CIDOC-CRM compatible
Temporal representation

December, 2009

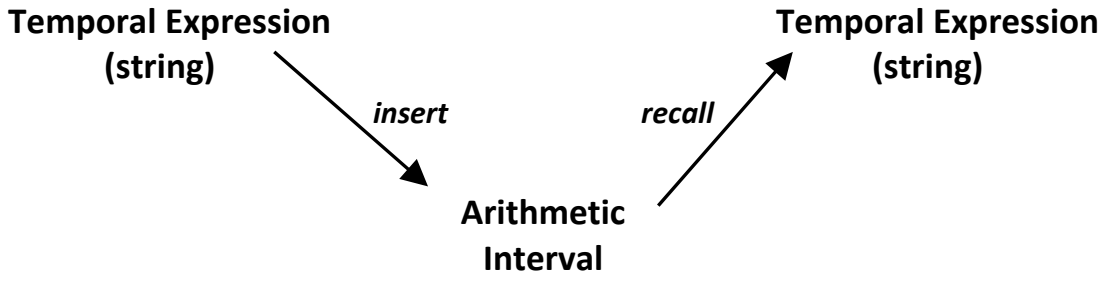
Information Systems Laboratory*
Institute of Computer Science (ICS)
Foundation for Research and Technology - Hellas (FORTH)

* This work is based on the work of Anthi Yiortsou [4] and Christina Gritzapi [5] .

Table of Contents

1 Introduction

In this report we describe a library which implements a primitive time data type. This library is available as source code, and as a DLL (dynamic-link library) for Windows platforms. An interval-based time model is adopted. Users declare temporal elements through certain temporal expressions that follow the rules of the Art and Architecture Thesaurus [3]. These expressions are parsed and converted into two integers, the lower and the upper boundaries of the corresponding arithmetic interval. These integers should be stored in our system. Upon recall of a temporal element the system reconstructs the exact temporal expression that was used to insert this element into the system. Graphically the above mechanism works as follows:



The originality of the above mechanism is the way it reconstructs the users' temporal expressions when temporal elements are recalled. For instance, "19th century" is expanded into 1800/1/1, 1899/12/31. Multiple expressions result in the same arithmetic interval, these expressions differ however in the intended precision. On recall the above data is represented again as "19th century" maintaining the precision of information.
Moreover the arithmetic intervals encoding temporal information and users' expressions maintain temporal relations and relations of distances.

2 How to write a Temporal Expression

There is a set of expressions that can be used in order to declare a Time. These expressions follow the rules of the Art and Architecture Thesaurus [3]. They can be grouped as follows:

2.1 Date expressions

These expressions have the format [Year Month Day]. One can declare only the Year, the Year and the Month or the whole date. Months must be designated verbally rather than numerically. If it is the case that the date is not fully declared, (i.e. only the year is given) the interval representation of this declaration is the minimum interval that can fully contain the given information. For example if the declaration [1974] is used, its internal representation will be the interval with bounds 1974/1/1 and 1974/12/31 respectively. This interval is the minimum one that contains every date within 1974. The following examples demonstrate the use of date expressions:

Declaration	Interval Representation
[1974 March 6]	(1974/3/6, 1974/3/6)
[1974 March]	(1974/3/1, 1974/3/31)
[1974]	(1974/1/1, 1974/12/31)

The abbreviated form of the era designation "Before Common Era", BCE, in full capitals and with no periods, is used for all dates before the year 1 (i.e. [1453 BCE]).

2.2 Decade expressions

These expressions declare the desired decade either absolutely (i.e. [decade of 1970]) or relatively (i.e. [first decade of 19th century]). The internal representation of these expressions is again the minimum interval that contains the declared decade. The following examples demonstrate the use of decade expressions:

Absolute declaration:

Declaration	Interval Representation
[Decade of 1970]	(1970/1/1, 1979/12/31)
[Decade of 1970 BCE]	(1979/1/1, 1970/12/31)

Relative declaration:

Declaration	Interval Representation
[First decade of 20th century]	(1900/1/1, 1909/12/31)
[First decade of 20th century BCE]	(1909/1/1, 1900/12/31)

For the relative declaration the keywords (first/second/third/ .../ninth/last) are used.

There is an exception in the correspondence between the decade expressions and their interval representation. For the first decade first century CE (the first decade first century BCE) the interval representation begins at the year 1 (ends at year 1). For example the expression [First decade of 1st century] is represented as (1/1/1, 9/12/31). This is so, because we do not expect year zero as a legal year.

2.3 Century expressions

These expressions have the format [(Number) century] (i.e. [19th century]).
Here are some century expression examples:

Declaration	Interval Representation
[1st century]	(1/1/1, 999/12/31)
[2nd century BCE]	(199/1/1, 100/12/31)
[16th Century]	(1500/1/1, 1599/12/31)

There is an exception in the correspondence between the century expressions and their interval representation. For the first century CE (the first century BCE) the interval representation begins at the year 1 (ends at year 1).
For example the expression [1st century BCE] is represented as (999/1/1, 1/12/31).

2.4 Period Expressions

These expressions declare a time period either absolutely or relatively. The absolute period expressions, declare the beginning and the ending of the time period explicitly. The beginning and the ending expressions can be any of the expressions mentioned above (date, decade e.t.c) and are separated with a dash. Their interval representation has lower bound the lower bound of the beginning expression and upper bound the upper bound of the ending expression. The following examples demonstrate the use of absolute period expressions:

Declaration	Interval Representation
[16th century - decade of 1970]	(1500/1/1, 1979/12/31)
[14th century BCE - 1300 August CE]	(1399/1/1, 1300/8/31)
[second decade of 1400 - 3rd century BCE]	(1419/1/1, 200/12/31)

The BCE designation should appear in the ending of the period expression if both the beginning and the ending expressions are before the year 1. If only the beginning expression is before the year 1, then the BCE designation is used for the first expression while the CE (Common Era) expression is used for the last one (as can be seen in the examples above).

A time period can be declared relatively as well. Relative period expressions have the following formats.

- **[Early/Mid/Late (number) century]**
  The interval representation for each of the keywords early, mid and late is the interval (0/1/1, 40/12/31), (30/1/1, 70/12/31) and (60/1/1, 99/12/31) respectively. This means that for the declaration [mid 16th century] the interval that represents the given information is (1530/1/1, 1570/12/31).

- **[1st/2nd half (number) century]**
  These expressions correspond to the intervals (0/1/1, 60/12/31) and (40/1/1, 99/12/31) respectively (i.e. the declaration [1st half 16th century] corresponds to the interval (1500/1/1, 1560/12/31)). Note that the duration of the period assigned to each interval is 60 years rather than 50 as someone would expect. This is so, because these periods represent uncertainty periods, thus their boundaries should overlap (one cannot determine exactly when the first half ends and when the second begins).

- **[1st/2nd/3rd/4th quarter (number) century]**
  For this expressions the intervals (0/1/1, 27/12/31), (25/1/1, 52/12/31), (50/1/1, 77/12/31) and (75/1/1, 99/12/31) are assigned respectively (i.e. the interval (1525/1/1, 1552/12/31) is assigned to the declaration [2nd quarter 16th century]).

All the keywords can be written with whatever combination of upper and lower case letters. So the declaration [Decade of 1970] can be written [decAde OF 1970] or [DECADE of 1970].

3 Handling the time primitive data type

In order to use the time primitive expressions we need to declare two integer variables that represent the lower and the upper boundaries of an actual time expression.
We provide three functions that convert a time expression from its alphanumeric format to its equivalent numeric format, and vice versa: *time_parse(), present(), and get_time_error_message().* These functions are available in C and Java programming languages.

time_parse()

Function *time_parse(),* takes as its first argument the time expression in its alphanumeric format and convert it into two integer (second and third) arguments that represent the lower and the upper boundaries of an actual time expression. It returns 0 on success.

present()

Function *present()* takes four arguments. The first two arguments represent the numeric lower and upper boundaries of the temporal element. The original alphanumeric temporal expression will be reconstructed using these numeric boundaries and will be returned to the third argument. The forth argument represent the language (English or Greek) in which the alphanumeric temporal expression will be presented.

get_time_error_message()

Function *get_time_error_message()* returns in its first argument the error message generated in case *time_parse()* did not succeed.

3.1 C language interface

Each c-file that makes calls to the time primitive routines should have the declaration:

```
#include "time_dll_api.h"
```

The above file contains the declarations of the functions (routines) that handle the time primitive expressions. These function declarations are:

```
void present (int lower, int upper, char *string, int lang);
int  time_parse (int *lower, int *upper, char * parse_str);
int  WINAPI get_time_error_message (char *str);
```

These time primitive functions come as a DLL library (Dynamic Link Library), named **time_dll.dll**. The time primitive DLL can be linked to a simple test program (named: **test_dll**, source code file: **test_dll.cpp**) with the following command:

```
gcc -o test_dll test_dll.cpp -L./ -ltime_dll
```

Note: time_dll.dll should be in windows %PATH% in order to be dynamically located and linked, upon program execution.

3.2 Java language interface

Each java-file that makes calls to the time primitive routines should have the declaration:

```
import sistime.*;
```

The java package **time_japi.jar** contains the object **Time**, which provides the functions (routines) that handle the time primitive expressions, along with **IntegerObject** and **StringObject** objects. These function declarations are:

void present(int lower, int upper, StringObject time_str, int lang);
int time_parse(IntegerObject lower, IntegerObject upper, String parse_time_str);
int get_time_error_message(StringObject message);

The java package **time_japi.jar** comes with two DLL libraries (Dynamic Link Libraries), named **time_dll.dll, sistime_Time.dll**, that contain the native c code, and wrapper code for the handling of time primitive.

The time primitive package can be compiled with a simple test program (named: **test**, source code file: **test.java**) with the following commands:

```
set PATH=%JAVAPATH%;%PATH%;..\class
set CLASSPATH=..\class\time_japi.jar;.

javac test.java
java test
```

Note: time_dll.dll and sistime_Time.dll should be in windows %PATH% in order to be dynamically located and linked, upon program execution.

4 Using Time Primitive in queries

For the presentation of each temporal element of time primitive type in a repository two fields are needed. These fields should have integer types and represent the lower and upper boundaries of a time expression.

4.1 Insertion of temporal elements into a repository

When we want to insert a temporal element in the repository:

1. We call the lexicographic analyzer *time_parse(),* which converts the temporal expression, used to define the temporal element, into its equivalent lower and upper numbers.
2. We store these two numbers to the appropriate fields

4.2 Presentation of a temporal expression which has been stored in a repository

When we retrieve a time value, which is stored in the repository, we have to reconstruct the corresponding temporal expression from its lower and upper boundaries. In order to achieve this we simply call the routine *present ().*

4.3 Temporal operators

In order to retrieve time primitive elements from a repository, two sets of temporal operators have been implemented. The first set contains operators that express relations between complete intervals. These operators are defined by J. F. Allen ([1],[2]) (***Allen operators*** from now on). The second set contains operators expressing possible relations between values within uncertainty intervals (***Uncertainty operators*** from now on).
Below we describe these operators. Especially for the ***Uncertainty operators,*** we show how they can be implemented using the ***Allen operators.***

Suppose that the user's temporal expression "User's Expression" is converted to the integers A and B, A being the beginning of the time declared by the temporal expression and B the end. Also suppose that with the term "Stored Temporal Element" we reference the values of the fields time_lower and time_upper of the repository. Finally suppose that the order of the "Stored Temporal Element", "User's Expression" and the operator between them is the following:

"Stored Temporal Element" *operator* **"User's Expression"**

Below follows the declaration of the ***Allen operators*** and the ***Uncertainty operators*** as well:

Allen Operators

1. **BEFORE**
   (Stored Temporal Element < User's Expression)
   Implementation : time_upper < A

2. **AFTER**
   (Stored Temporal Element > User's Expression)
   Implementation : time_lower > B

3. **EQUAL**
   (Stored Temporal Element = User's Expression)
   Implementation : time_lower = A *AND* time_upper = B

4. **MEETS**
   (The interval representing the Stored Temporal Element finishes when the interval described by the User's Expression starts)
   Implementation: time_upper = A

5. **MET_BY**
   (The interval representing the Stored Temporal Element starts when the interval described by the User's Expression finishes)
   Implementation: time_lower = B

6. **OVERLAPS**
   (The interval representing the Stored Temporal Element starts before the beginning of the interval described by the User's Expression and finishes after the beginning and before the end of this interval)
   Implementation: time_lower < A AND A<= time_upper <= B

7. **OVERLAPPED_BY**
   (The interval representing the Stored Temporal Element starts after the beginning and before the end of the interval described by the User's Expression and finishes after the end of this interval)
   Implementation: A <= time_lower <= B *AND* time_upper > B

8. **DURING**
   (The interval representing the Stored Temporal Element starts after the beginning of the interval described by the User's Expression and finishes before the end of this interval)
   Implementation: time_lower > A *AND* time_upper < B

9. **CONTAINS**
   (The interval representing the Stored Temporal Element starts before the beginning of the interval described by the User's Expression and finishes after the end of this interval)
   Implementation : time_lower < A *AND* time_upper > B

10. **STARTS**
    (The interval representing the Stored Temporal Element starts when the interval described by the User's Expression starts and finishes before the end of this interval)
    Implementation: time_lower = A *AND* time_upper < B

11. **STARTED_BY**
    (The interval representing the Stored Temporal Element starts when the interval described by the User's Expression starts and finishes after the end of this interval)
    Implementation: time_lower = A *AND* time_upper > B

12. **FINISHES**
    (The interval representing the Stored Temporal Element starts after the beginning of the interval described by the User's Expression and finishes when this interval finishes)
    Implementation: time_lower > A *AND* time_upper = B

13. **FINISHED_BY**
    (The interval representing the Stored Temporal Element starts before the beginning of the interval described by the User's Expression and finishes when this interval finishes)
    Implementation: time_lower < A *AND* time_upper = B

Uncertainty Operators

These operators can be grouped in two categories. The first category contains existential operators and they have the prefix **cb** (can be). The second one contains universal operators and they have the prefix **mb** (must be).
For the declaration of these operators we use the symbol S to represent the Stored Temporal Element interval while the symbol U represents the User' Expression interval. In short, we use the mathematical quantifiers ∃ **(there exists)** and ∀ **(for all)** in order to define the ***Uncertainty operators.*** For example the expression:
"S **cbeq** U ⇔ ∃ s ∈ S, ∃ u ∈ U : s = u" can be translated as "S can be equal U if and only if there exists a value s within S and a value u within U, such that s equals u".
For each operator we give the equivalent ***Allen operators***.

1. **Can Be EQual (cbeq)**
   S **cbeq** U ⇔ ∃ s ∈ S, ∃ u ∈ U : s = u.
   Equivalence to *Allen operators*: cbeq = ¬ (**before** *OR* **after)**

2. **Can Be Less Than (cblt)**
   S **cblt** U ⇔ ∃ s ∈ S, ∃ u ∈ U : s < u.
   Equivalence to *Allen operators*: cblt = ¬ (**met_by** *OR* **after**)

3. **Can Be Less Equal than (cble)**
   S **cble** U ⇔ ∃ s ∈ S, ∃ u ∈ U : s <= u.
   Equivalence to *Allen operators*: cble = ¬ (**after**)

4. **Can Be Greater Than (cbgt)**
   S **cbgt** U ⇔ ∃ s ∈ S, ∃ u ∈ U : s > u.
   Equivalence to *Allen operators*: cbgt = ¬ (**meets** *OR* **before**)

5. **Can Be Greater Equal than (cbge)**
   S **cbge** U $\Leftrightarrow$ $\exists$ s $\in$ S, $\exists$ u $\in$ U : s >= u.
   Equivalence to *Allen operators:* cbge = $\neg$ (**before**)

6. **Must Be EQual (mbeq)**
   S **mbeq** U $\Leftrightarrow$ $\forall$ s $\in$ S, $\forall$ u $\in$ U : s = u.
   Equivalence to *Allen operators:* mbeq = equal

7. **Must Be Less Than (mblt)**
   S **mblt** U $\Leftrightarrow$ $\forall$ s $\in$ S, $\forall$ u $\in$ U : s < u.
   Equivalence to *Allen operators:* mblt = before

8. **Must Be Less Equal (mble)**
   S **mble** U $\Leftrightarrow$ $\forall$ s $\in$ S, $\forall$ u $\in$ U : s <= u.
   Equivalence to *Allen operators:* mble = meets

9. **Must Be Greater Than (mbgt)**
   S **mbgt** U $\Leftrightarrow$ $\forall$ s $\in$ S, $\forall$ u $\in$ U : s > u.
   Equivalence to *Allen operators:* mbgt = after

10. **Must Be Greater Equal (mbge)**
    S **mbge** U $\Leftrightarrow$ $\forall$ s $\in$ S, $\forall$ u $\in$ U : s >= u.
    Equivalence to *Allen operators:* mbge = met_by

A graphical representation of all the above operators can be found in Appendix A.

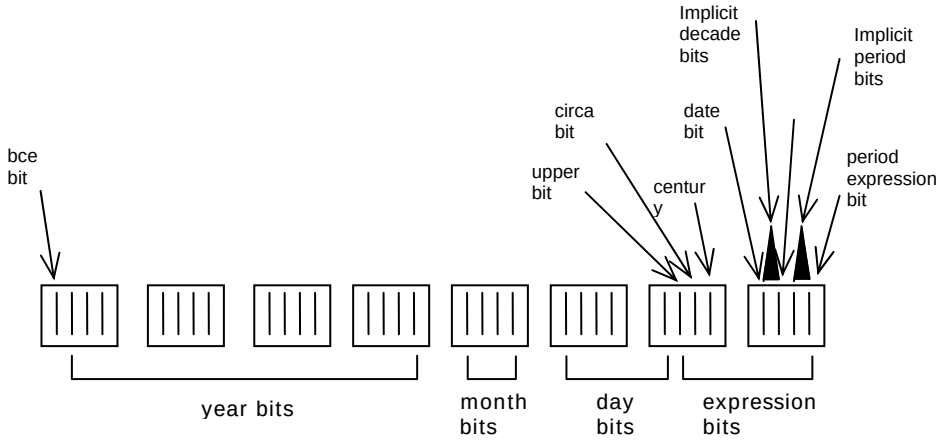
4.4 Internal representation and use in queries

The internal representation of the lower and upper boundaries of a time element is using the 7 least significant bits of the 8-byte integer used to model the time expression, while use the most significant bit to model time periods Before Common Era. The internal representation can be seen in the following scheme.

According to this notation the time primitive element “4th century” is stored internally into two 8-byte integers with the following values:

lower : 19665040 (decimal) , 012C1090 (hexadecimal)
upper : 26202064 (decimal) , 018FCFD0 (hexadecimal)

thus, representing the time period “300 January 1 - 399 December 31”



In order to implement any of the time primitive operations mentioned before, we must first apply a clear mask in order to "clear" the expression bits. The clear mask is the hexadecimal number 0xFFFFFFA0 (decimal -96), hereafter called CLEAR_FLAGS. So, the expressions to be used are :

lower &[†] CLEAR_FLAGS

and upper & CLEAR_FLAGS

It 's important to note that the masks should be applied in SQL queries with their decimal values .

4.5 Example sql queries

For example suppose that we have a relational database for monuments and a table COINS. In order to present the creation chronology of a coin we have the fields creation_chron_lower and creation_chron_upper. If a user wants to find all the coins which had been created before a specific chronology the question that will be formed in SQL is :

Example 1
select coin_name **From COINS** **where (creation_chron_upper & -96) < (parsed_time_val_lower & -96)**

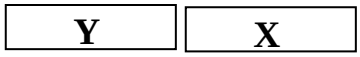
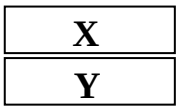
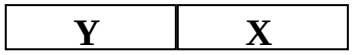
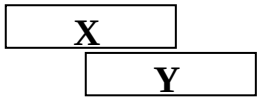
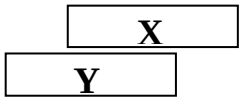
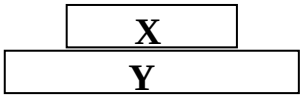
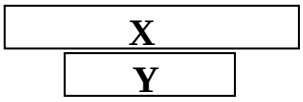
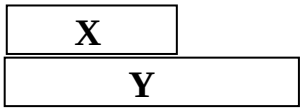
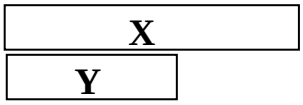
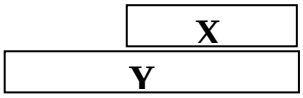
And if a user wants to find all the coins, which had been created after a specific chronology, the question that will be formed in SQL is :

[†] Bitwise AND operation

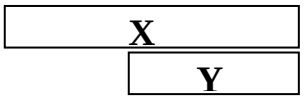
Example 2
select coin_name **from COINS** **where (creation_chron_lower & -96) > (parsed_time_val_upper & -96)**

The *parsed_time_val_lower* and *parsed_time_val_upper* are produced from the lexicographic analysis of the user's temporal expression through the function call *time_parse* ("User's Expression");

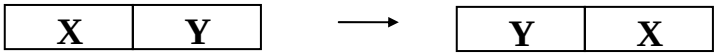
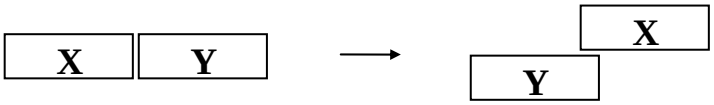
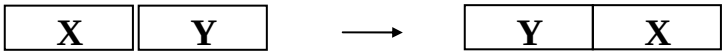
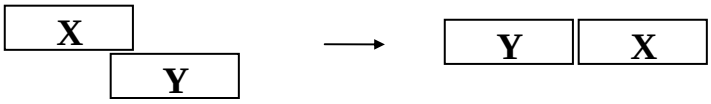
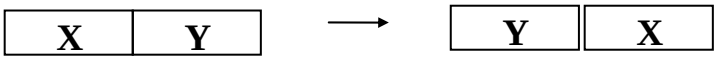
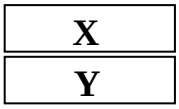
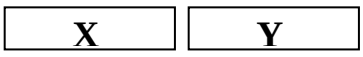
Appendix A

OPERATOR	GRAPHICAL REPRESENTATION
X *before* Y	
X *after* Y	
X *equal* Y	
X *meets* Y	
X *met by* Y	
X *overlaps* Y	
X *overlapped by* Y	
X *during* Y	
X *contains* Y	
X *starts* Y	
X *started by* Y	
X *finishes* Y	

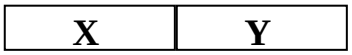
X *finished by* Y



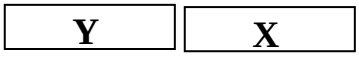
Relations that satisfy the ***Allen operators*** are illustrated in the above schema. A graphical representation follows for the ***Uncertainty operators***. They can cover all potential relations from snapshot A to snapshot B.

OPERATOR	**SNAPSHOT A**	**SNAPSHOT B**
X *can be equal* Y		
X *can be less than* Y		
X *can be less equal* Y		
X *can be greater than* Y		
X *can be greater equal* Y		
X *must be equal* Y		
X *must be less than* Y		

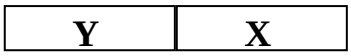
X *must be less equal* Y



X *must be greater than* Y



X *must be greater equal* Y



Bibliography

[1] J.F. Allen. Maintaining Knowledge about Temporal Intervals. *Comm. ACM*, 26: pp.832-843,1983

[2] J.F. Allen. Towards a General Theory of Action and Time. *Artificial Intelligence,* 23: pp. 123-154, 1984.

[3] T. Petersen and J. P. Barnett (editors). Guide to Indexing and Cataloging with the Art and Architecture Thesaurus. pp. 47-50, 1994.

[4] Anthi Yiortsou, Introducing Temporal Dimension in the Semantic Index System, 1998, *Technical Report FORTH-ICS/TR-231, October 1998.* Available in Greek: http://www.ics.forth.gr/isl/publications/paperlink/Introd_Temporal_Dimen_in_the_SIS.ps.gz

[5] Christina Gritzapi, Data transfer from a relational to an object-oriented database, 1996, *Technical Report FORTH-ICS/TR-168, May 1996*. Available in Greek: http://www.ics.forth.gr/isl/publications/paperlink/Data_transf_RDBMS_to_OODB.ps.gz