```
-----------------------------------------------
OBJ R-TREE (int m, int M, char dataset[LENGTH]);

Description
   In R-tree literature m and M are the minimum
   and maximum number of entries per node.
   The tree is suited for DIM dimensions. Assumed
   is that just after the creation of the R-TREE,
   it made persistent by its creator in dataset.

Declare
   R-NODE       root, node;
   float        box[DIM][2], sbox[DIM][2];

/* Assumption: The Add and Delete actions are  */
/* invoked by the (graphical data) objects     */
/* themselves and they send their correct      */
/* minimal bounding box in box.                */

Protocol
   ANY (box)  -> Add      +
   ANY (box)  -> Delete   +
   ANY (sbox) -> Select

Init
   root = NULL;

Actions
   Add = {
      if (root==NULL) {
         new root(m, M, true);
         persistent root in dataset;
         root.EntryAdd(sender, box);
      } else {
         ...
         new node(m, M, true);
         persistent node in dataset;
         ...
      }
   }
   Delete = {
      ...
      /* The deletion of a persistent node */
      /* implies removal from the dataset  */
      ...
   }
   Select = { root.Search(sender, box); }

EndOBJ R-TREE.
-----------------------------------------------
```

```
-----------------------------------------------
OBJ R-NODE (int m, int M, boolean leaf);

Description
   m and M have same meaning as in R-TREE.
   If leaf has value true, then this node is a
   leaf, else this is an internal node.

Declare
   float        box[DIM][2], sbox[DIM][2];
   ANY          id, sid;
   EntryType    entry[M];
   int          NrOfEntries, i;
   R-NODE       next;

Protocol
   (NrOfEntries<M): R-TREE(id,box)->EntryAdd    +
   (NrOfEntries>m): R-TREE(id,box)->EntryDelete +
   R-TREE(sid,sbox)               ->Search      +
   R-NODE(sid,sbox)               ->Search      +
   R-TREE()                       ->Full

Init
   NrOfEntries = 0;

Actions
   EntryAdd = {
      /* Assumption: R-NODE is not full */
      entry[NrOfEntries].box  = box;
      entry[NrOfEntries++].id = id;
   }
   EntryDelete = { ... }
   Full = { sender.(NrOfEntries==M); }
   Search = {
      for (i = 0; i < NrOfEntries; i++)
         if (overlap(sbox, entry[i].box)
            if (leaf)
               /* Return found object          */
               sid.InRegion(entry[i].id);
            else {
               /* Propagate search to lower    */
               /* level. This part of the code */
               /* causes the parallelism       */
               next = entry[i].id;
               next.Search(sid, sbox);
            }
   }

EndOBJ R-NODE.
-----------------------------------------------
```

[36] Peter van Oosterom. A Reactive Data Structure for Geographic Information Systems. In *Auto-Carto 9, Baltimore*, pages 665–674, April 1989.

[37] Peter van Oosterom and Eric Claassen. Orientation insensitive indexing methods for geometric objects. In *4th International Symposium on Spatial Data Handling, Zürich, Switzerland*, July 1990.

[38] Peter van Oosterom and Chris Laffra. Persistent graphical objects. In *Eurographics Workshop on Object Oriented Graphics*, June 1990. Submitted for Acceptance.

[39] Peter van Oosterom and Jan van den Bos. An object-oriented approach to the design of Geographic Information Systems. *Computers & Graphics*, 13(4):409–418, 1989.

[40] Peter van Oosterom, Marcel van Hekken, and Marco Woestenburg. A geographic extension to the relational data model. In *Geo '89 Symposium, The Hague*, October 1989.

[41] J.W. van Roessel. Design of a spatial data structure using the relational normal forms. *International Journal of Geographical Information Systems*, 1(1):33–50, 1987.

[42] André Weinand, Erich Gamma, and Rudolf Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, 1989.

[43] Andreas Wolf. The DASDBS GEO-Kernel, concepts, experiences, and the second step. In *Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, California*, July 1989.

Appendix – The R-tree in Procol

This appendix contains the Procol code of the objects `R-TREE` and `R-NODE`, which together implement a persistent multi-dimensional index structure; see section 6. In case of a GIS with attributes such as points, lines and regions in the plane the dimension of the R-tree is 2. A CAD system with solid objects, for example a polyhedron, will need a 3 dimensional R-tree. Higher dimensional R-trees are also possible, because the in some applications it is beneficial to interpret a combination k scalar attributes as one $k-$dimensional point attribute on with range queries may be formulated.

Not all the code is given here. This is indicated by three dots (...). Especially, some tricky parts of the insert and delete algorithms are omitted, but can be found in [19]. The code is non-trivial because the tree has to be kept in balance under the insert and delete operations. The purpose of this appendix is to show how the R-tree is implemented in an object-oriented manner. We only showed one query type: the box select (in 2D that is, a rectangle select), which returns the id of every object in the R-tree that overlaps the search box `sbox`. The result is sent back to the original object `sid` by invoking the action `InRegion` with the id of the found (graphical data) object. This happens once for each object that is found.

Note that we implemented a parallel or distributed version of the tree by using the object `R-NODE`. During the search operation several nodes can work in parallel on different processors. There is quite a bit of work in each node, because a typical value for `M` (maximum number of entries) is 100. This solution would probably not be very efficient for binary tree, because the overhead introduced by sending messages to other processors may require more time than the time that is gained by the parallel execution. Besides the search operation, the delete and insert operations might also benefit from parallel execution in case of node overflow and node underflow respectively. This is not shown in the code below.

```
#define DIM 2        /* or any other value > 1 */
#define LENGTH 256

typedef struct{
   ANY       id; /* R-NODE or graphical object */
   float     box[DIM][2];
} EntryType;
```

[12] Max Egenhofer and Andrew Frank. Panda: An extensible DBMS supporting object-oriented software techniques. In *Database Systems in Office, Engineering, and Scientific Environment, New York*, March 1989. Springer-Verlag.

[13] Max J. Egenhofer and Andrew U. Frank. Object-oriented modeling in GIS: Inheritance and Propagation. In *Auto-Carto 9, Baltimore*, pages 588–598, April 1989.

[14] Christos Faloutsos, Timos Sellis, and Nick Roussopoulos. Analysis of object oriented spatial access methods. *ACM SIGMOD*, 16(3):426–439, December 1987.

[15] Andrew U. Frank. Multiple inheritance and genericity for the integration of a database management system in an Object-Oriented approach. In *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems, Bad Münster am Stein-Ebernburg, FRG*, pages 268–273, September 1988.

[16] Andrew U. Frank and Renato Barrera. The fieldtree: A data structure for geographic information system. In *Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, California*, July 1989.

[17] Diane Greene. An implementation and performance analysis of spatial data access methods. In *IEEE Data Engineering Conference*, pages 606–615, 1989.

[18] Oliver Günther. *Efficient Structures for Geometric Data Management.* Number 337 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1988.

[19] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD*, 13:47–57, 1984.

[20] Independent European Programme Group – Technical Area 13 (IEPG TA-13). PCTE$^+$ C Functional Specification Issue 2, July 1988.

[21] Independent European Programme Group – Tecnical Area 13 (IEPG TA-13). Introducing PCTE$^+$, 1989.

[22] Setrag N. Khoshafian and George P. Copeland. Object identity. In *OOPSLA'86*, pages 406–416, September 1986.

[23] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, London, 1988.

[24] R. Morrison, A.L. Florianis, A. Dearle, and M.P. Atkinson. An integrated graphics programming environment. *Computer Graphics Forum*, 5:147–157, 1986.

[25] Norman W. Paton and Peter M.D. Gray. Identification of database objects by key. In *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems, Bad Münster am Stein-Ebernburg, FRG*, pages 280–285, September 1988.

[26] D. Jason Penney and Jacob Stein. Class modification in the GemStone Object-Oriented DBMS. In *OOPSLA'87*, pages 111–117, September 1987.

[27] Joel E. Richardson and Michael J. Carey. Persistence in the E language: Issues and implementation. *Software – Practice and Experience*, 19(12):1115–1150, December 1989.

[28] Hanan Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, 16(2):187–260, June 1984.

[29] Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. *ACM SIGMOD*, 15(2):340–355, 1986.

[30] Andrew Straw, Fred Mellender, and Steve Riegel. Object management in a persistent smalltalk system. *Software – Practice and Experience*, 19(8):719–737, August 1989.

[31] Sun Microsystems, Inc. SunINGRES Manual Set, January 1987.

[32] D.C. Tsichritzis and O.M. Nierstrasz. Fitting round objects into square databases. In *ECOOP '88*, pages 283–299, August 1988.

[33] Jan van den Bos. PROCOL: A protocol-constrained concurrent object-oriented language. *Information Processing Letters*, 32:221–227, September 1989.

[34] Jan van den Bos and Chris Laffra. PROCOL - A parallel object language with protocols. In *OOPSLA '89, New Orleans*, pages 95–102, October 1989.

[35] Peter van Oosterom. Spatial data structures in Geographic Information Systems. In *NCGA's Mapping and Geographic Information Systems, Orlando, Florida*, pages 104–118, September 1988.

not required in Procol. However, the use of illegal references is signalled at run-time. This is done by inspecting the surrogate and taking the appropriate measures.

If a persistent object contains a reference to a volatile object, this volatile object is not saved automatically. The proper way to program this case is to make the other object also persistent, if the referenced object is still needed in the future. The state of each object instance is stored as unity, that is in contiguous memory. Instances of different sizes are no problem. The state also contains some additional data, for example the creator. The application programmer decides whether instances of the same object type are "stored together" without instances of other types in one dataset or if a dataset contains instances of different types. The later is advantageous for representing complex objects in an efficient manner, because the instances of the different types that define a complex object are stored close together. Note that complex objects are important in CAD systems, because this is one of the main modeling tools.

10 Further Research

Besides an object-oriented programming language, Procol is also a parallel programming language. It is possible that the objects run in parallel on multiple processors. We have only used this in a few examples. Clearly, this topic deserves more attention and more research is needed in the context of highly interactive and graphical systems.

It is a small step from one single user Procol program with persistent objects to a system with multiple users. At least conceptually, because each object has its own protocol which regulates the communication. It should not matter from which program a message originates. However, we will have to reconsider some of the concepts.

The provided query facilities are very limited; with *retrieve* it is only possible to get hold of one "starting" object id of a specified type or to perform the selection of instances from one set (object type) at a time. More complex queries have to be programmed into the objects (the Procol program). Attention has to be paid to avoid object types becoming to specific. That is in contradiction with one of the basic principles of databases of data being independent of applications. More research in this area is necessary.

References

[1] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[2] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[3] F. Bancilhorn, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lécluse, P. Pfeffer, P. Richard, and F. Velez. The design and implementation of O_2, an Object-Oriented database system. In *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems, Bad Münster am Stein-Ebernburg, FRG*, pages 1–22, September 1988.

[4] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1973.

[5] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[6] E.H. Blake and S. Cook. On including part hierarchies in object-oriented languages, with an implementation in smalltalk. In *ECOOP '87*, pages 41–50, 1987.

[7] Peter Pin-Shan Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[8] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[9] Douglas Comer. The ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[10] Brad J. Cox. *Object-Oriented Programming – An Evolutionary Approach.* Addison-Wesley, Reading, Mass., 1986.

[11] Walter C. Dietrich, Jr., Lee R. Nackman, Christine J. Sundaresan, and Franklin Gracer. TGMS: An object-oriented system for programming geometry. *Software – Practice and Experience*, 19(10):979–1013, October 1989.

With this statement we can retrieve an instance of the same type of <object-id> from the dataset with identifier <dataset-key>, that satisfies the <discrimination-string>. If the dataset in question contains more than one instance of the required type, only one will be returned in <object-id>. Any successive execution of the **next** statement, will retrieve the *next* instance of the required type until all required instances have been retrieved from the dataset.

In general, <dataset-key> will be a string. This string can then be used to compose the filenames of the necessary dataset files. An example use of the provided persistence in the Declare section of an object type using the object `DRAWING`:

```
Declare

    object DRAWING drawing;

    allocate_drawing(char *name)
    {
        new drawing;
        persistent drawing in name;
    }
    read_old_drawing(char *name)
    {
        retrieve drawing from name;
    }
```

9.2 Internal Implementation

We will now motivate our design decisions for the internal implementation. Procol has been developed and implemented on a network of Sun workstations running under SunOS Release 4. Five possible implementations of the extension of Procol with persistent objects were considered, (compare with [32]): *bare* implementation (using the Unix OS-interface calls: open, close, read and write), *shared mapped memory* (virtual files), *Ingres* [31] (or other relational DBMS), *Postgres* [29] (or other extendable DBMS), *PCTE*+ [20, 21] (offers a powerful OMS derived from the Entity-Relationship Model of Chen [7]).

A file is mapped directly by the Unix OS on the address space of a process. A major advantage is that there is no difference between the "stored" object instances and their "running" counterpart, at least not at the level of the Procol kernel. At OS level there is a difference and this is the same as the difference between virtual memory pages that are in main-memory and the ones that are swapped on disk. We expect this implementation to be very efficient. In order to gain some experience we are currently converting a local application that uses explicit read and write statements, into a mapped memory implementation. First test results indicate that the elapse times decrease with about 30% in applications with a lot of read and write statements. A disadvantage of the mapped memory approach is that we still have to do the memory management ourselves. In [38] it is explained in more detail, why we decided to use the mapped memory approach for the prototype implementation of persistent objects in Procol.

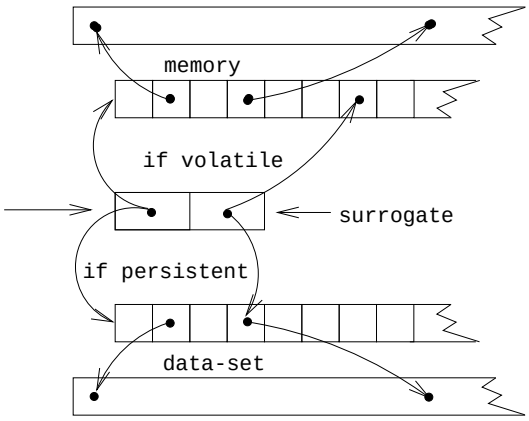

Figure 3: Object Reference with Surrogates

An object instance is identified by a *surrogate* [22]. That is, the object id is not the actual address in memory but we need an indirection [30] to locate the object. See figure 3 for a graphical demonstration of the process. Each surrogate contains an indication whether the object instance is volatile, persistent or deleted. When, during the execution of a Procol program, an object is referenced, we have to check the first part of the surrogate. If the the object instance is volatile, the surrogate contains a key than can be used to retrieve the memory address of the instance variables. If the object instance is persistent, the surrogate contains a dataset identifier, and a key. With this dataset identifier and the key, the OMS is able to retrieve the actual memory address of the the instance variables of the persistent object in question.

The disadvantage of surrogates is that there is an extra indirection. However, surrogates have several important advantages. They enable objects to be switched between volatile and persistent state, by modifying a part of the surrogate. We decided, based on the advantages and disadvantages mentioned in section 5, that the control of referential integrity is

and it may take quite a while to generate the complete answer.

- Another strategy is first to state the query and then retrieve the answer one by one (or perhaps in buffers of a fixed size). The first part of the answer will probably be ready sooner than the complete answer would be. This promotes parallelism and is also quite important in an interactive application, because the end-user can already see something on his screen then.

The problem with the second solution for returning a search result is that other objects might interfere with the set of objects that belongs to the queried object type. "Third-party" objects could change values and add new instances or delete existing ones. We still have to investigate whether this can be solved by applying the right protocols in the system (OMS) objects. This has to be solved before we decide on the syntax of a search query.

8 Object Instances of Different Sizes

In section 3 we saw that the wish to store a polyline or polygon with a minimum of overhead, implies that different instances of the same object type may have different sizes. So for example, the pure relational solution, presented by van Roessel [41] is not acceptable, because a polyline is scattered over several tuples in a table and first has to be aggregated before it can be used again. In [40] a solution in the context of the relational data model is presented.

Different sizes have an (enormous) impact on the implementation of persistent objects. In CO_2, the C implementation of O_2 [3], it was decided to prohibit object instances of different sizes. In contrast we would even like to have persistent objects whose sizes change dynamically. For example, to make it possible to remove points from or add points to a polyline. However, this would even further complicate the implementation. A decrease of the size of an object is not too hard, but an increase of object size means that an object does not fit in his (contiguous) part of memory and the memory after this object is probably occupied by another instance. The object will have to be moved to another larger place, because we want to treat an object as a unity and do not want to split it. This would have been impossible if the objects id is its address. However, this was already disapproved of because of reasons discussed earlier. In any case, growing persistent objects could introduce a lot of overhead; e.g. moving of objects.

A more feasible situation is that after the Init section, the size of an object may not vary any more. It is still possible to deal with dynamic problems. For example, use a pointer (id) to an object of type linked list. This object type has an "application" data-part and a pointer to the next list element. Each instance represents one list element and they all have the same fixed size. We can extend this approach and simplify our implementation of Procol, if we only allow the following data types as attributes: *Basic types* (int, char, float, ...), *References* (or links) to other objects, and *Arrays* with fixed size after the Init.

9 Persistent Objects in Procol

The question how to implement persistent objects in Procol can be divided into two sub-questions. The first is how to adapt the languages features (the external implementation). The second is how to implement this on the underlying platform (the internal implementation).

9.1 External Implementation

Our decisions according the external implementation of persistence in Procol included the introduction of the following new keywords:

1. **persistent** <object-id>
   in <dataset-key>
   With this statement a volatile object instance identified by <object-id> can be made persistent by coupling it to a dataset identified by <dataset-key>.

2. **volatile** <object-id>
   With this statement an object instance (identified by <object-id>), that has been made persistent before, can be made volatile. The result of this statement is that the persistent object instance is removed from its dataset.

3. **retrieve** <object-id>
   from <dataset-key>
   where <discrimination-string>
   and
   next <object-id>

a separate search tree (index) for each attribute for which efficient searches are required. This has to be made clear to the OMS before the queries are posed.

It is possible that the value of an attribute changes, after the search tree has been created for that attribute. In that case, the tree may have become incorrect or inconsistent. This can be solved by sending an (implicit) message to the search tree object(s) in the OMS, just after the attribute has changed. Upon receipt of this message the search tree adjusts itself.

7 The Procol Extension

This section presents some issues concerning the syntax and semantics of the constructs which might be added to Procol for the support of persistent objects. This is done here without worrying how this can be achieved in our implementation of Procol. From the users point of view, this extension should be as small and simple as possible. First, we have to decide how to indicate that an object is persistent. Some alternative possibilities are:

- *On the fly:* Make a volatile object instance persistent by applying a new Procol primitive *persistent*. Assume the variable x holds the id of an object instance; then this instance is made persistent by: *persistent x*. Note that there is a difference with the *save* operation of section 4, because the values (states) of a persistent object are always guaranteed to be up-to-date.

- *Per Object Instance:* At the moment an object is created, it is decided whether it will be persistent or not. A convention can be made that objects created with the *new* primitive are volatile and the ones created with the *persistent* primitive are persistent.

- *Per Object Type:* At the moment the object type is defined it is specified whether all instances of this type are persistent or not. A modified keyword *OBJ* could indicate this: *PERSISTENT_OBJ*.

A combination of these approaches is also possible. In the language E [27] the programmer has to indicate per type (class) that instances are optionally persistent. The programmer has to decide per instance if it is really a persistent object instance.

The advantage of persistence per object type is that only once, during the object type definition, there is a difference for the application programmer between persistent and volatile objects. In the other solutions it is required to indicate that the object is persistent for each object instance. The major drawback of the latter choice is that two different types have to be defined if we want to use both the volatile and the persistent variants of basically one object type. In the case of strong type checking this means that we can not freely interchange the use of volatile and persistent objects as arguments in messages and procedure calls.

In order to get hold of persistent objects with unknown id, Procol will be extended with the *retrieve* primitive. Perhaps it is better to take the following approach towards the primitives *new, delete,* and *retrieve*: consider them as messages to the object types themselves ("class methods"). These are system objects (partly) responsible for the OMS tasks. These system objects have to maintain index structures if requested by sending them an *create_index* message.

The *retrieve* primitive has some resemblance to the *new* primitive, because it also assigns the id of an object to a variable of the proper type. Unlike *new*, *retrieve* will not execute the Init section, because that already happened when this object was created for the first time. The protocol (expression) of the object regulating access to the object is matched starting at the current (saved) state.

A *discrimination condition* can be used, because the object type information may not be specific enough. Of course only visible attributes can be specified in the condition. A retrieve returns the id of the object of the proper type for which the discrimination condition evaluates True. If there is more than one object satisfying these criteria, only one is returned. If there is no object satisfying these criteria, a *NULL* object is returned.

It is a small step from the *retrieve* primitive to the associative search operation. In fact, it could be considered as an iteration over the retrieve operation. If fast replies are required, then in case of large set of objects, an index has to be used. This index could be a spatial index structure; e.g., needed for efficiently solving a "rectangle" query. There are several options for returning the answer of a search:

- Return one big set that contains the id's of all objects that satisfy the query. In case of large answers, a lot of temporary memory is required

6.2 Searching

The objects as presented so far are not suited for associative search operations. That is, searching based on the contents of an object instead of using the object id to find an object. This is especially useful for a program that want to use objects created by other programs, because the id's are unknown and have no semantic meaning. All that a program(mer) knows is about: object types (the kind of data he wants to use) and attribute values (restriction of instances).

Another use of associative searching is to solve the query: "How many inhabitants has the municipality with the name attribute 'Leiden'?". We have to look at all the instances of the municipality object type until we have found the proper one. This is an $O(n)$-algorithm. However, this problem can be solved with an $O(log(n))$-algorithm, if a binary search is used. In a relational database, efficient search is implemented by a B-tree [4, 9] for attributes on which an *index* is put. The B-tree has many useful properties, such as: it stays balanced under updates, it is adapted to paging (multiway branching instead of binary) and has a high occupancy rate.

The B-tree solution in an object-oriented environment is established by a set of auxiliary (system) objects. These objects do not contain the application data, but contain tree structures with references to the objects with the actual data. This B-tree has to be part of the OMS and, if possible, transparent to the "application" objects. Note that the OMS itself can be implemented in Procol as a set of objects.

There is some friction between the concepts behind the ADTs and the idea of associative searching, because associative searching requires knowledge of the internals of other objects. An object has to specify his query in terms of data-part that are inside other objects. To limit the damage, only so called *visible* attributes may be used in the query. These visible attributes become part of the specification of an object type (together with the actions of course), in contrast to the non-visible data-part which belong to the implementation. Note that an index may be put only on a visible attribute of an object type.

6.3 Multi-dimensional Data

The searching problem also applies to the graphic or geometric data. If no spatial structure is used, then queries such as "Give all municipalities within rectangle X" are hard to answer. A spatial data structure which is especially suited for the object-oriented environment is the R-tree [19]. This is because the R-tree already deals with objects; it only adds a minimal bounding rectangle (MBR) and then it tries to group the MBRs which lie close to each other; see Figure 2. This grouping process is reflected in a tree structure, which in turn may be used for searching. Several test results [14, 17] indicate that the R-tree is a very efficient spatial data structure.

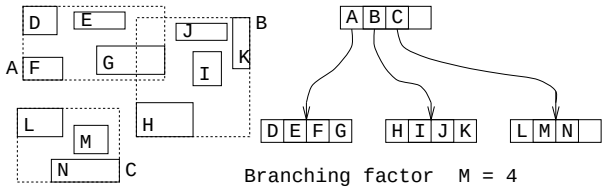

Figure 2: The R-tree

Not all known spatial data structures [35] are suited for this purpose. For example kd-trees [5], quadtrees [28], R+-tree, bsp-trees [36], cell-tree [18] and grid-files, are more difficult to integrate in the object-oriented environment because they cut the geographic objects into pieces. This is against the spirit of the object-oriented approach, which tries to make complete "units," with meaning to the user. The Field-tree [16], KD2B-tree and the Sphere-tree [37] are good candidates for integration in an object-oriented system, because they do not split the objects. Each of the spatial search structures has its own strengths and weaknesses, so if several alternatives are offered by the system, an application can use the structure that fulfills its needs the best.

In Procol, trees can be implemented in two different ways. The first method stores the entire tree in one single search object. The second method stores each node of the tree in a separate instance of the search object. The latter introduces overhead by creating a lot of search objects (nodes). However, it has the advantage of being suited for parallel processing in Procol because the search objects can run on parallel processors. This is useful for range queries: "Give all municipalities with more than 10.000 and less than 20.000 inhabitants." Appendix A contains a part of the Procol implementation of the R-tree. In this implementation each node is a separate object. In case of the R-tree this is a reasonable choice, because there is a fair amount of work in each node. The same would be valid for B-trees, but not for binary trees. In any case, for practical reasons, there has to be

count has a value greater than zero, the object will not be deleted and the creator is notified of this fact. The reference count mechanism introduces overhead, because the counters have to be updated in each assignment to an object variable. Problems are introduced by cyclic data structures.

- A slightly different approach, but also based on a reference count, is followed in O_2 [3]. The object deletion is not refused but postponed until the value of the reference count is zero. The creator does not have to worry about trying to delete the object another time.
- Dangling references can not occur if we prohibit the deletion of objects. This approach is taken in for example GemStone [26]. In order to avoid congestion of the system, *garbage collection* has to be performed. Two well known methods for this are:
  1. Using a reference count: when the count becomes zero, the associated object instance is automatically deleted by the system.
  2. Performing a sweep through the object space (a directed graph) in order to detect which objects are unreachable. A disadvantage is that the sweep is performed periodically and during this operation the system can not be used by the applications. This can be avoided by using an incremental version of the sweep algorithm.
- The maintenance of the reference count introduces overhead; both memory usage and execution time increase. Clearly, it is more efficient to omit a reference count and directly delete the object at request. However, in this case the system is not allowed to reuse the *id's* of deleted objects for new objects. So, if a message is being sent to a deleted object, the system can detect this, and the sender will be notified. This strategy implies that the address of an object can not be used as its id, because when the object is deleted we want to be able to reuse that part of the memory space for new objects.

It may be clear by now that we are biased towards the latter approach. In the context of Procol, dangling references are probably programming errors and the detection of the illegal use of dangling references during run-time is an adequate solution. Finally, it is interesting to note that PCTE+ [20, 21] offers links both with and without referential integrity. This is probably done for efficiency reasons. It is not stated in the PCTE+ documents how the referential integrity is maintained.

6 Object Management

In order to solve the administrative problems associated with the use of object id's, there is a need of an *Object Management System* (OMS) that takes care of the (persistent) objects. One of the responsibilities of the OMS to keep the references in the object system consistent. To be more precise, an object system is consistent if [22]:

- No two distinct objects have the same identifier (unique identifier assumption). In other words, the identifier functionally determines the type and the value of the object.
- For each identifier present in the system there is an object with this identifier (no dangling identifier assumption).

6.1 Object Identity

A uniform object identification mechanism has to be developed, capable of dealing with objects shared by multiple programs, multiple users or even multiple computers (in a network). There should be a mechanism to indicate in which persistent objects one is interested, so one is not bothered by non-interesting objects of others. One possible method could be to organize the object instances in "datasets" which are put in the normal hierarchical file system. This limits the scope and makes the task of finding the right communication partner easier for the OMS.

In relational databases [8] an identifier key is formed by one or more user-supplied attributes. Value based matching is a transparent technique for expressing relationships. However, it provides no support for referential integrity at all. By contrast, OOPLs support the notion of object identity which is independent of the attribute values [25]. Khoshafian and Copeland [22] describe several techniques for implementing object identity and they conclude that using so called *surrogates* is the best technique. Surrogates are system-generated, globally unique identifiers, completely independent of the physical location and data contents of an object.

4.1 Introducing Persistence

Normally, object instances are only present when the program is executing. Data will have to be loaded from a file or a database system into the (new) objects when the program is initialized. Just before the program stops, the data have to be saved again. This is, as argued in section 2, an inconvenient method, especially in the case of applications with huge amounts of data that are not entirely needed in each session. An object-oriented step in the right direction is to store the state of objects themselves. This can be compared with making a *core dump* of a single object. When an object is saved, a "snapshot" of the object instance is made. Changes made after the save operation are not propagated to this snapshot.

The suggestion to store the objects themselves is not as simple as one might expect. This is because objects usually contain references (in attributes or local variables) to other objects. A reference to an object is an *id* (identification of the proper type), assigned to that object by the operating system when it was created with the Procol primitive *new*. In some situations it is useless to save an object without also saving the related objects.

The "snapshot" method is used in several other OOPLs. In systems offering multiple inheritance the object type that also has to be persistent, inherits this property from a general object type with methods to save and load the object. Egenhofer and Frank [13, 15] suggest the object type *db_persistent* with methods *store, delete, retrieve* and *modify*. ET++ [42] has an object hierarchy with the object type *object* in the top of this hierarchy. The object type *object* has methods called *PrintOn* and *ReadFrom* which enable transfer to and from disk. These solutions work fine as long as the object types contain no references to other objects but only simple attributes, such as for example an array of coordinates describing a polygon.

The persistent data in PS-algol [2, 24] are organized into one or more databases. Each database has its own root and may contain values of different (complex) data types. The data are "imported" into a program with the *open.database* procedure which returns a pointer to the root. The root has the form of a name-value table in which the value is usually a pointer to another data structure. The actual data are accessed by following these pointers and it is assumed that the programmer has to know the structure of the database (though this is nowhere stated in the PS-algol papers). Once imported, the data can be manipulated in the same manner as volatile data. The procedure *commit* propagates the changes made so far to the database, if it was open for writing. Everything that is accessible from the root is stored. This means that values may change and data (structures) be added or removed.

In the OOPL Eiffel [23] an object type that inherits from the object type *STORABLE* gets this kind of behavior by means of the methods *store* and *retrieve*. If the method *store* is invoked in object instance x, the whole object structure starting at x is dumped (in a special format) to a file, even if the referenced object types in x do not inherit from *STORABLE*. Depending on x and the object structure of the application, it is possible to store the whole object structure, or just a part of it. Basically, this solution has two drawbacks. First, the application programmer has to indicate when to save or load the objects explicitly. So, if the program is stopped before the save, the latest data are lost. Second, updating one object in an object structure can become very expensive if all related objects have to be saved also, even if they did not change.

5 Referential Integrity

What happens if an object is deleted by its creator while other objects are still referring to this deleted object? A dangling reference is not a problem specific to persistent objects, it is a problem in the case of volatile objects too, but it manifests itself in a severe manner in combination with persistent objects. Assume, a persistent object contains a reference to a volatile object and the program is stopped. The next time the program is started, the reference to the volatile object is not valid any more (though it has not been deleted). By the way, dangling references can also occur in non-OOPL. For example, in C it is possible to have pointers to deleted data structures, which may be the cause of some severe errors in a program. Some systems guarantee *referential integrity*. An associated problem is that of an "unreachable" object, that is an object to which the last reference is lost. There are a number of possible approaches towards these problems:

- If we want to guarantee referential integrity, we at least have to be able to detect whether the integrity is damaged by the deletion of an object. This can be achieved by associating a *reference count* with each object instance. If the reference

elaborated in one of the later sections, in which case the section referred to will be mentioned here. Note that these requirements may neither be orthogonal nor independent of each other.

r1 Upward compatible. Persistent objects have to be introduced with a minimal change to Procol. This implies that existing Procol programs do not have to be changed in order to be compiled by the new version of the Procol compiler.

r2 Transparent persistent objects. Persistent objects are treated in the same manner as volatile objects by the application. Perhaps except for incompatible database facilities; for example, associative searching, that is object searching based on the contents or value of instances. Atkinson et al.[2] refine this by recognizing the following principles for persistent data (they assume several levels of persistence):

1. Persistence independence: the persistence of an object is independent of how the program manipulates that object. So, it has to be possible to call a procedure of which the actual parameters are sometimes persistent objects and other times volatile objects.
2. Persistence data type orthogonality: all objects are allowed the full range of persistence. This means that no matter how complicated the type is, its instances can still become persistent.

r3 Complex objects. This provides the system with sufficient modeling power; e.g. *part-of hierarchies.* In Procol complex object types are defined by means of links (or references) to other object types that together define the complex object. The complex objects are static in terms of the object type structure and dynamic in the sense of the object instances.

r4 Extendability with new ADTs. This wish might be a trivial one in the context of OOPLs or object-oriented databases, but certainly not in the context of the traditional DBMSs. The definitions of the new persistent ADTs also have to be stored somewhere, if we want to be able to manipulate its object instances in a sensible manner.

r5 Efficient handling of large amounts of objects. Long-lived systems allow time for data to accumulate. This, combined with the fact that we aim at developing interactive systems, justifies this efficiency requirement to be even more important than in other systems. Not only efficient retrieval by object id (which is very important in OOPL and object- oriented databases, as used in navigation links) is required, but also efficient associative searching has to be possible. This is realized, as usual, by indexing techniques such as B-trees [4, 9] or hashing.

r6 Object instances of different sizes. A polyline or polygon has to be stored with a minimum of overhead, because of the required time (and space) efficiency in interactive systems. This implies that different instances of the same object type may have different sizes. To treat an object instance as a *unity* means that it is stored in a contiguous part of memory. This may seem to be an implementation issue, but it is very important and by putting it in our wish list we emphasize this. This topic is further discussed in section 8.

r7 Highly interactive and graphical applications. The previous two wishes actually are part of this more general wish to make Procol suitable for this kind of applications. It has to be kept in mind that multi-dimensional data sometimes require other approaches than the data types encountered in traditional DBMSs. Also, the fact that Procol is designed as a parallel programming language should be exploited.

r8 Exchangeable objects. It should be possible to exchange object instances between different systems. Object instances created by one system must be directly applicable by other systems.

r9 Deal with referential integrity in a satisfactory manner. This is well-known problem in database and programming language research. The topic will be discussed in depth in section 5.

4 The First Attempt

In this section we describe our first (and quite simple) attempt to provide a mechanism for persistent objects, without trying to satisfy all the wishes of our list. The problems we encountered give some insight in the complexity of introducing persistent objects. Then we briefly compare our first attempt with the approaches taken in some other systems.

2 The Need Of Persistence

The computer science research in the area of programming languages emphasizes programming constructs and data structures. One of the most popular paradigms is that of Abstract Data Types (ADTs). Object-Oriented Programming Languages (OOPLs) encapsulate this paradigm in an elegant manner using object types to describe the ADTs. Access from outside to the data inside an object instance[1] is only possible through the methods or procedures defined for that object type.

The data stored in data structures (or objects) of a running program are in general volatile, that is, as soon as the program stops, the data are lost. However, in many applications the data itself are very important. An obvious solution is to save the data in a file by explicit write statements. The next time the program is started it first reads the data from file into the volatile data structures. Persistent objects make the program development more efficient, because the programmer does not have to worry about the reading and writing of data from and to disk. Also, the structure of the data file may become quite complex, resulting in possibly intricate parsing. Moreover, for large quantities of data this "file" solution becomes cumbersome during the execution of the program. Consider as an example an information system that registers bank accounts. A characteristic of this and many other information systems is that the objects are well structured, quite passive and occur in large quantities. Passive objects are objects that hardly ever send messages to other objects (except for replies); they only react to messages from outside. In the bank account example, an account object replies its current amount when asked for, and updates it when told so by a message from an authorized object.

Database Management Systems (DBMS) have been developed to deal with the large amounts of data mentioned above. DBMSs concentrate on the information representation and tackle related problems such as, integrity, security, redundancy, consistency, efficient searching, query formulation and concurrency control. A major drawback is that the DBMSs of the current generation are not extendable with new data types and operators. This makes the use of these DBMSs inconvenient in non-standard applications that need support for other data types.

[1] In this document we will use the term *object type* to indicate a class of objects and the term *object* to indicate one instance. In case more emphasis is needed we use the term *object instance* explicitly.

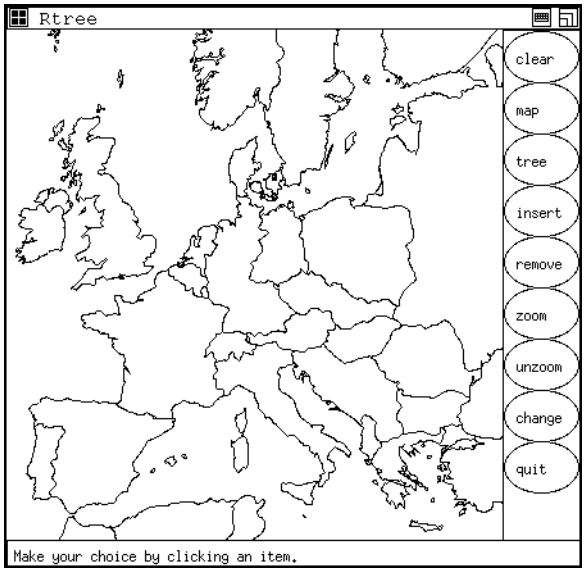

Figure 1: A Geographic Information System

The database research community has recognized this deficiency and is now trying to design systems that are more open [12, 29, 43]. At the other side the OOPL research community has recognized the need for persistent objects. Now, these two worlds meet. In some cases these encounters result in conflicts because of very different and incompatible principles. For example, explicit (navigation) links amongst instances are considered harmful by the database community, because they are hard to maintain. However, these same links form the backbone in representing complex objects in OOPLs. On the other hand, sometimes the combination of the two research communities result in a nice symbiosis. An example of this is that the concurrency control problem in the database is solved by the model of an object that accepts messages one by one (as in Procol).

We are interested in interactive graphical applications, such as: CAD systems, VLSI Design, and GISs, see Figure 1. Common aspects in these systems are: interactivity, graphics, and large amounts of data. In [39] we showed that the object-oriented approach offers a good data modeling and design environment for GIS applications. In the same paper the need for persistent objects in our object-oriented programming language Procol, was identified.

3 Our Wish List

This section shortly discusses the major requirements for persistent objects in Procol. Some will be

Persistent Graphical Objects in Procol

Peter van Oosterom and Chris Laffra

Department of Computer Science, University of Leiden
P.O. Box 9512, 2500 RA Leiden, The Netherlands
Email: {oosterom,laffra}@hlerul5i.bitnet

Persistent objects are objects whose contents may outlive the execution time of the program. This paper describes the process of introducing persistent objects in the object-oriented programming language Procol. The strength of persistent objects in an object-oriented programming language is the integration of a database system with a programming language. Persistent objects make the program development easier, because the programmer does not have to implement the explicit loading and saving of data. Besides the general functional aspects, special attention is paid to graphical applications in order to deal with their specific geometric requirements. For example, it must be possible to find, in an efficient manner, all graphical objects that fall within a given region. These issues, persistent objects and their geometric requirements, have not yet got the attention they deserve in the literature covering object-oriented graphical systems where modeling and functional aspects dominate.

1 Introduction

The fact that the object-oriented approach is so successful in computer graphics, is mainly due to the system modeling capabilities that the object-oriented paradigm offers. The specialization relationships which exist between the graphical objects in a system, can be modeled with inheritance or delegation which are present in many object-oriented development environments. Geometric data types, such as points, vectors and matrices may be implemented by abstract data types using object classes [6, 10, 11, 23]. However, in practice this modeling power is not enough when implementing CAD systems or Geographical Information Systems (GISs) which deal with large data sets. Integrated database facilities are required to support the graphical application in an efficient manner.

In this paper we describe a solution that is based on the introduction of *persistent* objects in the object-oriented programming language called *Procol*. A detailed functional description of the C–based language Procol can be found in [33] and some implementation aspects in [34]. The resulting system is also extendable with new (persistent) types and operators. In itself this approach is not new and has been described by several other authors [1, 27, 30], but we also tried to make the system suitable for highly *interactive and graphical* applications. This goal is achieved by putting emphasis on both time-efficiency (offering techniques such as: navigation, index structures, and parallelism) and the recognition of multi-dimensional data. As an example, not only one dimensional, but also multi-dimensional index structures are provided in Procol. The emphasis in this document is on the inclusion of persistent graphical objects in the syntax and semantics of Procol and on its implications for the technical realization; e.g., how Procol deals with problems such as referential integrity and associative searching.

We will state the problem area in section 2 and define our general requirements for persistence in an object-oriented language in section 3. Our first attempt to introduce persistence in Procol and comparisons with several other systems are described in section 4. Building on this experience the following topics are discussed in more detail: referential integrity, (multi-dimensional) search problems, Procol extension alternatives and object instances of different sizes in sections 5 through 8, respectively. In section 9 we present the solution as chosen in Procol. In nearly all the sections the requirements of graphics play an important role. The discussions are illustrated with examples based on graphical systems. Finally, section 10 indicates some topics where further research is required.