Recursive Query Processing in SBQL*

Tomasz Pieciukiewicz[1], Krzysztof Stencel[1,2], Kazimierz Subieta[1]

[1] Polish-Japanese Institute of Information Technology, Warsaw, Poland
[2] Institute of Informatics, Warsaw University, Warsaw, Poland
{ pietia, stencel, subieta }@pjwstk.edu.pl

Abstract. Recursive queries are required for many database applications. Among them we can mention Bill-Of-Material (BOM), various kinds of networks (transportation, telecommunication, etc.), workflows, processing semi-structured data (XML, RDF), and others. The support for recursive queries in current query languages is limited. In particular, this concerns the corresponding extensions of SQL in Oracle and DB2. In this paper we present recursive query processing capabilities for the object-oriented Stack-Based Query Language (SBQL). SBQL offers very powerful and flexible recursive querying capabilities due to the fact that recursive processing operators are fully orthogonal to other capabilities of this language. Recursive queries formulated in SBQL turned out to be much simpler in comparison to equivalent queries which could be formulated in other languages. This paper discusses SBQL constructs, such as transitive closures, fixed point equations and recursive procedures/views. Their main advantage is that they are seamlessly integrated with object-oriented facilities, computer environment and databases. We also consider a semi-strong typing system for the abovementioned recursive facilities.

1. Introduction

There are many important tasks which require recursive processing. The most widely known is Bill-Of-Material (BOM) which is a part of Materials Requirements Planning (MRP) systems. BOM acts on a recursive data structure representing a hierarchy of parts and subparts of some complex material products. Typical MRP software processes such structures by proprietary routines and applications implemented in a programming language. Similar problems concern computations on genealogic trees, stock market dependencies, various types of networks (transportation, telecommunication, electricity, gas, water, and so on), etc. The recursion is also necessary for internal purposes of computer systems, such as processing recursive metadata structures (e.g. CORBA Interface Repository), configuration management repositories, hierarchical structures of XML or RDF files, and so on.

Despite importance, recursion was not supported in SQL standards (SQL-89 and SQL-92). Beyond these standards, it was implemented (differently) in relational DBMSs, in particular, in Oracle and DB2, in the form of transitive closures and linear recursion. The newer SQL standard SQL:99 introduces some recursive capabilities. Unfortunately the standard is very huge and eclectic, thus many database professionals doubt if it will ever be fully implemented. The ODMG standard for object-oriented databases and its query language OQL do not mention any corresponding facilities. Recursion is considered a desirable feature of XML-oriented and RDF-oriented query languages, but current proposals and implementations do not introduce corresponding features or introduce them with many limitations. In XQuery one can simply use XPath expressions or recursive functions, but the former is too week to express simplest BOM queries, while the latter represents just the 3GL programming style rather than high level database querying.

The possibility of recursive processing has been highlighted in the field of deductive databases. An example of these is Datalog which has sound theoretical foundations (see e.g.

* This work is supported by European Commission under the 6th FP project e-Gov Bus, IST-4-026727-ST

[1] and thousands of papers which cannot be cited here). However, it seems that Datalog falls short of the software engineering perspective. It has several recognized disadvantages, in particular: flat structure of programs, limited data structures to be processed, no powerful programming abstraction capabilities, impedance mismatch during conceptual modelling of applications, poor integration with typical software environment (e.g. class/procedure libraries) and poor performance. Thus practical mission-critical Datalog applications are till now unknown. Nevertheless, the idea of Datalog semantics based on fixpoint equations seems to be very attractive to formulate complex recursive tasks. Note however that fixpoint equations can be added not only to languages based on logic programming, but to any query language, including SQL, OQL and XQuery.

Besides transitive closures and fixpoint equations there are classical facilities for recursive processing known from programming languages, namely recursive functions (procedures, methods). In the database domain a similar concept is known as recursive views. Integration of recursive functions or recursive views with a query language requires generalizations beyond the solutions known from typical programming languages or databases. First, functions have to be prepared to return bulk types that a corresponding query language deals with, i.e. a function output should be compatible with the output of queries. Second, both functions and views should possess parameters, which could be bulk types compatible with query output too. Currently very few existing query languages have such possibilities, thus using recursive functions or views in a query language is practically unexplored.

This paper discusses three different approaches to recursive querying: transitive closure operators, least fixed point equation systems (fixpoint equations, for short), recursive procedures and views. For the first time, we describe all three approaches to recursive processing within a unified framework: the Stack Based Architecture (SBA) for object-oriented query/programming languages [2, 3, 4]. SBA treats a query language as a kind of programming languages and therefore, queries are evaluated using mechanisms which are common in programming languages. SBA introduces an own query language Stack-Based Query Language (SBQL) based on abstract, compositional syntax and formal operational semantics. SBQL is equipped with a strong type system.

We have implemented all these three recursive facilities within the framework of SBA and smoothly integrated them with object-oriented ideas, computer environment and databases. The research has been done within the currently developed object-oriented database platform ODRA devoted to Web and grid applications.

We also present a proposal of a typing system for transitive closures and fixpoint systems. The type checking of fixpoint systems is not a straightforward task, since types in a system of fixpoint equations can also require possibly infinite type inference. We adopted an engineering trade-off as the solution to this problem (which corresponds to the spirit of so called semi-strong type checking). If types can be reconstructed in the first iteration of the fixpoint computation, then the system of fixpoint equations is considered correct. Otherwise, the user is obliged to declare types explicitly. However, in spite of the possibility to reconstruct types in systems of fixpoint equations, we recommend to declare them explicitly in the application code. After years (or even months) of code maintenance, even relatively simple fixpoint equations may turn to be incomprehensible. Explicit declarations of types can help in the analysis and modification.

The rest of paper is organized as follows. Section 2 contains some remarks on our view of workflow systems which will be used as the running example in this paper. In Section 3 we shortly describe the type system of SBA/SBQL. Section 4 describes transitive closures in SBA. Section 5 presents fixpoint systems of SBA while in Section 6 we consider type

checking of fixpoint systems. In Section 7 we show that recursive queries can be formulated using recursive procedures and views. Section 8 concludes.

2 Workflow Systems and BPQL

Within the European project ICONS (2002-2004) a new complex workflow management system has been developed. It is currently commercialized under the name OfficeObjects®WorkFlow [5] by the Rodan Systems S.A. and deployed among more than 40 business and public administration clients in Poland. OOW is based on the WfMC architecture [6, 7] and XML Process Definition Language (XPDL), the standard process definition language proposed by WfMC. Perhaps the most interesting and powerful feature of OOW is the Business Process Query Language (BPQL) [8], currently embedded in XPDL. BPQL was provided as a tool for tracking and monitoring workflow processes. BPQL queries can be issued within running workflows, thus they support flexibility of dynamic workflow changes based on retrieving information from the entire workflow environment. In this context flexibility means that it is possible to express complex dynamic requests that depend on the structure of processes, processes' execution history, state and anticipation, as well as current organisational and application data. For many quite complex requirements, such as workflow participant assignments and transition conditions, this approach seems to be the closest to the real business process behaviour. BPQL has been equipped with a set of more than 30 workflow monitoring functions (written in BPQL), which aggregate typical tasks, e.g. finding the best participant assignment. BPQL has appeared to be very useful when an application is already operating but client requirements are changing. Due to BPQL some changes are matters of some minutes rather than several days of hard coding and testing.

An inevitable part of any query language, including BPQL, is a database schema that is a reference to all the data that queries have to address. Following WfMC, in case of workflows such a schema is referred to as *metamodel*, because it concerns internal data describing the state of workflow processes. The metamodel describes data on the workflow environment, process definition and process enactment.

Because the metamodel is presented by a very complex UML class diagram (see [9]) the decision was that the repository storing the workflows internal state should be object-oriented. Mapping a metamodel class diagram into a relational schema has advantages and disadvantages. An advantage is that such a schema can be queried by SQL. However such a complex relational schema would be very difficult to understand by a typical workflow programmer. Moreover, SQL queries could be very complex (traversing many tables by means of joins), difficult to write, read and change. Another disadvantage of such a mapping is that a lot of information is lost, e.g. information on inheritance, associations and cardinalities. Our experience has shown that in comparison to a relational schema an object-oriented UML-like schema requires much shorter time to understand and queries in an object-oriented query language are much shorter, more legible and more efficient in comparison to SQL. Thus the decision that the metamodel and BPQL are to be object-oriented. Current BPQL is a subset of the Stack-Based Query Language (SBQL) that is developed and implemented within our research group.

Currently we are involved in next European projects having workflow layers as significant work packages. Relying on our experience from BPQL we would like to develop a business process query language that will be more adequate to further workflows' designer and programmer needs.

The problem of making a good business process query language has a lot of aspects. One of them is the problem that we encountered during querying a workflow metamodel containing

recursive structures. For instance, a typical structure of a workflow process is a graph, perhaps with cycles, which on the UML schema is represented as a cyclic structure of classes and associations. Some queries addressing such structures, e.g. "check which and how many participant roles will be necessary for the given processes in December 2006" requires recursive traversing process graphs, including some data on human resources, and subsequent collecting all the information. The most natural approach to graph processing is recursion. As queries are more convenient than proprietary routines and applications implemented in a programming language, a query language supporting recursive processing may be a desirable facility for any workflow system. In this paper we will use workflow systems as the running example. We will try to convince the reader that the recursive capabilities of SBQL facilitate writing clearer and more comprehensible workflow queries.

3. SBQL Semi-Strong Type System

SBQL is based on a new strong typing theory. It distinguishes *internal* and *external* type systems. The internal type system reflects the behaviour of the type checking mechanism, while the external type system is used by the programmer. A static strong type checking mechanism simulates run-time computations during compile time by reflecting the run-time semantics with the precision that is available at the compile time. Roles of the SBQL typing system are the following: compile-time type checking of query operators, imperative constructs, procedures, functions, methods, views and modules; user-friendly, context dependent reporting on type errors; resolving ambiguities with automatic type coercions, ellipses, dereferences, literals and binding irregular data structures; shifting type checks to run-time, if it is impossible to do them during compile time; restoring a type checking process after a type error, to discover more than one type error in one run; preparing information for query optimisation by properly decorating a query syntax tree. The internal SBQL type system includes three basic data structures that are compile-time counterparts of run time structures: a metabase, a static environment stack and a static result stack. Static stacks process type signatures – typing counterparts of corresponding run time entities. Signatures are additionally associated with attributes, such as mutability, cardinality, collection kind, type name, multimedia, etc. For each query/program operator a decision table is provided, which determines allowed combinations of signatures and attributes, the resulting signature and its attributes, and additional actions.

4. Transitive Closures in SBQL

A transitive closure in SBQL is a non-algebraic operator having the following syntax**:**

q_1 **close by** q_2

Both q_1 and q_2 are queries. The query is evaluated as follows. Let *final_result* be the final result of the query and $\cup$ the bag union. Below we present the pseudo-code accomplishing abstract implementation of q_1 **close by** q_2:

final_result := *result_of* (q_1);
for each $r \in$ *final_result* do:
- push *nested*(r) at top of ENVS.
- *final_result* := *final_result* $\cup$ *result_of* (q_2);
- pop ENVS;

ENVS stands for environment stack. Note that each element r added to *final_result* by q_2 is subsequently processed by the *for each* command. The above operational semantic can be

described in the denotational setting as the least fixed point equation (started from *final_result* = ∅ and continued till fixpoint):

$$final\_result = q_1 \cup final\_result.\, q_2$$

Similarly, the semantics can be expressed by iteration (continued till *result_of* (q_2) = ∅):

$$final\_result = q_1 \cup q_1.q_2 \cup q_1.q_2.q_2 \cup q_1.q_2.q_2.q_2 \cup ....$$

Naive implementation of the **close by** operator is as easy as the implementation of the dot operator. Note that if q_2 returns a previously processed element, an infinite loop will occur. Checking for such situations in queries is sometimes troublesome and may introduce unnecessary complexity into the queries. Another operator **close unique by** has been introduced to avoid infinite loops due to duplicates returned by q_2.

As q_1 and q_2 can be any queries, simple or complex, the relation between elements which is used for transitive closure is calculated on the fly during the query evaluation; thus the relation needs not to be explicitly stored in the database.

In Fig.1 we depict a simple data schema used in our examples. It is a small part of a possible workflow metamodel. An *Activity* has a *name*, *type* and *state*, as well as collections of *pre-* and *post- conditions* and *constraints*. Execution of an activity takes no more than *maxTime*. Activities are connected with each other via *Transitions*. Transitions may specify *conditions*. Activities may also require *Resources* (which have *names*), the *amount* of resources required by each activity is specified in *RequiredResources*.

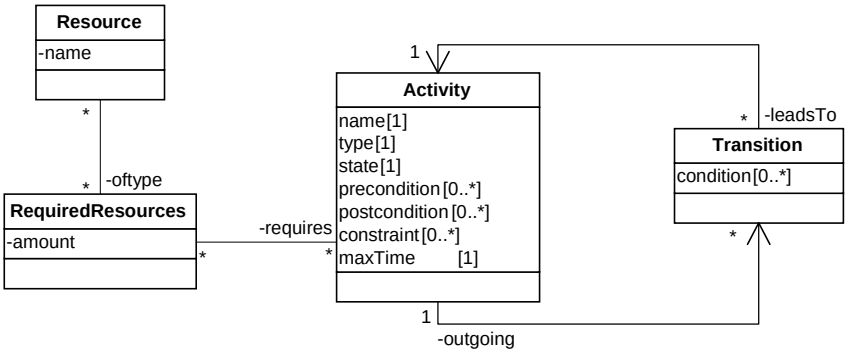

Fig. 1. A sample data schema

The simplest transitive closure SBQL query over this schema finds all activities that may be reached via transitions from an activity named "Planning".

(*Activity* **where** *name* = "Planning") **close by** (*outgoing.Transition.leadsTo.Activity*)

This query first selects activities having with *name* equal to "Planning". The transitive closure relation is described by the subquery (*outgoing.Transition.leadsTo.Activity*). It returns all *Activity* objects which may be navigated to via the pointers that lead to and from appropriate *Transition* objects.

As we know the maximum time for execution of each activity, we may want to calculate the maximum amount of time that may be required to complete a given business process. Let's assume that the given process starts with „Planning" and all activities that may be reached via transitions from this activity are parts of this process.

First we augment the above query so that it returns pairs. Each such pair constructed by means of the dependent **join** consists of an activity and the time needed to complete this activity counted from the beginning of the whole workflow.

(((*Activity* **where** *name*=”Planning”) **as** *x*) **join** (*x.maxTime* **as** *howLong*))
close by
((*x.outgoing.Transition.leadsTo.Activity* **as** *x*) **join** ((*x.maxTime* + *howLong*) **as** *howLong*))

The query uses a named value in order to calculate the time. The maximum time the initial activity may require is named *howLong* and paired with the initial activity. In subsequent iterations the *howLong* value from parent object is used to calculate the value (that will be assigned the same name) to be paired with child elements.

With the above query in hand we can compute the maximum length of the workflow. We simply restrict the result of this query to the part *howLong* and calculate the maximum:

max((((*Activity* **where** *name*=”Planning”) **as** *x*) **join** (*x.maxTime* **as** *howLong*))
close by
((*x.outgoing.Transition.leadsTo.Activity* **as** *x*)
join ((*x.maxTime* + *howLong*) **as** *howLong*))
.*howLong*)

This query computes the time needed to complete the critical path but does not return the activities along it. Such information is very important in most cases. SBQL allows for collecting this information. Another named variable is needed which will store the collection of activities along the particular path. Here is the query which collects the path items:

(((*Activity* **where** *name*=”Planning”) **as** *x*) **join** ((*x.maxTime* **as** *howLong*), *x* **as** *path*))
close by
((*x.outgoing.Transition.leadsTo.Activity* **as** *x*)
join
((*x.maxTime* + *howLong*) **as** *howLong*, (*path* **union** *x*) **as** *path*))

Now, we can use this query in order to select the critical path. We just group the result of the above query under the name *paths*. Then we find the path with the longest completion time (i.e. the value of the variable *howLong*):

(((((*Activity* **where** *name*=”Planning”) **as** *x*) **join** ((*x.maxTime* **as** *howLong*), *x* **as** *path*))
close by
((*x.outgoing.Transition.leadsTo.Activity* **as** *x*)
join
((*x.maxTime* + *howLong*) **as** *howLong*, (*path* ∪ *x*) **as** *path*))) **groups as** *allPaths*)
.((*allPaths* **as** *result*) **where** *result.howLong* = *max*(*allPaths.howLong*))**.***result***.***path*

The results of recursive queries may be also used as a starting point for other calculations – for example, the type and amount of resources required by a specific business process.

((*Activity* **where** *name* = ”Planning”) **close by** (*outgoing.Transition.leadsTo.Activity*))
.((*requires.RequiredResource***.**(*oftype.Resource.name* **as** *res*, *amount* **as** *howMany*))
group as *requiredResources*)
.(**distinct**(*requiredResources.res*) **as** *resource*)
.(*resource*, *sum*((*requiredResources* **where** *res*=*resource*).*howMany*))

This query first finds all the activities that are a part of a business process, then finds all resources required in various activities pairing them with the required amounts, groups all those pairs in a collection named *requiredResources* and processes the collection in order to merge pairs with duplicate resource names.

Another variant of the transitive closure is the operator **leaves by** which returns only leaves of the traversed graph. An example use of this operator is the query which returns only the *reachable* final activities of the workflow (such that they have no outgoing transition):

(*Activity* **where** *name* = ”Planning”) **leaves by** (*outgoing.Transition.leadsTo.Activity*)

Cycles in the queried graph (which may be encountered in graphs representing workflows) can be easily dealt with by means of another variant of the **close by** operator – **close unique by**. This variant removes duplicates after each closure iteration, thus cycles do not imply infinite loops. Therefore the following query stops even if the workflow contains cycles:

(*Activity* **where** *name* = ”Planning”) **close unique by** (*outgoing.Transition.leadsTo.Activity*)

Another variant of the **close by** operator is the **leaves unique by** operator. It is a combination of the two previous variants. It returns only leaf objects, while preventing problems with graph cycles.

As we have shown, even relatively complex workflow queries can be easily programmed, if we formulate them in SBQL. Furthermore, the compositional nature of SBQL allows building well-structured queries which can be read and understood without excessive difficulties.

In order to type check a transitive closure expression first we have to determine the signature of the query q_1. Then we determine the signature of query q_2 in context of q_1. To pass the type checking, the signature of q_1 and the signature of q_2 in context of q_1 have to match.

5 Fixpoint Systems in SBQL

Fixpoint equations of the form $x = f(x)$ originated in mathematics as a very useful and general model of recursive systems, e.g. context-free grammars or recursive functions. In SBQL we use fixpoint equations in a different role: as a convenient option for programmers. SBQL provides querying capabilities based on fixpoint systems, i.e. queries of the form x :- $q(x)$, where x is a variable, q is an arbitrary SBQL query dependent on x. Starting from some initial x the system iteratively calculates $q(x)$ till the fixpoint is reached, i.e. till $x = q(x)$. The idea of SBQL fixed point equations is rooted in Datalog, with essential differences and extensions: (1) no limitation concerning an object model and (2) free combining fixpoint equations with other SBQL operators, including imperative constructs and abstractions; (3) simple pragmatics of applications. Majority of tasks that are expressed in Datalog can be very similarly expressed as SBQL fixpoint equations. However, for some tasks the conceptual and semantic differences may result in totally different practice. In contrast to Datalog that mostly attracted researchers with theoretical preferences, SBQL fixpoint equations are motivated by purely practical reasons: how to simplify programming of complex recursive tasks in business applications and how to simplify the maintenance of corresponding solutions.

A system of fixpoint equations can have arbitrary number of variables. The syntax of an SBQL fixpoint system is as follows:

$$\textbf{fixpoint}(x_{i1}, x_{i2},\ldots, x_{in})\ \{x_1 :- q_1;\ x_2 :- q_2;\ldots\ x_m :- q_m;\}$$

where:

- $x_1, x_2,\ldots, x_m$ are names of variables in this equation system,
- $x_{i1}, x_{i2},\ldots, x_{in}$ are returned variables, $\{x_{i1}, x_{i2},\ldots, x_{in}\} \subseteq \{x_1, x_2,\ldots, x_m\}$,
- $q_1, q_2,\ldots, q_m$ are SBQL queries with free variables $x_1, x_2,\ldots, x_m$;

The semantics of this language construct is the following:

1. Variables $x_1, x_2,\ldots, x_m$ are initialized to empty bags.
2. Queries $q_1, q_2,\ldots, q_m$ are evaluated.

3. If the results of $q_1, q_2,\ldots, q_m$ are equal to the values of $x_1, x_2,\ldots, x_m$, then stop (the fixpoint is reached). Otherwise assign the results of $q_1, q_2,\ldots, q_m$ to $x_1, x_2,\ldots, x_m$ and go to step 2.
4. The values of $x_{i1}, x_{i2},\ldots, x_{in}$ are returned as the result of the fixpoint query.

As queries $q_1, q_2,\ldots, q_m$ can reference variables $x_1, x_2,\ldots, x_m$, the fixpoint system provides recursive capabilities.

The simplest use of a fixpoint system in a query is the calculation of transitive closure. The query below uses a fixpoint system to find all activities connected via transitions with an activity named "Planning" (the query addresses the schema shown in Fig.1):

```
fixpoint (activities){
   activities :- (Activity where name="Planning") union
                 (activities .outgoing.Transition.leadsTo.Activity);
}
```

The system returns all activities that are reachable in the workflow graph from the Planning activity. Fixpoint systems are regular SBQL queries, and as such may be used as parts of other SBQL queries. The query below uses a fixpoint system as a part of a SBQL query, in order to find the longest single activity that may be reached from Planning:

```
max( (  fixpoint (activities){
           activities :-  (Activity where name="Planning") union
                          (activities .outgoing.Transition.leadsTo.Activity);
        }
      ).maxTime)
```

A fixpoint system may use some variables as a way to break down the problem into smaller, more manageable parts. The query below does that in order to find the length of the critical path of a business process starting with Planning:

```
fixpoint (final) {
   start  :- ((Activity where name="Planning") as x) join ((x.maxTime as howLong));
   path   :- start union   (      (path.x.outgoing.Transition.leadsTo.Activity as x)
                                  join ((howLong + x.maxTime) as howLong));
   final  :- max(path.howLong)
}
```

Only variable *final* is returned as the fixpoint result. The other two variables are used only to perform calculations, as their final values are inessential to the user. The variable *start* is used to find the top element of the hierarchy (the "Planning" activity), while *path* is the variable in which the results of recursive calculations are stored. The variables *final* and *start* do not have to be calculated recursively.

Fixpoint systems in SBQL fit well with the rest of the language. When compared with transitive closures, fixpoint systems used to express more complicated queries seem to be more readable, as decomposition of the problem is easier.

6 Type Checking Fixpoint Systems

Like other queries, a fixpoint system has to be typed. In the case of fixpoint systems the type inference is not as easy as with other queries. Since the type of each equation's result depends not only on the equation itself, but also on the other equations in the system, the signature of equation's result may also change in each of iterations. This means that the approach to type

inference used in other SBQL queries cannot be used in fixpoint systems. To determine the correct approach let us first to consider a fixpoint system with only a single equation, e.g.:

fixpoint (*activities*){
activities **:-** (*Activity* **where** *name*=”Planning”) **union**
(*activities .outgoing.Transition.leadsTo.Activity*); }

If we assume the activities variable to be of *any* type for the time being, then we can reduce the equation for type inference purposes to the following equation:

activities **:-** (*Activity* **where** *name*=”Planning”);

We can determine the signature of this equation. It is $i_{Activity}$[*card*=0..*]. In the next step, we have to verify, that the inferred type is correct. In order to do this, we use the original equation and determine its signature, using the signature inferred in the previous step for the *activities* variable. If the signature calculated in this step is the same as the signature inferred in the previous step, then the inference is correct.

When a fixpoint system consists of more than one equation, additional steps are necessary. Let’s consider a simple fixpoint system that consists of three equations:

fixpoint (*final*) {
start **:-** ((*Activity* **where** *name*=”Planning”) **as** *x*) **join** ((*x.maxTime* **as** *howLong*));
path **:-** *start* **union** ((*path.x.outgoing.Transition.leadsTo.Activity* **as** *x*)
join ((*howLong* + *x.maxTime*) **as** *howLong*));
final **:-** *max*(*path.howLong*)
}

If we assume all three variables in those equations to be of *any* type, and try to infer the signatures of all three equations simultaneously, we encounter a problem. Although the inference of the *start* variable’s signature works correctly, the *path* and *final* variables’ signatures will be inferred incorrectly. In order to infer the *path* equation’s signature we need the “real” (already inferred) signature of *start* equation, and to infer the *final* equation’s signature we need the “real” signature of *path* equation. To solve this problem we need first to perform stratification of the fixpoint system. This particular fixpoint system will be divided into three strata:

1. *start* **:-** ((*Activity* **where** *name*=”Planning”) **as** *x*) **join** ((*x.maxTime* **as** *howLong*));
2. *path* **:-** *start* **union** ((*path.x.outgoing.Transition.leadsTo.Activity* **as** *x*)
join ((*howLong* + *x.maxTime*) **as** *howLong*));
3. *final* **:-** *max*(*path.howLong*)

Now, we may proceed with the inference, starting with the lowest stratum (1). Each stratum will be processed separately. The results of type inference in a lower stratum will be used to infer the types in a higher stratum. Stratification, however, not always solves the problem. Consider the following fixpoint system:

fixpoint (*odd, even*){
odd **:-** (*Activity* **where** *name*=”Planning”) **union**
(*even .outgoing.Transition.leadsTo.Activity*);
even **:-** (*odd .outgoing.Transition.leadsTo.Activity*);
}

The *odd* and *even* equations depend on each other, so they will be placed in the same stratum. The first iteration of the inference algorithm will not produce correct results. The correct result for this (and similar) fixpoint system could be achieved in one of subsequent iterations.

However treating fixpoint system's type inference as another fixpoint system is dangerous. Some evaluations may never reach the fixpoint, e.g.:

x **:-** 1 **union** (*x* **as** *a*)

This equation generates infinitely nesting binders. The possibility of a type checking system falling into infinite loop is unacceptable. Thus, a fixpoint-style signature inference cannot be used. In order to solve this problem, we give the user the possibility to explicitly define types of equations. The fixpoint definition becomes as follows:

fixpoint(x_{i1}, x_{i2},…, x_{in}) {x_1 : sig_1 :- q_1; x_2 : sig_2 :- q_2;… x_m : sig_m :- q_m;}

where:

- x_1, x_2,…, x_m are names of variables in this equation system,
- x_{i1}, x_{i2},…, x_{in} are returned variables, $\{x_{i1}, x_{i2},\ldots, x_{in}\} \subseteq \{x_1, x_2,\ldots, x_m\}$,
- sig_1, sig_2, … , sig_m are optional signature definitions,
- q_1, q_2,…, q_m are SBQL queries with free variables x_1, x_2,…, x_m;

This approach allows explicitly defining the equation signatures in situations in which they are too difficult for the type inference algorithm to determine, while allowing the user to omit the definition for simpler ad-hoc queries. The type inference algorithm works as follows:

1. Stratify the equation system
2. Process each stratum, starting from the lowest:
   a. Assume variables in this stratum to be of type *any* unless explicit type is provided
   b. Infer the equations' signatures using this assumption
   c. Verify the inference
   d. If the verification is not successful, report a type checking error

Such an option is currently implemented in ODRA.

7 Recursive Procedures and Views in SBQL

The SBQL philosophy allows for seamless integration of imperative language constructs, including recursive procedures and functions with query operators. This allows utilizing the most popular recursive processing technique, without sacrificing any of the benefits of a query language. In contrast to popular programming languages the new quality of SBQL concerns types of parameters and types of functions output. The basic assumption is that parameters are any SBQL queries and the output from functional procedures is compatible with query output. Thus SBQL procedures and functions are fully and seamlessly integrated with SBQL queries. Statements in SBQL procedures use SBQL queries too. An SBQL query preceded by an imperative operator is a statement. Statements such as *if, while, for each*, etc. can be more complex, see [3]. SBQL includes many such imperative operators (object creation, flow control statements, loops, etc.).

Below we present a recursive procedure which finds all activities connected that may be reached via transitions from a specified activity. It consists of a single return statement. The returned value is an empty collection or the result from recursive invocation of the same procedure. For simplicity in the examples we skip typing.

```
procedure ActivityGraph(startingActivity) {
  return if count(startingActivity) = 0 then bag()
         else (startingActivity
              union ActivityGraph(startingActivity.outgoing.Transition.leadsTo.Activity)); }
```

The procedure takes a reference to activity or a collection of such references as the parameter (*startingActivity*). An example procedure call is the following:

ActivityGraph (*Activity* **where** *name*=”Planning”)

An advantage of recursive procedures is simplicity of the problem decomposition. A recursive task can be easily distributed among several procedures (some of which may be reused in other tasks). For example, a procedure calculating the total amount of resources required by unfinished activities following a given activity is shown below. The procedure utilizes the previously defined *ActivityGraph* procedure in order to perform the recursive processing and then performs calculations, on local variables (introduced by **create local**).

```
procedure RequiredResources(startingActivity) {
  if not exists(startingActivity) then return bag();
  create local ActivityGraph (startingActivity) as activities;
  create local (activities.requires.RequiredResources) as reqResources;
  create local (reqResources.(oftype.Resource.name as res, amount as howMany))
                                                              as reqResDetails;
  create local distinct(reqResDetails.res) as resList;
  create local ((resList as resource).
                  (resource, sum((reqResDetails where res=resource).howMany))) as result;
  return result;
}
```

Recursive procedures in SBQL offer many advantages when compared to stored procedures in relational DBMSs. Most of them are consequences of the fact that procedures in SBQL are based on SBA, working on the same principles and evaluated by the same evaluation engine, while in relational systems stored procedures are add-ons to the system evaluated separately from SQL queries. SBQL queries are valid as expressions, procedure parameters, etc. The type system is the same and there is no impedance mismatch between queries and programs.

SBQL updateable views are based on procedures and as such can be recursive and can utilize any other SBQL option, in particular parameters. Note that recursion without parameters makes little sense, thus if one assumes that views can be recursive then they must have parameters too. Recursive parameterized views are not available in any query language but SBQL. A simple read-only view, returning all subparts of parts which names are passed as a parameter, is shown below.

```
create view SubActivitiesDef {
  virtual objects SubActivities (whichActivity){
      if count(whichActivity) = 0 then return bag();
      create local (Activity where name ∈ whichActivity) as a;
      return (a union SubActivities(a.outgoing.Transition.leadsTo.Activity.name)) as b;
  }
  on_retrieve do { return deref(b); }
}
```

Below we present an example view invocation.

SubActivities (“Planning”) **where** *status* = “completed”

SBQL updateable views are discussed in detail in several sources, e.g. in [10, 3].

8 Conclusion

We have presented recursive query processing capabilities for the object-oriented Stack-Based Query Language (SBQL). As SBQL offers very powerful and flexible recursive querying capabilities, we could use this language e.g. to express many workflow-oriented queries. Although they are inherently complex, their formulation in SBQL is compact and readable in comparison to other methods.

The transitive closure allows one to write queries which are more powerful and easier to read than SQL queries when compared with Oracle and DB2 SQL variants of transitive closure operators. This also concerns XML-oriented repositories. SBQL makes XML data processing much easier, especially concerning recursive tasks. Fixpoint systems provide SBQL with recursive capabilities similar to deductive query languages. However SQBL offers much more freedom, as there is no restriction on operators which may be used within the queries.

We have described type inference technique for transitive closures and fixpoint systems. It amounts to an engineering trade-off between the desire for full type inference and limited computational complexity of the type checking process. If the types cannot be automatically inferred, the user is obliged to state the types explicitly. Explicit type declaration (regardless of possible positive type inference) is strongly encouraged in application code for the sake of readability and maintainability.

We have also shortly presented recursive procedures (functions) and recursive views in SBQL. Although recursive procedures are available in almost all professional programming languages, SBQL makes important extensions: parameters for procedures can be queries and all statements in procedures can use queries too, including a *return* statement from a function.

References

[1] S.Abiteboul, R.Hull, V.Vianu: *Foundations of Databases*. Addison-Wesley 1995

[2] K.Subieta, Y.Kambayashi, and J.Leszczyłowski. Procedures in Object-Oriented Query Languages. in: *Proc. VLDB Conf.,* Morgan Kaufmann, 182-193, 1995.

[3] K.Subieta. *Theory and Construction of Object-Oriented Query Languages* (in Polish), PJIIT - Publishing House, 2004, 522 pages

[4] K.Subieta. Stack-Based Approach (SBA) and Stack-Based Query Language (SBQL). http://www.sbql.pl, 2008

[5] Rodan Systems S.A.: OfficeObjects®Workflow, 2006. http://www.rodan.pl/en/produkty/officeobjects/?dzial=workflow/

[6] Workflow Management Coalition, Workflow standard, The Workflow Reference Model, WfMC-TC-1003 issue 1.1, Jan 1995.

[7] Workflow Management Coalition, Workflow standard, Workflow process definition language – XML process definition language, WfMC-TC-1025 draft 0.03a, May 2001

[8] M.Momotko, K.Subieta. *Business Process Query Language - a Way to Make Workflow Processes More Flexible*. Proc. 8th East-European Conference on Advances in Databases and Information Systems (ADBIS), September 2004, Budapest, Hungary, Springer LNCS 3255, pp.306-321

[9] M.Momotko. *Tools for Monitoring Workflow Processes to Support Dynamic Workflow Changes*. PhD Thesis, 2005, http://www.sbql.pl/phds

[10] H.Kozankiewicz. *Updateable Object Views*, PhD Thesis, Institute of Computer Science, Polish Academy of Sciences, 2005, http://www.sbql.pl/phds