Querying Semistructured Temporal Data

Carlo Combi[1], Nico Lavarini[2], and Barbara Oliboni[1]

[1] Dipartimento di Informatica, Università degli Studi di Verona
[2] Creactive Consulting S.p.A.

Abstract. In this paper we propose the *GEM* Language (*GEL*), a SQL-like query language, which is able to extract information from semistructured temporal databases represented according to the Graphical sEmistructured teMporal (*GEM*) data model.

1 Introduction

In the last years the database research community has devoted some efforts to the development of methods for representing and querying *semistructured data* [1] (i.e., data that have no absolute schema fixed in advance, and whose structure may be irregular or incomplete). In this context, several approaches have been proposed, in which labeled graphs are used to represent semistructured data [9, 11]. Recently, it has been recognized and emphasized that time is an important aspect to consider also in the semistructured data context; thus, the problem of representing and querying changes in semistructured data has been considered in the database research field and some temporal data models, based on labeled graphs [5–7], have been studied.

In this work, we consider the Graphical sEmistructured teMporal (*GEM*) data model [6], which is general enough to include the main features of semistructured data representation. GEM allows one to model either valid or transaction times: the *valid time* (VT) of a fact is the time when the fact is true in the modeled reality, whereas the *transaction time* (TT) of a fact is the time when the fact is current in the database and may be retrieved [10].

In this paper we propose the *GEM* Language (*GEL*), a SQL-like query language, which is able to extract information from semistructured temporal databases represented by means of the *GEM* data model. *GEL* is a language designed for semistructured data and, similarly to Lorel [2], can be seen as an extension of OQL [4]. In particular, the most important features which add some novelty to *GEL*, with respect to the other main proposal, namely Chorel [5], are the following: we allow for querying databases which manage either valid time — and we will focus on it in the rest of the paper — or transaction time, while most of the others only allow for managing transaction time; we can exploit the generality of the *GEM* model, and thus build and query about general relationships between objects: it provides expressive power both to the model and to the language, without constraining the database to be tree-shaped and to use just the containment relationship; we introduce suitable temporal predicates and specific clauses and keywords for allowing the user to manage temporal aspects of data.

2 Related Work

In [2], the authors propose the *Lorel* query language, a semistructured query language based on the Object Exchange Model (OEM) [11]. OEM is a simple graph-based data model, with objects as nodes and object-subobject relationships represented as labeled arcs. Nodes are not labeled, labels are represented only on the edges and represent the node the edge point to. In the OEM data model each entity is represented by means of an object with an *oid* (*object identifier*). Lorel queries are intuitive, based on a syntax similar to that of the statement `SELECT FROM WHERE` of OQL [4], and use *Path Expressions*. A path expression represents a path on the graph, and thus it identifies the objects composing the path itself.

Chorel (*Change Lorel*) [5] is a query language for semistructured temporal data and is an extension of the Lorel query language. The Chorel query language is based on the *DOEM* (*Delta-OEM*) [5] data model, which is a temporal extension of OEM. Change operations (i.e., node insertion, update of node values, addition and removal of labeled arcs) are represented in DOEM by using *annotations* on nodes and arcs of an OEM graph for representing the history. Intuitively, annotations are the representation of the history of nodes and edges as it is recorded in the database: indeed, this proposal takes (implicitly) into account the *transaction time* dimension. DOEM and Chorel are implemented by means of a method that encodes DOEM databases as OEM databases and translates Chorel queries into equivalent Lorel [2] queries over the OEM encoding. Chorel queries are similar to Lorel queries, and can contain *annotation expressions*. For example, the query

```
SELECT Guide.<add>restaurant;
```

requires the restaurants having an `add` annotation, i.e., those restaurants which have been added to the database after its creation.

Chorel is a very flexible and powerful language, but is limited by the data model it is based on. As an example, neither OEM nor DOEM allow for the representation of general relationships: they represent only the containment relationship.

In [14], the authors extend the XPath [13] data model and query language to include valid time. In particular, they extend XPath's data model by adding to each node a list of disjoint intervals or instants representing valid time, and impose that the valid time of a node is constrained to be a subset of the valid time of a node's parent. Moreover, a valid-time axis is added to the query language to retrieve nodes according a valid time view. The valid-time axis of a node contains the valid-time information of the node itself. The main focus of [14] is the extension of the XPath data model to represent valid time, and thus the authors do not introduce any extension of XPath with temporal predicates and aggregates.

In [8], the author presents an extension of XPath to support transaction time. The proposed extension allows the representation of the history of an XML document as a sequence of XML documents representing the versions of the considered XML document. According to the data model extension, the

author extends the query language to query the transaction time. At this aim several new axes, node tests, and temporal constructs are added.

3 Representing temporal semistructured data

In this work, we suppose to represent temporal semistructured data by means of the Graphical sEmistructured teMporal data model (*GEM*), which represents semistructured temporal data by means of rooted, connected, directed, labeled graphs, where the temporal dimension is explicitly reported on node and edge labels and is described as an interval. *GEM* allows the database designer to model either transaction or valid times, by properly defining suitable constraints [6].

In designing *GEL*, we focus on the valid time dimension, as we want to focus on query aspects related to changes in the represented real world. In this section, we briefly describe the data model, by considering an example taken from a medical scenario. Figure 1 shows a *GEM* graph, representing information about the patient David Johnson.

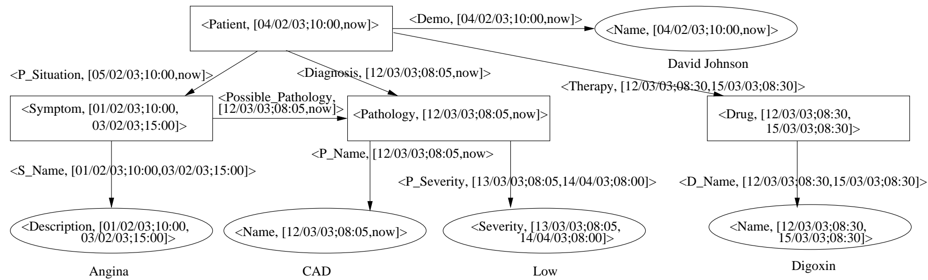

Fig. 1. An example of a *GEM* graph

A *GEM* graph is composed by two kinds of nodes: *complex* and *simple* nodes. The former represent abstract entities, whereas the latter represent primitive values and are leaves. Complex nodes are depicted as rectangles, while simple nodes are depicted as ovals. In the *GEM* data model [6], the symbol `now` is used to define respectively the objects that are valid at the present time in the considered reality, when considering the valid time dimension. Considering the example depicted in Figure 1, the node *Patient* is a complex node, while the node *Name* (child of *Patient*) is a simple node having value *David Johnson*.

In Figure 1, the nodes *Patient* and *Name*, and the edge *Demo* have valid time interval[1] *[04/02/03;10:00,*`now`*]*. The time interval represents that *David Johnson* becomes a *Patient* from 10:00 of 04/02/03, and he is still a *Patient*.

Several constraints and relationships could exist between valid times of nodes and edges: as an example, the valid time of a simple node could be contained in

[1] In this paper we adopt the format `DD/MM/YY;HH:mm` for timestamps.

the valid time of the related complex node: indeed, the simple node represents a property of the related complex node. Figure 1 depicts the simple node *Severity* having a valid time interval contained into the valid time of its related complex node *Pathology*. As a further example, the valid time of the node *Symptom* starts before the valid time of the related node *Patient* (i.e., the symptom appeared before the patient was enrolled), and the valid time of the edge between *Patient* and *Symptom* represents the fact the symptom has been reported after the patient was enrolled.

4 A query language for semistructured temporal data

Semistructured (temporal) data may be irregular and incomplete and do not necessarily conform to a fixed schema, thus flexibility in querying is needed. *GEL* is able to manage irregularity by means of flexible statements and allows one to extract and evaluate temporal information. Moreover, *GEL* supports the filtering of query results based on temporal information.

GEL is similar to Lorel [2] and to OQL [4], and has a SQL-like syntax. *GEL* queries are composed through the classical clauses `SELECT`, `FROM`, `WHERE`. The expressions specified in each clause are *path expressions*, i.e., expressions representing paths, which allows one to reach a given object on the *GEM* graph.

In an OEM graph [11], edges between nodes represent only the containment relationship, thus Lorel path expressions are based on this kind of relationship, and use the "dot notation"; in a *GEM* graph, edges represent different relationships and thus in the *GEL* syntax we decided to adopt the object-oriented notation related to methods. For example, the *GEL* path expression `Patient.has(Symptom)` can be read as "the `Patient has` the `Symptom`", and more formally, "the object `Patient` is related, by means of the relationship `has`, to the object `Symptom`".

An example of a *GEL* query is the following:

```
SELECT Patient.Demo(Name)
FROM Patient
WHERE Patient.P_situation(Symptom).S_name(Description) = "Angina"
```

Intuitively, this query requires to extract the `Name` object, identified by the path expression in `SELECT` clause, starting from the object reported in the `FROM` clause, but only if the required object satisfies the constraint imposed in the `WHERE` clause. Temporal clauses `TIME-SLICE` and `MOVING WINDOW` can be used to specify temporal features of required data, as detailed in Section 4.5.

As for the type system, in the semistructured data context flexibility is needed. In order to convert data having different types, *GEL* adopts the Lorel [2] approach, based on *type coercion*.

The `FROM` clause could be left implied in the *GEL* queries. This characteristic, rising from Lorel [2], derives from the fact that usually the value of this clause is the object from which the path expressions, in the `SELECT` clause, start. If the `FROM` clause is not specified, then it is automatically produced from the `SELECT` clause, introducing in the `FROM` clause a path expression for each path expression

in the `SELECT` clause. If the `FROM` clause is implied, then all the path expressions in the `SELECT` clause must start with the root of the graph. For example, the query

```
SELECT Patient.Diagnosis(Pathology)
WHERE Patient.Demo(Name) = "Smith"
```

becomes

```
SELECT Patient.Diagnosis(Pathology)
FROM Patient
WHERE Patient.Demo(Name) = "Smith"
```

In this way, the queries can be simplified leaving implied the `FROM` clause.

4.1 *GEL* statements

A *GEL* query is based on the `SELECT` statement, as for OQL and SQL [4, 12]. In the same way as OQL and SQL, in *GEL* the expression in the `SELECT` clause states for the objects of the database which have to be retrieved, the expression in the `FROM` clause specifies the objects to consider in the search, and the expression in the `WHERE` clause represents the constraints the retrieved objects have to satisfy. Obviously, like in other syntactically similar languages, when there is no constraint to be used in the `WHERE` clause, the whole clause itself can be missing. In the following, we will adopt a *current* semantics for the query evaluation [12], i.e., the query will return only the current objects which satisfy the query, for all the cases where time dimensions are not referred to in the query. When variables are used in the query to refer to the temporal dimensions of nodes/edges, the adopted semantics will be the *non-sequenced* one [12]: all the objects of the database, even the non current ones, will be considered for the query evaluation.

The result of a *GEL* query is a *multiset* or *bag* of tuples of attribute values. Each value can be either an atomic value or a node identifier; each attribute is named according to the content of the `SELECT` clause. It is possible to use set operators (`intersect`, `union`, `except`) to combine different queries (SQL-style).

4.2 Attribute naming

In *GEL*, attributes can be renamed, as in SQL statements:

```
SELECT P.Diagnosis(Pathology).P_name(Name) as Path_name
FROM Patient P
```

in this case, `Path_name` is the label of the retrieved object, i.e., in the original graph the pathology name is labeled `Name`, while in the result it is labeled `Path_name`.

4.3 Variables

Variables can be used also inside an expression and can have different roles. *GEL* allows one to use variables (i) as *aliases* to avoid to repeat long expressions, and (ii) to identify each element to be instantiated in the query evaluation.

```
SELECT P.P_name(Name) as Path_name
FROM Patient.Diagnosis(Pathology) P
```

In this case, the variable `P` is used to identify the `Pathology` object, in order to avoid to refer to it repeating all the expression, as for example:

```
SELECT Patient.Diagnosis(Pathology).P_name(Name) as Path_name
FROM Patient.Diagnosis(Pathology)
```

The variables inside the path allow for retrieving (and for naming) the needed objects, without the need of repeating the common portions of the path for each expression we want to name, as in the following example.

```
SELECT P.Diagnosis(Pathology)<T>.P_name(Name)<N>
FROM Patient P
WHERE T.P_Severity(Severity)="high" AND N="CAD"
```

This query allows one to retrieve the objects `T` and `N`, without specifying two similar expressions in the `FROM` clause; at the same time `T` and `N` are used as names for the corresponding objects.

The query without using Intra-path variables should be

```
SELECT P.Diagnosis(Pathology) as T,
       P.Diagnosis(Pathology).P_name(Name) as N
FROM Patient P, P.Diagnosis(Pathology) T,
       P.Diagnosis(Pathology).P_name(Name) N
WHERE T.P_Severity(Severity)="high" AND N="CAD"
```

Variables in the path expressions can be used to force two expressions to be distinct. For example, the query

```
SELECT P.Demo(Name)
FROM Patient P
WHERE P.Diagnosis(Pathology).P_name(Name) = "CAD" AND
      P.Diagnosis(Pathology).P_name(Name) = "Pneumonia"
```

cannot be used to extract the patients who suffer of `CAD` and `Pneumonia`, because the two paths in the `WHERE` clause have a common prefix (in this case the complete path expression) and thus they are instantiated on the same objects. For this reason the previous query does not retrieve the desired result.

It is possible to assign two distinct variables to the two desired results, thus they are instantiated separately on the two paths of the graph, related to two (possibly) different nodes with label `Name` (if they exist), in the following way:

```
SELECT P.Demo(Name)
FROM Patient P
WHERE P.Diagnosis(Pathology)<X1>.P_name(Name) = "CAD" AND
      P.Diagnosis(Pathology)<X2>.P_name(Name) = "Pneumonia"
```

In this way, the two paths `P.Diagnosis(Pathology).P_name(Name)`, which are considered as distinct by the user, are not forced to be the *same* path.

The variables can be used also to identify a constraint expressing the fact that an object must be reachable on several paths. Indeed a GEM database, as the one in Figure 1, is not expressed as a tree, but as a DAG (Directed Acyclic Graph): in our case, for example, edges `Possible_Pathology` and `Diagnosis` are directed to the same node `Pathology`. In this case, we can use variables to define queries on objects with several ingoing edges:

```
SELECT Patient.Demo(Name)
FROM Patient.Diagnosis(Pathology)<X>,
  Patient.P_Situation(Symptom).Possible_Pathology(Pathology)<X>
```

The previous query asks for the name of patients having a diagnosed pathology and presenting also a connected symptom.

4.4 Wildcards

The path expression power, with respect to objects and paths representation, can be increased by using *wildcards*.

The simplest type of wildcard comes directly from SQL, and is represented by the special characters '`#`'[2] and '`%`'.

As in SQL, these characters can be considered "special" in the *pattern matching* between the query strings and the labels of *GEM* graph elements, as they can be used for comparisons with string literals and also, differently from SQL, as wildcards for the edge labels.

The '`#`' character represents *any character*. The '`%`' character represents a sequence of characters with an arbitrary length. These wildcards can be used in the node and edge names contained in the path expressions, or instead of their names. Moreover, in a path expression, specifying an edge/node, having label "`%`", means "an edge/node with any label". For example, the path expression `Patient.%(Pathology)` means that we are looking for a node with label `Patient`, linked by means of any edge, to a node with label `Pathology`. In the same way, the path expression `Patient.%(%).P_name(Name)` means that we are looking for a node with label `Patient` linked by means of any edge to any node, which has an outgoing edge with label `P_name` ending in a node with label `Name`. Since in the path expressions, nodes and edges are represented in different ways, when a node or an edge is only represented by the wildcard '`%`' we can abbreviate it with an empty string.

Another kind of wildcard allows one to specify some properties of each element in the path expression by using some simple *regular expression*.

It is possible to specify how many times a given edge has to appear in a sequence, and to give a choice between different possible labels. For example the

[2] The choice of the character '`#`' instead of the traditional SQL '`_`' to mean any character, is based on the fact that '`_`' is often used as word separator in labels for nodes and edges, and thus it is too used to be protected by escape sequences.

path expression A.[(B)]?.(C) means that between the nodes A and C, either 0 or 1 nodes B can exist.

The | character, used between two or more elements, allows one the choice of any element in the set. This operator is particularly effective when it is used together with the previous one, in the characters [and]. For example A.[(B)|(C)]*.(D) means that between the nodes A and D, a sequence exists, and is either empty or each element of it is B or C

These regular expressions can be combined with the string wildcards to obtain a powerful and flexible query system. As an example, the following query

```
SELECT S.%(Name)
FROM Symptom S
WHERE S.[Possible_Pathology(Pathology)]+.P_name(Name) = "Hepatitis"
```

extracts the Name of the Symptom (connected by any edge to the node) which is related (to a degree) to Hepatitis. The WHERE clause actually requires the node S to be connected by a path made of Possible_Pathology edges and Pathology nodes to the simple node labeled Hepatitis. The length of the path must be greater than 0, so a path Symptom.p_name(Name) would not make the clause true, while a path Symptom.Possible_Pathology (Pathology).P_name(Name) would do.

The following query

```
SELECT P.Demo(Name)
FROM Patient P
WHERE P.[P_situation(Symptom)|Diagnosis(Pathology)].%(%) = "Hemicrania"
```

extracts the name of a Patient for which the symptom or the pathology is connected to any node labeled Hemicrania. In this case what is required is the existence, starting from a Patient node, of the edge P_situation and the node Symptom, or of the edge Diagnosis and the node Pathology. When this condition is met, any edge connected to any simple node labeled Hemicrania would make the clause true.

4.5 Temporal aspects

Nodes and edges of a *GEM* graph have in the label a time interval representing their validity with respect to a considered time dimension (see Section 3). In particular, in this work, we focus on the valid time dimension, i.e., the time when the fact represented by the object is true in the considered reality.

Temporal clauses

In the WHERE clause, conditions expressing constraints that must be satisfied by the requested objects are specified. To consider temporal aspects, we introduce the new clauses TIME-SLICE and MOVING WINDOW. Moreover, temporal constraints can be specified in the WHERE clause.

The TIME-SLICE clause allows the user to query along the temporal dimension of nodes and edges, by considering only those nodes and edges having a

valid time interval intersecting the specified interval. In general, the result of the query will be composed by nodes (edges), requested in the `SELECT` clause, satisfying general constraints expressed in the `WHERE` clause, and having a time interval intersecting the time interval specified in the `TIME-SLICE` clause.

In the `TIME-SLICE` clause, the considered time interval can be specified in different ways:

- by the `FROM ... TO ...` keywords it is possible to require objects intersecting an interval starting at a given instant, and ending at a second given instant. For example, the query requiring the name of the patients considering only the period from 08:00 of 01/02/03 to 08:00 of 01/03/03 is:

  ```
  SELECT Patient.Demo(Name)
  FROM Patient
  TIME-SLICE FROM 01/02/03;08:00 TO 01/03/03;08:00
  ```

- by the `FROM` keyword it is possible to require objects having an interval ending after a given instant. In this case, the ending time is not specified. For example, the query requiring the name of the patients after 08:00 of 01/02/03 is:

  ```
  SELECT Patient.Demo(Name)
  FROM Patient
  TIME-SLICE FROM 01/02/03;08:00
  ```

- by the `TO` keyword it is possible to require objects having an interval starting before a given instant. For example, the query requiring the name of the patients before 08:00 of 01/03/03 is:

  ```
  SELECT Patient.Demo(Name)
  FROM Patient
  TIME-SLICE TO 01/03/03;08:00
  ```

The interval constraint expressed by the `TIME-SLICE` clause can be forced to be either a strict containment, or a left strict containment, or a right strict containment. For example, in the first case, the objects satisfy the requested constraints only if they have a time interval strictly contained in the specified time interval. This kind of containment is expressed by means of the keyword `STRICT`.

```
SELECT Patient.Demo(Name)
FROM Patient
STRICT TIME-SLICE FROM 01/02/03;08:00 TO 01/03/03;08:00
```

In Figure 2 we suppose that $[t_i, t_j]$ is the time interval of a node (edge) and $[t_h, t_k]$ is the time interval specified in the `TIME-SLICE` clause, and report examples of time intervals satisfying the temporal constraints expressed in the `TIME-SLICE` clause with respect to the specified keywords.

The `MOVING WINDOW` clause allows the user to consider nodes and edges through a (moving) temporal window. The window is specified in the clause and moves along the temporal axis. The general constraints expressed in the other

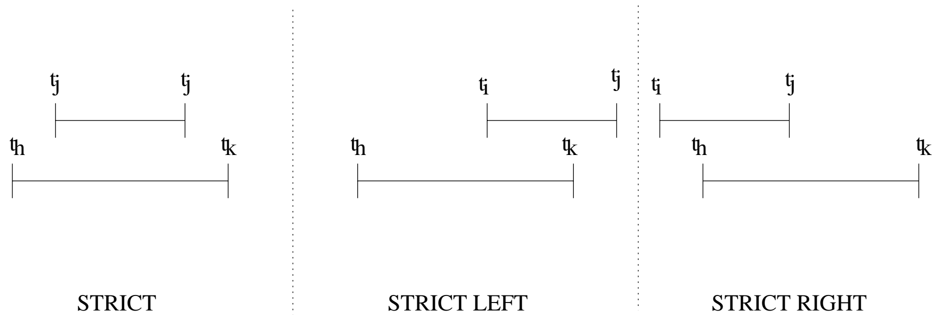

Fig. 2. Examples of time intervals satisfying constraints in the `TIME-SLICE` clauses

clauses are checked only on the nodes and edges visible through the window, i.e., only on the nodes and edges having a time interval satisfying the temporal constraints expressed in the `MOVING WINDOW` clause. For example, the following query requires the name of the patients having had both CAD and pneumonia within a period of 40 days:

```
SELECT P.Demo(Name)
FROM Patient P
WHERE P.Diagnosis(Pathology)<X1>.P_name(Name) = "CAD" AND
      P.Diagnosis(Pathology)<X2>.P_name(Name) = "Pneumonia"
MOVING WINDOW 40 days
```

Temporal predicates
To compare valid times of different nodes and edges, *GEL* provides the support of standard comparison predicates both for intervals, instants, and for comparing intervals and instants.

Variables can be suitably assigned both to the overall valid time and to the starting and ending instants of the valid time. The syntax is based on the symbol `@`, and is used as in the following example:

```
Patient.Diagnosis@[X1,X2](Pathology).P_name(Name)@[Y1,Y2]
```

This path expression identifies a graph element and extracts the values of the start and end times of the element itself. In particular, it identifies the edge `Diagnosis` and extracts its start and end times, and the node `Name` and extracts its start and end times.

To assign a single variable to the overall valid time, the previous symbol `@` must be used as in the following example:

```
Patient.Diagnosis@[X](Pathology).P_name(Name)@[Y]
```

This path expression identifies a graph element and extracts the interval values of the valid time of the element itself. In particular, it identifies the edge `Diagnosis` and extracts its time interval, and the node `Name` and extracts its time interval.

The variables used to extract the times could be used as selection for the query, as in the following case:

```
SELECT N as PatientName, X1 as DiagStart, X2 as DiagEnd
FROM Patient.Diagnosis@[X1,X2](Pathology).P_name(Name)<N>
```

The result of this query is a set of tuples; each tuple is composed by the string representing the name of the patient, the start time and the end time of the diagnosis related to the patient itself.

This is a case, where the query evaluation considers all the nodes/edges and not only the current ones, with the application of a non-sequenced temporal semantics [12]: the condition in the query may involve the explicit comparison of nodes/edges at different times.

GEL also offers the way to compose temporal predicates on time instants and intervals in the `WHERE` clause. The temporal predicates can be *point-point predicates*, which compare two time instants, *point-interval predicates*, which compare a time instant with a time interval, and *interval-interval predicates*, which compare two time intervals.

A *point-point predicate* is composed by a variable, a temporal comparison operator, and a time instant, which can be either a variable or a constant. The temporal comparison operators are: `=`, `<>`, `<`, `<=`, `>`, `>=`.

Point-interval predicates verify whether a time instant belongs to a time interval. This time interval is represented with the *GEL* syntax by means of a couple of time instants separated from a comma, and contained in "`[`" and "`]`". One or both of these instants can be replaced by a variable, which can be extracted from another element. Thus, a point-interval predicate is composed by a variable, an interval operator, and an interval (variable or constant). The interval operators are `in` (the instant belongs to the interval) and `out of` (the instant does not belong to the interval).

Interval-interval predicates verify whether two time intervals satisfy the well known Allen's relations (before, meets, overlaps, ...) [3].

As a final example, let us consider the following query requiring the name of the patients having had CAD either starting or ending during pneumonia and pneumonia holding on an interval overlapping the period from May 23, 2003 8:00 a.m. to July 21, 2003 8:00 a.m.

```
SELECT P.Demo(Name)
FROM Patient P
WHERE P.Diagnosis(Pathology)<X1>@[I1,I2].P_name(Name) = "CAD" AND
      P.Diagnosis(Pathology)<X2>@[T].P_name(Name) = "Pneumonia" AND
      (I1 in T OR I2 in T) AND
      T overlaps [23/05/03;08:00,21/07/03;08:00]
```

5 Conclusions

In this work, we proposed the temporal query language *GEL* for semistructured data, which explicitly considers the temporal dimensions of data and their comparison as well as specific temporal clauses and keywords. As for future work, we

plan to focus on some main topics, such as allowing the specification of a (temporal) graph structure for the query result, i.e., providing the language with the closure property, supporting several time semantics for graph-based data models, and extending the query language to deal with both valid and transaction times, to obtain a fully fledged bitemporal query language.

References

1. S. Abiteboul. Querying Semi-Structured Data. In *Proceedings of the International Conference on Database Theory*, volume 1186 of *LNCS*, pages 262–275, 1997.
2. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.
3. J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.
4. R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, , and Fernando Velez. *The Object Data Standard: ODMG 3.0.* Series in Data Management Systems. Morgan Kaufmann Series in Data Management Systems, 2000.
5. S. S. Chawathe, S. Abiteboul, and J. Widom. Managing historical semistructured data. *Theory and Practice of Object Systems*, 5(3):143–162, 1999.
6. C. Combi, B. Oliboni, and E. Quintarelli. A graph-based data model to represent transaction time in semistructured data. In *Proceedings of DEXA 2004*, volume 3180 of *LNCS*, pages 559–568. Springer-Verlag, Berlin, 2004.
7. C. E. Dyreson, M. H. Böhlen, and C. S. Jensen. Capturing and Querying Multiple Aspects of Semistructured Data. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 290–301. Morgan Kaufmann, 1999.
8. Curtis E. Dyreson. Observing transaction-time semantics with ttxpath. In *WISE (1)*, pages 193–202, 2001.
9. M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. STRUDEL: A web site management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26,2 of *SIGMOD Record*, pages 549–552. ACM Press, 1997.
10. C. S. Jensen, C. E. Dyreson, and M. H. Bohlen et al. The consensus glossary of temporal database concepts - february 1998 version. In *Temporal Databases: Research and Practice*, volume 1399 of *Lecture Notes in Computer Science*, pages 367–405. Springer, 1998.
11. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260. IEEE Computer Society, 1995.
12. R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL.* Series in Data Management Systems. Morgan Kaufmann, 2000.
13. World Wide Web Consortium. XML Path Language (XPath) version 1.0. http://www.w3.org/TR/xpath.html. W3C Reccomendation 16 November 1999.
14. Shuohao Zhang and Curtis E. Dyreson. Adding valid time to xpath. In *Databases in Networked Information*, volume 2544 of *LNCS*, pages 29–42, 2002.