MVA'94 IAPR Workshop on Machine Vision Applications Dec. 13-15, 1994, Kawasaki

ANALYSIS AND REPRESENTATION OF COMPUTER VISION SYSTEMS BY THE OBJECT-PROCESS METHODOLOGY

Dov Dori
Department of Information Systems Engineering
Faculty of Industrial Engineering
Technion, Israel Institute of Technology
Haifa 32000, Israel

ABSTRACT

Computer vision involves a host of problems, algorithms and techniques dealing with all aspects of capturing scenes and converting them into meaningful interpretations. A computer vision system may include a pattern recognition subsystem and be itself embedded within a more complex system. Machine vision systems feature a combination of complexity on one hand and a balance between structure and behavior on the other hand. Analysis and design of computer vision systems calls therefore for a methodology that represents equally well structure and behavior within a unified frame of reference and has adequate tools for complexity management. This paper discusses the object-process analysis (OPA) as an approach to tackle this task. Following the introduction of the basic OPA principles, we use the Image Understanding Environment (IUE) Project documentation to demonstrate the principles, use and the benefits of the methodology. The result is a series of consistent, inter-related object-process diagrams that gradually expose the details of the system. Complexity is managed through visibility control, which is obtained by a host of options for scaling object process diagrams. The ease of application of object-process analysis to the case in point suggests that it can be successfully applied to analyze, understand and communicate complex software systems such as the IUE Project .

INTRODUCTION

The domain of machine vision involves a host of problems, algorithms and techniques dealing with all aspects of capture and conversion of scenes into meaningful interpretations. Frequently, pattern recognition subsystems are embedded within computer vision systems. The latter, in turn, may be subsystems in yet more complex systems, such as industrial inspection, autonomous navigation and robotics. Machine vision systems feature two main characteristics. The first is system complexity, and the second is the balance between structure and behavior.

The structure/behavior balance characteristics is due to the fact that machine vision systems have two main aspects: structural (static) and procedural (dynamic). The structural aspect pertains to the objects present in the systems and the long-term relations among them, such as aggregation-particularization (whole-part), generalization-specialization (gen-spec), and characterization. The procedural aspect relates to the time-varying behavior of the system, which is usually guided by algorithms, procedures or routines. Even though most of the behavior of a machine vision system can be described in terms of algorithms, the traditional tools for algorithm description, such as flow diagrams or structured pseudo-code, lack proper representation of the objects involved. Since most machine vision systems feature substantial structural and behavioral aspects, understanding them requires a representation that strikes a balance between structure and behavior within a unified representation scheme. Objects and processes should have adequate weight, and the procedural relations that link objects to processes should be explicitly stated.

The complexity characteristic of computer vision systems implies that they have a large number of objects and processes. Both the objects and the processes are at various abstraction levels. At the lowest level, objects are as simple as pixels, variables, parameters, etc., while processes can be a pixel neighborhood averaging or convolving an image. Intermediate processes are of the nature of vectorization, segmentation, skeletonization and edge detection. Accordingly, objects at this level are regions, edgels, arcs, skeletons and other recognized segments leading to high-level understanding. At the highest level, the objects are stationary or moving 3-D structures, decision about a robot's trajectory change for obstacle-avoidance, acceptance or rejection of an inspected product, etc. Frequently, objects from different abstraction levels interact because of the inherent non-linear nature of the pattern recognition process. For example, many computer vision systems have feedback cycles that enable hypotheses from higher levels to be tested at lower levels.

The object-process analysis (OPA) can be used to fully understand computer vision systems, explain them, and communicate their analysis results. The OPA paradigm extends object-oriented analysis to meet the challenge of handling complexity and striking a balance between a system structure and behavior. OPA combines the representation of the structure and behavior of machine vision systems within an integrated framework and enables complexity management by scaling within this framework.

As a case study, a portion of the Image Understanding Environment (IUE) Project that deals with the design and implementation of image objects and their related methods is analyzed and presented using OPA.

SYSTEM ANALYSIS PARADIGMS

First attempts in system modeling concentrated on the dynamics of the system. A notable example of these early approaches is the data flow diagram (DFD) method [De Marco, 1978], emphasizing processes as the major theme of the analysis. Only later approaches, e.g., entity-relationship diagrams [Chen, 1976] and object-orientation (OO) [Coad and Yourdon, 1991; Embley, 1992; Shlaer, 1992; Rumbaugh, 1991; Nerson, 1993] put objects at the center of the analysis.

The object-oriented methodology is the currently accepted paradigm for system analysis, design and programming. It is built on the premise that every thing in any domain can be represented as an object. Processes (referred to as "methods" or "services") are encapsulated within objects and are activated via messages passed among objects. This "objectification" of the universe is adequate for describing the structure of a system, but it lacks appropriate tools to explicitly model the dynamic, procedural aspect of systems. To model the system's behavior, OO methodologies use flow charts [Coad, 1991], DFD or some variation thereof, such as Action DFD [Shlaer, 1992], state charts [Rumbaugh, 1991], or a combination of these methods. These methods are generally not directly related to the object model. Consequently, the analyst is forced to separate structure from behavior in the analysis process. The results of the analysis are also communicated separately using different diagramming methods and symbol sets. This is unnatural, because in reality objects and processes go hand in hand and are inseparable: Objects are created, changed and destructed by processes, and processes have no meaning without the objects upon which they act.

The integration between structure and behavior within the system is of particular significance in computer vision systems, because almost by definition all machine vision systems deal with objects—scenes, images, transforms and interpretations, and the processes that convert one type of object to the other—imaging, scanning, image processing, recognition, etc.

OBJECT-PROCESS ANALYSIS

Object-process analysis (OPA) [Dori et al., 1994; Dori, 1995] is a superset of OOA that has been designed to respond to the two main challenges of systems analysis discussed above: (1) integration between structure and behavior, and (2) provision of tools for complexity management. To meet the first challenge, OPA is based on the description of how objects interact with each other through processes. This provides the basis for the structure/behavior integration. The second challenge is met by scaling, which enables controlling the visibility and level of detail of objects and processes. Scaling is a complexity management tool, that works in both directions. Up-scaling (zooming-in) and down scaling (zooming-out) increases and decreases the level of detail, respectively.

The universe consists of a collection of *things*. Things are objects and processes, and may be simple or compound. Simple (atomic) things cannot be decomposed into things nor are they characterized by other things. Compound things consist of other things and/or specialize into other things and/or are characterized by other things. The *universe of interest* is a subset of the universe that is relevant to the system under consideration. Its modeling is guided by two complimentary aspects: structural and procedural. The structural aspect explains what are the objects in the universe of interest that play a meaningful role in the system under consideration, and how they relate to each other structurally, i.e. in the long run. The structural aspect pertains to such questions as which is the whole and what are its parts, which object is the general and what are its specializations, etc. The procedural aspect explains how the objects in the universe of interest behave and how they interact with each other through processes. It provides a time-varying image of how processes affect object states, including their construction and destruction.

An *object* is a persistent thing which is constructed (generated), changed and destructed (eliminated) by one or more processes. A *process* is a transient thing which requires one or more objects to enable its occurrence and affects at least one object (possibly an enabling object). A *feature* is a thing that characterizes a higher-level thing. A feature is related to the class which it characterizes through a *characterization* relation.

A *class* is the set of all the things that are characterized by the same set of features. An *object (process) class* is the set of all the objects (processes) that are characterized by the same set of features. An object (process) is a typical member of the object (process) class to which it belongs. An *instance* is a particular member of a class. Features are attributes and services. An *attribute (service)* is an object (process) class that characterizes a higher level class. Being an object class, an attribute has a number of instances (legal values). At any given point in time, the *state* of an object is the set of attribute values that are valid at that time.

A major benefit of OPA is its ability to handle complexity through visibility control. Controlling visibility of analysis and design details is done via a host of scaling options. Each compound thing, which is called *seed* in the context of scaling, can be scaled up to yield an embedded object-process diagram, called *plant*, in which the inner structure and behavior of the seed is exposed. Conversely, a collection of things—the plant—can be scaled down to a seed, thereby obtaining a higher-level, more compact view of the system. Hence, the instances of the attribute *direction* of scaling are "up" and "down." The second attribute of scaling, *seed preservation*. has three instances: (1) no seed preservation, with explosion for up-scaling and implosion for down-scaling, (2) background seed preservation, with blow-up and shrinking, and (3) root seed preservation, with unfolding and folding.

THE IMAGE UNDERSTANDING ENVIRONMENT

The Image Understanding Environment (IUE) project [IUE, 1994] is a five year program, sponsored by the US Defense Advanced Research Project Agency (DARPA), to develop a common software environment for the development and algorithms and application systems for image understanding. The primary purpose of the IUE is to facilitate exchange of research results within the Image Understanding community by providing a standard interface for the development and sharing of code. It is designed to support evolution of IU approaches and an effective programming environment for rapid proptotyping. As the IUE Overview Document notes [IUE, 1994], "the central approach to the design of the IUE is the use of object-oriented design principles."

To demonstrate how OPA can be applied to facilitate the analysis and design of IUE, we selected images which are "among the most fundamental objects in image understanding, since they are the ultimate source from which all other iconic and symbolic representations are computed" [IUE, 1994]. Chapter 4 of the IUE Class Definition [IUE, 1994a] provides a detailed description of the image classes in IUE. To satisfy a number of design goals, IUE distinguishes four types of classes: generic classes, data classes, interface classes and implementation classes. All but the implementation classes are abstract classes designed to inherit attributes to the implementation classes. Figure 1 is the Object-Modeling Technique (OMT) [Rumbaugh, 1991] image class hierarchy diagram, provided in the IUE documentation [IUE, 1994a]. The diagram contains 32 classes divided into four types with all the relations among them. The complexity of this OMT diagram makes it difficult to follow and understand. In an attempt to clarify it, lacking formal tools for complexity management, dashed lines separate the 32 classes into four types, and the name of each class type is written within the area containing the corresponding classes.

THE IMAGE CLASS HIERARCHY

Figure 2 is a top-level object-process diagram (OPD) describing the four class types and the structural relations among them. Each class type is itself an object class—a seed—which contains the classes belonging to the corresponding type. This high-level view enables easy tracking of the relations among class types. As Figure 2 shows, the only structural relation in this OPD happens to be the generalization-specialization (gen-spec, or "is-a") relation, which is denoted by a blank triangle whose apex and base are connected to the generalized and specialized classes, respectively. This corresponds with OMT [Rumbaugh, 1991] and OOSE [Embley, 1992]. The meaning of direct and indirect gen-spec relations is explained below.

The gen-spec relation gives rise to *inheritance*, which is one of the cornerstones of object-oriented analysis and design. Inheritance means that any feature (attribute or service) that characterjzes the general class also characterizes all of the specialized classes. The features are inherited from the ancestor (general) class to the descendents (specialized classes). Each specialized class may have its own features in addition to the ones it inherits from the general, ancestor class. Since inheritance spans across any number of generations, the result of the overall structure is a directed acyclic graph, whose nodes and edges are the classes and the gen-spec relations, respectively. To remain faithful to the description in Figure 1, we must distinguish between two types of gen-spec relations, direct and indirect, as shown in the legend of Figure 2. The direct gen-spec relation is the common one and is used when the specialized class acquires directly all the features of the the general class. The indirect gen-spec relation is used when only a proper subset of the classes which are specializations or parts of the specialized class acquire all the features of the the general class. Applied to the image classes in IUE, the OPD in Figure 2 shows, the source of all the image classes is **IUE-object** and the destination is **implementation-class. IUE-object** inherits its features directly to **generic-class** and to **data-class. Interface-class** inherits directly from **generic-class** and indirectly from **data-class.** Finally, **implementation-class** inherits indirectly from both **data-class** and **interface-class.** These fundamental relations among the classes are very important to note. However, deducing them from the OMT class hierarchy of Figure 1 is not straightforward because of the multitude of connections among the 32 classes in the diagram. In the top-level OPD of Figure 2, which has only five classes, those relations are explicitly expressed and easily tracked.

Next, we wish to refine the top-level OPD. To this end, we scale it up. As noted, up-scaling can be done in three ways: blow-up, unfolding and explosion. We start by applying blow-up. However, blowing up all the four high-level classes at once would result in an OPD which is still too complex, because it would include all 32 classes of Figure 1. Therefore, we blow up only **generic-class**. The resulting OPD is shown in Figure 3. The class that was blown up is shown in a gray thick blow-up frame, with its name in a corner. Inside the

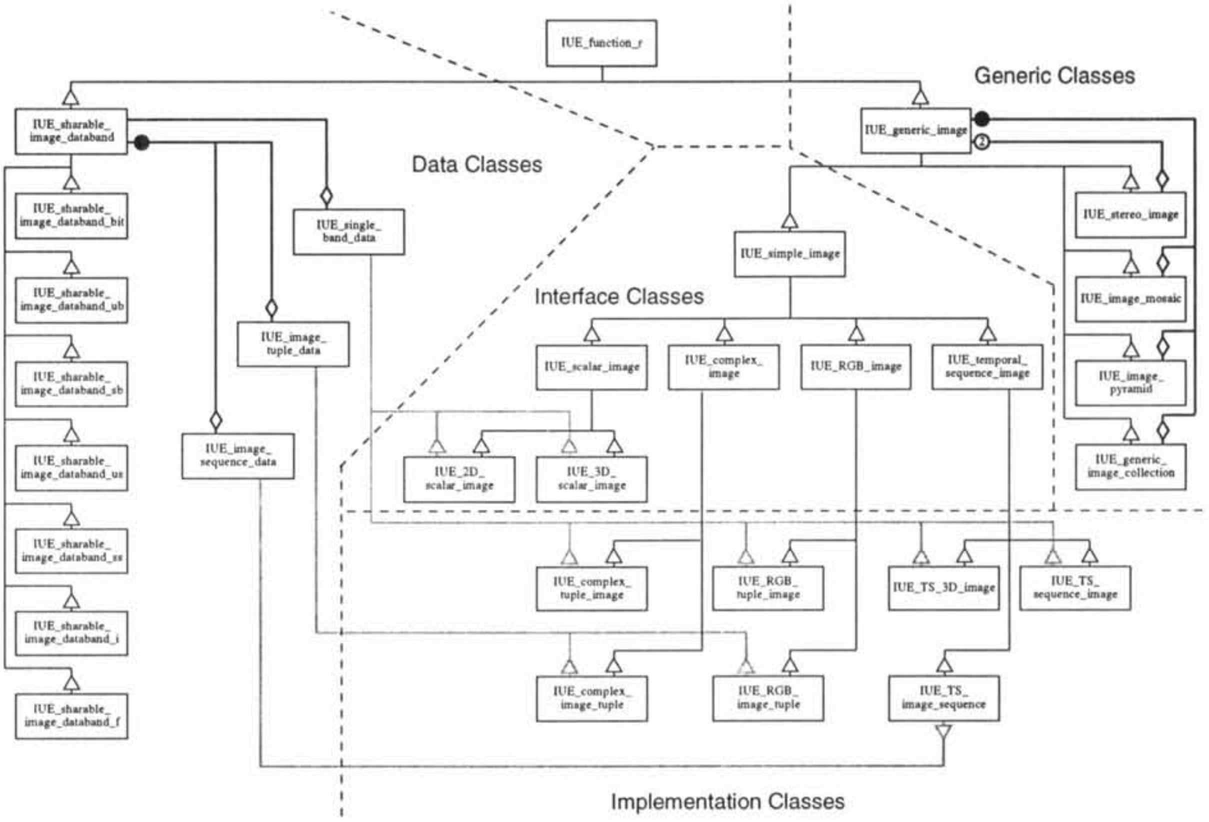

Figure 1. The Object-Modeling Technique (OMT) IUE image class hierarchy [IUE, 1994a]

blow-up frame are the lower-level classes belonging to the class that was scaled up. To verify that the up-scaling operation was done correctly, note that if we apply the inverse operation of blow-up, called shrinking, we get back the OPD of Figure 2.

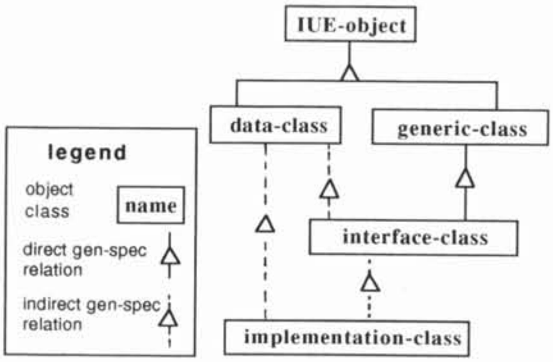

Figure 2. A top-level OPD of the IUE image class hierarchy

Figure 3 is in accord with the "Generic Classes" portion of the OMT diagram of Figure 1. **generic-image** specializes into four types: **stereo-image**, **mosaic-image**, **pyramid-image** and **generic-image-collection**.

As in OOSE [Embley et al., 1992], the black triangle denotes the aggregation-particularization (whole-part) structural relation. Each one of these four image types consists of a number of **generic-images**. The number is denoted by the participation constraint written along the aggregation link. Thus, the number 2 means that **stereo-image** consists of exactly two **generic-images**, while 1..m implies a relation of one-to-many between either one of **mosaic-image**, **pyramid-image** and **generic-image-collection** and **generic-image**.

Having understood the **generic-imgae** specializations, we can now shrink back the blow-up frame of **generic-imgae** and blow up other classes. This time we blow up **data-class** and **interface-class**. The result is shown in Figure 4, where the seven **sharable-image-data-band** specializations which appear stacked on the left hand side of Figure 1 have been omitted to enhance readability. Since the parts of **data-class** and the specializations of **interface-class** are now specified, we can specify exactly the direct gen-spec relations between the lower-level classes such that the indirect gen-spec link between **data-class** and **interface-class** (denoted by the dashed line dashed) is no longer needed, and is replaced by the solid line denoting the direct gen-spec link between **single-band-data** and **scalar-image**. In Figure 1, **single-band-data** is linked separately to both **2D-scalar-image** and **3D-scalar-image**, but the latter two are specializations of **scalar-image**, it is more correct from a software engineering viewpoint and more economic in graphic symbols to establish the gen-spec link at the highest level possible.

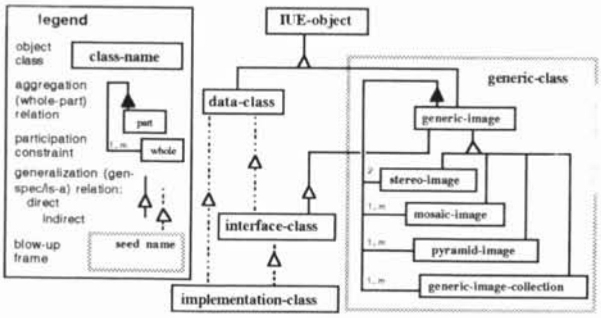

Figure 3. An OPD resulting from blowing up **generic-class** *in the top-level OPD of Figure 2*

As noted, blow-up is one of the three possible up-scaling operations, the other two being unfolding and explosion. The difference between blow-up and unfolding is that while in blow-up the seed (the class that is scaled up) remains in the background of the plant (the scaled-up OPD) with the blow-up frame enclosing all the lower level classes, in unfolding, the seed remains as the root of the aggregation hierarchy. Explosion is similar to blow-up, except that rather than having the blow-up frame with the blown-up class name in the plant, neither the frame nor the name appear in the plant. Although the difference among the different types of up-scaling is graphical rather than semantic, applying various types of up-scaling to the same system may provide important insight into the structural relations which can lead to more economic and concise modeling, as we show below. Figure 5 shows unfolding of the top-level OPD of Figure 2. It is a combination of figures 3 and 4, where instead of blow-up we applied unfolding.

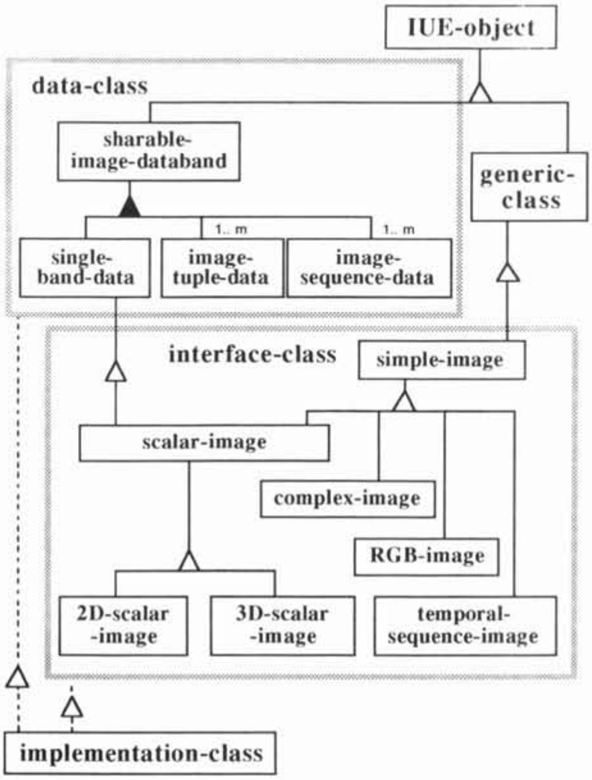

Figure 4. An OPD resulting from blowing up **data-class** *and* **interface-class** *in the top-level OPD of Figure 2*

Comparing Figure 3 and 4 to Figure 5, we see that instead of the seed appearing within the blow-up frame in the plant, each one of the three classes that were unfolded now appears in the plant as the root of the aggregation tree. Note that the gen-spec links, which connected the blow-up frames of data-class and interface-class, now originate from the corresponding classes themselves, such that figures 3 and 4 are semantically identical to Figure 5. Examining the aggregation trees of Figure 5,

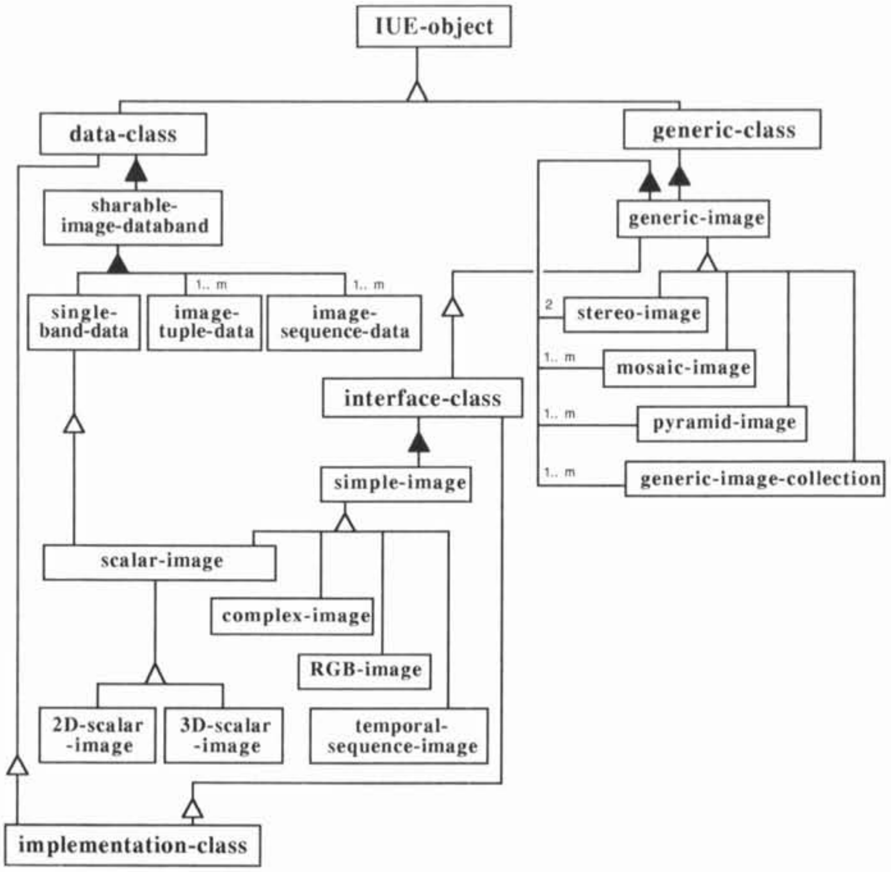

Figure 5. An OPD resulting from unfolding **generic-class**, **data-class** *and* **interface-class**

we see that the root of each tree has just one outgoing edge with the whole-part relation: **generic-class** has just **generic-image** as its part, **data-class** has just **sharable-image-databand**, and **interface-class** has just **simple-image** as its part. This suggests that the insertion of **generic-class**, **data-class** and **interface-class** may be redundant in the first place. Looking back at Figure 1, we see that this is indeed the case with these three classes but not with **implementation-class**. Hence, we redesign the top-level image class hierarchy of Figure 2 as shown in the OPD of Figure 6. Finally, we blow-up **implementation-class** and get the OPD of Figure 7, where the complete gen-spec lattice, as specified in Figure 1, is shown.

THE GENERIC-IMAGE CLASS

We continue our analysis by looking into the details of the object **generic-image**, which is the root of the generic class hierarchy. Figure 8 is a detailed portion of the class hierarchy depicted in the blow-up frame of **generic-class** in Figure 3. Each class appears in a box divided into one, two or three compartments. If only one compartment exists, then it contains just the class name, as in **generic-image-collection**. The thick line under the **generic-image-collection** class denotes the fact that it has neither specific attributes nor specific methods. If there are two compartments, then the first contains the class name and the second—the class attributes, as in **mosaic-image**, which has the attribute *image-set* : **set-of-generic-images**. If a third compartment exists, it holds the list of the class methods (services), as in **stereo-image**, which has three attributes and three methods.

We have adopted the same font conventions and C++like syntax used in [IUE, 1994a]: class names are in boldface roman or times (e.g., **generic-image**), attributes are in times italic font followed by a column followed by an object name (e.g. *stero-sensor:* **sensor-model**), method names are in sans-serif or helvetica font and preceded by a double semi column followed by the method input in parentheses, semicolon and the output (e.g., ::get-window(this: **generic-image**, location: **1d-array-of-int**, window: **value**): **value**) , and pointers are in typewriter or courier font followed by a semicolon (e.g., location:).

A typical method specification looks as follows:

::pixel-in-bounds (this: **simple-image**, location: **1d-array-of-int**): **boolean**

The interpretation of this specification is that ::pixel-in-bounds is a method of the object class **simple-image**, which takes as input **1d-array-of-int**, which is a one-dimensional array of integers, denoting the pixel location, and returns **boolean**, denoting whether or not the the pixel is within the image bounds.

Section 4.6 of the IUE Class Definitions [IUE, 1994a] is the image class definitions. Subsection 4.6.1, which holds 3.5 pages, describes the details of all the generic classes. The equivalent of these pages (excluding some documentation) is given in Figure 8, which provides more details about the structural relations among the classes than what is given in the IUE Class Definitions document. The latter specifies only superclasses and subclasses for each class, i.e., it refers to the gen-spec relations, but gives no detail about the whole-part relation, which is specified in Figure 8.

Using the above IUE conventions, all the generic image class hierarchy with all the attributes and methods are compactly and concisely recorded in their proper locations in Figure 8 to reflect superclasses, subclasses, whole-part relations with the participation constraints. The class **generic-image** has no attributes and eight methods, all of which are inherited to all the four

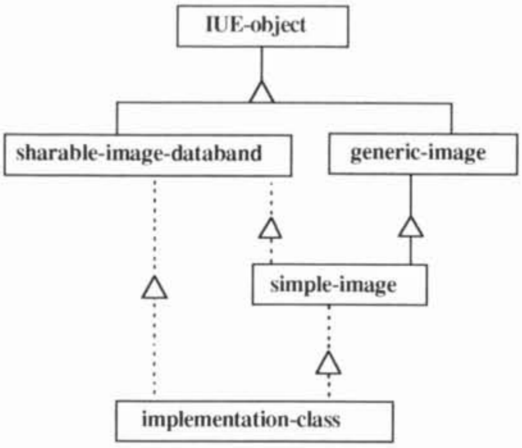

Figure 6. A revised top-level OPD of the IUE image class hierarchy

subclasses.

Attributes in IUE are classified into hard attributes and soft attributes. Hard attributes are assigned to the class as it is constructed, while soft attributes are computed results that are recorded to save unnecessary computations. **Stereo-image** has three hard attributes: *left-image* and *right-image,* each of which is a pointer to **generic-image**, and *Stereo-sensor*, which is a **sensor-model**. Since generic image is a generalization of **simple-image**, **stereo-image**, **pyramid-image**, **mosaic-image** and **generic-image-collection**, it is possible to construct a stereo image each of whose left and right images are themselves **pyramid-images,** for example. This is a very good generalization. However, there should be a constructor of **stereo-image** which will prevent the generation of a stereo-image with left and right images of different classes or with two stereo images. No such constructor is listed in the IUE Class Definition document.

CONSTRUCTORS AND METHODS

Figure 9 shows the generalization hierarchy of **generic-image**, **simple-image** and **scalar-image** and the methods associated with each class. The methods ::get/set pixel and ::get/set window of **simple-image** are designed to get and set the value of a single pixel or a window (a rectangular array of pixels), respectively. **1d-array-of-int** is a one-dimensional array of integers—a vector of integers—which is used to store and pass the location of a pixel or a window or the extents (size) of an image. Therefore **1d-array-of-int has** is of length 2 for 2D images and 3 for 3D images.

A **scalar-image** is an image known to consist of pixels which are scalars. Therefore, it is possible to devise for this class specialized methods that make pixel access more efficient. Hence, it contains methods like ::get-pixel-i and ::get-pixel-f, which are used when the image is known to consist of integer and float data types, respectively. The corresponding ::set-pixel methods do not require the suffix -i or -f, as their input parameter list already contains the appropriate type. This is also true for the two ::get-pixel methods that have **ref(int)** and **ref(float)** in their input parameter list.

simple-image and **scalar-image** are classified as interface classes. This means that, like **generic-image**, they are abstract classes, i.e., they are not instantiable—no instances can be created for these classes. The simplest implementation, instantiable class is **2d-scalar-image**. Therefore it is the first one which has constructor methods—methods used to construct new instances of the class. Figure 10 is an OPD (object-process diagram) that depicts the three possible methods by which an instance of **2d-scalar-image** can be constructed. Construction is a process, which is denoted by an ellipse. The rest are objects, which are denoted by boxes. Arrows incoming to and outgoing from a process are *effect links*, denoting the object(s) required for the process to occur and the object(s) resulting from the process occurrence. Solid and dashed lines connecting incoming effect links denote a logical **and** and a logical **or** connector, respectively. Thus, Figure 10 expresses the fact that the three constructors: (1) **2d-scalar-image**(file-name: **iue-string**), (2) **2d-scalar-image**(x: **int**, y: **int**, datatype: **generic-image::Pixel-Type**), and (3) **2d-scalar-image** (pIS: **const pointer** (**noniue** (**IMAGE-STRUCT**)) are alternative, and that while constructors (1) and (3) require one object input each, constructor (2) requires three object inputs: two objects, x and y of the class **int** for the image size and one object, datatype of the object **generic-image ::Pixel-Type**. If no logical connector connects incoming effect links, then the default is logical **and**: all input classes must participate in the process (which may be an abstract process,a method, a constructor, or an operator).

Figure 11 is an OPD of the complete set of the 39 methods of **2d-scalar-image**, enumerated in [IUE, 1994a]. The reason Figure 11 shows only 14 ovals is that many methods are represented through inheritance, thereby reducing the number of methods in the OPD by a factor of almost 3. The equivalent textual description in [IUE, 1994a] occupies almost four pages.

Since the OPD describes methods of **2d-scalar-image**, each method has one effect link incoming from the object this: **2d-scalar-image**, which appears at the top left corner of the OPD. Each methods has in addition at least one incoming effect link. For example, the method ::pixel-in-bounds accepts also position, which, as the legend specifies, is a generalization of coordinates and location: **1d-array-of-int**. coordinates is an aggregation of x: **int** and y: **int**. Neither position nor coordinates appear in the IUE Class Definitions. Therefore they are not preceded by a semicolon. Rather, they were added to take advantage of generalization and aggregation, respectively, for obtaining a high-level view of the set of methods and reducing the amount of required documentation. Thus, due to inheritance, the arrow outgoing from position into the ::pixel-in-bounds oval stands for the following two ::pixel-in-bounds methods, specified in [IUE, 1994a]:

(1) ::pixel-in-bounds (this: **2d-scalar-image**, location: **1d-array-of-int**): **boolean**, and

(2) ::pixel-in-bounds (this: **2d-scalar-image**, x: **int,** y: **int**): **boolean**.

The double-headed arrow is a reflective effect link: the result of the method is fed back into the object which was provided as an input. For example, as the OPD of Figure 11 specifies, the assignment operator, ::operator=, gets the objects this: **2d-scalar-image** and im: **2d-scalar-image**, and assigns this: **2d-scalar-image** to im: **2d-scalar-image**.

An effect link surrounded by square brackets is optional. Thus, the OPD specifies two ::get-copy-slice methods:

(1) ::get-copy-slice (this: **2d-scalar-image**,

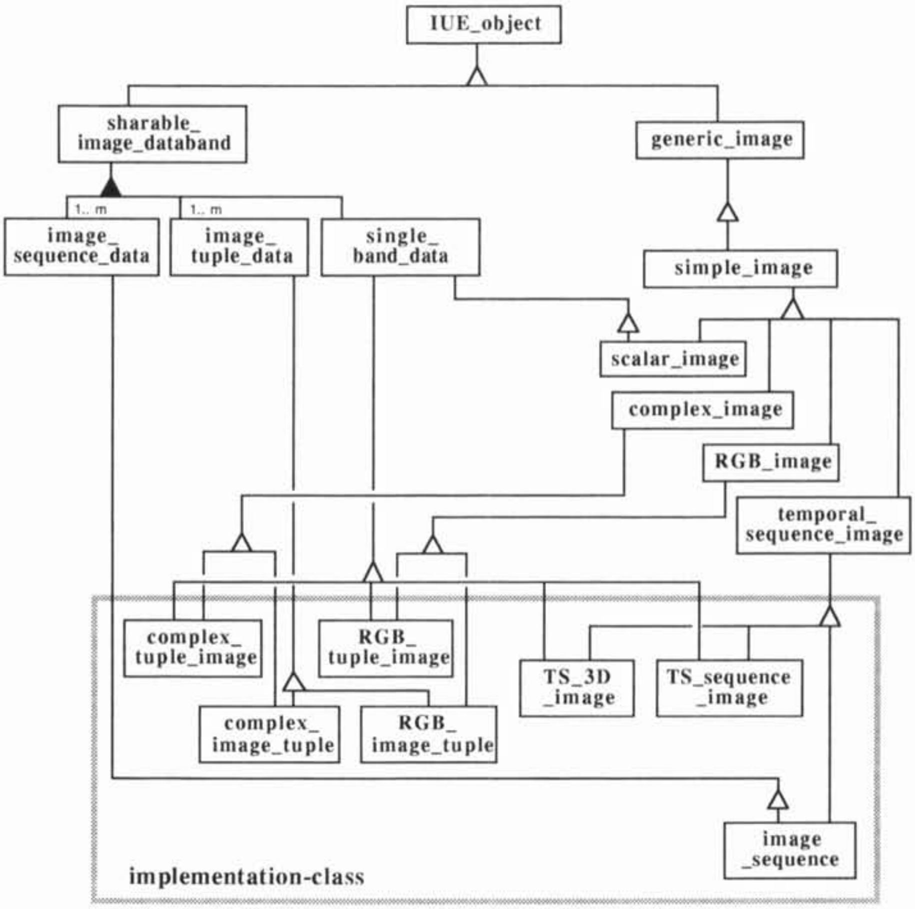

Figure 7. **Implementation-class** *blown up, showing the complete gen-spec-lattice for its constituent classes*

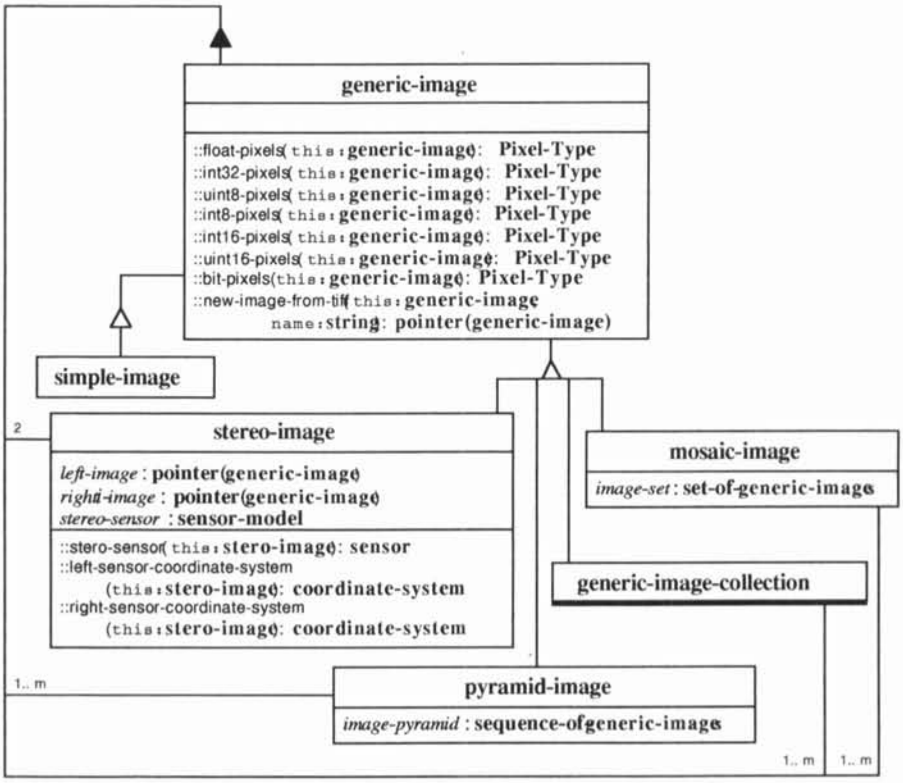
generic-image
::float-pixels(this : generic-image): Pixel-Type
::int32-pixels(this : generic-image): Pixel-Type
::uint8-pixels(this : generic-image): Pixel-Type
::int8-pixels(this : generic-image): Pixel-Type
::int16-pixels(this : generic-image): Pixel-Type
::uint16-pixels(this : generic-image): Pixel-Type
::bit-pixels(this : generic-image): Pixel-Type
::new-image-from-tiff(this : generic-image
name : string): pointer(generic-image)
simple-image
2
stereo-image
left-image : pointer(generic-image)
right-image : pointer(generic-image)
stereo-sensor : sensor-model
::stero-sensor(this : stero-image): sensor
::left-sensor-coordinate-system
(this : stero-image): coordinate-system
::right-sensor-coordinate-system
(this : stero-image): coordinate-system
mosaic-image
image-set : set-of-generic-images
generic-image-collection
1.. m
pyramid-image
image-pyramid : sequence-of generic-images
1.. m
1.. m

Figure 8. The attributes and methods of **generic-image** *and its specializations*

offx: **int**, offy: **int**, copy-image: **2d-scalar-image**): **2d-scalar-image**, and

(2) ::get-copy-slice (this: **2d-scalar-image**, offx: **int**, offy: **int**, xstep: **int**, ystep: **int**, copy-image: **2d-scalar-image**): **2d-scalar-image**.

As the documentation states, the first method fills data into copy-image, starting from [offx, offy], while the second does the same, stepping [xstep, ystep].

The continuation of Chapter 4 of the IUE Class Definition has the definitions of the classes **3d-scalar-image**, **complex-image**, and the other implementation classes, each with over 4 pages of documentation (the entire Class Definition Document [IUE, 1994a] is 338 pages). The vast majority of the methods for these classes is identical or almost identical to that of **2d-scalar-image**. For example, the two ::get-copy-slice methods for **3d-scalar-image** which are completely analogous to those of **2d-scalar-image** are:

(1) ::get-copy-slice (this: **3d-scalar-image**, offx: **int**, offy: **int**, offz: **int**, copy-image: **3d-scalar-image**): **2d-scalar-image**, and

(2) ::get-copy-slice (this: **3d-scalar-image**, offx: **int**, offy: **int**, offz: **int**, xstep: **int**, ystep: **int**, zstep: **int**, copy-image: **3d-scalar-image**): **3d-scalar-image**.

While it is apparent that these methods are mere extensions of the corresponding methods from 2D to 3D, the documentation [IUE, 1994a, p.70] erroneously states that the first method "fills data into copy-image, starting from [offx, offy]," while the second does the same, "stepping [stepx, stepy]". The offz and the stepz were left out, apparently because they were copied without alteration (including the wrong names stepx and stepy, instead of xstep and ystep), from the **2d-scalar-image** documentation. Errors of this type are unfortunately not rare in the document. Working with object-process diagrams and the abstractions of displacement and step, it is possible to avoid both the redundant documentation and the susceptibility to errors this redundancy is bound to cause. It is not clear why, at least conceptually, ::get-copy-slice is not defined at the level of **scalar-image**, and inherited with the proper dimension to **2d-scalar-image** and to **3d-scalar-image**.

CONCLUSION

Machine vision systems feature complexity and a substantial behavioral aspect beside the structural one. The work suggests that a holistic view of machine vision systems, which is sometimes set aside, should be adopted. An object-process analysis (OPA) approach, which is an extension of object-oriented analysis, has been suggested as a methodology for analysis, representation and communication of machine vision systems. The main benefit of OPA is presentation of the structural and behavioral aspects of a system within a single, coherent frame of reference at any level of detail without loss of consistency and links among the various abstraction levels. A small but fundamental subset of the documentation of the Image Understanding Environment (IUE) project was analyzed as a case in point to demonstrate how OPA can be used both to exhibit within one frame of reference the systems structure and behavior and to manage the inherent complexity of such systems. The analysis is presented in a top-down fashion, showing concurrently objects in the system, how they relate to each other structurally and how they interact with each other through processes.

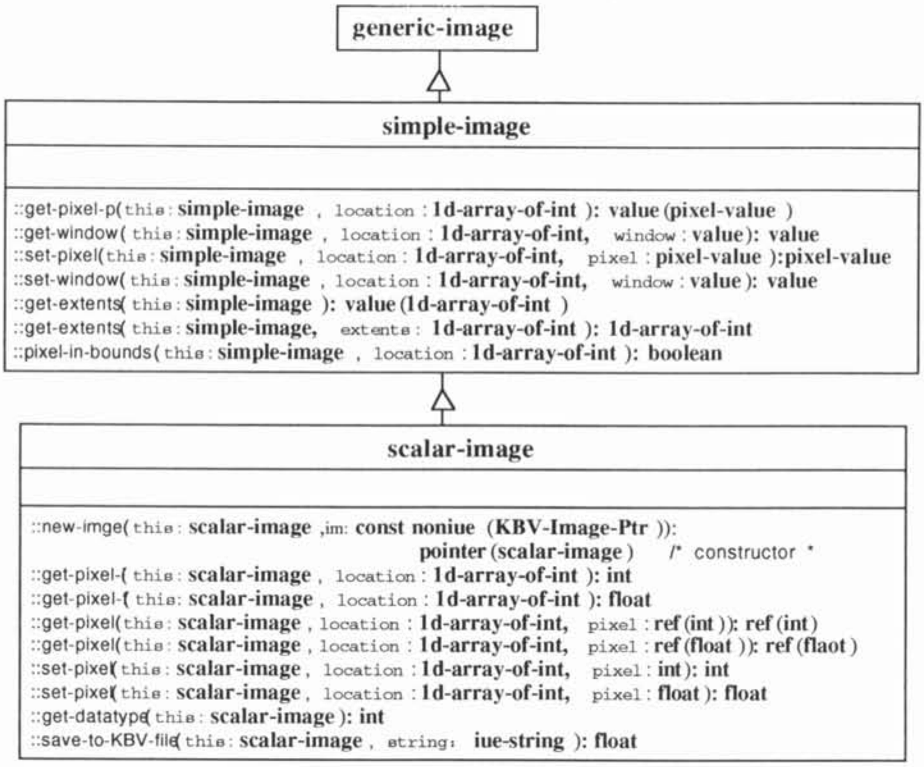
generic-image
simple-image
::get-pixel-p(this : simple-image , location : 1d-array-of-int): value (pixel-value)
::get-window(this : simple-image , location : 1d-array-of-int, window : value): value
::set-pixel(this : simple-image , location : 1d-array-of-int, pixel : pixel-value):pixel-value
::set-window(this : simple-image , location : 1d-array-of-int, window : value): value
::get-extents(this : simple-image): value (1d-array-of-int)
::get-extents(this : simple-image, extents : 1d-array-of-int): 1d-array-of-int
::pixel-in-bounds(this : simple-image , location : 1d-array-of-int): boolean
scalar-image
::new-imge(this : scalar-image ,im: const noniue (KBV-Image-Ptr)):
pointer (scalar-image) /* constructor *
::get-pixel-(this : scalar-image , location : 1d-array-of-int): int
::get-pixel-(this: scalar-image , location : 1d-array-of-int): float
::get-pixel(this : scalar-image , location : 1d-array-of-int, pixel : ref (int)): ref (int)
::get-pixel(this : scalar-image , location : 1d-array-of-int, pixel : ref (float)): ref (flaot)
::set-pixel(this : scalar-image , location : 1d-array-of-int, pixel : int): int
::set-pixel(this : scalar-image , location : 1d-array-of-int, pixel : float): float
::get-datatype(this : scalar-image): int
::save-to-KBV-file(this : scalar-image , string: iue-string): float

Figure 9. The hierarchy of **generic-image**, **simple-image** *and* **scalar-image** *and their associated methods*

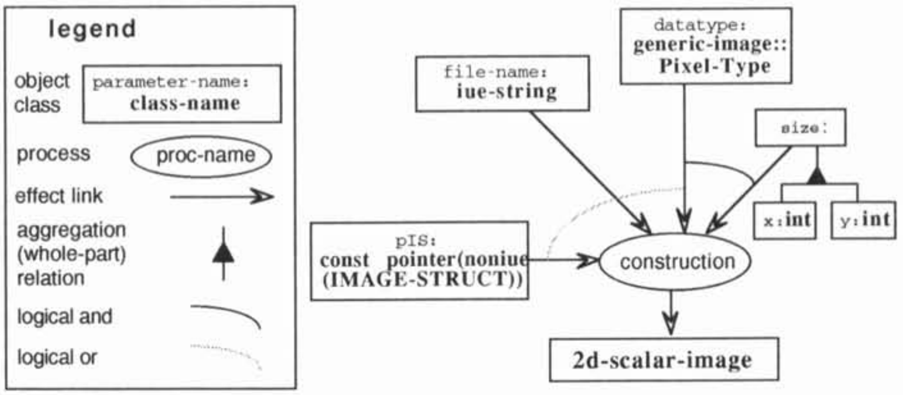
legend
object class
parameter-name: class-name
process
proc-name
effect link
aggregation (whole-part) relation
logical and
logical or
file-name: iue-string
datatype: generic-image:: Pixel-Type
size:
x:int
y:int
pIS: const pointer(noniue (IMAGE-STRUCT))
construction
2d-scalar-image

Figure 10. The three different constructors of **2d-scalar-image**

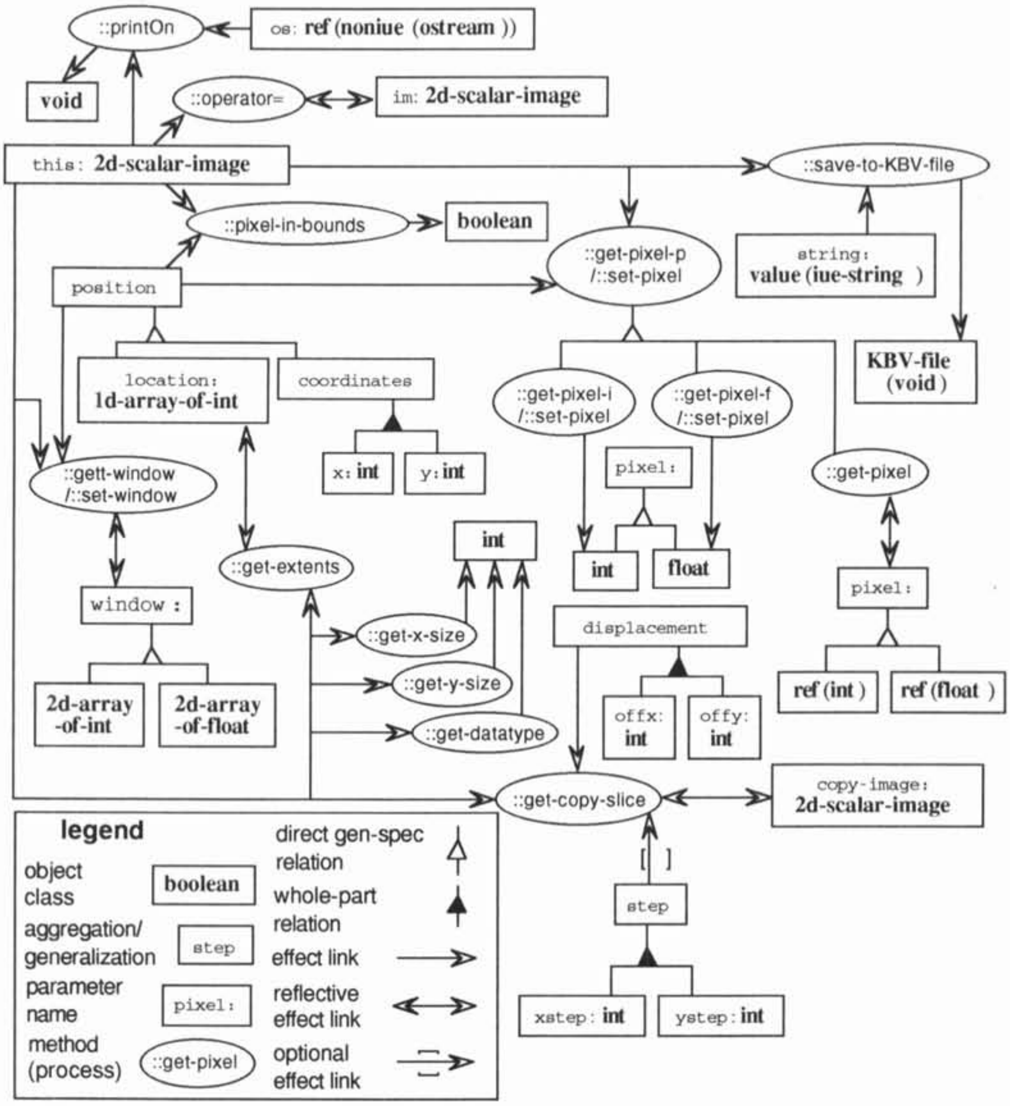

Figure 11. The complete set of methods of **2d-scalar-image**

The top-down presentation was made possible by the use of scaling. Scaling is a powerful tool for complexity management, as it provides for controlling the visibility and level of detail of objects and processes of interest. As we proceed down the hierarchy, we get from the high-level, abstract classes to actual implementation classes. At this low level, OPA extends the set of symbols to cover cases, such as reflective and optional effect links, to abstract and compact the documentation. The resulting documentation is a set of object-process diagrams that are about threefold more compact. The set of OPDs provide a comprehensive, graphic representation of portions of the system that are at the focus of interest, while keeping the reader oriented as to where in the system the focus is on. This way, the "large picture" is not lost in a myriad of minute details. The graphic representation potentially inspires ideas for further abstractions that can be implemented in code, such as position, coordinate and step in IUE.

Acknowledgement This research was supported by the Technion V.P.R. Fund.

REFERENCES

[Chen, 1976] P.P. Chen, The Entity Relationship Model: Toward a Unifying View of Data. *ACM Trans. on Data Base Systems*, Vol. 1 No. 1, pp. 9-36, 1976.

[Coad, 1991] P. Coad and E. Yourdon, *Object Oriented Analysis, (2nd Ed.)*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[De Marco, 1978] T. De Marco, *Structured Analysis and System Specification*, Yourdon Press, New York, 1978.

[Dori, 1995] D. Dori, Object-Process Analysis: Maintaining the Balance Between System Structure and Behaviour.*Journal of Logic and Computation*. 5, 2, April 1995 (to appear).

[Dori et al., 1994] D. Dori, I. Phillips and R.M. Haralick, Incorporating Documentation and Inspection into Computer Integrated Manufacturing: an Object-Process Approach. To appear in *Applications of Object-Oriented Technology in Manufacturing*, S. Adiga (Ed.), Chapman & Hall, London, 1994.

[Embley, 1992] D.W. Embley, B.D. Kurtz, and S.N. Woodfield, *Object Oriented Systems Analysis*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[IUE, 1994] J. Mundy, T. Binford, T. Boult, T. O'Donnel, M.C. Chiang, S. Fenster, A. Hanson, R. Beveridge, R. Haralick, V. Ramesh, C. Kohl, D. Lawton, D. Morgan, K. Price, T. Start, The Image Understanding Environment Program—IUE Overview. *FTP from ftp.aai.com (192.190.241.40) in directory /pub/iue/doc/manuals*. Version used: April 1994.

[IUE, 1994a] J. Mundy, T. Binford, T. Boult, T. O'Donnel, M.C. Chiang, S. Fenster, A. Hanson, R. Beveridge, R. Haralick, V. Ramesh, C. Kohl, D. Lawton, D. Morgan, K. Price, T. Start, The Image Understanding Environment Program—Class Definitions. *FTP from ftp.aai.com (192.190.241.40) in directory /pub/iue/doc/manuals*. Version used: April 1994.

[Nerson, 1993] J. M. Nerson, Applying Object Oriented Analysis and Design, *Communications of the ACM*, 35, 9, pp.63-74, 1993.

[Rumbaugh et al., 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Shlaer, 1992] S. Shlaer, and S.J. Mellor, *Object Lifecycles: Modeling the World in States*, Yourdon Press, PTR Prentice Hall, Englewood Cliffs, NJ, 1992.