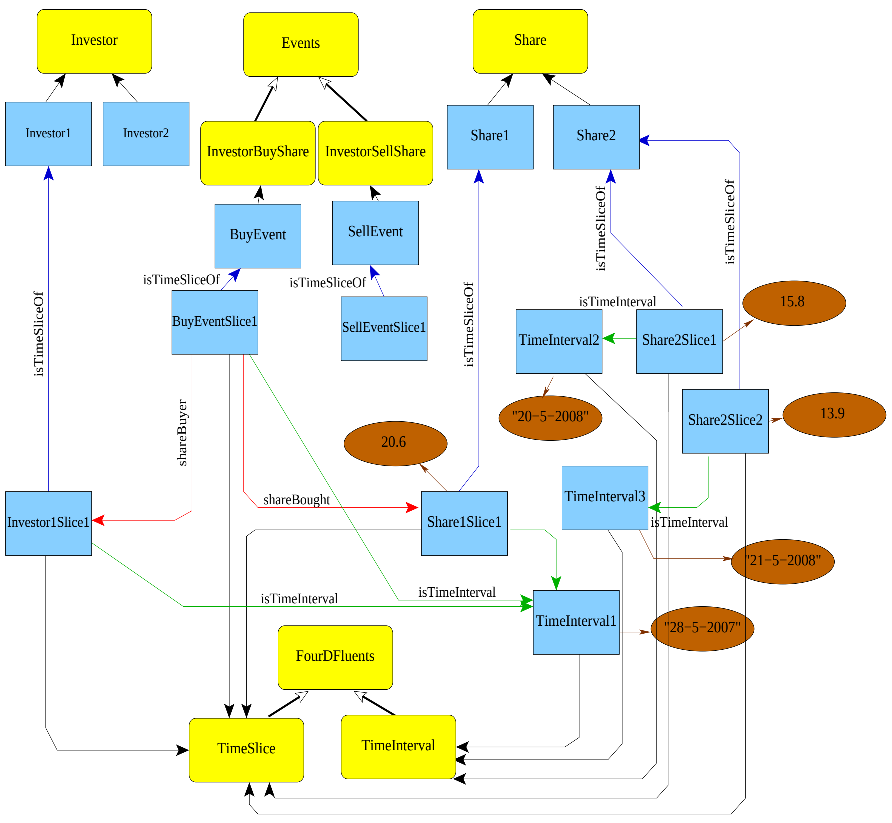


Figure 1: The Share Management Example

an instance of the TimeInterval class is created, and it is connected to the new instance of the TimeSlice class.

Assume that at time T an investor decides to sell a share. When such an event occurs (InvestorSellShare) three new instances of the TimeSlice class are created. The first one is connected to the investor, the second to the share and the third to the event (InvestorSellShare). The timeslice of the InvestorSellShare is also connected to the timeslice of the Investor through the object property shareSeller and it is also connected to the timeslice of the Share through the object property shareSell. Finally an instance of the TimeInterval class is created (which holds the time that the InvestorSellShare occurred) and it is connected to all the new instances of TimeSlice.

2.2. Formalisms

The most important formalisms for reasoning about actions and changes are: Situation calculus [21], fluent calculus [22], event calculus [13], action languages and action calculus, and temporal action logic (TAL) [20]. Situation calculus is the most popular approach. It is a second order language which has been designed for representing changes taking place in a world of interest. All changes that happen in a world are the result of the execution of some actions. The world is described by fluents (predicates and functions). A likely evolution of the world is a sequence of actions that is represented by a first order term which is called a situation.

Description Logic languages are viewed as the core of knowledge representation systems, considering both the structure of a DL knowledge base and its associated reasoning services. The standard inference problems that a DL reasoner is able to provide an answer for, are: subsumption, satisfiability, consistency and instantiation.The performance of reasoning algorithms depends on the expressiveness of the logic implemented.

2.3. A Temporal Description Logic

In this subsection we briefly introduce a class of interval-based temporal Description Logic, **TL-ALCF**, proposed by Artale and Franconi [1, 2]. They show that the subsumption problem is decidable and supply sound and complete procedures for computing subsumption.**TL-ALCF** is composed by the temporal logic TL which is able to express temporally quantified terms and the non-temporal Description Logic **ALCF** extending **ALC** with features (i.e., functional roles). In this formalism an action is represented through temporal constraints on world states where each state is a collection of properties of the world holding at a certain time. The intended meaning of **TL-ALCF** is explained in the following share managment system example:

InvestorBuyShare=$\diamond$(x y) ($\sharp$ f x)($\sharp$ m y). (($\star$ SHARE : Available)@x $\sqcap$ ($\star$ SHARE : Bought)@y)

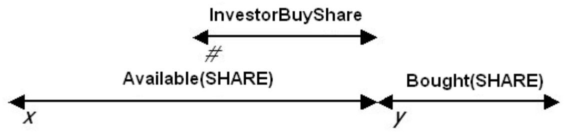
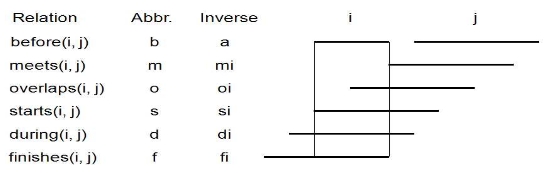


Figure 2: Temporal dependencies of the intervals in which InvestorBuyShare holds

Fig. 2 shows the temporal dependencies of the intervals in which the concept InvestorBuyShare holds. InvestorBuyShare denotes any action occurring at some interval involving a ⋆ SHARE that was once available and then it was bought (by an investor), where ⋆ SHARE is a parametric feature and Avaiable and Bought are non-temporal concepts. The parametric feature ⋆ SHARE plays the role of formal parameter of the action, mapping any individual action of type InvestorBuyShare to the SHARE to be bought, independently from time. Temporal variables are introduced by the temporal existential quantifier ⋄ excluding the special temporal variable ♯, usually called now, and intended as the occurring time of the action type being defined. The temporal constraints (♯ f x)(♯ m y) state that the interval denoted by x should finish with the interval denoted by ♯ and that ♯ should meet y, where f and m are Allen's temporal relations of Fig. 3.



Relation	Abbr.	Inverse
before(i, j)	b	a
meets(i, j)	m	mi
overlaps(i, j)	o	oi
starts(i, j)	s	si
during(i, j)	d	di
finishes(i, j)	f	fi

Figure 3: Allen's interval relationships

As the evaluation of concept at the interval, (⋆ SHARE : Available)@x and (⋆ SHARE: Bought)@y state that ⋆ SHARE: Available is qualified at x and ⋆ SHARE: Bought is qualified at y. In the concept description, the operator is the selection of feature, which is the role quantification that is interpreted as a partial function. Details onTL-ALCF syntax can be found in [1, 2].

2.4. DL Reasoners

Most current reasoners target the OWL-DL subset. Examples of well-known DL reasoners are FaCT [8] and Racer [3]. An interesting category of OWL-reasoners are based in Prolog. Other interesting categories of reasoners are Bossam [9] and Pellet [10] whose reasoning algorithm is based on description logic. These reasoners implement a tableau-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval, conjunctive query answering). However, they cannot handle concepts that evolve in time. For example, Bossam and Pellet handle time just like every other property. Also, they do not support relations over time intervals other than some basic ones such as comparissons between two time points, e.g. $func : after(time - const1, time - const2)$ returns true if $time - const1$ follows $time - const$ and $func : before(time - const1, time - const2)$ returns true if $time - const1$ precedes $time - const2$, $func : containedIn(time - const1, time - begin, time - end)$ returns true if $time - const1$ is in the duration formed by $time - begin$ and $time - end.$ Racer combines description logics reasoning with reasoning about temporal relations within the nRQL A-box query language [3].

Another category of reasoners includes Golog [23] a high-level agent programming language, Goncolog [24] (Concurrent Golog) which incorporates concurrency, interrupts, and exogenous actions into Golog. Hence, it allows the design of more flexible controllers for agents living in complex scenarios, etc. IndiGolog(Incremental Deterministic (Con)Golog) [25] is a high-level programming language where programs are executed incrementally to allow for interleaved action, planning, sensing, and exogenous events. IndiGolog provides a practical framework for real robots that must react to the environment and continuously gather new information from it. To account for planning, IndiGolog provides a local lookahead mechanism with a new language construct called the search operator.

GOLOG and CONGOLOG are two situation calculus based languages, extended versions of which can represent time explicitly. They use a sophisticated logic of actions (based on Situation Calculus),which allows the specification of effects of actions and constraints about the world, and can also reason with incomplete information about the world.The GOLOG and CONGOLOG [26, 27] interpreters are similar to hierarchical task networks (or HTNs) as they both take a (usually incomplete specified) plan as input and produces a complete plan as output. HTNs though , have some unique features that are absent in GOLOG/CONGOLOG :

1. It is straight-forward to express partial ordering between actions in HTN. In comparison the non-deterministic constructs in Golog/ConGolog are limited and do not allow us to easily specify a partial order between a set of actions and let the interpreter pick a particular total order.
2. Besides allowing pre and post conditions, HTNs also allow the specification of particular kind of temporal conditions, where fluents are required to hold between two (not necessarily consecutive) action steps.
3. Finally, since HTNs have been used extensively in real planning systems, it is perhaps important that the later execution languages are upward compatible with HTNs.

These temporal reasoners for KBs are ideaquate for OWL-time because the based in specific format of KBs. We propose PROTON a complete system which takes as input the knowldge in owl-time ontology and transform them to prolog clauses (which are the KB for temporal reasoning).

3. PROTON: A Prolog Reasoner for Temporal ONtologies

In the last five years we have conducted reasearch "the area of reasoning about action and change in temporal settings [15, 16, 17, 18, 19]. More specifically we have studied the frame, ramification and qualification problem in temporal databases. In these previous work we have proposed an extension the situation calculus in order to encapsulate time.

3.1. Our Extension to the Situation Calculus

In this work a temporal reasoner is built on situation calculus. Below we presented our Extension to Situation Calculus [15] in order to encapsulate time:

- In each fluent f, the argument L is added, where L is a list of time intervals $[a, b], a < b$.
- Each $[a, b]$ represents time instances x: $\{x\{a < x < b\}\}$.
- The fluent f holds true in all time intervals that are contained in list L.
- We define functions $start(a)$ and $end(a)$, where a is an event(action). The former returns the starting time point of action a while the later returns its end.
- Events are ordered as following: $a_1 < a_2$, when $start(a_1) < start(a_2)$.
- The predicate $eventHappen(a, t)$ means that action a is executed at time moment t.
- The so called "temporal situation" is defined as a situation with the list of time intervals for which the fluents are true.
- Function $Holding(S, t)$ returns all fluents that are true at time t. For a functional fluent the $Holding(S, t)$ returns the value which the function has at time point t. The situation S is a temporal situation.

- We define as a non-temporal situation S of a time point t the situation $S = Holding(S', t)$. [1]

- A transition from a situation to another could happen when the function $Holding(S,\ t)$ returns a different set. [2]

PROTON is based on the idea of transforming the representation of a temporal ontology into a set of equivalent prolog predicates. Queries concerning temporal information are addressed towards the prolog database. PROTON is capable of manipulating relationships between time instances or time intervals as well as for computing inferences

3.2. The Implemetation of PROTON

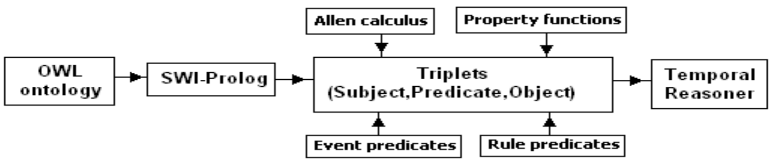


Figure 4: PROTON Architecture

Figure 4 illustrates the architecture of PROTON. It takes a temporal ontology in OWL as input and transforms it into Prolog predicates. PROTON consists of several modules, the most important of them being the following:

1. SWI-Prolog for transforming temporal OWL concepts to prolog clauses
2. Allen calculus for computing relations over time intervals
3. A set of functions for computing property values at any instance of time
4. A set of predicates than determine when an event takes place and
5. A set of predicates which execute rules when an event takes place or when a change in the value of a property occurs.

[1]Notice that many different temporal situations could refer to the same situation at a specific time point.

[2]For example $\{f_1([[7,9]], \neg f_1([[10,\infty]]), ...\}$ means that at time point 10 we have transition from one situation to another because the truth value of the fluent f_1 changes. Notice that the transition happens without an action taking place.

The Transformation from temporal OWL to clauses

PROTON is based on transforming OWL statements to facts of the form “predicate(subject,object)”. First,we transform the OWL ontology to triplets (**subject predicate object**) using SWI-Prolog.The following is the list of possible transformations:

1. Triplets of the form: (S,’http://www.w3.org/1999/02/22-rdf-syntax-ns#type’,O) are transformed to O(S)
2. Triplets of the form: (X,Y,literal(Z)) are transformed into Y(X,Z)
3. Triplets of the form: (X,Y,Z) are transformed into facts Y(X,Z)
4. Triplets of the form: (S,’http://www.w3.org/2000/01/rdf-schema#subClassOf’,O) are transformed into **O(X) :- S(X)**
5. Triplets of the form: (S,’http:/www.w3.org/2000/1/rdf-schema#subPropertyOf’,O) are transformed to **O(X,Y) :- S(X,Y)**

For example, an OWL entity of the form

```
<owl:ObjectProperty rdf:ID=”tsTimeInterval”>
 <rdfs:domain rdf:resource=”#tsTimeSliceof”/>
 <rdfs:range rdf:resource=”#TimeInterval”/>
</owl:ObjectProperty>
```

is first transformed into the triple such as
(tsTimeSliceof, onproperty, ...)
(tsTimeSliceof, disjointwith,tsTimeInterval)
and then to the following KB facts:

$$fact(disjointwith('tsTimeSliceof',' tsTimeInterval')).$$

$$fact(onproperty('\_\_Description2', tsTimeSliceOf)).$$

After having transformed all OWL entities to facts, we can take advantage of the mechanisms inherent to Prolog for the implementation of the reasoner. As such, the KB in Prolog can be further enriched with the necessary inference rules for reasoning. The above process can be modified to produce a knowledge base for the reasoning model in use (i.e., situation calculus in this work).

In the following we discuss PROTON using the share management system of Section 1 as an example. The KB consists of predicates produced automatically, as discussed above. Some of these predicates are common for all ontologies while others are application specific. The temporal relations of Allen (before, equals, meets, overlaps,during, starts,finishes) for handling temporal relationships are also implemented. First, we introduce two time-handling predicates:

$date(D, t)$**: .** The first argument is a time interval and the second is a timestamp. In our implementation timestamps are strings of the form "$yyyy - mm - ddThh : mm : ss$" and time intervals are lists of the form $[t1, t2]$ where $t1$ is the starting timestamp of the time interval and $t2$ is the ending timestamp. The predicate succeeds if $t1 < t < t2$ (if the second argument belongs in the time interval D.

$before(t1, t2)$**: .** Both arguments are timestamps. The predicate determines the earlier between those two timestamps. It succeeds if timestamp $t1$ is earlier than $t2$.

Next we show how property values are computed. For this, the following predicates are defined:

$value(X, Y, Z)$**:.** Argument X is the name of the share in a string format (called a caption), argument Y is and integer (the value of caption X) and argument Z is a timestamp. The predicate succeeds if the value of caption X is Y at the timestamp Z. An example where the predicate succeds is illustrated in Fig. 5a.

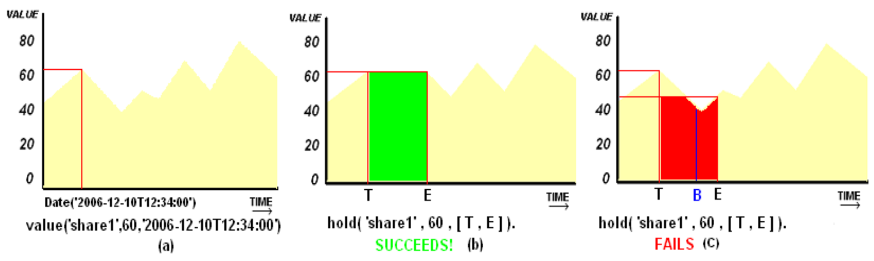


Figure 5: Examples of the hold and value predicates

$hold(X, Y, D)$**:.** [3] Argument X is a string (caption), argument Y is the value of the caption and D is a time interval. The predicate succeeds if the caption X holds the value Y during the time interval D and change it's value at the end of D. More specifically,this means that Value(X,Y,t) succeeds for every

[3]Here we implement our representation $f(L)$ that we have defined in section 3.1

t, $t1 < t < t2$ and $Value(X, Y, t2)$ fails. For example, in Fig. 5b, '$share1$' has value 60 in the time interval $[T, E]$ and the predicate evaluates true. while in Fig. 5c the predicate $hold(\text{'}share1', 60, [T, E])$ evaluates to false because the share doesn't maintain it's value during the time interval $[T, E]$.

In Fig. 6 we present predicates that handle events. In our share management system example an event occurs when an investor buys or sells a share.Every event E is associated to a unique pair of an investor X and a share/caption Y which are connected with timeSlices.

$eventHappen(E, t, L)$:. [4] It takes as input an event E, a timestamp t and returns list L (initially empty) in its output. The predicate succeeds if an event E (e.g.,buying or selling a share) occurs at timestamp t. This means that there is an investor I who buys or sells a share S at timestamp t. The investor and share are connected with each other with time slices and then they are connected to the event the same way (Fig.4). If the predicate is succesfull then the investor and the share that are associated to the event are added to list L ($L = [I, S]$).

$sales(t, V, L)$:. this predicate is an example of how indirect effects of actions (events) are captured. Takes as input a timestamp t, an integer value $V > 0$ and a list L as in the previous predicate. The predicate succeeds if there is at least one share S in the database that its price has increased by $V\%$ (compared to its previous price) until the timestamp t. In order to check which shares satisfy this constraint, all the shares in the database are scanned one by one and those that make the predicate succeed are added to the L list.

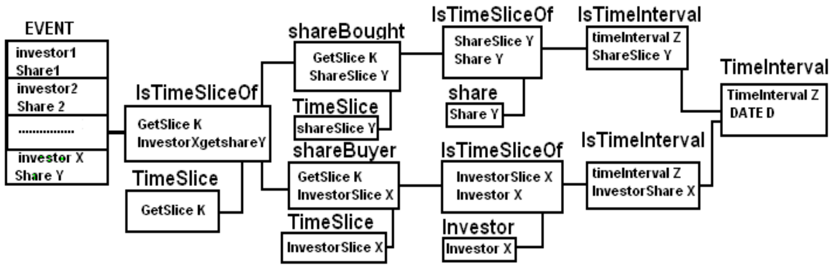


Figure 6: The correspondence among Time-Events and Situations

[4]Here we implemend the predicate $eventHappen$ that we have defined in section 3.1

$holding(CL, caption_i, value_i)$**:.** [5] Takes as input the caption list CL which is a list with all the captions in the database, a list with a pair of a caption and a value $(caption_i, value_i)$ and a timestamp t. The predicate determines the value of all captions at a specific timestamp (This is very important in order to address the frame problem). The predicate scans the database and determines the value of each caption using the following algorithm: If a $value(caption_i, value_i, t)$ predicate exists for timestamp t then the captions value is defined by this predicate. If there is not a $value(caption_i, value_i, t)$ for timestamp t, then the caption naturaly has the value that was determined by the last occurance of the value predicate for this caption(the last time the caption's value was changed) at a timestamp $t1 < t$. To find this value the holding/3 predicate calls the $find\_last\_value$ predicate.

$find\_last\_value(CP, V, T)$**:.** Takes as input a caption CP,a value V and a timestamp t. The predicate scans the knowledge base to find a past value $V$of caption CP with the precondition that the timestamp $t1$ when the caption changed value is $before$ the timestamp $t$and there is no other timestamp $t2$ so that $t1 < t2 < t$.

The $find\_last\_value$*.* predicate implements one very important feature of PROTON because it makes it capable to answer questions which cannot be answered by other reasoners. For example let's asume that we have a share $Share_1$ and the following fact is registered in the database : $value(Share_1$,50,'*11-5-2009T23:00:00*'). If we wanted to determine the value of $share_1$ at '*12-5-2009T23 :00:00*' (which is after *11-5-2009T23:00:00*) we would get a negative answer from most reasoners while PROTON would be able to retrieve the previous value of $share_1$ and answer '50'.

run **:.** It is the simulator of the share management system and it takes not arguments The predicate scans the database for events and executes the aproporiate actions for each event.This is illustrated in Fig. 7. For every timestamp *date_i*,that exists in the knowledge base, the *run* predicate determines:

1. The value of all the shares at this timestamp (using the $Holding$ predicate)
2. All the events that occur (using the $EventHappen$ predicate)

[5]Here we implemend the predicate *holding* that we have defined in section 3.1

3. All the shares that have raised there value for more than a specified value (using the *sales* predicate).

For example lets assume that our database consists of the following : Two date predicates ("20-5-2009T12:00:00" and "20-5-2009T14:00:00"), two shares ("share_1" with value "100" and "share_2" with value 200) and one investor ("investor_1"). Also lets assume that the following two events occur:

1. "Investor_1" buys "share_1" at "20-5-2009T12:00:00".
2. "share_2" raises it's value by 50%.

Now, by executing the run predicate , our reasoner will answer the following :

```
Date : 20-5-2009T12:00:00
share_1 100
share_2 200
[investor\_1,share\_1]

Date : 20-5-2009T14:00:00
share_1 100
share_2 300
[share_2]
```

This is a small example using only two date predicates.In a more realistic simulation of the system,this process continues for every date predicate in the knowledge base.

PROTON can be generalized to handle any application by supplying predicates specific for the application in use. Notice that only application specific predicates (such as *sales*) need to be changed.

4. An example of simulator

In this example we simulate some instant and periodic events that may occur to a public employee. Instant events are events that may occur at a random time point and periodic events are events that always occur after a predefined period of time passes (e.g the employee receives salary every month). Each of the events has direct and indirect effects on the employee's status. Below we describe the event predicates and their effects in more detail :

$misdemeanor(P, t)$: . When the employee P commits a misdemeanor his status changes to *illegal* for a time period of $[t, t + n]$. During this period the employee cannot receive salary and all the periodic events are postponed by n time units. For example lets asume that an employee takes his salary every 60 time unints and he commits a misdemeanor and becomes suspended during the time interval $[40, 45]$. As an indirect effect of his misdemeanor, the employee will receive his salary 5 time unints later than usual, at time point 65.

take_pardon(P, t) :. This event can only occur if the employee P is under suspension. When it occurs the employee's status turns to legal again. Using the previous example , if the employee P is suspended during the period $[40, 45]$ and he receives a pardon at the time momment 43, then he will receive his salary at time momment 63.

good_grade(P, t):. When an employee P receives a good grade and he is not suspended or has got a bad grade then he receives a bonus at time t.

bad_grade(P, t):. When an employee P receives a bad grade at time t then he does not receive a bonus.

The following two predicates simulate the periodic events of our example :

$take_increase(P, t)$:. This predicate defines if the employee P can receive an increase to his salary at time t. Every empoyee receives an increase to his salary every $t1$ time units. The predicate is executed at every time momment increasing an internal counter until the counter equals $t1$. When this happens then ,as an direct effect, the employee receives an increase to his salary and the counter is set to 0 again. The increase of the salary may be affected by misdemeanor commitments of the employee. In such cases the increase is postponed. For example let's asume that the employee usually receives an increase every 100 time units and that he commits a misdemeanor during the intervals $[30, 35]$ $[60, 65]$. Under ideal circumstances (if he had not commited any misdemeanors) the employee would receive his first increase at time point 100 but now after two misdemeanors he will receive the increase at time point 110.

$take_promotion(P, t)$:. Similar to the take_increase predicate this predicate is used to increase an employee's ranking in the hierarchy after a predifined number of $t2$ time units. Again, if the employee has commited any misdemeanors, the promotion is postponed. For example if the employee's current ranking is 0 and the promotion period is 100 time units if the employee commits a misdemeanor and becomes suspendedfor the period $[30, 35]$ then he will receive a promotion at time point 105.

After defining the application dependant predicates, we can use the *run* predicate which we described before as a simulator. The *run* predicate will go through all the date predicates of the knowledge base executing the approporiate rules for each date. In the following lines we show an example of such a simulation test :

Before the test we need to make some clarifications about the events and the employees.

1. We use 3 employees in this test : p1,p2,p3.
2. The test is ran in the time interval [0,100]
3. Every employee is considered to be legal and also a good employee at the beginning of the test so the legal and good_employee initial predicates look like that :
   (a) $p1 : legal(p1, [[0, 100]]).good\_employee(p1, [[0, 100]])$.
   (b) $p2 : legal(p2, [[0, 100]]).good\_employee(p2, [[0, 100]])$.
   (c) $p3 : legal(p3, [[0, 100]]).good\_employee(p3, [[0, 100]])$.
4. The initial predicates for the salary and promotion status for the employees are :
   (a) $p1 : salary(p1, 500, 0).p1 : position(p1, 0, 0)$.
   (b) $p2 : salary(p2, 600, 0).p2 : position(p1, 1, 0)$.
   (c) $p3 : salary(p3, 700, 0).p3 : position(p1, 2, 0)$.

   The salary and position values are incremented by one every time the take_increase and take_promotion events occur.
5. The initial time intervals for this test are :
   (a) $timeInterval([0, 19],'timeInterval\_1')$.
   (b) $timeInterval([20, 39],'timeInterval\_2')$.
   (c) $timeInterval([40, 54],'timeInterval\_3')$.
   (d) $timeInterval([55, 79],'timeInterval\_4')$.
   (e) $timeInterval([80, 99],'timeInterval\_5')$.

1. 10. misdemeanor p3 : The employee p3 becomes suspended for the next 5 time units so we have a change in the data base , $legal(p3, [[0, 100]])$ changes to $legal(p3, [[0, 10], [15, 100]])$

2. 25. take increase p1,p2 : During the time interval [0-24] the salary predicates of the employees p1,p2 were increased by one in every time stamp salary(p,s,n) changes to salary(p,s,n+1) so when t=24 we have :
   (a) $salary(p1, 500, 24)$ changes to $salary(p1, 501, 0)$
   (b) $salary(p2, 600, 24)$ changes to $salary(p2, 601, 0)$

   So the employees p1 and p2 receive their increase normally but p3 who had been suspended for 5 time units has at this point this salary predicate: salary(p3,700,19) because his salary increase counter stopped at t=10 and started to count again at t=15.
3. 30. take increase p3 : After 5 time units from the normal salary increase time employee p3 receives an increase and his salary predicate changes : $salary(p3, 700, 24)$ changes to $salary(p3, 701, 0)$
4. 41. misdemeanor p3 : Employee p3 commits a misdemeanor and turns illegal for the next 5 time units. We have a change in the legal predicate : $legal(p3, [[0, 10], [15, 100]])$ changes to $legal(p3, [[0, 10], [15, 41], [46, 100]])$.
5. 43. Employee p3 (who is currently illegal) receives a pardon. So , he becomes legal again and we have a change in the legal predicate. $legal(p3, [[0-10], [15-41], [46-100]])$ changes to $legal(p3, [[0-10], [15-41], [43-100]])$.
6. 50. Employees p1 and p2 receive their salary increase normally because they havent turned illegal until now.
   (a) $salary(p1, 501, 24)$ changes to $salary(p1, 502, 0)$
   (b) $salary(p2, 601, 24)$ changes to $salary(p2, 602, 0)$

   Again here, employee p3 does not receive an increase cause he is postponed for 7 days because of his two misdemeanors.
7. 57. Employee p3 receives his salary increase after 7 days of the time when the normal increase occurred. This is because he committed two misdemeanors (the first lasted 5 time units and the second only 2 because of the pardon that occurred). So : $salary(p3, 701, 24)$ changes to $salary(p3, 702, 0)$
8. 60. misdemeanor p1: Employee p1 is suspended for 5 time units and the legal predicate changes : $legal(p1, [[0, 100]])$ changes to $legal(p1, [[0, 60], [65-100]])$. The employee p1 was about to receive promotion in the next time unit but this misdemeanor will postpone his promotion 5 time units later.

9. 61. take promotion p2 : This is the first time that the employees would receive a promotion if they hadnt been committing misdemeanors. So , only p2 receives a promotion for now and the position predicate changes as follows : $position(p2, 1, 60)$ changes to $position(p2, 2, 0)$
10. 66. take promotion p1 : After being postponed by 5 time units the employee p1 receives his promotion. $position(p1, 0, 60)$ changes to $position(p1, 1, 0)$
11. 68. take promotion p3 : After 7 time units (5 +2) employee p3 receives his promotion also: $position(p3, 2, 60)$ changes to $position(p3, 3, 0)$
12. 75. take increase p2 : Employee p2 receives a salary increase : $salary(p2, 602, 24)$ changes to $salary(p2, 603, 0)$
13. 80. take increase p1 : Employee p1 receives a salary increase : $salary(p1, 502, 24)$ changes to $salary(p1, 503, 0)$
14. 82. take increase p3 : Employee p3 receives a salary increase : $salary(p3, 702, 24)$ changes to $salary(p3, 703, 0)$
15. 84. bad grade p2 : Employee p2 receives a bad grade and is considered a bad employee. If an employee gets a bad grade he is not able to receive bonuses.
16. 87. misdemeanor p1,p3 : Employees p1,p3 both commit a misdemeanor at the same time. So the legal predicates change like this :
    (a) $legal(p1, [[0, 60], [65{-}100]])$ changes to $legal(p1, [[0, 60], [65{-}87], [92, 100]])$
    (b) $legal(p3, [[0 - 10], [15 - 41], [43 - 100]])$ changes to $legal(p3, [[0 - 10], [15 - 41], [43 - 87], [92, 100]])$.
17. 90. good grade p2 : Employee p2 receives a good grade and is able to receive bonuses from now on.

5. Conclusion

In this paper we presented PROTON a reasoner implemented in prolog for managing temporal information over OWL ontologies. As future work we intend to extend PROTON to handle the ramification problem (e.g., handle the indirect consequences of actions) and also apply PROTON to application domains such as medicine and finance.

Acknowledgements

The authors are supported by project TOWL (FP6 STREP, Contract No. 26896) of the EU.

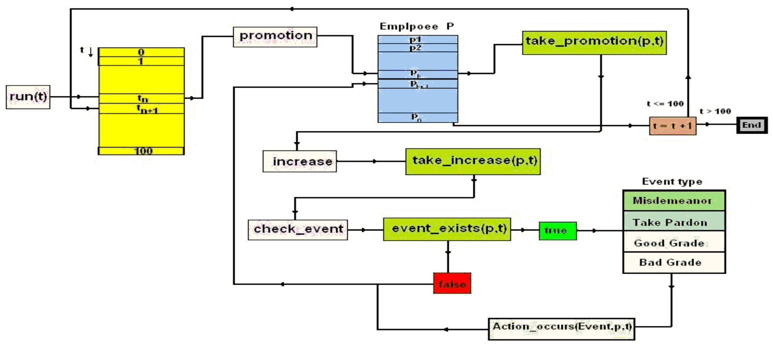


Figure 7: PROTON simulator.

References

[1] A. Artale and E. Franconi, ”Temporal description logics”, Handbook of Time and Temporal Reasoning in Artificial Intelligence. MIT Press, 2000.

[2] A.Artale and E.Franconi, ”A Temporal Description Logic for Reasoning about Actions and Plans”, JAIR, 1998, volume 9, pages 463-506.

[3] V. Haarslev, R. Moller, and M. Wessel. Querying the Semantic Web with Racer + nRQL. In KI 2004 Germany, 2004.

[4] T.Matzner, P.Hitzler “Any-World Access to OWL from Prolog”, KI 2007, pp. 84-98.

[5] http://www.towl.org/

[6] V. Milea, F. Frasincar, U. Kaymak and T. di Noia. An OWL-Based Approach Towards Representing Time in Web Information Systems. In Proceedings of the 4th International Workshop on Web Information Systems Modelling (WISM 2007) and the 19th International Conference on Advanced Information Systems Engineering (CAiSE 2007), pages 791-802, Tapir Academic Press, 2007.

[7] T. Di Noia, E. Di Sciascio, F. M. Donini. Semantic Matchmaking as Non-Monotonic Reasoning: A Description Logic Approach Journal of Artificial Intelligence Research (JAIR), Volume 29, page 269–307 - 2007

[8] D.Tsarkov and I.Horrocks. FaCT++ Description Logic Reasoner: System Description. In IJCAR 2006, vol. 4130, pp 292-297.

[9] Bossam Rule/OWL reasoner http://bossam.wordpress.com/

[10] Pallet an owl-reasoner, http://pellet.owldl.com/

[11] J.Avery, J.Yearwood "DOWL: A DYNAMIC ONTOLOGY LANGUAGE", University of Ballarat,2003

[12] C.Welty , R.Fikes and S.Makarios "A Reusable Ontology for Fluents in OWL", IBM Research paper, 2004.

[13] R.A. Kowalski, F. Sadri, Reconciling the event calculus with the situation calculus, Journal of Logic Programming 31 (1–3) (1997) 39–58.

[14] Nikos Papadakis and Yannis Christodoulou. A tool for addressing the ramification problem in spatial databases: A solution implemented in SQL. Expert Systems with Applications, Volume 37 , Issue 2 (March 2010), pp. 1374-1390.

[15] N.Papadakis,D.Plexousakis.Action with Duration and Constraints:The Ramification problem in Temporal Databases. IJTAI, 12, 315-353,2003.

[16] Nikos Papadakis and Dimitris Plexousakis. Action with Duration and Constraints: The Ramification problem in Temporal Databases. 14th IEEE ICTAI, 2002 , Washington D.C.

[17] Nikos Papadakis ,Dimitris Plexousakis and Grigoris Antoniou, The Ramification problem in Temporal Databases: Chaning the Beliefs About the Past, Journal of Data and Knowledge Engineering, 59, pp. 397-434, November 2006.

[18] Nikos Papadakis Grigoris Antoniou Dimitris Plexousakis. The Ramification Problem in Temporal Databases: Concurrent Execution with Conflicting Constraints, 19th IEEE ICTAI, 2007 pp 274-278, Greece.

[19] N. Papadakis, D. Plexousakis, G. Antoniou, M. Daskalakis and Y. Christodoulou, The Ramification Problem in Temporal Databases: A Solution Implemented in SQL, pp. 381-388, ISMIS 2008.

[20] E. Giunchiglia, V. Lifschitz, Action languages, temporal action logics and the situation calculus, Elect. Artic. in Computer and Information Science 4,1999.

[21] R. Reiter, Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, MIT Press, Cambridge, MA, 2001.

[22] M. Thielscher, From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem, AI, pp. 277–299,1999.

[23] C. Fritz, J. Baier S. McIlraith, ConGolog, Sin Trans: Compiling ConGolog into Basic Action Theories for Planning and Beyond, In Proceedings on the 11th International Conference on Principles of Knowledge Representation and Reasoning, 600–610, Sydney, Australia, September 16–19, 2008.

[24] C. Fritz S. McIlraith, Decision-Theoretic GOLOG with Qualitative Preferences, In Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR06), 153-163, Lake District, UK, June 2006.

[25] S. Sardina, G. De Giacomo, Y. Lesperance and H. J. Levesque, On the Semantics of Deliberation in IndiGolog from Theory to Implementation, Annals of Mathematics and Artificial Intelligence, Volume 41, Numbers 2-4 / August, 2004

[26] Extending ConGolog to Allow Partial Ordering, Section III:Agent Languages, Book Intelligent Agents VI. Agent Theories Architectures, and Languages, Volume 1757/2000,Lecture Notes in Computer Science, December 2006.

[27] R. Reiter Sequential, temporal golog. In Principles of Knowledge Representation and Reasoning: Proceedings of the 6th International Conference (KRgS), 547-556, 1998.