Using Active Objects for Query Processing *

Ralf Jungclaus
Gunter Saake
Abt. Datenbanken, Techn. Universität Braunschweig
P.O. Box 3329, 3300 Braunschweig, FRG
Cristina Sernadas
Departamento de Matemática, Instituto Superior Técnico
1096 Lisboa Codex, Portugal

Abstract

Most object-oriented databases can only be accessed by a programming language rather than a query language, although an ad hoc query facility is a desirable feature of such systems. In this paper, we present a way to process queries to extract data from a collection of interacting objects. We adopt the object-oriented paradigm to describe the operational semantics of queries given as expressions of a query algebra. An expression can be transformed into a collection of active objects performing the operations. We show that we are able to implement query processing using the existing framework for collections of objects, which is based on objects modeled as processes. Our approach makes parallel query processing possible.

1 Introduction

A central feature of modern database systems is an *ad hoc query facility* (cf. [Mai83, Dat86]) offering efficient access to stored information. Although the field of *object-oriented database systems* (OODBS) is rapidly growing [DD86, Dit88], query languages for OODBS are still rough (cf. [ZM89]). Most researchers in the field of OODBS however do consider an ad-hoc query facility to be a mandatory feature of such systems [ABD$^+$89]. A major difficulty seems to be the contrast between the associative access to objects (being provided by relational query languages currently available) and the encapsulation principle for objects.

We propose a query algebra for object-oriented information systems. The query algebra tries to integrate relational query concepts with the OBLOG model for information systems [SSE87]. The query algebra should be regarded as a formal basis for a high-level query interface for an OODBS. In the OBLOG model, an information system is conceptually modeled by a society of interacting objects [SFSE89]. These objects retain, carry,

*The work was partially supported by ESPRIT BRA WG 3023 IS-CORE (Information Systems – COrrectness and REusability)

generate or compute information, i.e., they are the basic building blocks of information systems. The model has been formalized resulting in a concept of object as an observable process [ESS88, ESS90].

The evolvement of an object society is driven by the activity of certain objects. Hence, the system environment rather than queries is responsible for the manipulation of objects. An object society may be regarded as an active database (cf. [Day88]).

We regard an ad hoc query facility to be a means to extract information from the *current state* of a collection of objects. The current state of an object society includes only structural information, i.e. objects and their attributes. The current state is evaluated and represented by *meta-objects* keeping track of changes in the population of the object society and providing access to the state of objects.

In contrast to other approaches [SZ89, CDLR89], the result of queries in our approach are not objects but *data*. Information *about* objects is not enough to constitute virtual objects derived from queries. The derivation of objects from an existing object society is very difficult, since objects in OBLOG are regarded as *observable processes*. Hence, an object is not only described by its structure but also by the set of all its possible life cycles, which must be computed then. This is the motivation to capture queries by a separate framework and language. This approach is used e.g. in IRIS [F^+87] and in O_2 [CDLR89]. In contrast, a computational complete language capturing both object definition and object manipulation is used e.g. in GemStone [MS87].

Queries against object societies are processed by *active objects*, which interprete the operations and are responsible for the evaluation of query algebra terms. Thus, we are representing query operations by objects and hence adopt the object-oriented paradigm to describe query processing. Since objects are processes, we are able to execute queries *in parallel* and *distributed* by means of interaction. Our schema of execution resembles graph reduction techniques [TBH82]. The computation of terms is postponed until necessary, which is known as *lazy evaluation*. A similar approach to query processing was presented in [BFN82].

The query system is conceptually an object society of its own. It is defined using the same concepts being used for any other object society. Therefore, it can be implemented *using an existing environment* and can thus be easily integrated into the environment. The interface to the society being queried is made up by the meta-objects mentioned above. Thus, we are able to implement a query system as just another application in an environment for active object societies. Furthermore, query objects may be used by ordinary objects to issue queries themselves.

In this paper, we are focussing on the operational semantics of queries. In the next section, we will introduce object societies briefly. The third section gives a short introduction to our query algebra. In section 4, we will give an overview of the query processing using active objects. Section 5 deals with the meta-objects representing the state of an object society. In section 6, we are discussing the interpretation of queries by active query objects. Before we give a short outlook on issues yet to be investigated, we describe the execution of queries in section 7.

2 Object Societies

In the OBLOG approach, an information system is conceptionally regarded as a society of interacting objects. The concept of object in OBLOG is based on the observation that the parts of an information system (i.e. storage components as well as process components) have a distinct temporal and spatial existence [SFSE89]. Objects enable uniform description of both passive and active components. The *state* of each object can be described by data, the *evolvement* can be described by changes in the current state of an object. Thus, the object concept in OBLOG covers both structural and dynamic aspects of objects.

The internal state of an object is described by attribute values. The set of *attributes* describes the structure of the object. Since objects are dynamic entities, the state can be changed. This evolvement of objects takes place in a discrete manner by the occurrence of *events*. A modification event changes the values of certain attributes and corresponds to a set of basic operations on the state of an object. Events also bring objects into existence (these events are called *birth events*) and destroy objects (the *death events*).

A sequence of events beginning with a birth event describes the life of an object. To specify the behavior of an object, the set of possible life cycles has to be specified. Hence, the behavior of an object is described by a *process* (cf. [Hoa85, Hen88]).

An object is semantically described by a set A of attributes, a set X of events, a set Λ of permitted life cycles over X (which may be finite or infinite) and a valuation mapping giving the values of attributes after sequences of events. Objects are defined over a universe of data types being either predefined or user-defined data types.

In the remainder, we will use an easy to understand subset of the OBLOG language for specifying sample objects. As an example, consider an object `Dept` representing a department of a company.

Example 2.1

```
object Dept
  data types
    money;
    |PERSON|
  attributes
    turnover: money;
    employees: set(|PERSON|);
    manager: |PERSON|;
  events
    birth establish_dept;
    death close_dept;
    assign_turnover(money);
    assign_manager(|PERSON|);
    hire_employee(|PERSON|);
    fire_employee(|PERSON|);
  valuation
    variables P: |PERSON|, m: money;
      [assign_manager(P)]manager = P;
      [hire_employee(P)]employees = insert(P, employees);
      [fire_employee(P)]employees = delete(P, employees);
      [assign_turnover(m)]turnover = m
  safety
    variables P: |PERSON|;
      {sometime(after(hire_employee(P)))}fire_employee(P);
end object Dept;
```

□

In our abstract view of a department, only the turnover, the manager and the set of employees working in that department are of interest to us – these are the attributes of the object. The state of a department can be changed by the events given under the keyword **events**. The valuation equations give the values of the attributes changed by certain events. The rules under the keyword **safety** are used to specify the set of permitted life cycles in a declarative way. In this case, an employee can only be fired if he/she has been hired before. Note that the domain for the attribute `employees` are sets of identifiers for objects of type `PERSON`, i.e. the attribute is object-valued. So is the attribute `manager`. |`PERSON`| denotes the data type of surrogates for the object type `PERSON`.

The object `Dept` is *passive* in the sense that it does not evolve by its own initiative. The evolvement events are triggered from outside a passive object. If an object is capable to make certain events happen by its own initiative, the object becomes *active*. We call these special events *active initiatives*. Whenever an active initiative is permitted to occur, one of the possible alternatives will be chosen to be fired. If we require the permitted life cycles to include certain events, we have to state a *liveness requirement* which indicates the events which have to occur before the object is allowed to die. In order to fulfill a liveness requirement the object forces the occurrence of other events which have to precede the obligatory events according to the permitted life cycles.

Consider for example a calendar. The object representing it has to count the days by its own initiative. Thus, we have to specify an event `next_day` which is an active initiative. Note that the operator $*$ denotes infinite iteration.

Example 2.2

object `Calendar`
  attributes
    `day`: [1..31];
    `month`: [1..12];
    `year`: nat;
  events
    birth `start`;
    `next_day`;
  valuation
    ...
  initiatives
    `next_day`$*$
end object `Calendar`;

□

Developing an abstract view of a Universe of Discourse, we tend to group similar objects into *types*, i.e. we regard such objects to be occurrences of an archetype. To distinguish these occurrences, we need to have a way to identify them uniquely. Therefore, we associate with an object type an abstract data type such that the carrier set of values is the set of possible *surrogates* for the occurrences.

An occurrence (or *instance*) of a type is an object *template* qualified by a surrogate. The occurrences of a type have the same schema and can be regarded as isomorphic copies of an archetype. As an example, consider the type `PERSON`: Each person is identified by his/her name and birthdate. The set of all possible tuples of names and birthdates constitutes the surrogate space for persons. We refer to the set of all possible surrogates of type PERSON using the notation |PERSON|. Here, we give a fragment of the specification of the object type PERSON.

Example 2.3

```
object type PERSON
  surrogate
    name: string;
    birthdate: date;
  template
    attributes
      soc_sec_no: integer;
      ...
    events
      birth birth;
      death death;
      ...
      become_manager;
    ...
end object type PERSON;
```

□

Objects and object types are not enough to constitute a society of objects. Objects may have a complex structure and they may be related in various ways. Last, but not least, they have to be able to *interact*.

Relationships between objects are based on *inheritance*. An inherited object is *included* preserving encapsulation, since attributes of the inherited object may only be changed by events local to it. These events may, however, be called from outside this object.

If we are dealing with *roles* of objects, we regard the specialized object to be a *view* of the parent object. The view object however does still have the properties of the parent object, i.e. it inherits it. Specialization in contrast is *static*, i.e. the specialized object is born as a special kind of the parent object and remains of this kind for it's entire life.

Consider a manager as a view of a person. A particular manager is a particular person at the same time. Syntactically, we have to add attributes, events, and rules to describe the special structure and behavior of the view object. Semantically, the object representing the manager as a person is inherited. A specialization of a person in contrast is e.g. a woman.

Extending the view concept to object types, we have to state that the current set of surrogates of the view type ot_1 is contained in the current set of surrogates of the parent type ot_2 and each instance ob of ot_1 is a view of the same instance ob of ot_2, i.e. we constitue an IS-A relationship between instances.

The following example shows the specification of the view type MANAGER as a view of type PERSON.

Example 2.4

```
object type MANAGER
  view of PERSON;
  template
    attributes
```

```
        ...
    events
        birth PERSON.become_manager;
    ...
end object type MANAGER;
```

□

Objects in a society do not always have a simple structure. The OBLOG approach allows *aggregation*, i.e. composition of objects from sets or lists of objects as well as from tuples of objects. For further information on aggregating objects see [SFSE89, ESS90]. Also possible are arbitrary relationships between objects.

An object society is a possibly large collection of objects which interact. Interaction thus is the mechanism for integrating objects into the society. A simple case of interaction between objects is *event calling*. To call an event means to synchronize life cycles on events being triggered by the calling object.

Consider for example the promotion of a person to become a manager of a department. The `assign_manager`-event of the department calls the `become_manager`-event of the corresponding person identified by the surrogate P:

Example 2.5

```
global interactions
    variables P: |PERSON|;
        Dept.assign_manager(P) >> PERSON(P).become_manager;
```

□

Note that event calling is also be used for interactions with inherited objects.

3 The Query Algebra

Since we only want to retrieve structural information from an object society, we define the *global state* of an object society used as a representation of the current state of such a society.

The global state maps singular *objects* into tuples consisting of the attribute values, and *object types* into sets of surrogates (of the currently existing objects of that type) together with the corresponding tuples. *View relations* and *specialization relations* are derived from the signature of the object society together with the extension of the objects and object types, associating a tuple containing local as well as inherited attribute values to each view object. We denote the state of objects and object types by their name surrounded by brackets (e.g. [`PERSON`]).

The use of a query algebra is motivated by the advantages of the relational algebra for accessing relational databases. In contrast to the one-sorted classical relational algebra, our query algebra is *many-sorted*, i.e. it contains algebras for basic data types and instantiations of parameterized constructors like *set* and *list* as subalgebras. We use the well-known basic query operators like **select** and **project** as well as operators for additional structural concepts. Complex objects give rise to the introduction of concepts based on nested relational algebras (cf. [SS86]). In the remainder of this section, we shortly introduce some operators of our query algebra using small examples.

Since the data types of the society are subalgebras, we are able to use values and operators of these algebras in query terms. Additionally, we can use the operators and predicates associated with type constructors like e.g. `set_of`, `tuple(a:...)`. The set-oriented operations like **union** or **intersect** known from relational algebra hence are special cases of data type operations.

The selection operator **select** selects elements of a set using a boolean predicate with a free variable `SELF` of the inputtype. Suppose we want to select the tuples associated to objects of type `PERSON` having the name 'Smith'. The corresponding algebra term is

$$\textbf{select}_{\texttt{SELF.name='Smith'}}([\texttt{PERSON}])$$

In the remainder of this section, we will omit the variable `SELF` if no ambiguity arises.

The operator **project** extracts certain indicated attributes of tuples in a relation. As an example consider the algebra term computing a result relation from the relation [`PERSON`] having only the attributes `name` and `soc_sec_no`:

$$\textbf{project}_{\texttt{name,soc\_sec\_no}}([\texttt{PERSON}])$$

The result of this term is of type '`set(tuple(name:string, soc_sec_no:nat))`'.

As a generalization of the project operator, instead of a list of attributes a list of functions on the attributes on the input type together with component names can be given as proposed in [SZ89]. We call the operator **construct**. The following term computes for each department the turnover per employee. Note that we use an operation **Card** defined for the parameterized type constructor `set_of` computing the cardinality of a set.

$$\textbf{construct}_{\texttt{name:dept\_name,tpp:(turnover/}\textbf{Card}\texttt{(employees))}}(\texttt{DEPT})$$

In the OBLOG model, there are more interesting relations between objects than there are in the relational model. Hence, we introduce additional operators of type `join`.

The general `join`-operator computes the join of two input relations depending on a boolean predicate with free variables `LEFT` and `RIGHT` of the corresponding input types. As a special case we have the join without qualification with a predicate equal to **true**. It is similar to the cartesian product of sets except for the elimination of one tuple construction step.

$$[\texttt{PERSON}]\ \textbf{join}\ [\texttt{DEPT}]$$

The next example computes the managers who have been promoted on the establishment date of a department using an equijoin:

$$[\texttt{MANAGER}]\ \textbf{join}_{\texttt{LEFT.promotion\_date=RIGHT.est\_date}}\ [\texttt{DEPT}]$$

The **viewjoin**-operator joins elements of view types if they are views of the same base object. An example is the algebra term representing the query *"All managers currently being a patient"*. We assume the type `PATIENT` to be another view of type `PERSON`.

$$[\texttt{MANAGER}]\ \textbf{viewjoin}\ [\texttt{PATIENT}]$$

The result tuples contain all attributes of `MANAGER`, `PATIENT` and of the base object type `PERSON`.

The **apply**-operator applies a function to a set of tuples. It has the same semantics as the **image**-operator in [SZ89]. Examples for the use of that operator are

$$\mathbf{apply}_{\mathtt{turnover/}\mathbf{Card}\mathtt{(employees)}}(\mathtt{[DEPT]})$$

giving the set of turnover per employee values or

$$\mathbf{apply}_{\mathtt{employees}}(\mathbf{select}_{\mathtt{SELF.name='Sales'}}(\mathtt{[DEPT]}))$$

giving the set of employees of the sales department.

We conclude the presentation of the algebra operators with the **materialize**-operator. It delivers the relation associated with a set of objects identified by its surrogates. The idea is based on the `PROPERTY`-operator proposed by Codd [Cod79]. The following term computes the relation containing the tuples associated with the employees of the sales department:

$$\mathbf{materialize}(\mathbf{apply}_{\mathtt{employees}}(\mathbf{select}_{\mathtt{SELF.name='Sales'}}(\mathtt{[DEPT]})))$$

Further extensions of the algebra include the introduction of the parameterized data type constructors `bag_of` and `list_of` in order to make operations like sorting and duplicate handling possible (cf. [SLPW89]). Aggregate functions like `average` can be introduced as functionals on *collections* as a generalization of the data type constructors `list_of` and `bag_of`.

4 Overview of the Query System

The initial idea was to define a query system for societies of objects using the same framework which is used for the object society itself, i.e. to adopt the OBLOG model to describe the operational semantics of queries.

As mentioned before, we employ active objects for the processing of operations in a query expression. These objects are called *query objects*. A query object represents one operation in the query expression. Since terms can itself be arguments of operators, an expression of the query algebra can be represented by a hierarchy of query objects. Thus, we transform each node of the operator graph representing the query expression into a query object which inherits the query objects representing subterms. Query objects are described in detail in section 6.

The innermost operations of a query expression access the current state of an object society. The current state has to be evaluated and represented. In our approach, we use *state objects* to accomplish this task. One state object is associated with each object type in the society being queried (we treat single objects as object types with just this one object being the population). State objects are monitoring the population of the corresponding object class and collect the local states of the instances on demand. Reading the values of attributes does not violate encapsulation of the instances because we consider events and attributes to be the interface of an object. In the following section, state objects are explained in detail.

State objects are particularly defined for the object types being present in a society. They are not uniformly defined. Thus, they cannot be inherited by query objects requesting access to the state of an object class because we cannot specify inheritance in advance. Since the query objects are defined in advance, a state object dynamically inherits the query object having the state of an object class as input operand. The relation between query objects and state objects is described in section 6.

To give an impression for the conceptual representation of a query, consider the following one:

Example 4.1

> *Get the turnover values of all departments led by managers called X currently being patients.*

The corresponding algebra expression is:

$$
\begin{array}{l}
\mathbf{apply}_{\texttt{turnover}}\ (\\
\quad ((\mathbf{select}_{\texttt{name}=X}[\texttt{MANAGER}])\ \mathbf{viewjoin}\ [\texttt{PATIENT}]) \\
\quad)\ \mathbf{join}_{\texttt{LEFT.Dept=RIGHT.id}}\ [\texttt{DEPT}] \\
)
\end{array}
$$

Note that we now consider `DEPT` to be the object type representing departments. The surrogate is denoted by `id`.

The structure of the query objects representing the query given above is shown in figure 4.1. The edges denoting inheritance of query objects yield a straightforward representa-

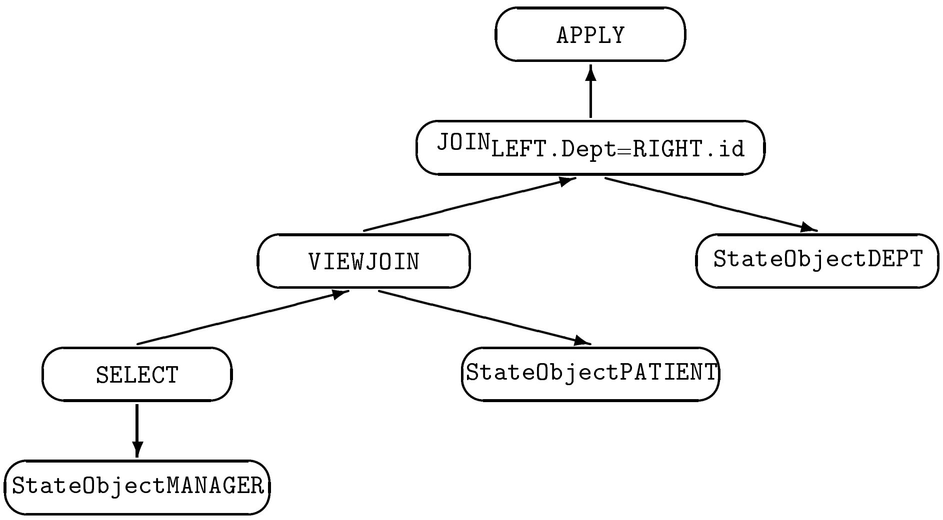

Fig. 4.1: Structure of query objects representing a sample query

tion of the operator graph for the query. Details on computation can be found in section 7.

5 Representation of the Global State

The query algebra operates on the global state of an object society. In our approach, we use state objects for evaluation and representation of the global state. A state object is associated with each object type being defined for an object society.

Each object (instance) currently existing in the society is inherited by the state object of the corresponding object type. Thus, the state object is capable to read the attribute values describing the state of each instance in the object class.

A state object keeps track of changes in the current population of the type. We specify events to alter the attribute containing the set of surrogates of the existing object instances by the events `insert_object` and `delete_object`. The birth event of every instance calls the corresponding `insert_object`-event, the death event calls the `delete_object`-event of the state object.

Consider the specification of the (passive role of the) state object for the object type `PERSON`. Note that the **inheriting**-clause describes encapsulated inclusion of all current instances of type `PERSON`.

Example 5.1

```
object StateObject_PERSON
   inheriting P in PERSON as EXTENSION;
   attributes
      population: set(|PERSON|);
   events
      birth become_SO_PERSON;
      insert_object(|PERSON|);
      delete_object(|PERSON|):
      request;
      ready;
   valuation
      variables P: |PERSON|;
         [insert_object(P)] population = insert(P, population);
         [delete_object(P)] population = delete(P, population);
   interaction
      variables P: |PERSON|;
         EXTENSION(P).birth >> insert_object(P);
         EXTENSION(P).death >> delete_object(P);
end object StateObject_PERSON;
```

□

Whenever a state object is requested to deliver the state of the corresponding object type, it becomes active. We are modelling this behavior using the concept of *phase transition*. That is, we are specifying the active phase as a view of the state object. The view object comes into existence when a state object is called to deliver the state, i.e. the corresponding event in the life of a (passive) state object is the birth event of the active view object. The active phase of the state object for the object type `PERSON` is defined as follows:

Example 5.2

```
object Active_SO_PERSON
   view of StateObject_PERSON;
      attributes
         state: set(tuple(...));
      events
         birth StateObject_PERSON.request;
         death StateObject_PERSON.ready;
         insert_state(|PERSON|, tuple(...));
      valuation
         variables P: |PERSON|, t: tuple(...);
            [insert_state(P, t)] state
```

```
              = insert(t, state);
    safety
        variables s:tuple(...); P:|PERSON|;
            {[in?(P, StateObject_PERSON.population)]⇒
                (exists r:tuple(...)) sometime(after(insert_state(P,r)))
                } StateObject_PERSON.ready;
            {¬ sometime(after(insert_state(P,s)))} insert_state(P,s);
    liveness
        goals
            StateObject_PERSON.ready
    initiatives
        variables P: |PERSON|, t: tuple(...);
            StateObject_PERSON.ready;
            insert_state(P, t);
end object Active_SO_PERSON;
```

□

The safety requirements state that the `ready`-event is only allowed to happen after every object instance of that type has been asked for its state and that each instance's state is only inserted once.

Since the `ready`-event is a liveness requirement and an active initiative, the active view of the state object has the obligation to query all instances to fulfill that requirement.

When the event `request` of the passive state object `StateObject_PERSON` happens, the active view of it is created. The active object then collects the state of every instance and dies after having delivered the state of the object class. The passive object `StateObject_PERSON` then changes to the state `waiting`.

Regarding inheritance in the OBLOG model, we are able to specify state objects for complex object types, view types and specializations. The state of a type then contains the inherited attributes, too. Consider the subtype `MANAGER`. The relation representing the state contains the attributes of `PERSON` and the attributes of `MANAGER`.

6 Query Objects

As outlined in section 4, we use active query objects to perform the evaluation of queries against object societies.

If we were to interpret a query expression at runtime, the specifications for query objects would have to be generated, compiled, and executed *at runtime*. Hence, we would have to extend the existing environment for object societies. Additionally, the specifications for query objects could not be reused.

To overcome these disadvantages, we introduce *operator object types* (OpOTs). An OpOT is a parameterized object type to represent an operator of the object algebra. This enables us to define a query system in advance. At runtime, instances of OpOTs are created according to the operators in the query expression. The query then can be regarded as an object society which is interacting with the object society to be queried. This approach allows *reusability* of query object specifications and the *use of the existing environment* for object societies.

Since structure and dynamics of query objects look similar, OpOTs are *specializations* of an object type `QUERY_OBJECT`. This object type can be regarded as a uniform interface to each query object.

Since every query object is an instance of an OpOT, it has to have a surrogate value as identifier. Every query object can be identified by the term it represents. Thus, the surrogate space for the type `QUERY_OBJECT` is the set of all correct *query algebra expressions*. We do not introduce the definition of the suitable data type `QUAL_Term` here, but suppose it to be available. Note that the data type of the attribute `result` is the parameterized type `inputtype` which is instantiated whenever a query object is created.

The object type `QUERY_OBJECT` is defined as follows:

Example 6.1

```
object type QUERY_OBJECT
   data types QUAL_Term, inputtype;
   surrogate id: QUAL_Term;
   template
      attributes
         state: enum{waiting, working, ready, error};
         result: set(inputtype);
      events
         birth Start;
         death Throw_away;
         Request_Input;
         Input_Ready;
         Insert_into_Result(inputtype);
         Ready;
         Error;
      safety
         variables t:inputtype;
         {sometime(after(Request_Input))} Input_Ready;
         {sometime(after(Input_Ready))} Insert_into_Result(t);
         {(state=ready) or (state=error)} Throw_away;
         {¬ sometime(after(Error))} Request_Input;
         {¬ sometime(after(Error))} Input_Ready;
         {¬ sometime(after(Error))} Insert_into_Result(t);
         {¬ sometime(after(Error))} Ready;
         ...
      valuation
         variables t:inputtype;
         [Insert_into_Result(t)]result = insert(t, result);
         [Insert_into_Result(t)]state = working;
         [Ready]state = ready:
         [Error]state = error;
         ...
      liveness
         goals
            Ready or Error;
      initiatives
```

```
        variables t:inputtype;
        Request_Input;
        Insert_into_Result(t);
        Ready or Error;
end object type QUERY_OBJECT;
```

□

The state of a query object is described by the current state of execution (attribute `state`) and the result relation.

Query objects are active. Thus, their evolvement has to be driven by events being active initiatives. Before being able to evaluate the operation it represents, the query object has to request the input operand(s) by firing the `Request_Input`-event. Then, the result tuples have to be inserted into the result relation according to the operation. A successful life cycle includes the `Ready`-event, which is allowed to happen after each input tuple has been processed. Thus, the liveness requirement states that the object either has to finish the evaluation (i.e. the event `Ready` has to occur) or it has to indicate an error. This is reported to the parent query object or the environment. The query object then has accomplished its mission.

The permitted life cycles are specified by the rules under the keyword **safety**. Note that the `Error`-event can occur at any time, but only the death event `Throw-away` is allowed to happen after an error has been indicated.

Each query object *inherits* those query objects representing a nested term. Nested terms are computed by the operation `sub_term`(Q) for an element Q of the ADT `QUAL_Term`. Note that a query object uses the attribute `result` of the inherited query object(s) as input operand(s). To improve readability, we rename the result attribute.

As an example for an OpOT we give the definition of the type SELECT since we can explain the main concepts of OpOTs using one example:

Example 6.2

```
object type SELECT
  data types Cond_Type;
  specializing from QUERY_OBJECT where op(self.id) = 'SELECT';
  constants condition: Cond_Type;
  template
    inheriting Q in QUERY_OBJECT
      where Q.id = sub_term(self.id) as SUBQUERY
        renaming result as input;
    safety
      variables t:inputtype;
      {in(t,input) and fulfill(t,condition)} Insert_into_Result(t);
      {¬ sometime(after(Insert_into_Result(t)))} Insert_into_Result(t);
      {sometime(after(Input_Ready)) and
        (for all(s:inputtype) in(s,input) and fulfill(s,condition) ⇒
          sometime(after(Insert_into_Result(t)))
        )
      } Ready;
    interaction
      calling
```

```
            Request_Input >> SUBQUERY.Start;
            SUBQUERY.Ready >> Input_Ready;
            SUBQUERY.Error >> Error;
end object type SELECT
```

□

Since the object type SELECT is a specialization of the type QUERY_OBJECT, semantically everything specified for the type QUERY_OBJECT is valid here.

The boolean expression being the **select**-condition remains constant through the lifetime of an instance and is therefore represented by the constant identified by `condition`. We assume a suitable data type for representation of conditions to be available which includes the predicate `fulfill` evaluating to **true** if an input instance fulfills the condition.

In order to synchronize the behaviors of the connected query objects, we have to specify interaction. The `Request_Input`-event calls the birth event of the inherited object thus initiating the evaluation of the pending subquery. When the subquery has been evaluated, the `Input_Ready`-event is called by the inherited object.

The computation of the result is described by the first safety rule: if a tuple representing an object's state fulfills the select condition, the event describing the insertion of the tuple into the result relation is allowed to occur. The third rule states that the `Ready`-event is only allowed to occur after the `Input_Ready`-event has occurred and each tuple in the input relation fulfilling the select condition has been inserted into the result.

Please note that query objects may easily be linked to or inherited by ordinary objects. Therefore, objects may issue queries and may access results of queries. Results of queries can be made persistent by assigning them to an attribute of a persistent object.

7 Execution of Queries

A query is represented by an operator graph. Since each node of the operator graph is transformed into a query object, we achieve a structure of query objects to process a query. This corresponds to the representation of an expression by a graph connecting subexpressions by pointers. It is then possible to employ a *graph reduction technique* [TBH82] for query evaluation. The graph reduction technique introduces *horizontal parallelism* whenever it is possible. The execution is demand-driven, i.e. intermediate results are evaluated only when they are needed. The evaluation of terms is postponed until necessary (*lazy evaluation*). The technique is well-known for functional languages and has been proposed for the processing of database queries using the functional query language FQL [BFN82].

A reduction step consists of bringing into existence subordinate query objects to evaluate subqueries. Whenever an input operand is to be evaluated, the query object representing the current term has to wait until the input operand is available, i.e. until the subquery has been evaluated. As outlined in the previous section, event calling is used for synchronization purposes.

To give an example for the execution of a query, consider the query given in example 4.1. The processing starts with the creation of an instance of type `APPLY` which represents the outermost operation of the query expression. Since the argument of the operation itself is a term, the first reduction step involves the creation of an instance of type `JOIN`. One argument of this operation is the state of the object type `DEPT`, so the corresponding state object is triggered to deliver the type's state. The other argument is again a reducable term, so simultaneously the query object creates an instance of type `VIEWJOIN` and waits for the delivery of the result.

The query objects are created in a top-down manner as shown in fig. 6.1.

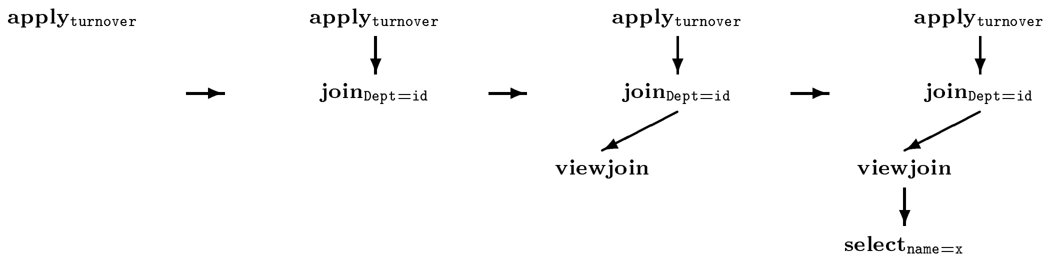

Fig. 6.1: Top-down creation of query objects

The data flow is shown in fig. 6.2. Note that the current states of the types `DEPT` and `PATIENT` can be collected *concurrently* to the creation of the query objects of type `JOIN`, `VIEWJOIN` and `SELECT`.

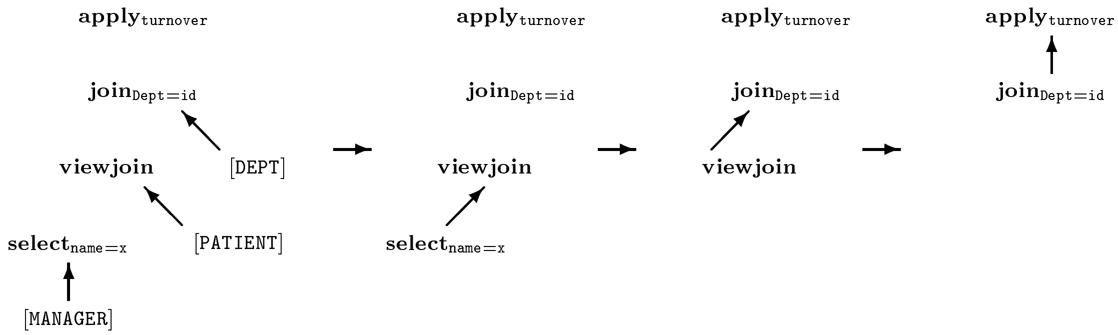

Fig. 6.2: Data flow during query processing

Efficient query processing can mainly be achieved through algebraic optimization of query expressions. Since objects are represented by processes, an efficient implementation of an environment for object societies already has to care of scheduling of objects (processes), memory allocation, secondary storage access methods and access planning. The query system uses the facilities of the existing environment because it is realized using the same framework. Thus, we already achieve quite efficient query processing without the introduction of methods for further optimization like access plans or cost estimations (cf. [JKS85]). A runtime optimization using more sophisticated query object specifications can, however, be introduced without leaving the proposed framework. A possible extension would be the introduction of vertical parallelism (pipelining). By means of interaction elements of intermediate results can immediately be forwarded to the query object performing the outer operation.

8 Conclusion and Further Issues

In this paper, we presented an algebra for querying object societies and we proposed the use of active objects for query processing. Defining object types for the algebra operators, we managed to specify a predefined, universal query system which can be implemented using the environment for object societies already at hand. The operational interpretation of a query by a hierarchy of query objects makes parallel query processing possible.

Future research directions on the topic include the definition of a high-level declarative query language being compiled into the query algebra and the optimization of both the query expression and the processing. We expect to combine algebraic optimization of query expressions with techniques optimizing data access and representation of the global state. Furthermore, we plan to make use of techniques for distributed query processing and to introduce more parallelism during the execution of queries. First ideas on the latter issues include the decomposition of intermediate result relations in order to operate on the parts simultaneously and to minimize the amount of data passed between distributed query objects.

Currently, we are developing a denotational semantics of object queries fitting to the denotational semantics of OBLOG specifications as introduced in [SS89]. Furthermore, we started to develop an environment to implement OBLOG specifications. The approach to query processing proposed in this paper allows easy integration of a query facility and a monitor for object societies into such an environment.

Acknowledgements

We would like to thank our IS-CORE colleagues, in particular Hans-Dieter Ehrich, Udo Lipeck, Robert Meersman and Amilcar Sernadas, for many helpful comments on our approach to query processing. Thanks are also due to the anonymous referees giving useful remarks on how to improve the paper.

References

[ABD+89] Atkinson, M.; Bancilhon, F.; DeWitt, D.; Dittrich, K. R.; Maier, D.; Zdonik, S. B.: The Object-Oriented Database System Manifesto. In: Kim, W.; Nicholas, J.-M.; Nishio, S. (eds.): *Proc. Int. Conf. on Deductive and Object-Oriented Database Systems*, Kyoto, Japan, December 1989. pp. 40–57.

[BFN82] Buneman, P.; Frankel, R.; Nikhil, R.: An Implementation Technique for Database Query Languages. *ACM Transactions on Database Systems*, Vol. 7, 1982, pp. 164–186.

[CDLR89] Cluet, S.; Delobel, C.; Lécluse, C.; Richard, P.: Reloop, an Algebra Based Query Language for an Object-Oriented Database System. In: Kim, W.; Nicholas, J.-M.; Nishio, S. (eds.): *Proc. Int. Conf. on Deductive and Object-Oriented Database Systems*, Kyoto, Japan, December 1989. pp. 294–313.

[Cod79] Codd, E. F.: Extending the Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, Vol. 4, No. 4, 1979, pp. 397–434.

[Dat86] Date, C. J.: *An Introduction to Database Systems*. Addison-Wesley, Reading, MA, 1986.

[Day88] Dayal, U.: Active Database Management Systems. In: *Proc. Conf. on Data and Knowledge Bases*, Jerusalem, 1988. pp. 150–169.

[DD86] Dittrich, K. R.; Dayal, U. (eds.): *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, 1986. IEEE Computer Society Press, Washington, 1986.

[Dit88] Dittrich, K. R. (ed.): *Advances in Object-Oriented Database Systems.* Lecture Notes in Comp. Sc. 334. Springer Verlag, Berlin, 1988.

[ESS88] Ehrich, H.-D.; Sernadas, A.; Sernadas, C.: Abstract Object Types for Databases. In: Dittrich, K. R. (ed.): *Advances in Object-Oriented Database Systems*, Bad Münster am Stein, 1988. LNCS 334, Springer Verlag, Berlin, 1988, pp. 144–149.

[ESS90] Ehrich, H.-D.; Sernadas, A.; Sernadas, C.: From Data Types to Object Types. *Journal on Information Processing and Cybernetics EIK*, Vol. 26, No. 1/2, 1990, pp. 33–48.

[F$^+$87] Fishman, D. H. et al.: Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, January 1987, pp. 48–69.

[Hen88] Hennessy, M.: *Algebraic Theory of Processes.* The MIT Press, Cambridge, MA, 1988.

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[JKS85] Jarke, M.; Koch, J.; Schmidt, J. W.: Introduction to Query Processing. In: Kim, W.; Reiner, D.; Batory, D. (eds.): *Query Processing in Databases.* Springer Verlag, Berlin, 1985, pp. 3–28.

[Mai83] Maier, D.: *The Theory of Relational Databases.* Pitman, London, 1983.

[MS87] Maier, D.; Stein, J.: Development and Implementation of an Object-Oriented DBMS. In: Shriver, B.; Wegner, P. (eds.): *Research Directions in Object-Oriented Programming.* MIT Press, Cambridge, MA, 1987, pp. 354–392.

[SFSE89] Sernadas, A.; Fiadeiro, J.; Sernadas, C.; Ehrich, H.-D.: The Basic Building Blocks of Information Systems. In: Falkenberg, E.; Lindgreen, P. (eds.): *Information System Concepts: An In-Depth Analysis*, Namur (B), 1989. North-Holland, Amsterdam, 1989, pp. 225–246.

[SLPW89] Saake, G.; Linnemann, V.; Pistor, P.; Wegner, L.: Sorting, Grouping, and Duplicate Elimination in the Advanced Information Management Prototype. In: Apers, P. G. M.; Wiederhold, G. (eds.): *Proc. 15th Int. Conf. on Very Large Databases VLDB'89*, Amsterdam, 1989. Morgan Kaufmann, Palo Alto, 1989, pp. 307–316.

[SS86] Schek, H.-J.; Scholl, M. H.: The Relational Model with Relation-Valued Attributes. *Information Systems*, Vol. 11, No. 2, 1986, pp. 137–147.

[SS89] Sernadas, C.; Saake, G.: Formal Semantics of Object-Oriented Languages for Conceptual Modelling. IS-CORE Report, INESC, Lisbon, 1989. *Submitted for publication.*

[SSE87] Sernadas, A.; Sernadas, C.; Ehrich, H.-D.: Object-Oriented Specification of Databases: An Algebraic Approach. In: Hammerslay, P. (ed.): *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, Brighton (GB), 1987. Morgan-Kaufmann, Palo Alto, 1987, pp. 107–116.

[SZ89] Shaw, G. M.; Zdonik, S. B.: An Object-Oriented Algebra. *Data Engineering*, Vol. 12, No. 3, 1989, pp. 29–36.

[TBH82] Treleaven, P. C.; Brownbridge, D. R.; Hopkins, R. P.: Data-Driven and Demand-Driven Computer Architecture. *ACM Computing Surveys*, Vol. 14, No. 1, 1982, pp. 93–143.

[ZM89] Zdonik, S. B.; Maier, D. (eds.): *Readings in Object-Oriented Database Systems.* Morgan-Kaufmann, Palo Alto, CA, 1989.