OMEGA: AN OBJECT-ORIENTED IMAGE/SYMBOL PROCESSING ENVIRONMENT

Mark J. Carlotto & Jennifer B. Fong
The Analytic Sciences Corp. (TASC)
55 Walkers Brook Dr.
Reading, MA 01867

ABSTRACT

A Common Lisp software system to support integrated image and symbolic processing applications is described. The system, termed Omega is implemented on a Symbolics Lisp Machine and is organized into modules to facilitate the development of user applications and for software transportability. An object-oriented programming language similar to Symbolics Zetalisp/Flavors is implemented in Common Lisp and is used for creating symbolic objects known as tokens. Tokens are used to represent images, significant areas in images, and regions that define the spatial extent of the significant areas. The extent of point, line, and areal features is represented by polygons, label maps, boundary points, row- and column-oriented run-length encoded rasters, and bounding rectangles. Macros provide a common means for image processing functions and spatial operators to access spatial representations. The implementation of image processing, segmentation, and symbolic processing functions within Omega are described.

1. INTRODUCTION

Applications involving the analysis of imagery, maps, and other kinds of spatial information requires a software environment that can provide the full range of image processing, segmentation, and symbolic processing functions. One approach to developing such applications is to partition the computation into parts, performing numerically-intensive image processing on array processors or vector machines and symbol processing on Lisp machines, for example. Unfortunately, such an approach does not lead to tightly coupled and highly integrated software systems; i.e., there tends to be artifical boundaries established in the software architecture. For example, where should segmentation functions be implemented since they serve as the bridge between the pixel and symbolic domains?

This paper describes an integrated image-symbol processing environment that has been implemented on a Symbolics Lisp Machine. Similar systems have been implemented at SRI[1], ADS[2], and VPI[3] to name just a few. The key features of Omega may be summarized as follows:

<u>Modular design</u> - the software is organized into software modules with machine-independent functions written in Common Lisp for software transportability, and machine-dependent software written in Symbolics Common Lisp and Zetalisp;

<u>Object-oriented programming</u> - key data structures for representing images, regions that define the spatial extent of significant areas in images, and symbolic data are implemented using our own object-oriented language that is similar to Zetalisp/Flavors;

<u>Spatial representations</u> - a variey of spatial representations are provided for delineating points, lines, and areas; conversion between spatial representations occurs transparently to the programmer;

<u>Uniform access</u> - macros provide a uniform means for region-of-interest-directed image processing functions and spatial operators to access the spatial representations;

<u>Spatial indexing</u> - auxillary data structures are provided to facilitate the transfer of information between the pixel and symbolic domains.

The organization of this paper is as follows: Section 2 describes the organization of the Omega software system. Section 3 outlines our implementation of an object-oriented programming language in Common Lisp which is used extensively in Omega for representing images, regions, and symbolic quantities. The representation of spatial information is discussed in Section 4. Section 5 describes macros for accessing spatial information and their use by image processing functions and spatial operators. Segmentation and associated data structures, and symbol processing are also discussed there. Section 6 summarizes our work to date and outlines future plans.

2. SOFTWARE ORGANIZATION

The Omega software is structured into a hierarchy of modules as shown in Fig. 1. This is done for two reasons: 1) customization, allowing an application to extract and use selected functions, and 2) transportability of Omega software to other Lisp environments. Common Lisp *packages*[4] are used to implement the software organization. All functional interfaces between packages is through inheritance. A parent package inherits the functions of the child packages that it uses; therefore, parent packages are dependent upon the children but the children are independent of the parents. As shown in Fig. 1, the software has been organized into the following packages:

TOKENS - providing the object-oriented data structures and operators used throughout Omega for representing images, regions, and symbolic quantities;

OMEGA STORAGE REPRESENTATION - machine-independent software that defines and maintains objects known as refresh memories that are manipulated by the image processing and segmentation functions in Omega;

SYMBOLICS STORAGE REPRESENTATION - machine-dependent functions responsible for image memory allocation and access optimization and for loading/storing images from/to the host and remote file systems;

COLOR DISPLAY - machine-dependent software providing the necessary interfaces (lookup tables, cursor control) and graphics (text and polygon overlays) for image and symbol display on the Symbolics color monitor;

REGIONS - data structures and operators for representing 2d information in the form of label maps, run-length encoded rasters, boundary points, polygons, and bounding rectangles;

SEGMENTATION - functions that perform histogram-based image segmentation, region growing, connected pixel labeling, etc.

IMAGE PROCESSING - functions that perform general point and local operations, statistics generation, image filtering and restoration, etc.

SPATIAL ROUTINES - operators for computing geometrical, topological, and relational properties between significant areas;

GRAPHICS - functions for 1d, 2d, and 3d (isometric) plotting;

USER INTERFACE - provides scrollable and pull-down menus based on the Symbolics window system for accessing basic functions within the Omega environment and for building custom applications.

The OMEGA package inherits all the above packages. Custom applications may inherit any related subset of Omega packages.

In creating a general software development system, easy maintenance and transportability to other machines were major design considerations. Although Omega is currently hosted on Symbolics hardware[5], the majority of the software is written in Standard Common Lisp[4]. As shown in Fig. 1, the software is explicitly divided into machine-dependent and machine-independent packages. This breakout of code is done to minimize the effort in rehosting the software on other machines and to help focus the developer of an application system on optimization and interface issues. The hierarchical organization of software packages explicitly defines dependencies between Omega functions, and allows the user to readily identify device dependent software.

3. OBJECT-ORIENTED PROGRAMMING

Since Omega is designed to be easily transportable to other lisp environments, the underlying data structures and their manipulations requires an implementation in Standard Common Lisp. Uniformity among data structures is captured using object-oriented programming[6]. Data types are defined in terms of object class descriptions and procedures (operators) for handling messages to instances of object classes. In Omega, objects are known as *tokens*, and are used to represent images, significant areas within images, regions defining the spatial extent of significant areas, and other symbolic quantities.

We developed our own object-oriented programming language for the Omega System. Other Common Lisp object-oriented programming languages such as Common Loops[7] or CLOS[8] were not available at the time of the development. Our implementation is based on the use of Common Lisp *structures*. A structure is defined with the creation of a new token class using the form:

(deftoken token-class (mixins) (attributes))

where *mixins* are other token classes and *attributes* are the possible slots for *token-class*. Mixins allow tokens to be constructed hierarchically from other tokens, inheriting attributes, values, and operators. A default token class, similiar to the Vanilla flavor in Zetalisp, can be mixed into any token definition, and provides operators for describing a token instance or token definition. Messages to tokens are handled by operators attached to a structure attribute. An operator is defined with the form

(defop (message token-class) (arguments) body)

and would be called as

(send-message token-instance message (arguments)).

Default operators to read and write attribute values are created when the token class is defined.

The performance of our own object-oriented programming language has been compared to Flavors and Common Loops (not implemented in microcode). It is about three times slower than Flavors (which is implemented in microcode) and comparable in speed to Common Loops. Functions for storing/loading token instances to/from disk are also provided in Omega.

As noted earlier tokens are used to build many of the data structures used within Omega. *Images* are represented by tokens known as *refresh memories* which contain slots for the image array, header information (e.g., image name, description), instances of region objects which define regions of interest within the image, and a lookup table which specifies how the image is to be displayed. The spatial extent of regions within images are represented by other tokens known as *regions* which contain slots for region labels, polygons, boundary point lists, row- and column-oriented run-length encoded rasters, and bounding rectangles. Regions are discussed in more detail in Section 4.

Where refresh memories and regions are quite specific, tokens can, conceivably, be used to represent almost any kind of spatial or non-spatial data in Omega. For example, significant areas in images can be represented by tokens that contain slots for: an instance of a region to describe the spatial extent of the token, attributes which describe the token itself (e.g., area, perimeter, centroid, average value), pointers to other tokens to describe relationships such as adjacency and containment, and hypotheses (e.g., values to denote the probability or evidence that a particular token belongs to a given class). One such example,

```
token-43
   label          6
   compactness    .8
   area           1034
   contains       (token-232 token-103)
   region         region-81
   building       .3
   forest         .7
   open-area      .1
```

might represent an area that has been identified by some inference mechanism to be a forest.

4. SPATIAL REPRESENTATIONS

A variety of spatial representations are provided to support region-of-interest-directed image processing and spatial operations. These representations are stored as slot-values in region tokens. Regions may be attached to RFs to define regions of interest within images as mentioned above, or be attached to significant areas to describe their spatial extent. Each spatial representation has an associated *looping macro* which is used for accessing the pixels within that representation. The use of looping macros is discussed in Section 5.

Six kinds of spatial representations are currently defined: label maps, x-rasters, y-rasters, polygons, boundary points, and bounding rectangles. In the spirit of object-oriented programming, conversions between representations takes place transparently as they are needed. For example, if the boundary points of an area are needed, if the x- and y-rasters exist then the boundary points are computed from the intersection of the x- and y-raster representations; if the polygon exists, it is computed by generating the intermediate points between the polygon's vertices; otherwise, the programmer is notified that the boundary points cannot be computed from the available information.

Label maps are the most direct way of representing points, lines, and areas. They may be spatially connected (4-, 6-, or 8-connected) or disjoint. All segmentation functions in Omega output a label map and an associated list of tokens. In the label map, pixels are assigned values to correspond to the segments they represent. These same values are stored as slot-values in the corresponding tokens. The macro, *LOOP-MASK*, is used to sequentially generate the addresses of all pixels in the label map with a given value.

X- and y-rasters are used to represent connected regions using row- and column-oriented run length encoding schemes. They may be generated from label maps or from polygons. X-rasters are defined by a y offset (the y coordinate of the topmost row of the region) and an array of x coordinates. The elements of the array contain the lists of the x coordinates which define the beginning and end points of runs of pixels in each row. The y coordinate of a run is equal to the array index plus the y offset. The length of the array is thus equal to the number of rows in the connected region. In Fig. 2, the x-raster representation of the bitmap (a) is depicted in (b). A similar approach is used to represent y-rasters which are defined by an x offset (the x coordinate of the leftmost column of the region) and an array of y coordinates which define the beginning and end points of runs of pixels in each column (c). The looping macros, *LOOP-X-RASTER* and *LOOP-Y-RASTER*, are used for accessing pixels in connected regions sequentially by row and by column.

Boundary points are the pixels which comprise the boundary of a polygon, or of a region represented in a bitmap. The boundary point list defines a tightly closed boundary, similar to the one discussed by Merrill[9], where adjacent points are no farther than a diagonal pixel distance apart. The boundary

points of a region are the union of the x- and y-rasters (Fig. 2d). The looping macro, *LOOP-BOUNDARY*, is used for accessing the boundary pixels of a connected region.

Polygons provide a means for representing information that may be manually specified (e.g., a region of interest). They are stored as a list of vertices and may be converted to rasters, boundary points, or bounding rectangles. Bounding rectangles are a special type of polygon which are used to facilitate the access of subimages in image processing and to speed up certain kinds of spatial operations. The looping macros, *LOOP-RECT* and *LOOP-ROI* are used for accessing the pixels within bounding rectangles.

5. PIXEL-SYMBOL PROCESSING

In Omega, a major design goal was to develop a highly integrated software system, i.e., one in which pixel and symbol processing functions share the same basic data structures, access these data structures in the same manner, and follow common conventions so as to allow information to pass easily between the pixel and symbolic domains. Earlier sections addressed the development of common data structures for representing images, regions, etc. This section discusses how image processing functions and spatial operators access these common structures and describes methods to facilitate the transfer of information between the pixel and symbolic domains.

5.1 Looping Macros

Looping macros are lisp forms which are used for accessing pixels via the various representations described in the previous section. They provide a common mechanism for image processing functions and spatial operators to access regions and their underlying spatial representations. Six kinds of looping macros are available:

LOOP-MASK - used to generate the addresses of all pixels in a label map with a given value;

LOOP-X-RASTER and LOOP-Y-RASTER - used for accessing pixels in connected regions sequentially by row and column;

LOOP-BOUNDARY - used for accessing the boundary pixels of a connected region;

LOOP-RECT - used for accessing the pixels within a bounding rectangle;

LOOP-ROI - used for accessing input image data within a rectangular region-of-interest at one position in the image space and output image data within a different rectangular region-of-interest at another position in the image space.

For example, the form

(LOOP-X-RASTER region w_x w_y *array &body body)*

allows the pixels in a region to be sequentially accessed via the region's x-raster representation where *region* is an instance of a region token, w_x and w_y are the x- and y- dimensions of a processing window, *array* is the image array over which the pixel addresses are being generated, and *body* contains the lisp forms that are to be evaluated within the macro. All looping macros provide arguments for specifying the array being accessed within the body of the macro to insure that pixel addresses outside the array are not generated by the macro. All macros, with the exception of LOOP-MASK, allow a processing window (which is used in conjunction with another macro described below for local or "sliding window" processing) to be defined.

5.2 Image Processing

Image processing functions, while using regions and other data structures, typically transform and alter image objects only. Image processing functions are generally built with the macro

(LOOP-IMAGE w_x w_y *in-array out-array in-roi out-roi loop-mode &body body)*

which allows the programmer to access any of the above looping macros by specifying a *loop-mode*. Depending on the looping mode, *in-array* and *out-array* may be different images (for LOOP-ROI and LOOP-MASK), or the same image (for LOOP-X-RASTER, LOOP-Y-RASTER, LOOP-BOUNDARY, and LOOP-RECT). Similarly, *in-roi* and *out-roi* may be the same or different regions, or *nil* in the case in which we wish to access the entire image array. The programming example in Fig. 3 shows how a region-of-interest-directed image processing function can be easily built with such a macro. The example shows a function to compute the local maximum of an image within a rectangular window. Embedded within the LOOP-IMAGE macro is a second macro, *LOOP-WINDOW*, which generates pixel addresses within a w_x by w_y window. The w_x and w_y, array, and region arguments to the LOOP-IMAGE macro are provided to ensure that LOOP-WINDOW does not generate pixel addresses outside the arrays being accessed (Fig. 4).

5.3 Spatial Operators

There are two basic classes of spatial operators: those that compute information about regions, and those that compute information about relationships between regions. Fig. 5 is an example of a spatial operator definition (defop) for computing the average value of an image over a region. Since the computation involves only one token at a time, it is referred to as a *unary* operator. Other kinds of unary operators compute region properties such as area, perimeter, centroid, etc. Compilation of the operator in Fig. 5 creates a lisp form that is evaluated whenever the message, *:average*, is sent to a token instance. For example,

(*send-message :average token-56 rf-1*)

causes the region representation and area for *token-56* to be retrieved (if the area is not known it would be automatically computed provided an area operator has been defined), the macro to loop over all pixels in the region bound to *token-56*, the average value to be computed over the image bound to *rf-1*, and the result stored in the slot, *average*, for *token-56*.

Relational, or N-ary operators involve more than one token at a time. There are at least four ways to compute relational properties. The first way involves simply comparing the values of unary attributes, e.g., finding the distance between two regions by computing the Euclidean distance between their centroids. The second way is to use only the region representations; e.g., to determine if one token is within the bounding rectangle of, or contained within, another token. The third way, is to use only label maps. The label map, in conjunction with another data structure known as the *TOKEN-FROM-LABEL-TABLE* discussed below, uniquely maps pixels into tokens. Therefore, it is possible to determine, for example, what tokens are adjacent to other tokens by examining adjacent pixels in the label map. A problem with using only label maps is that it is not suited to the problem of determining adjacency for a specific token. This leads us to the fourth method which uses region representations to index into label maps. So, to determine which tokens are adjacent to a given token, one simply loops around the boundary of the token in the label map accumulating a list of all unique labels, and hence tokens, found.

5.4 Segmentation

Segmentation is the process of converting iconic descriptions (images) into symbolic descriptions (lists of tokens). Moving "bottom-up", i.e., from pixels to symbols, Omega contains basic functions for edge extraction, region growing, hierarchical region splitting, connected pixel labeling, and others. Each of these functions provides two basic outputs: a label map and a list of instantiated tokens.

Two other data structures not yet discussed are used extensively by segmentation functions. The *TOKEN-FROM-LABEL-TABLE*, which was mentioned earlier, is used to spatially access tokens via the label map as shown in Fig. 6. The TOKEN-FROM-LABEL-TABLE is a hash table which associates a token with each non-zero value in the label map. Another data structure, the *DISPLAY-TABLE* is used by functions and operators to regenerate the spatial extent of tokens from their symbolic description. For each symbolic attribute, the DISPLAY-TABLE associates symbolic-values, bitmap-values, and display-values. Fig. 7 shows how tokens can be regenerated from their region representation using the DISPLAY-TABLE.

5.5 Symbolic Processing

A discussion of symbolic inferencing mechanisms is application-dependent and beyond the scope of the present paper. However, in the applications that are currently under development, an important capability is the ability to query a symbolic database. The query function,

(*? tokens constraint*)

returns those tokens from the list *tokens* (i.e., the symbolic database) which satisfy *constraint*. The form of the constraint expression depends on the type and arity of the constraint. Unary constraints involve only one token at a time; e.g.,

(*? a-tokens :area '> 10*)

returns all tokens from the list *a-tokens* whose area is greater than 10. Binary constraints involve two tokens at a time; e.g.,

(*? a-tokens :distance-between-any '< 2 b-tokens*)

returns all tokens from the list *a-tokens* that are less than 2 from any token in *b-tokens*. An example of a constraint with a higher arity is

(*? a-tokens :contains b-tokens c-tokens d-tokens*)

which returns all tokens from the list *a-tokens* that contain at least one token from each of the three lists: *b-tokens, c-tokens,* and *d-tokens*.

Rule-based inference systems can easily be built using the query function to select those tokens in a domain that satisfy the constraint(s) in the "if-part" of a rule. Another function not discussed here known as *assert* is then used to update those tokens as specified in the "then-part" of the rule. Further discussion and application of the above ideas are topics of a future paper.

6. SUMMARY

A Common Lisp software environment for integrated pixel-symbol processing has been described. The system is currently being used to support the development of applications such as object/change detection, multi-spectral image analysis, and knowledge-based geographic information retrieval.Future plans are to migrate Omega to other systems such as the TI Explorer and SUN Symbolic Processing Environment. We are also considering extensions to parallel hardware such as Thinking Machine's Connection Machine.

REFERENCES

1. L. Quam, IMAGECALC: Reference Manual, Stanford Research Institute, Menlo Park, CA, 1984.

2. C. McConnell, P. Nelson, and D. Lawton, "Constructs for Cooperative Image Understanding Environments," Proc. DARPA Image Understanding Workshop, 1986.

3. E. Garland and R. Ehrich, A GIPSY Primer, Spatial Data Analysis Laboratory, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1985.

4. G. Steele, Common Lisp, Digital Press, 1984.

5. Symbolics Programming and Color System Manuals (Genera 7), Symbolics Inc., Cambridge, MA, 1986.

6. M. Stefik and D. Bobrow, "Object-Oriented Programming: Themes and Variations," AI Magazine, Vol. 6, No. 4, 1986.

7. D. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel, "CommonLoops: Merging Common Lisp and Object-Oriented Programming," ISL-85-8, Xerox Palo Alto Research Center, 1985.

8. D. Bobrow, D. Moon, et al ,"Common Lisp Object Specification," ANSI X3J13 Document 87-002, American National Standards Institute, Washington, DC, 1987.

9. R. Merrill, Representation of Contours and Regions for Efficient Computer Search," Comm. ACM, Vol. 16, No. 2, 1973.

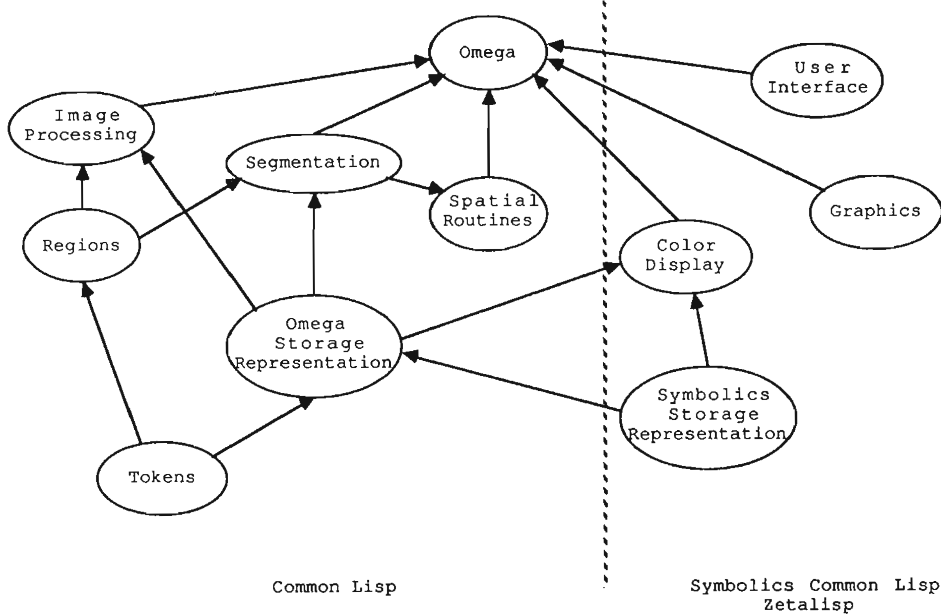
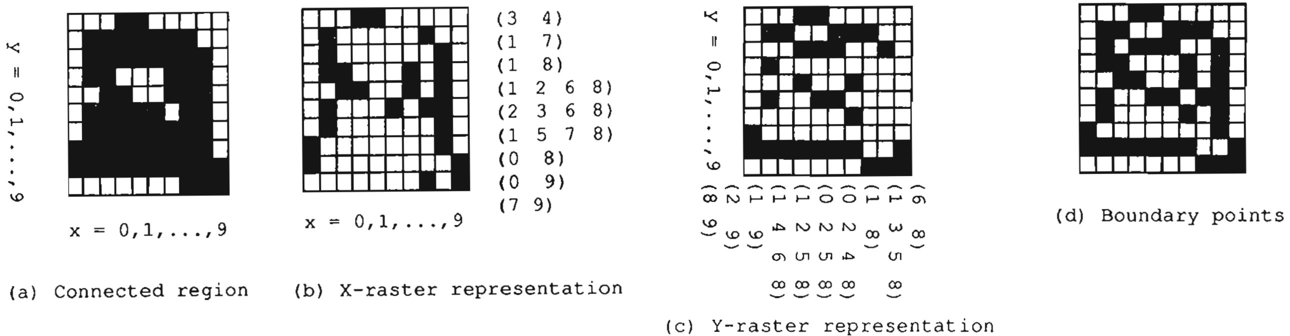


Fig.1 Hierarchical organization of Omega software into packages.



(a) Connected region (b) X-raster representation (c) Y-raster representation (d) Boundary points

Fig.2 Spatial representations based on run-length encoded rasters

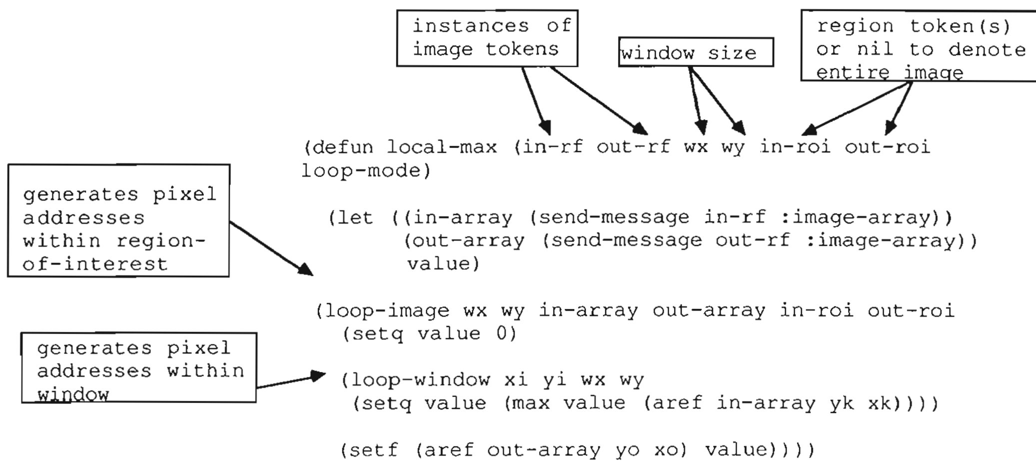


Fig.3 Example of region-of-interest-directed image processing function implemented with a looping macro

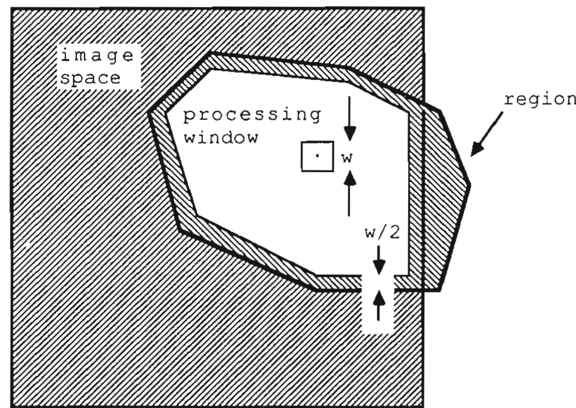


Fig.4 Effects of image array, region, and processing window on the effective region-of-interest for an image processing function or spatial operator

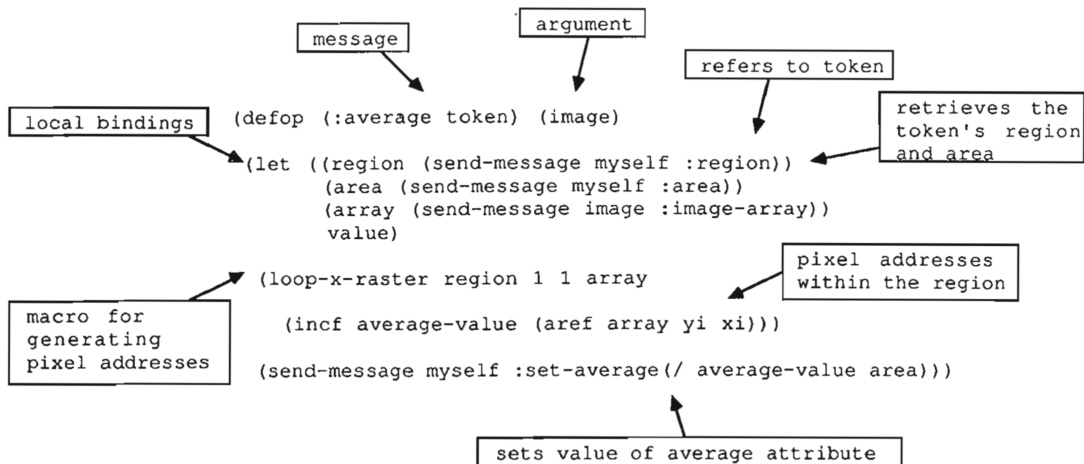


Fig.5 Example of spatial operator implemented with a looping macro

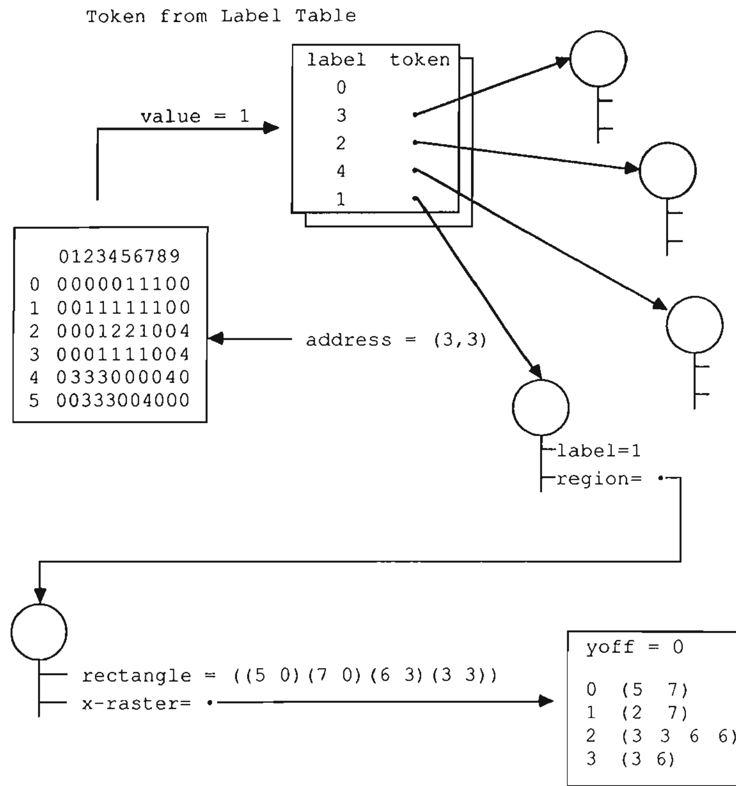


Fig.6 Spatially accessing a symbolic description via the TOKEN-FROM-LABEL-TABLE

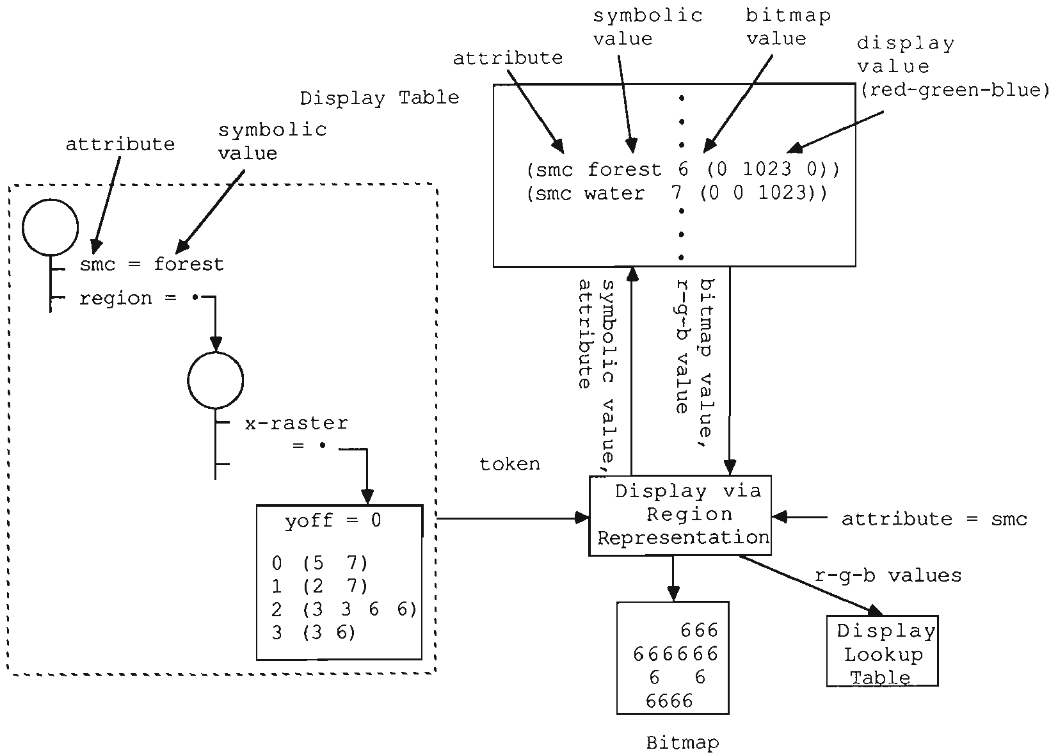


Fig.7 Use of the DISPLAY-TABLE in regenerating the spatial extent of a token from its region representation