Knowledge-Based Computer Vision

Integrated Programming Language and Data Management System Design

Andrew M. Goodman, Robert M. Haralick, and Linda G. Shapiro
University of Washington

Vision systems use a variety of complex data structures to represent their knowledge of the images being processed and the application domain. A new programming language and data management system approach is suggested.

The languages that come closest to providing the tools needed for a comprehensive computer vision system fall under the category of "object-oriented database languages." As in the cases of CAD/CAM environments, software engineering environments, and engineering database environments, our data is too complex to model with traditional relational database management systems. Such systems, as well as systems providing data modeling using semantic or functional models, use languages specifically designed for interaction with highly structured databases. In addition, they don't often provide the semantics of a full, imperative language such as Common Lisp or C.

For this reason, we do not mention all the contrasts between our language/data management system and those of the relational, semantic, and functional sort. Most of the comparisons we make are between our language and the others with object orientation and persistence. Ideas for the data model have come from earlier work on data models for geographic information systems.[1]

Through a survey of people who have vision applications, we have found that the following properties are needed in a language for programming computer vision tasks:

(1) Persistent data objects (types, operations, and instances of types) should appear to the programmer no differently than do transient objects. The only difference between a persistent and a transient object of the same type is that the persistent one continues to exist after the current user session terminates. If the applications programmer's task is to implement abstract data types for vision applications, he or she should not have to write file utilities for each new type that might have persistent instances. In defining types and their behaviors, the programmer should not have to define a persistent version of the type and a nonpersistent version. Someone writing an operation should not have to worry about whether an argument passed to the operation is persistent. There should be a minimal performance penalty for all this.

(2) The programmer must be able to build and specify behavior for highly connected networks of objects, yet at the same time have type constructors such as sets, sequences, arrays, and relations that impose order on these possibly arbitrary graphs of objects. If someone needs to implement a threaded tree or a heap, a unified, full, and imperative syntax should be available. We do not want two languages (a database manipulation language and a procedural language), as is the approach with relational DBMSs.

(3) With such interconnected data sets, it becomes too much of a task for the programmer to keep track of the references to objects and free their space when no longer referenced. Garbage collection on both transient and persistent objects is necessary. Some systems, such as E[2] and Extra/Excess,[3] require explicit deletion of

0018-9162/89/1200-0043$01.00 © 1989 IEEE

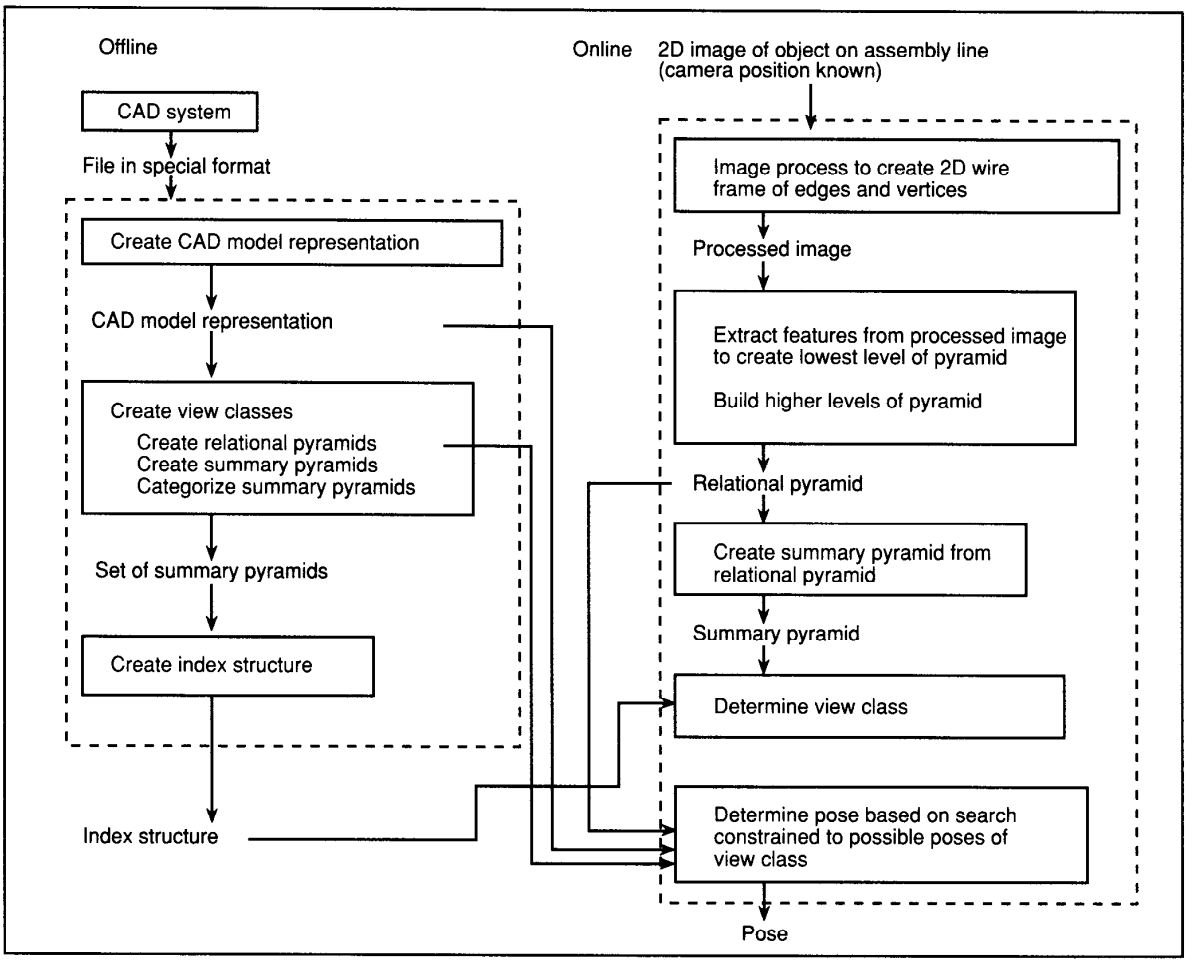


Figure 1. An example of a task in automatic pose determination. Boxed text indicates processing steps. Unboxed text indicates the type of the data object produced by the preceding processing steps.

persistent objects. For good reason, there don't appear to be any systems that use reference counting as a collection mechanism.

(4) The object-oriented approach to data modeling and programming can provide much of the solution to our programming and data modeling needs. However, we require a set-and-relation semantics that supports uniqueness constraints as powerful as those in the relational model. No object-oriented languages to date support such semantics and programmers encounter interesting problems when trying to support them in an object-based system.

(5) An expert system shell (written in the language) is needed for those who, although unfamiliar with the implementation and interrelationships of the abstract data types developed for an application domain, want to do computer vision tasks. This means that an interpreted environment is vital. However, a compiler is also needed to reasonably perform expensive vision tasks. Furthermore, since expert system shells generally perform symbolic computation, an underlying language good for symbolic computation would be helpful. Some object-oriented database systems, such as E and Adaplex,[4] only provide compilation. GemStone and Orion provide both compilation and interpretation.

(6) Much of the database integrity support that industrial-strength object-oriented databases and relational DBMSs provide is overkill for what we need to do. For the most part, data objects used by more than one person are read, not written, in processing steps. For example, several users might use the same images to test algorithms. They only read them. We do not have the same problems a group working on a CAD design has, when multiple users are performing updates to the same objects. In most cases, when a database of objects (for example, some images) is being used by one person and a new user wants to access the database, the new user does so at the risk that someone else will commit a transaction after the new user has committed his or her own. The situation is analogous to editing a document in a multiuser system. If a user wants to insure that his or her changes are not overwritten, the user makes his or her own copy of the database. Thus far, we have not seen the need for transaction and concurrency support beyond what Unix (or a comparable system) provides.

(7) A user should be able to create per-

sistent data objects, copy them, and make them accessible to other executables on other machines. Some systems, such as GemStone and E, impose the constraint that there is one large database with each object in it having a unique ID over all space and time. This makes it difficult to share objects between executables using different databases because the system views the objects it has created as the only ones in existence.

(8) There is no good reason to invent a new programming language to do these things if one exists that comes close to providing the necessary semantics. Taxis,[5] Excess, and S (a language implementing persistent objects for statistical use) are languages designed from the ground up. Adding minor extensions to the syntax of an existing language will attract more programmers and enable us to spend more time on the semantics specific to persistence, data modeling, and computer vision and less on garbage collection and compiler bugs. For this reason, we have chosen to extend the implementation (leaving the syntax intact) of a Common Lisp as a bed for providing all of the properties mentioned above.

In the following sections, we use an example from automatic pose estimation to illustrate the necessity for the above properties in a language system, we indicate where other languages and data management systems fail to provide these properties, and we discuss the type constructors used in our system and features of the language semantics of our system.

Example of an application

Pose estimation is the process of determining the position and orientation of an object from an image of the object. In model-based vision, the vision system works with stored models of the objects it expects to see. It extracts features such as line segments, holes, and corners from the image and creates a data structure representing the features, their attributes, and their interrelationships. It then matches the data structure extracted from the image to the stored models to

(1) determine the identity of the object and
(2) find the correspondence between the features of the image and the features of the correct model so the pose can be determined.

The *on-line* phase of the system is the process of extracting features from an image, constructing the data structure, performing the matching operation, and determining the pose. Generally, the on-line phase has to execute rapidly to meet the time constraints of the application task. As vision systems become more and more automated, many systems also employ an *off-line* phase to construct the stored models from external data and determine the features that will be used in the matching process.

Figure 1 shows the complete description of a pose estimation system. The input to the off-line phase of the system is a CAD model of an object that will be coming down an assembly line. The input to the on-line phase is an image of the known type of object. The task is to determine the position and orientation of the object relative to the camera. We can, from this example, see much of what a programming language needs to provide to easily support writing such an application.

In Figure 1, the sequence of operations on the left-hand side represents the off-line construction of a set of view classes. The sequence of operations on the right represents the on-line pose determination phase. These sequences closely resemble the work of Lu and Shapiro.[6]

The off-line phase can be summarized as follows: The 3D CAD model that comes from a geometric modeling system is converted to a vision model, here deemed the "CAD model." This representation is three-dimensional, and the image (and thus the extracted features used in matching) is two-dimensional.

To facilitate the matching, the CAD model is converted into a set of view classes, each representing one of the topologically distinct views of the object. Each view class is represented by a relational pyramid, a hierarchical relational structure that describes a view class in terms of the features that will appear in an image of that class and the relationships among them. Primitive features appear at level zero. Level one features represent relationships among level-zero features. In general, level-k features represent relationships among features from levels 0 to k-1.

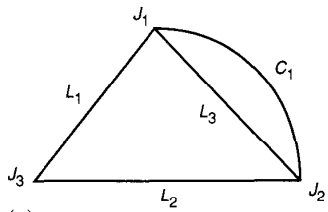
Figure 2a is a line drawing of an object, and Figure 2b shows a simplified *relational pyramid* structure describing the line drawings of that view class. To rapidly determine the view class of the object in the image, a second structure, called a *summary pyramid*, is constructed from the relational pyramid. If the relational pyramid has a relation R with c tuples $\{\{(N_1,t_{1j}), (N_2,t_{2j}), ..., (N_n,t_{nj})\}, j = 1, ..., c\}$, where the N_is are relation names and the t_{ij}s are entities from lower levels, the summary has a corresponding relation R with a single tuple $\{(N_1, ..., N_n), c\}$. Figure 2c illustrates the summary pyramid for the relational pyramid of Figure 2b.

In the on-line phase, the input image is processed to a point where the features can be extracted and a relational pyramid can be built. Next, a summary pyramid is constructed from this relational pyramid describing the features extracted from the image. At this point, this summary pyramid is compared (via an index structure) to the summary pyramids of the various view classes, and one is chosen based on a similarity measure. The correspondences determined from matching the relational pyramid of the image with the relational pyramid of the selected view class determine the pose of the object.

Without knowing more about the operations on and representation of CAD models, relational pyramids, summary pyramids, and index structures, we can make several observations about aspects of this system that require underlying support from the implementation language:

(1) Traditional image-processing systems do not provide enough support in the way of data types and operations. Note that a relatively small part of the application is low-level image processing. Computer vision systems use data structures that are far more complex than plain images. At the same time, structures such as multidimensional images must be easily representable and manipulatable with the tools provided. An extension of the core system will provide structures and operations for low-level image processing.

(2) Data objects with complex structure need to exist between applications and computers. The two dotted boxes (see Figure 1) represent the on-line and off-line parts of the application domain. They are distinct processes. For each index structure produced by the off-line phase in an industrial system using the application, many objects will come down an assembly line whose poses are unknown and must be determined by the system. The two processes are possibly on the same or different machines/networks. We see that the on-line phase uses data objects produced by the off-line phase. These objects need to



(a)

Level 2:
junction adjacency
{(three-line, J_1), (three-line, J_2)}
{(three-line, J_1), (two-line, J_3)}
{(three-line, J_2), (two-line, J_3)}

Level 1:
three-line junctions
J_1 : {(straight, L_1), (straight, L_3), (curve, C_1)}
J_2 : {(straight, L_2), (straight, L_3), (curve, C_1)}

two-line junctions
J_3 : {(straight, L_1), (straight, L_2)}

Level 0:

Straight	curve
L_1	C_1
L_2	
L_3	

(b)

Level 2:
junction adjacency
{(three-line, three-line), 1}
{(three-line, two-line), 2}

Level 1:
three-line junctions
{(straight, straight, curve), 2}

two-line junctions
{(straight, straight), 1}

Level 0:

Straight	curve
3	1

(c)

Figure 2. An example of the low-level features of an object (a), the relational pyramid constructed from it (b), and the summary pyramid constructed from the relational pyramid (c).

persist between processes.

Some persistent object systems, such as GemStone,[7] use one global database giving unique identifiers to each object created by the system. Two separate instances of the system cannot share objects, since both have used the same object identifiers for different objects. This problem can be eschewed by making object identifiers so long (greater than 100 bits) that the identifier can contain a stamp of the executable that created it and hence guarantee uniqueness across more machines than will exist in the next thousand or so years. GemStone, however, only uses 32 bits for object IDs.

(3) There are objects with identical structure, some persistent and some transient. In Figure 1, when the pose of the unknown object is being determined in the last step of the on-line phase, the system uses two separate relational pyramid structures as its input. The first is a relational pyramid created by the off-line phase, which must be in permanent storage before the on-line phase begins. The second, which is a relational pyramid created earlier in the on-line phase, is not persistent. It does not exist after the on-line phase finishes executing.

The issue of whether an object is persistent should not affect the design of types and the coding of functions. When programmers write a non-system-level function, we feel that usually they would like it to work on actual parameters (arguments) that are either local to the process or persistent, as long as they are of the right type.

Furthermore, when a programmer designs a new type, he or she should not have to create two of them that only differ in whether instances of the type are persistent. Of course, both in writing functions and creating new types, we might want to override this feature.

An obvious implication of all of this is that the programmer should never have to worry about how to write a type of object to a file or whether the provided disk representation is compact enough.

The system model — detachable databases

As shown in Figure 3, the base system is a set of databases, an interpreter, and a compiler. The system databases contain the elementary types and functions. They also contain various objects used by the system. Programmers can add databases, put functions and types in the databases that they create, and create instances of these types and put them in the databases.

The idea is that the system is tailorable. If a cell morphology package exists on the computer system, a user can access the database in which it is contained. It is not a permanent part of the system, as is the case when new types are added to a Smalltalk or GemStone system. When a cell morphology package is added, the package's programmer does not start from scratch. The programmer builds on top of packages that other users have created. In this case, the cell morphology package programmer is probably using functions and types from an image-segmentation package. These types would be two-dimensional images, contingency tables, symbolic images, and region relations (sets of objects where each object contains all sorts of relevant information about a region of a symbolic image).

In some other object-oriented languages, such as GemStone's OPAL and E, each executable (OPAL interpreter session or E program) is connected at runtime with one database containing a universe of objects that all users share. It is difficult to build a subsystem, such as a database, with only image-processing operations and types. We prefer to think of each database as an address space containing objects referencing either local objects (in the current database) or objects in outside databases. In this way, our universe of persistent objects is partitioned into modules with related objects that can be transferred between machines and executables without severe problems.

The core system, of course, will be designed with the foresight that many of the derived systems will be doing matrix manipulation as well as more symbolic computation. Traditionally, programming languages have been designed with syntax, semantics, and implementations that emphasize one of these types of computation over the other.

Some type constructors

Figure 4 shows basic type constructors. They can be used to build the types needed for applications. Below, the word *object* refers to an instance of an atomic type, an array, a relation, a record, a sequence, or a set. As a description of some entity, the word object further implies that it can be

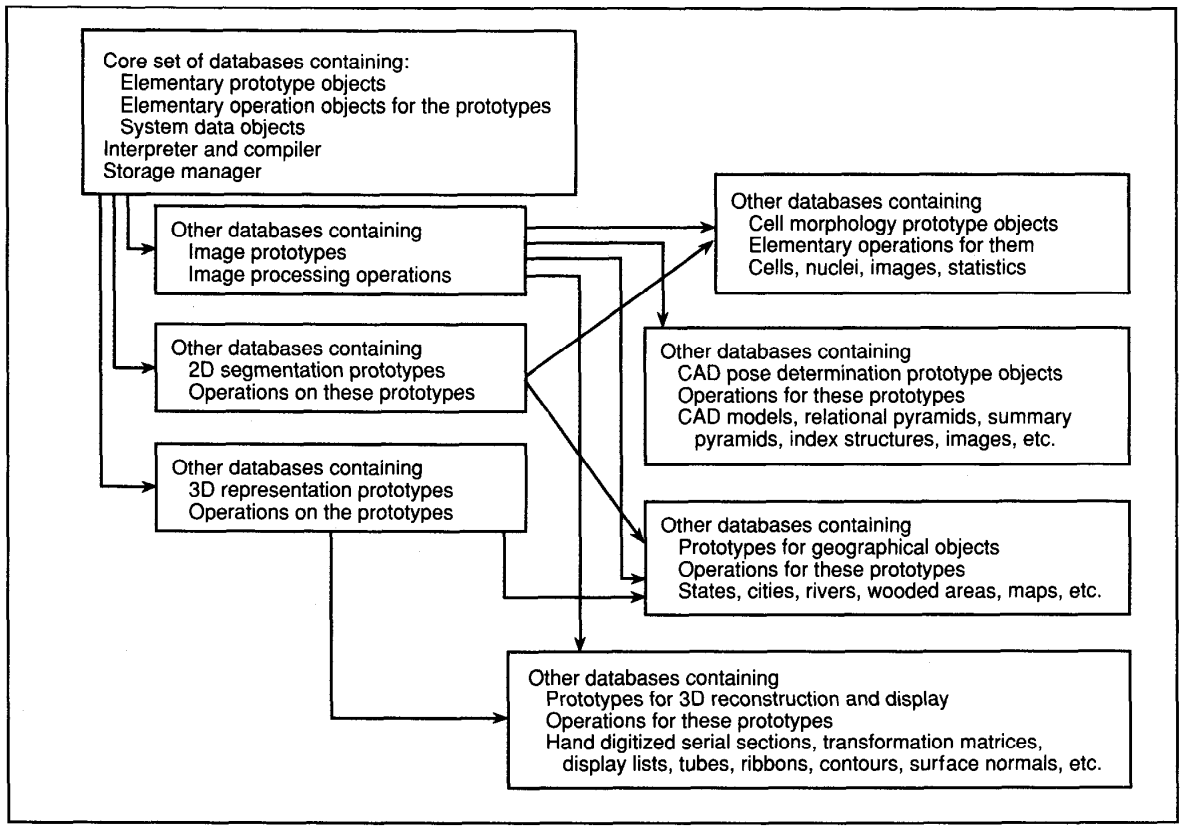


Figure 3. How user environments can be derived from the core system.

referenced from other objects. In contrast, the word *value* refers to an entity to which other objects may not hold references. The elements of an array, for example, may be values in some cases. In what follows, we give informal definitions of the type constructors for objects available in the language.

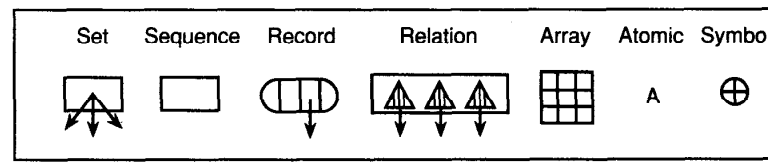


Figure 4. Some type constructors for modeling the highly structured data we find in applications.

• *An instance of an atomic type* is an instance of an integer, float, double, character, or enumerated type. Such an instance is an object and may have multiple references to it. There is a distinction between the sort of values sometimes held in the cells of arrays and fields of records, on one hand, and instances of atomic types, on the other hand. Instances of atomic types are objects and may have references to them. The elements of arrays and records may or may not have object identity (depending on the type of array or record). Their values may only be accessed by operations on the array or record containing them (which *is* an object).

• *An instance of an array type* is a rectilinear arrangement of either atomic values or references to objects. Arrays are a parameterized type parameterized on the type of the cell value. Strings and vectors are instances of array types.

• *An instance of a record type* is like a "struct" (a multifield structure) in C. One can set and retrieve the values of its fields using the names for its fields. The various record types form a lattice representing an is-a relationship (also called a generalization hierarchy). Both fields and functionality are inherited through this lattice. The fields of a record can be atomic or references to objects.

• *An instance of a sequence type* is like a one-dimensional array except that insertion into and deletion from arbitrary positions in the sequence are allowed. Sequences are a parameterized type in the same way as arrays.

• *An instance of a set type* is an unordered collection of references to unique objects, where an object is, in general, an instance of a record type. The elements referred to by a set may be referenced from other objects. Sets are a parameterized type parameterized on the type of the elements referenced. (The meaning of the expression "an object is unique within a set" will be discussed later in this article.)

• *An instance of a relation* is similar in some respects to an instance of a set. The

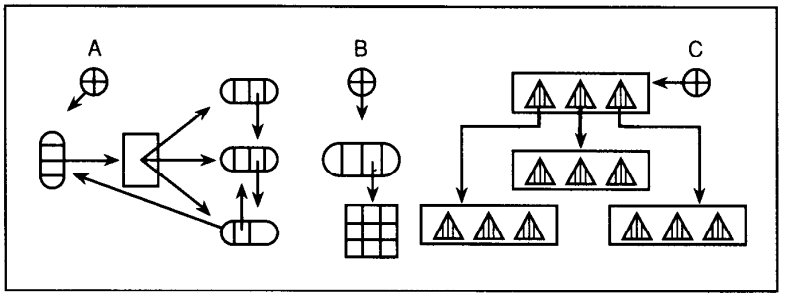


Figure 5. Some typical data structures derived from the type constructors.

main difference is that the elements of a relation do not have identity in the way records do. It is impossible to change or retrieve their attribute values without performing an operation on the relation as a whole. Other objects cannot maintain references to the tuples of a relation. Relation types are parameterized similarly to sets. One feature of relations that allows somewhat more flexibility than in standard relational models is that the elements in a domain of a relation may be object references.

• *An instance of a symbol* is an entity that allows the programmer to give a name to the object bound to the symbol (this object would be the value of the symbol). Symbols play the role of variables in some programming languages. The difference is that, whereas variables in most languages have a type associated with them, symbols are untyped. It is the object to which a symbol is bound that has a type.

Tasks in computer vision often involve transformations of data that may be generally in the form of arrays to data that can take a variety of forms. More array-like data is often produced in image-processing steps. At subsequent steps, such as in mid- and high-level vision, data with more interesting structure is produced.

Figure 5 shows examples of the forms data can take. A, B, and C are symbols that refer to objects.

A is a record that has a set-valued field. The set refers to three records, each referring to one of the others. A is interesting in that it shows that, using the type constructors, we can have data that is an arbitrary graph. At the same time, structure can be imposed on the graph by using a set, which would allow, for example, iteration over the nodes of the graph. Soon, we will provide an example of an application that heavily uses data in this form.

B represents a record that refers to an array object. This type of construct might be, for example, a symbolic image (an image where each pixel's value represents the region to which it belongs). In one field, the record object would contain information about the meaning of the symbol values. Another field might contain information regarding the maximum and minimum symbol values. Another field refers to the array itself.

In contrast to A, where data is nodal and roughly forms a graph, C represents data that is hierarchical. It is similar to what has been called (in database literature) a nested relation (or NF^2 relation[8]). C differs in that its subrelations have object identity, whereas an NF^2 relation does not have object identity. It is also possible for the elements of the tuples of a relation to refer to objects other than relations.

A, B, and C are fairly typical of the forms data takes for the vision applications we have surveyed. In most cases, one or more symbols point into a connected graph of objects. We can refer to the connected graph a symbol refers to as an *object graph*. Symbols can be part of an object graph. The object graph is also often constrained by the procedures that manipulate it to be a tree, list, or directed acrylic graph (DAG). Object graphs are further constrained by a strict typing system that restricts what an individual reference (arrows in Figure 5) may point to. The destination of a reference is constrained to a specified type and that type's subtypes. Since types form a singly rooted lattice, with root t, a reference constrained to be of type t is, in fact, completely unconstrained.

The CAD model

Figure 6 shows the record types for a world of CAD objects. One of these worlds contains a single instance of the CAD-World type, and then multiple record instances of the types Object, Edge, Vertex, Arc, and Face. In a world of CAD objects, there are no record instances of the types Boundary, Surface, 1Dpiece, and 2Dpiece because each of these types is used only as a parameter to a relation object in some other type. This means there are only tuples of the above types, and no records.

All of the types except CAD-root-type are subtypes of CAD-root-type. In keeping with a standard object-oriented paradigm, this means that all types contain the fields, as does CAD-root-type. More specifically, if we define the type CAD-World as follows (using a more readable syntax than Common Lisp)

```
CAD-World := Type( is-a = CAD-root-
    type ) {
  objects: Set( type = Object );
  transformation: Function;
  points: Array ( dimensions = <*,3>,
    type = Float );
}
```

the following function and macros

```
CAD-World.make := Function( name:
    Symbol;
  bounding-box: box(dimensions = 3);
  objects: Set (type=Object);
  transformation: Function;
  points: Array (dimensions = <*,3>,
    type = Float);)
  returns CAD-World
{ ... }

objects := Macro( Arg: CAD-root-
    type ) returns Set(type = Object)
{ ... }
transformation := Macro( Arg: CAD-
    root-type ) returns Function
{ ... }
points := Macro( Arg: CAD-root-type )
returns Array ( dimensions = <*,3>,
    type = Float )
{ ... }
```

are automatically created by the system and can't be redefined without redefining the CAD-World type.

If we do the following:

```
some-CAD-World := CAD World.
    make (some-name, some-objects,
    some-box, some-transformation,
    some-points);
```

and then want to know the objects in "some-CAD-World," we just say

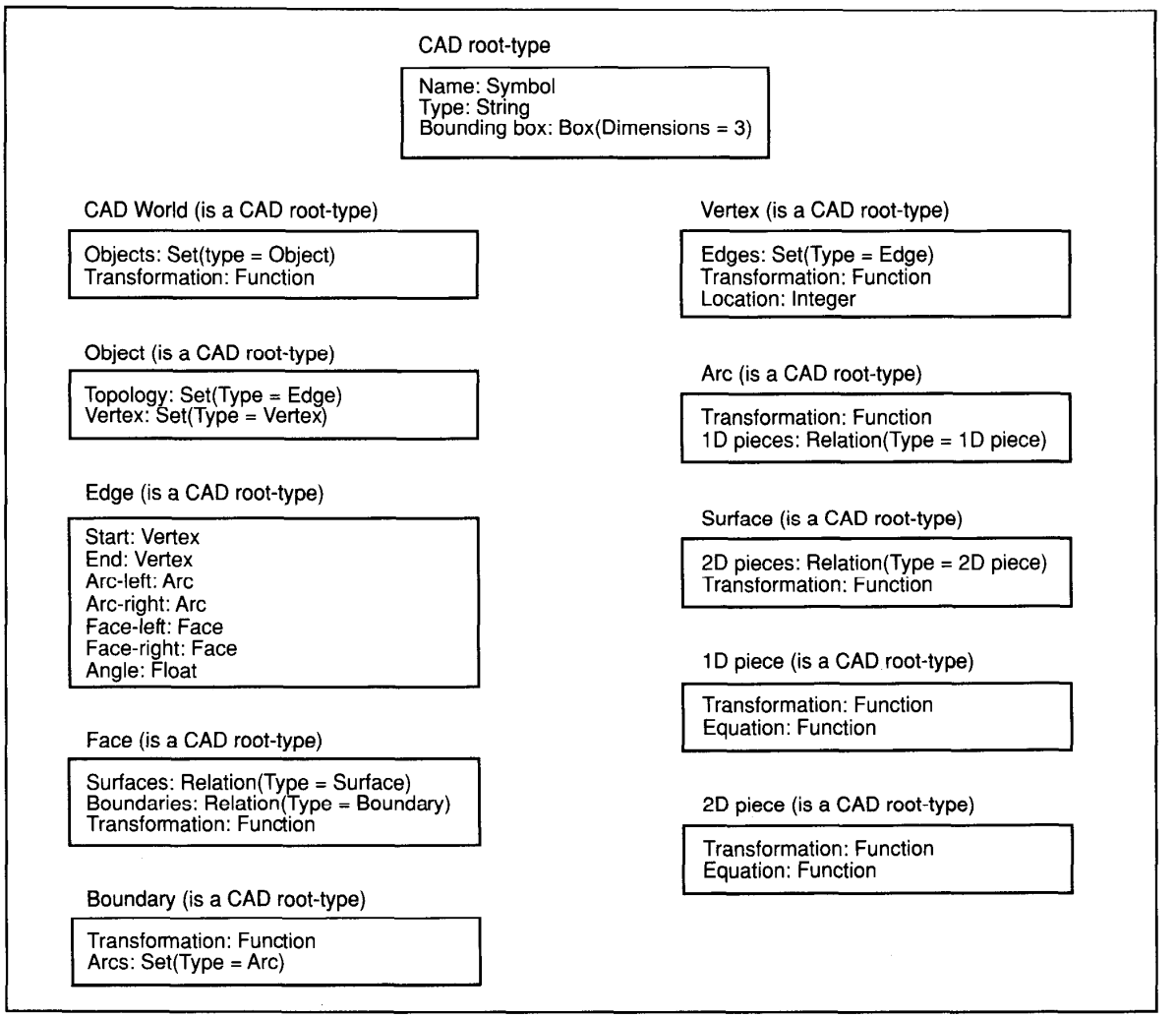


Figure 6. The types for a world of CAD objects. Each box contains the names and types of fields for a record type.

```
objects-in-some-CAD-World:=
    objects(some-CAD-World);
```

When the type CAD-root-type was created, the macro bounding-box was defined as follows:

```
bounding-box := Macro( arg: CAD-
    root-type ) returns box(dimensions
    = 3)
{ ... }
```

If we want to know the bounding box for "some-CAD-World," we just say

```
bounding-box-for-some-CAD-World
  := bounding-box(some-CAD-
  World);
```

Since the type of some-CAD-World is a subtype of CAD-root-type (which is the required argument type for the function bounding-box), the call to bounding box is legal. It is possible that the other type, say Foo, that is not a subtype of CAD-root-type will have a field called bounding-box whose type is, say, Bar. If this happens, the following function definition will occur:

```
bounding-box := Macro( arg: Foo )
    returns Bar
  { ... }
```

The bounding-box function, as well as all the other functions described so far, are generic and hence such a redefinition does not cause a problem.

Figure 7 shows an example of a function that performs a specific kind of query on a CAD-World. Get-adjacent-faces takes as parameters a CAD-World and a face and returns all faces adjacent to the face in the particular CAD-World. The "For" statements allow iteration over the elements of one or more sets or relations, selecting those records or tuples that satisfy the predicate in the "Where" clause (the predicate defaults to logical true if no "Where" clause is specified). When appropriate keys exist on the sets over which selection is being performed, they are used to select the elements satisfying the predicate. Otherwise, straightforward iteration over all elements must be used.

We can make several observations about

```
Get-adjacent-faces := Function(some-CAD-World: CAD-World; some-face:
Face){
        result: Sequence;
        For (i in some-CAD-World.objects) {
                For (j in i.topology) {
                        if (j.face-left = some-face)
                                push(j.face-right, result);
                        else push(j.face-left, result);
                } Where (j.face-left = some-face || j.face-right = some-face);
                break;
        }
        return(result);
}
```

Figure 7. An example of a query on an instance of a CAD-World data structure.

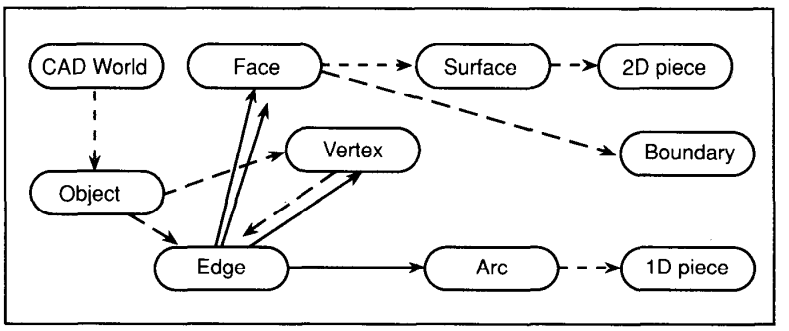


Figure 8. The relationships between instances of types in a CAD model. A node labeled *A* pointing to a node labeled *B* with a solid arrow implies that each instance of type *A* references an instance of type *B*. A node labeled *A* pointing to a node labeled *B* with a dashed arrow implies that each instance of type *A* references a set or relation of instances of type *B*. (The difference in dash length means nothing.) This figure leaves out many of the types in the CAD model.

the code example above:

• *Macros.* The language supports a powerful macro facility that the compiler can exploit to create efficient code. Our high-level extensions to Common Lisp will use macros. One of the most useful features is that macros can be used to define new functions.

• *The ":=" assignment syntax for types, functions, and macros.* This syntax in the example is meant to make it clear that types, functions, and macros are data objects themselves and can be persistent or nonpersistent. The types, functions, and macros of the Common Lisp kernel are viewed as a collection of persistent objects by the system.

• *Why extend Common Lisp? Why not extend CLOS instead?* At first glance, it seems it might be more reasonable to extend CLOS[9] (Common Lisp Object Standard) directly to provide persistence, since CLOS includes many more object-oriented features than does Common Lisp. However, in CLOS, the emphasis is on slot-based objects. We think of the set, sequence, and relation types of our language as parameterized types. It would be unnatural to implement these types as slotted objects, as you would have to do using CLOS. Another point is that we would like to use some of the strict typing features of Common Lisp (such as typed arrays) that would be somewhat awkward to use in CLOS. Furthermore, a straightforward extension of CLOS cannot provide certain features that we think would eventually be useful (for example, parameterized types). Lastly, since we will be providing a superset of Common Lisp, it will be fairly straightforward to build CLOS on top of our Common Lisp if there are those who want to use a CLOS with persistent objects.

• *Protection of the underlying representation of abstract data types.* In many languages, the operations belonging to a class have access to the class' instance variables, whereas nonmember functions do not and instead have to rely on the interface created by the member functions. In our model, we follow the approach to record access that Common Lisp uses for structure access and CLOS uses for slotted object access. This entails that, for any function, all access to the data contained within an object is through a functional interface. To create privacy, the privacy semantics of standard object-oriented languages in CLOS or Common Lisp, a type can be created in a package with the accessors made private to the package.

Interconnectedness of object graphs

Each arrow in Figure 8 represents an attribute of the type at the base of the arrow. This sort of picture shows up in a lot of semantic data modeling literature. In the ER model as discussed in Tsichritzis and Lochovsky,[10] each arc would be represented as a named relationship between objects of the types at either end of the arrow. In the functional data model implemented in Daplex,[4] each arrow would be a function with the type at the tail representing the domain and the object at the head, the range.

In each case, these data models began as database designing tools. Often, a query language was supplied that allowed queries on a database to be embedded in another procedural language. In the Daplex case, a language was built around the data model that was more expressive than a query language, but not as flexible as a standard procedural or functional language.

From the outset, our plan has been to bring the data modeling tools closer to what is found in standard programming languages. Hence, our objects explicitly reference what they need to. The language for data modeling should be uniform for defining the types of both persistent and transient objects.

As a result, we provide a more complete semantics for sets than is found in nonpersistent languages by allowing the programmer to have control over what constitutes uniqueness of a set's members. We provide less constraints than do those languages exclusively for database programming.

In normal programming, the data objects have no constraints (except type constraints) unless the programmer builds them into the operations that change these objects. We take the same approach. The data model provides a minimum number of constraints so the difference between programming with persistent objects and normal in-core data structures becomes minimal.

In some ways, this effort resembles that of GemStone, since the idea behind GemStone was to add persistence to Smalltalk. GemStone tries to treat persistent objects the same way as transient ones.

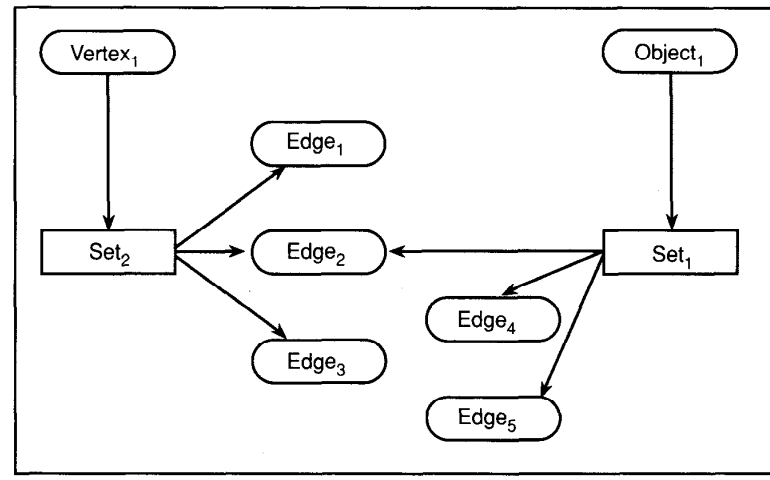


Figure 9. How objects can be members of more than one set. Ellipses indicate record objects; squares indicate set objects.

The lack of set semantics in object-oriented languages

Figure 9 shows part of an instance of a CAD model. It illustrates that objects can often belong to more than one set. In this case, an instance of an Object type and an instance of a Vertex type maintain sets of Edge objects. The instance of type object maintains its set through the topology attribute. The instance of type Vertex maintains its set through the edges attribute.

This sharing of objects does not occur just between sets. Sharing can occur between two nonset attribute values or between a set object and a nonset attribute. An example of the latter case happens between Object instances and Edge instances. Object instances contain a set of instances of type Vertex (representing the vertices in the object). Edge instances refer to a vertex for their start and end points. Often the same vertex will be referenced both through the set object and through an attribute of the Edge object.

We can see, then, that to support our applications, it is important to implement the abstract data type of a set whose elements may be referred to from elsewhere.

These set semantics are provided by object-oriented languages such as GemStone and Smalltalk. But the languages providing these semantics fail to provide other set semantics that are also necessary. In these languages, a set is a collection of unique objects where uniqueness has the following definition: Objects *a* and *b* are unique if they do not have the same object identifier. This, of course, implies that for sets of this type, there could be multiple members with different object IDs but exactly the same attribute values. We see a need to support uniqueness constraints beyond the requirement of unique object IDs for set members.

Providing such uniqueness constraints is something relational DBMSs do well. When we create a set or relation, we would like to be able to create uniqueness constraints similar to the uniqueness constraints that exist in relational DBMSs.

Three basic problems come up when attempting this in an object system such as ours:

(1) The members of sets and relations can have fields that are nonatomic. This is not true in the relational model. As a result, what it means for a member of a set or relation to be unique with respect to the others is necessarily more complicated than in the relational model.

(2) Since the members of sets (which are records) are objects, insuring that sets are not corrupted by updates done through outside references is a problem. While a record is a member of a set, it can be updated through another object that has a reference to it. If a field guaranteed to be unique in a set is updated to a nonunique value, the set is corrupted.

(3) Since we want detachable databases (as compared with a one-world model such as GemStone) in our model, a set possibly exists in a database not currently linked that contains a record in a database that is linked. If a key field of this record is updated through some outside reference, the uniqueness constraints of the set could be corrupted.

Conventional programming languages do not provide set semantics either. There are no programming languages that implement sets of records like objects in a complete way. The set type constructor, as we use it, is pervasive in the example application domains that we have looked at. Hence, support must be provided for it at the language level.

If we do not extend the semantics of some already existing language, or create a new language that supports set semantics, programmers who write vision applications will end up doing it repeatedly as a procedural interface in each new domain.

Edge example. The set-theoretic definition of a relation requires a relation to contain unique *n*-tuples (where two *n*-tuples are unique if they do not have identical attribute values). For practical purposes, this is a minimal and infrequently used constraint. In reality, relational database schemas impose stronger constraints on what may be inserted in a particular set or relation by specifying that no two tuples will have the same value on certain fields or combinations of fields. Each such field, or combination of fields, is called a key. When no key is specified, uniqueness is commonly not enforced on the attributes. Instead, a system-maintained, globally unique tuple identifier is used as an implicit additional field and serves as a key. In general, some sort of index structure is maintained for each key.

We would like to be able to impose such constraints on the set and relation types in

our language. For example, we would like to impose the constraint on the topology field for the type Object in Figure 6 that no two edges have the same start and end point. In particular, we would like a run-time error to be signaled if an insertion into the set of edges violates one of these constraints or if an instruction is executed that changes the value of start or end in such a way that the record is no longer unique according to the above constraints.

We could define the set of constraints as follows:

```
k1 := Key(type = edge) {
   start,end                    : UNIQUE
}
```

Then, the topology field of the Object type could be changed to

```
topology: Set(type = Edge, keys = k1);
```

Before making this change, uniqueness would have been guaranteed on the object identifiers of the individual edges over the topology set. This is less of a constraint because it is possible that a user could have instantiated two edge objects with identical start and end points. If topology had been declared with the k1 constraint, upon insertion of the second edge record with identical start and end points, the k1 constraint would not have been met and the insertion would not have been allowed.

Perhaps we would like to be even more specific about what constitutes a unique edge over the topology set. If we created another key and used it in the definition of the topology relation:

```
k2 := Key(type = edge) {
   start.location, end.location
                                : UNIQUE;
}
```

we would protect the set against the possibility of there being two edges with different object IDs whose start and end points have different object IDs but the same value for their location (the same index into the array of points).

Continuing with the example, if record types had been defined for R-tree nodes[11] and operations written that implemented insertions and deletions into an R-tree, we might want to index the topology field on the bounding boxes of the edges. One bounding box possibly corresponds to more than one edge, so we would not specify such a key as a uniqueness constraint. Instead, we might say

```
k3 := Key(type = edge) {
   start.location, end.location
                                : UNIQUE;
   bounding-box: NONUNIQUE;
}
```

Given an object ID of a bounding box, this would allow us to find all edges for which it is a bounding box in logarithmic as opposed to linear time, since a key would exist mapping the object IDs for bounding boxes to edge objects.

The full semantics of keying, as discussed above, is not supported in data models other than the relational model:

- **In other object models, the uniqueness constraint is on the object ID of members.** Were we to simply put the constraint that the object IDs of the edges must be unique, we could end up with more than one edge with all the same field values, or the same start and end points, or the same left face and right face.
- **In other object models, keys on attributes (if allowed) are on single fields.** We would like to be able to key over a combination of fields. Taking the Edge type in Figure 6 as the example, we would like to be able to specify that it is only the combination of start and end that is required to be unique over a set. There can be more than one set member with the same value for start or the same value for end. No other object models allow this sort of keying.
- **A constraint should apply to a set, not all of the instances of the type in the universe.** In some models, such as GemStone's, a uniqueness constraint applies to all the objects of a type in the universe, not just to those in a particular set. The uniqueness constraints are actually part of the record type and are inherited in subtypes. We prefer to look at the constraints for a set as a parameterization of the set, not the type of the element in the set.
- **You should be able to put different constraints on different sets of a particular type.** In our example, we might want to enforce k3 on the set of edges for each object, but not enforce the constraint on the set of edges connected to a Vertex instance. The reason for this would be that if, as a rule, the software using the types in the CAD example inserts edges into the topology relation of an object as soon as they are created, there will be no duplicate edges (in the sense of k3) for an object. Now, when a vertex is instantiated, the edges inserted into the edges relation of the vertex are unique on start and end if they have been inserted successfully into the topology relation of the object. There is no need for the overhead of keeping the additional constraint.

Expert system assistance in computer vision

Predominantly, we design and implement a programming language to support writing vision applications so we can use the language to develop expert system tools. In turn, we use those tools to perform vision tasks for end users unfamiliar with the software configuration of the underlying system. We envision end users to be those who do not assist in the development of the software implementing the abstract data types used in a particular vision system.

As we see it, the plight of the end user is to extract high-level information from a starting set of low-level data that the user brings to the system from sensing devices (digitizers, for example) or software tools (CAD/CAM systems, for example) used in the user's discipline. From this point, he or she would like to be able to accomplish something without learning a new programming language and then learning about all the abstract data types implemented and their interrelationships.

Users may possibly want to become acquainted with the system and its details, but we feel it's more likely they will not. In trying to use the bare system, users may encounter one of the following two barriers:

(1) Users will know what form of data they would like to produce but not the needed programming steps (the sequence of operations required and the parameters for those operations) to produce data in this form from the initial data.

(2) Users will want to perform some operation they have heard exists on the system but will have data that is not in precisely the right form for use.

For example, as shown in Figure 10, a typical user may have a feeling for what the various subtypes of type Rectilinear (Histogram, Matrix, Contingency-table, and Symbolic-image) are and what sorts of functions should operate on them. In addition, users may know conceptually what a Map is. But, they might not know the difference between the various subtypes of

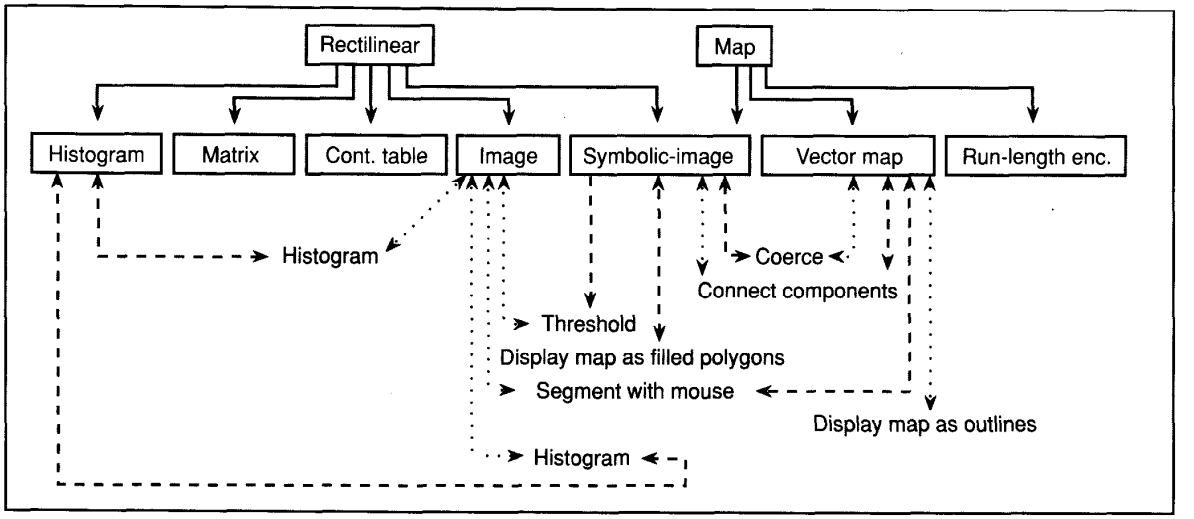


Figure 10. Some types and operations for a two-dimensional image-segmentation package. These types form a sublattice of a larger type lattice. Each operation maps an object of the type connected to it by a dashed arrow to the type connected to it by a dotted arrow. A double-headed arrow between a type and a function indicates that the type maintains a reference to the function (through a set attribute containing all functions for which that type is a parameter) and that the function maintains a reference to the type (through its set of parameters and the parameter types).

Map (Symbolic-image, Vector-map, and Run-length-encoding).

If this user has an image of interest, he or she might want to do the following:

(1) Segment the image using the mouse, and then
(2) Display the resulting map with outlines around the regions.

In this case, users would know they want to use two operations: Segment-w-mouse, and Display-map-as-outline. There are no problems in this instance, since the type of the result of the first operation is the type of the input to the second.

But, if users want to take the result of the segmentation and display it as filled regions (using Display-map-as-filled-polygons), they have a problem because Display-map-as-filled-polygons requires a Symbolic-image as its argument. In this case, the shell would be able to find an appropriate coercion path to carry out this operation. In particular, it would apply the operation Coerce to the Vector-map.

In general, several paths could qualify as coercions. Some coercions are complex in that they might require additional arguments since they are not simple "filters" that convert from one type to another. If the user attempts to use Display-map-as-filled polygons, but supplies as an argument an object of type Image instead of Symbolic-image, the following multiple coercion paths exist:

(1) First threshold, then display. (Thresholding requires additional arguments for which the system would query the user.)
(2) First segment with the mouse, then coerce to Symbolic-image, then display.

In such cases, user interaction would be requested to choose the appropriate path. The system can give hints as to which paths will yield the best performance.

Implementation details

Ultimately, the questions of whether to use single or multiple inheritance, whether to implement parameterized classes, and whether to create a syntax and semantics such as shown previously, affording strict typing on parameterized types (sets, sequences, relations, arrays), record types, and atomic types, will be very important to us. However, in the immediate future, we feel that it is much more important to provide a general programming language that treats persistent and transient general-object graphs in an easy-to-use fashion with reasonable performance and main memory usage. For now, Common Lisp provides enough object-oriented features; all data and functions are objects with self-contained type information, and the structs of Common Lisp allow for a form of inheritance and code reuse.

We are using the struct type of Common Lisp heavily for our implementation of records and tuples. Since updates of records can trigger updates of sets and checks of set integrity, the implementation of records and tuples is not a straightforward extension of Common Lisp structs. Persistent sets with no keys will be implemented as persistent hash-tables where the keys are the object IDs of the elements in the set. Each additional key will be implemented as one or more B+ trees where the nodes of a tree are treated as persistent objects by the storage system. Such trees allow for range queries where a hashing approach does not. In general, when an object *o* is needed from disk, the objects recursively reachable from it are brought in up to a user-controllable depth. *Stubs* are created for the objects reachable from *o* that are one level of depth beyond those in main memory. When a stub is referred to, a fault occurs and the object and other reachable objects are read from disk.

Through this article, we have justified the need for a new programming language and data management system suited to the needs of programmers in application domains such as computer vision.

We have shown where existing systems fail to provide much needed features and where existing systems provide unneeded features that will burden a system fully loaded with the large data objects and the spectrum of operators and types encountered in vision applications. Additionally, we have shown the need for type constructors not found in other object-oriented systems such as relations and sets.

Finally, we have concluded that, with relatively small extensions to an existing language (Common Lisp) that already provides many of the facilities we need, we can provide the semantics a language needs to make development of applications an easier task. ■

Acknowledgments

This research was supported by the National Science Foundation and DARPA under grant DMC-8714809 and via a gift from Texas Instruments.

References

1. L.G. Shapiro and R.M. Haralick, "A Spatial Data Structure," *Geo-Processing*, Vol. 1, 1981, pp. 313-317.

2. J.E. Richardson and M.J. Carey, "Persistence in the E Language: Issues and Implementation," Tech. Report 791, Dept. of Computer Sciences, Univ. of Wisconsin, Madison, Wis., Sept. 1988.

3. S. Vandenverg, M.J. Carey, and D.J. DeWitt, "A Data Model and Query Language for Exodus," Dept. of Computer Sciences tech. report, Univ. of Wisconsin, Madison, Wis., Dec. 1987.

4. M. Atkinson and O.P.Buneman, "Type and Persistence in Database Programming Languages," *ACM Computing Survey*, Vol. 19, No. 2, June 1987, pp. 106-190.

5. J. Mylopoulos and H.K.T. Wong, "Some Features of the Taxis Data Model," *Proc. 6th Int'l Conf. Very Large Data Bases*, Oct. 1980, CS Press, Los Alamitos, Calif., Order No. 322, pp. 399-410.

6. H. Lu and L.G. Shapiro, "Model-Based Vision Using Relational Summaries," *Proc. SPIE Conf. Applications of Artificial Intelligence*, Mar. 1989, pp. 662-675.

7. *Programming in OPAL*, Servio Logic Development Corp., Beaverton, Ore., Mar. 1986.

8. P. Dadam et al., "A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies," *Proc. ACM SIGMod*, 1986, pp. 356-364.

9. S.E. Keene, *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley, Reading, Mass., 1988.

10. D.C. Tsichritzis and F.H. Lochovsky, *Data Models*, Prentice Hall, Englewood Cliffs, N.J., 1982.

11. N. Roussopoulos and D. Leifker, "Direct Spatial Search on Pictorial Databases Using Packed R-Trees," *Proc. ACM SIGMod*, S. Navathe, ed., Oct.1985, pp. 17-32.

Andrew M. Goodman is a member of the Intelligent Systems Laboratory at the University of Washington and a graduate student in the university's Department of Computer Science. His technical interests include programming languages, programming environments, database systems, and medical applications for computers.

Goodman received his BA in computer science from Cornell University with distinction in 1985.

Robert M. Haralick is the Boeing Clairmont Egtvedt Professor in Electrical Engineering at the University of Washington. His recent work is in shape analysis and extraction using the techniques of mathematical morphology, robust pose estimation, and techniques for making geometric inference from perspective projection information.

Haralick received a BA in mathematics in 1964, a BS in electrical engineering in 1966, an MS in electrical engineering in 1967, and his PhD in 1969, all from the University of Kansas. He is an IEEE fellow, a member of the IEEE Computer Society, serves on the Editorial Board of *IEEE Transactions on Pattern Analysis and Machine Intelligence*, and is the computer vision area editor for *Communications of the ACM* and an associate editor for *Computer Vision, Graphics, and Image Processing* and *Pattern Recognition*.

Linda G. Shapiro is a professor of electrical engineering and adjunct professor of computer science and engineering at the University of Washington. Her research interests include computer vision, artificial intelligence, pattern recognition, robotics, and spatial database systems.

Shapiro received a BS in mathematics from the University of Illinois, Urbana-Champaign, in 1970, and MS and PhD degrees in computer science from the University of Iowa, Iowa City, in 1972 and 1974, respectively. She is a senior member of the IEEE, a member of the IEEE Computer Society, the ACM, the Pattern Recognition Society, and the American Association for Artificial Intelligence, and is editor of *Computer Vision, Graphics, and Image Processing* and an editorial board member for *Pattern Recognition*. She was co-program chair of the IEEE Computer Vision Workshop in 1982, general chair of the IEEE Computer Vision Workshop in 1985, and general chair of the IEEE Conference on Computer Vision and Pattern Recognition in 1986.

The authors can be contacted at the Intelligent Systems Laboratory, Dept. of Electrical Engineering, FT-10, University of Washington, Seattle, WA 98195.