CONCEPT AND IMPLEMENTATION OF AN OBJECT-ORIENTED FRAMEWORK FOR IMAGE PROCESSING*)

by Ingwer C. CARLSEN and Detlef HAAKS

* Philips GmbH Forschungslaboratorien, Forschungsabteilung Technische Systeme Hamburg, Vogt-Kölln-Straße 30, D2000 Hamburg 54, Germany

Abstract

This paper describes the concept of an Iconic Kernel System which provides a general software toolbox allowing image processing algorithms to be implemented independently of computer hardware and operating systems. Following a strict object-oriented design philosophy, it contains a basic set of data structures and operations from which the application-specific software environments can be built. Special attention has been paid to the incorporation of non-object-oriented software into object-oriented environments and to the integration of communication objects for the integration of distributed system components and user interface systems. The concept has been implemented in a DEC-VAX/VMS environment as a working prototype using several programming languages.

Keywords: adaptive information systems; image processing systems; mixed-language programming; object-oriented design; software portability.

1. Introduction

The Iconic Kernel System at the Philips Forschungslaboratorium Hamburg is a software environment which enables the user to implement image processing algorithms independently of specific computer hardware and operating systems (to distinguish this environment from a related standardization effort in the German standardization organization, it has been called the IKS-PFH by Haaks and Carl森1). It is a reaction to the current software crisis in image processing caused by the fact that so far software for image processing has been mainly designed and developed for the following:

*) This paper is an extended version of an invited paper on the same subject which has been accepted for publication in Computers and Graphics, 15 (4), 473–481 (1991).
As a consequence, this software can hardly be ported to other hardware configurations or operating systems, and it is difficult to adapt it to other applications or user demands. This lack of flexibility and reusability inevitably leads to expensive and error-prone multiple developments and implementations of similar software.

In the past this could be tolerated, and, to a certain extent, it was even justified. Limited performance and high costs of the image processing equipment forced the software designers to exploit system capabilities to their limits in order to achieve the performance needed for image processing. Input/output operations had to be closely coupled to the operating systems to reduce storage overheads and to speed up data transfer. Algorithms were programmed to make optimal use of special hardware architectures of processors. All this made the resulting software not only highly system dependent, but, even worse from today's point of view, the system-specific parts were often tightly interwoven with the remaining parts of the software. Changes in system-specific parts thus normally spread out over large (often unpredictable) parts of the software.

This was a considerable, but still manageable, effort. Compared with today's requirements, the complexity of these picture processing applications was rather low, and only a small number of people were involved in the development process. In most of the cases those few image processing experts who implemented the software were also its end users. This type of user had the knowledge necessary for quickly performing intricate changes in the system, and these users were quite tolerant to errors or imperfections in "their" systems. Generally accepted standard methods for image processing did not exist, i.e. there was no demand for distributing and reusing picture processing algorithms—every implementation more or less started from scratch again.

During recent years these conditions have changed considerably. The lifecycle of image processing hardware has become shorter. New equipment offering higher performance at constant or even declining costs has to be integrated into existing environments more often and makes the use of hardware-specific features for ultimate performance obsolete for many applications. The rapidly growing complexity of imaging applications demands more experts and effort in developing software. In parallel, an increasing number of image processing algorithms of general interest have emerged. Cost effectiveness thus demands that as much existing software as possible be reused.

—special target hardware and operating systems;
—special and rather limited application areas;
—users who are experts in image processing.

As a consequence, this software can hardly be ported to other hardware configurations or operating systems, and it is difficult to adapt it to other applications or user demands. This lack of flexibility and reusability inevitably leads to expensive and error-prone multiple developments and implementations of similar software.

In the past this could be tolerated, and, to a certain extent, it was even justified. Limited performance and high costs of the image processing equipment forced the software designers to exploit system capabilities to their limits in order to achieve the performance needed for image processing. Input/output operations had to be closely coupled to the operating systems to reduce storage overheads and to speed up data transfer. Algorithms were programmed to make optimal use of special hardware architectures of processors. All this made the resulting software not only highly system dependent, but, even worse from today's point of view, the system-specific parts were often tightly interwoven with the remaining parts of the software. Changes in system-specific parts thus normally spread out over large (often unpredictable) parts of the software.

This was a considerable, but still manageable, effort. Compared with today's requirements, the complexity of these picture processing applications was rather low, and only a small number of people were involved in the development process. In most of the cases those few image processing experts who implemented the software were also its end users. This type of user had the knowledge necessary for quickly performing intricate changes in the system, and these users were quite tolerant to errors or imperfections in "their" systems. Generally accepted standard methods for image processing did not exist, i.e. there was no demand for distributing and reusing picture processing algorithms—every implementation more or less started from scratch again.

During recent years these conditions have changed considerably. The lifecycle of image processing hardware has become shorter. New equipment offering higher performance at constant or even declining costs has to be integrated into existing environments more often and makes the use of hardware-specific features for ultimate performance obsolete for many applications. The rapidly growing complexity of imaging applications demands more experts and effort in developing software. In parallel, an increasing number of image processing algorithms of general interest have emerged. Cost effectiveness thus demands that as much existing software as possible be reused.
and adapted for new imaging applications (it has been estimated by Meyer\(^2\)) that less than 15% of new code serves an original purpose).

In a similar way the requirements for user interfaces of picture processing systems have changed drastically\(^3\). A growing number of users have their expertise in other application domains and regard an image processing system merely as a tool for accomplishing their task. For this new group of users, the quality of the user interfaces will be decisive for acceptance and successful use of imaging systems. Different user groups will differ in their needs, e.g. a specialist developing new algorithms will demand a larger functionality than a user performing routine investigations, and a novice user will demand more support and help than an experienced user. A well-accepted and successful system will have to cover these varying needs—users are no longer willing to adapt their behavior to the user interfaces provided, but are expecting the user interfaces to be adapted to their specific demands instead. Therefore, human factors, principles of software ergonomics\(^4,5,6\) for computer-supported work, and the corresponding design goals\(^7\) have to be taken into consideration from the very first design steps.

Several attempts have already been taken to cope with these changing requirements. Special data formats have been agreed upon to facilitate at least the exchange of images\(^8\). Libraries have been offered, e.g. by Tamura et al.\(^9\), providing subroutines of generally applicable image processing methods to be called from user programmes. Several extensions of conventional programming languages have been proposed, e.g. by Duff et al.\(^10\), to cover the specific demands of image processing applications. Recently, a very interesting operator model has been proposed by Gemmar et al.\(^11\) allowing for a unified description of general image processing operations. In addition, virtual machines for image processing have been proposed, providing abstract data types and generic operations for image processing and hiding system-specific details of displays, e.g. by Dreschler-Fischer et al.\(^12\). Sophisticated and flexible user interfaces for the analysis of image sequences have been demonstrated by Haarslev\(^13\).

However, these attempts address only part of the problem. What is still missing is a flexible, yet efficient, framework for image processing supporting the complete software life-cycle from design through implementation to final application. From this, the demand for a unified and broad basis for a general iconic kernel system emerged.

This issue was taken up in 1988 by the Joint Technical Committee 1 (JTC1)\(^14\) of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). In its Subcommittee SC24 “Computer Graphics and Computer Imaging”, a project for an “Image Pro-
cessing and Interchange Standard—IPI” was established with the goal of arriving at a generic imaging standard by 1994. The IPI standard will comprise mainstream imaging functionality and will complement other generic standards in text, graphics and related areas. The standard will initially consist of three parts:

— a Common Imaging Architecture derived from an abstract imaging model and providing the common basis for the subsequent parts;
— an Application Programming Interface (API), describing in detail data objects and associated image processing functionality, and providing the basic building blocks upon which applications can be built;
— an Image Interchange Format (IIF) defining syntax and semantics of image attributes, image data and ancillary application data for interchange purposes.

The contributions to these parts are currently based on contributions by the national standardization bodies of Germany (DIN), Great Britain (BSI), Japan (JISC) and the USA (ANSI) resulting from many years of national standardization efforts.

Standardization work in Germany started in 1986 when an ad hoc commission Bildverarbeitungskernsysteme was initiated by DIN. This commission produced a draft proposal for an Iconic Kernel System—IKS. Work was continued in the ITG-Fachgruppe Mustererkennung formulating a proposal for the IKS. This proposal together with an independently developed Format for the Transfer of Colored Raster Pictures—FTCRP formed the basis of the work of the DIN Arbeitskreis Imaging. This task group was inaugurated in 1989 to formulate the DIN contributions to the later ISO/IEC IPI standardization effort. The German contribution is based on abstract (generic) data types (compound data, general look-up tables, histograms and regions-of-interest, highly structured image attributes and descriptors, images of structured picture elements, compound images etc.), and a very general and flexible model for iconic operations.

Comparable work in ANSI started in 1988 when the X3H3.8 Imaging Applications Programmer Interface Task Group was established which in 1989 proposed a Programmer’s Imaging Kernel—PIK. After several revisions its latest version “Strawman V8” forms the second major contribution to the IPI standard. The ANSI approach is less general than the IKS in that it is based on up to 5-dimensional matrices of scalar picture elements, look-up tables, histograms, regions-of-interest, and an extensive list of fundamental iconic operations.
The third contribution to the IPI has been introduced by BSI in the form of a report of the Alvey Project MMI/27 proposing in June 1990 an Image Processing Algorithms Library—IPAL, an extension of the well-known NAG library by a considerable number of iconic operations and structured image descriptors.

In Japan standardization work in imaging started in April 1989 in the Information Technology Standards Commission of Japan of the Information Processing Society of Japan (IPSJ/ITSCJ). Standardization-like work had previously been done on SPIDER (a library of over 400 FORTRAN IV subroutines developed in 1980), SPIDER II (extending SPIDER by another 350 FORTRAN IV subroutines in 1986), and the Standard Format for Digital Images on Magnetic Tape in Japan (SIDBAMT76)—in this format also an image database of 1300 typical images is available from the Institute of Industrial Science of the University of Tokyo.

The IKS-PFH is a test implementation of certain concepts introduced by the Philips Forschungslaboratorium Hamburg into the German standardization contribution. It provides a set of basic operations and data structures plus the necessary tools to build the final system whose functionality meets the demands of the final application. IKS-PFH has been designed as an interactive environment providing different user groups with their specific functionality and allowing for the flexible adaptation to changing applications and user demands. Section 2 briefly summarizes the main requirements and constraints our object-oriented system design had to fulfil. Section 3 then describes the architecture of the various system components. Details of their mixed-language implementation will be given in Sec. 4. Section 5 then summarizes our experiences with the object-oriented, mixed language implementation of an IKS-PFH prototype.

2. Requirements

2.1. Hardware and operating system independence

The IKS-PFH has to be designed independently of hardware and system software aspects in order to be portable and adaptable to other environments. Logical interfaces to peripherals, operating systems, and underlying hardware have to be provided to shield the special properties of these components from the remaining IKS-PFH parts. These interfaces have to support the integration of new external subsystems without affecting existing components.

2.2. Extendibility and reusability

The dynamic development of image processing applications demands open
systems that are able to grow with changing or new requirements. New operations and data structures should be easily integrated (perhaps substituting older components) without affecting other system parts.

The IKS-PFH has to be designed as a minimal, but complete, system which provides the basic tools and components as building blocks for applicationspecific methods and data structures. In addition, it should provide a framework for the integration of software coded in some high-level programming language to use the vast amount of existing reliable and efficient software. These integrated routines should not sacrifice any performance and should be encapsulated to behave as original IKS-PFH components.

2.3. Human factors

The IKS-PFH has to be easily adapted to users of different degrees of experience, different work procedures and different preferences in interaction styles. In particular we address the following user groups:

(1) users who only use a fixed set of methods;
(2) users who adapt existing methods to changes in work procedures;
(3) users who develop and integrate new methods and data structures.

The user interfaces have to be transparent, consistent and predictive in their behavior, i.e. the user must always have control over the system, be informed on its status, and be able to predict its future behavior. The user should feel comfortable with the offered functionality—too limited a functionality is as frustrating as one that is too broad! The user must be able to choose between different interaction techniques and must be able to make minor modifications. Help and advice information have to be adapted to the level of the user's expertise.

2.4. Complete image processing environment

A complete image processing environment has to support image processing from its initial design to its final application. Image processing experts will use the whole potential of IKS-PFH to develop and test new components by

—coding algorithms in some high-level programming language and subsequently embedding them into IKS-PFH or,
—formulating new operations and data structures using the object-oriented implementation language of the IKS-PFH.

The novice or inexperienced end-user will eventually use a version of the IKS-PFH adapted to his specific needs by expert and/or experienced users.
A complete image processing environment has to support the distributed development of large image processing application packages, i.e. different parts of the system developed separately at different locations and by different teams before integration into the final system. In an object-oriented design each of these system parts will have its own set of communicating objects, an "object world". These can be interfaced by providing general communication objects translating messages exchanged between the various object worlds—only within these communication objects does information about the respective system component and the iconic core reside. In this way, changes in one system component do not affect other system parts, but can be shielded by the communication objects (fig. 1).
2.5. Interface to symbolic image processing

No definition of iconic and symbolic image processing has been generally accepted. Iconic processing is mostly associated with pixel-oriented manipulations, whereas symbolic processing operates on more abstract representations of images. A possible distinction may also be based on the application-specific knowledge involved in the sense that iconic image processing normally does not rely on application-specific knowledge, whereas symbolic image processing uses it to a large extent.

The IKS-PFH covers the iconic picture processing domain only. Nevertheless, it is very important with respect to an open system architecture to specify a clear interface to symbolic processing

— to define knowledge-based iconic operations which make use of symbolic information generated by symbolic image processing, as well as
— to provide symbolic data generated by iconic operations for further symbolic processing.

Symbolic information is stored in a variety of data structures, e.g. sets, lists and graphs, and the iconic–symbolic interface should provide the most significant of them as abstract data types.

3. Architecture

The requirement for the extendibility of a picture processing system demands the definition of compact modules and a clearly structured communication between them. In this way specific system parts can be altered without affecting other components. According to Booch\(^6\), Cox\(^7\) or Tazelaar\(^8\), an object-oriented architecture provides the necessary degree of modularity, safety and extendibility, and its inheritance mechanisms support the extendibility by allowing one to specialize and reuse existing classes of the IKS-PFH.

The first step in object-oriented design consists of mapping the picture processing application onto separate system components. The IKS-PFH is built around the four major object categories shown in fig. 2: the data, operator, communication and peripheral objects.

3.1. Data objects

Two categories of data objects are provided: basic data objects and image objects. Basic data objects are characters, strings, numbers, vectors, matrices,
sets, lists and records. These data types are either provided by the implementation language of the IKS-PFH or they are easily constructed from available data types.

The central idea behind image objects in IKS-PFH is the integration of raw image data and related information into one compact unit. With reference to the concepts of the ACR-NEMA Standard\(^9\) the inheritance hierarchy shown in fig. 3. is proposed.

The properties common to all image objects are collected in the most general class “Image Object” having the following attributes:
I.C. Carlsen and D. Haaks

Fig. 3. Image object hierarchy.

—a descriptor uniquely identifying an image (e.g. composed of an identifier, date and time of recording, recording characteristics etc.);
—a format description holding information about the dimensions, the resolution and the coding format of an image;
—access rights controlling the access to images in large image databases;
—a location description giving information on where the image currently resides in a distributed environment;
—an image history recording the operations performed on the original image (e.g. to support undo/redo operations).

This general image object serves as an abstract superclass\(^{20})\) to be used for subsequent specializations only, but never for actual instantiations. In order to define real-world image objects we have to add properties (slots) for storing image data (e.g. the raw data) and methods to access and manipulate them. For that, single image objects and compound image objects are provided.

3.1.1. Single image objects

Single-image objects contain exactly one image. The class “Single Image” is a specialization of the abstract superclass “Image Object” with respect to three possible representations of images:

—An iconic image object contains an image as a regularly arranged set of representations of the individual picture elements (pixels). Pixels are not limited to scalar data types, but can be data types of arbitrary complexity, e.g. to represent color images as RGB-triplets. Pixels may hold information
Object-oriented framework for image processing

Fig. 4. MR image set and associated methods.

about non-iconic aspects (e.g. region or segment labels) and thus represent a mixed form of iconic and symbolic information.

-A **symbolic image object** stores the image content on higher abstraction levels, e.g. in the form of augmented transition networks, semantic networks and region adjacency graphs etc. It is important for the generality of an IKS-PFH to provide a basic and extendable set of symbolic image objects in order to support later iconic/symbolic picture processing.

-A **graphic image object** stores the image content as a set of graphical primitives (e.g. lines, polygons and regions) to support graphic operations for viewing purposes and to allow for explicit reference to graphics standards (e.g. GKS, PHIGS).

### 3.1.2. Compound image objects

A **compound image object** contains an arbitrary number of images and other data objects which are related to each other in some application-specific sense. Each compound image stores references to its constituent single images, and provides methods which are only applicable to the compound image object as a whole (e.g. special reconstruction techniques). With respect to the relationship of the single images three types of compound image objects are distinguished:

- An **image set object** represents a set of images which are related to each other with regard to their contents. It is not necessary that all the single images are different from each other. An image set can contain images of any image type (i.e. varying dimensions and pixel formats).

  An example is a set of magnetic resonance (MR) images (fig. 4) obtained with different pulse sequences. The specific methods of this MR image set include methods to calculate images representing the spin density $\rho$ and the
relaxation times $T_1$ and $T_2$ from the original spin-echo (SE) and inversion-recovery (IR) images.

—An **image sequence object** is an image set object with an additional ordering relation, e.g. with respect to their temporal (movie), spatial (stack of slice images) or frequency (multi-spectral) relationship. Methods for the analysis of image sequences on the basis of their relations will be incorporated within this class, e.g. for motion detection, 3D reconstruction.

The example shows a computer tomography (CT) image sequence (fig. 5). This image sequence represents a spatial relationship among the individual images. The specific method provided in this example for CT image sequences is 3D-reconstruction technique.

—An **image record object** is a collection of an arbitrary number of images of any kind and other related data objects. The main objective of this class is to comprehend data objects related with regard to their content and specialized evaluation methods into one entity.

The image record example describes an image record containing the diagnostic data of one patient gathered by a medical information system (fig. 6). The diagnostic data include an MR image set, a CT image sequence, an ultrasound (US) image and an electrocardiogram (ECG). Algorithms for the multimodality analysis of these data can be incorporated as well (e.g. geometric registration, multi-sensorial segmentation).

The image objects contained in a compound image object can themselves be compound image objects again. This recursive nature allows an application programmer to specify any image structure required by his specific demands. Furthermore the images contained can be of any type, i.e. iconic, graphic or symbolic. So different aspects of a real-world image can be collected in a compound image object allowing for the definition of highly specific operations on these collections.
There are many ways to represent image structures using these constructs, e.g. a 2D RGB-color image can be represented as an image record of the individual color planes or as a single image whose pixels consist of RGB-triplets. This is one aspect of the important flexibility provided by the IKS-PFH. The application programmer decides which representation to choose.

3.1.3. Communication between IKS components

The embedded operator methods show the communication with the related abstract operator objects and the maintenance of history objects. The basic functionality of these operations is included in the corresponding matrix objects. Let us take the “Average” method as an example (fig. 7.), in which the grey value of every pixel in the output picture is determined as the average over the grey values of some neighboring pixels in the input image. Firstly, this method ensures that the corresponding operator object (“averageOperator”) is instantiated (our object names follow the general naming conventions of “Smalltalk” and “Objective C”). Secondly, it delegates the average task to its matrix object providing its region of interest and the neighborhood having been received as a parameter. The matrix object itself will call the external average algorithm with the appropriate parameters. Eventually the “averageOperator” object is activated to extend the image history (for further implementation details refer to Sec. 4.4).

3.2. Operator objects

Operator objects are designed to confine all information on generic (polymorph\(^2\)) image processing operations, which may be distributed over several image objects. Typically image processing methods applicable to several image types will be turned into operator objects, whereas basic functions specific to one image type should be encapsulated in the image object.
The distinction between operator and data objects does not impose a separation of methods from data. Both the data objects and the operator objects are objects composed of data and related methods. The distribution of the functionality between data and operator objects is left to the application programmer to meet his requirements.
Three categories of operator objects are provided: embedded, alien and IKS operator objects.

**Embedded operator objects** manage generic operations distributed as methods over several image objects. A user who invokes such an operation should not be aware of its multiple definitions in a diversity of image objects. To tackle this problem we propose this abstract operator object whose principal purpose is to confine in one object all information about a generic operation which is common to all its distributed realizations. It does not provide the functional behavior itself; instead it only includes information (and related methods) for updating the history of an image and specific help functions for the user.

**Alien operator objects** serve to integrate existing picture processing algorithms externally coded in high-level programming languages, e.g. Fortran, C, and Pascal, into the IKS-PFH. In order to support language-specific features (e.g. storage formats, parameter passing mechanisms etc.) one specialization of the alien operator is provided for each programming language containing specific converting routines and activation templates (fig. 8). These templates unify the communication with the other IKS-PFH objects so that clients of an alien operator object do not recognize any difference to operations coded within the IKS-PFH.

This allows a generic operation to be partially implemented within the IKS-PFH and using other languages outside of the IKS-PFH depending on image types, e.g. for efficiency reasons or support of special processing hardware.

The **IKS operator object** describes abstract image processing operations similar to those introduced by Gemmar and Hofele:

---Control flow objects--- to achieve code factoring and separating specific from general parts of an iconic operation. Separating the control structure of an iconic operation, e.g. specifying the sequence of accessing individual picture
elements, from its functional parts hides details of the image representations from the application developer and, when being a common part of many imaging operations, avoids code duplication. For example, it allows well-structured access to parallel processing facilities to be shared by many IKS operators, e.g. providing parallelization strategies for special classes of imaging operations such as convolution-type operations.

- **Local operations objects** define the detailed numerical behavior of an imaging operation. A local operation object is applied as described by the control flow object (e.g. conventional top-to-bottom scanning) to pixels selected by a neighborhood object (e.g. rectangular neighborhood).

- **Neighborhood objects** provide the context for the local operations and allow the internal definitions of neighborhood and local operations to be disentangled. This allows the actual neighborhood to be dynamically changed independently of the local operation while processing the image.

The separate specification of control flow strategies, local operations and neighborhood objects supports programming by (ex)changing small software parts (exploratory and incremental programming). A local averaging operation may again serve as a simple example for the above concepts. As above, the averaging operation determines the grey value of every pixel in the output picture as the average over the grey values of some neighboring pixels in the input image. We have defined the average operation by using an abstract filter object. The abstract filter first checks its operand to be a 2D iconic image and generates (if not provided) a new image object for output with the same characteristics (i.e. same dimensions, same pixel type etc.). The abstract filter object includes a control flow strategy which systematically executes the local averaging operation in the conventional top-to-bottom scanning as specified by the control flow. For every processed pixel the local operation consists of the calculation of the local averages over pixels whose coordinates have been provided by the neighborhood (fig. 9).

From this example it is easy to recognize the flexibility and extendibility inherent to this approach. By changing one of the units the user is able to define a new (or to modify an existing) operation. Furthermore he is able simultaneously to influence (e.g. improve the performance of) a large number of operations by redefining for example only the control flow strategy. The functional unit approach provides few, but powerful, basic building blocks for picture processing operations.

### 3.3. Communication objects

Communication objects are used to interconnect major building blocks of
the imaging system. These building blocks are usually developed in different places by different people. So, a well-defined interface structure is needed in order to provide a safe mechanism for interconnecting these building blocks (see fig. 1) and to shield changes in one block from other parts of the system. To illustrate the functionality of communication objects, we use in the following paragraphs the coupling of a stand-alone user interface system with the different building blocks of the IKS as an example.

Because of their inherent complexity, user interfaces should be designed as stand-alone user interface systems (UIS). Like IKS-PFH, the UIS should be organized as a user interface framework (UIF) of basic building blocks so that the UIS can support the same degree of extendibility and reusability as the IKS-PFH itself. As a consequence, an application developer is able to reuse user interface components in addition to the picture processing components to
arrive at a uniform and consistent system behavior in various application domains\(^\text{24,25}\). For the connection of a UIF to the IKS-PFH we propose a communication interface consisting of communication objects. If a new UIS is to be connected to the IKS-PFH, only the communication objects have to be adapted.

For the separation of user interfaces from applications, user interface management systems (UIMS), as described by Kasik et al.\(^\text{26}\), have gained wide acceptance. Early UIMS concepts proposed a formal specification of user interfaces including the definition of presentations and the description of valid dialogues and their mapping to application function invocations. The proposed specification methodologies include command language grammars and state transition networks. One objective was to enable non-programmers to specify user interfaces. Such UIMSs had early successes in implementing command languages and menu-driven systems, but with the advent of more complex interfaces (e.g. window managers and direct manipulation style interfaces), problems of principle have arisen. One problem is semantic feedback for direct manipulation interfaces, i.e. whether to handle the feedback semantics within the UIMS (because feedback is a UIMS task) or within the application representing the general semantics. Other problems arise from multitasking and usage of multiple I/O channels. To cope with these problems, a general shift from a formal specification UIMS to object-oriented user interface development systems can be observed in the UIMS research area\(^\text{27-29}\).

The design of a UIS as an object-oriented framework nicely fits into our ideas of an object-oriented picture processing framework and may serve as a general paradigm for structuring communication between system components ("object worlds"). To support the inherent flexibility and extensibility of the IKS-PFH, such a user interface framework has to provide a similar degree of flexibility and extensibility\(^\text{30}\). We assume the existence of an extendible application-independent UIF (e.g. ICpak 201 by Stepstone) which serves as a basis for defining an object-oriented UIF for picture processing applications (fig. 10). This UIF itself can make use of a user interface toolbox (e.g. the X Toolkit) and a window system (e.g. the X Window System) resulting in a hierarchy of abstraction levels and providing a high degree of reusability for user interface software. A picture processing specific UIF can be designed without making any assumptions about the underlying basic user interface components and window systems. The IKS-PFH only realizes the existence of communication objects which connect it to arbitrary user interfaces. These communication objects provide a logical view of a UIS to the IKS-PFH and vice versa and may be generalized to logical views onto the various system components allowing for the translation of mutually exchanged messages (fig.
Fig. 10. Proposed overall structure of a UIS and its communication with IKS.
To limit the necessary amount of knowledge about the connected user interfaces and related system components within IKS-PFH as far as possible, proper control and data structures are required for exchanging information.

According to Thomas, two control strategies can principally be distinguished: internal and external control. In our case, internal control means IKS-PFH being master with respect to the UIS, and external control means the UIS being the master of the IKS-PFH. For many applications, however, it is advantageous to follow a mixed control strategy, e.g. that proposed by Hurley and Silbert. The UIS and the IKS-PFH are modelled as systems having equal rights and communicating with each other—the roles of master and slave changing continuously. A possible implementation uses asynchronous processes for each system. We propose to locate the main control within the communication interface. Only for exception handling will the IKS-PFH initiate abstract interaction routines to inform the user.

The IKS-PFH and UIS include, independently from each other, data structures necessary for their respective functionality. The communication objects have to establish a mapping from IKS-PFH entities to UIS entities and vice versa. The communication of both systems via the communication objects should be done by translating and passing of messages. In this sense, the role of communication objects can be generalized to safe and flexible communication mechanisms for coupling not only to different user interface environments, but also to other system components providing groups of
Object-oriented framework for image processing

Fig. 12. Only the communication objects contain knowledge about the IKS and the user interface, e.g. one and the same IKS object can be displayed in different ways.

closely connected objects ("object worlds"). This separation of large systems into independent object worlds, which communicate with each other only via dedicated communication objects providing mutual logical views, is a natural extension of an object-oriented design.

To give an example, the IKS-PFH object representing an image sequence recognizes only the existence of a related communication object, but not that it may be presented in different ways according to a user's needs. The image sequence communication object "knows" that an image sequence consists of a multitude of simple image objects and is able to map these images onto the appropriate UIS components, e.g. windows. Furthermore it is able to handle other UIS objects, e.g. buttons. The communication object related to an image sequence holds information about the respective IKS-PFH object and is able to map its parts needed for presentation to different versions of UIS objects according to user preferences (fig. 12).

4. Implementation

The major goal of our implementation was to evaluate the concepts outlined above. The object-oriented design principles of the proposed IKS-PFH architecture are most efficiently implemented using object-oriented programming languages. We selected "Objective C" for the implementation of the iconic parts, and Prolog and Common Loops for the implementation of two communication interfaces. In addition, image processing algorithms coded in
Fortran, Pascal and C have been incorporated from external libraries. The general idea of using mixed language programming, besides the reuse of existing software, is to gain an appropriate balance of efficiency and flexibility (fig. 13).

The implementation has been deliberately limited to the crucial architectural aspects of the object-oriented approach to image processing. Special attention was paid to a more detailed analysis of the connection of different UISs to the IKS-PFH. To achieve a working system for real applications a lot of detailed work needs to be done.

4.1. Data objects

The root class definition of all data objects is located in the “IKSObject” class. The “IKSObject” provides the behavior common to all objects within the IKS-PFH and inherits the functionality common to every object in the “Objective-C” environment as defined in the class “Object”.

For a working prototype we implemented as basic data objects different pixel and matrix objects. A general pixel object has been implemented as the class “Pixel” and specialized to 8 and 16 bit integers in the classes “Integer Pixel” and “Byte Pixel”. These pixel objects provide the basic arithmetic and conversion functionality needed by IKS operator objects. For the representation of the image object data, a general matrix object is provided by the class “Matrix”. Its associated functionality allows uniform access for formatting, indexing and accessing individual elements.

The image object “Bild” contains the general format description and image histories to be used by IKS operator objects and general methods for initializ-
Object-oriented framework for image processing

ing, loading and saving image raw data. These have been specialized in the class Pixel "Bild" as the most prominent representation of an iconic image containing matrices of pixels.

Only single image objects have been realized, the extension to compound image objects is straightforward.

4.2. Operator objects

As examples of embedded operator objects classes for the addition, subtraction, median-filtering and averaging of images have been implemented. The actual functionality of these operator objects resides with the image classes they operate on. These operator objects share the behavior of a common superclass "Operator", but provide specialized methods for help information and updating of image histories.

Using alien operator objects, we have incorporated several edge-detection algorithms taken from external libraries coded in Fortran, C and Pascal.

IKS operator objects have been implemented allowing for simple scanning as control strategies. The control flow strategies use index calculation methods provided by the matrix objects to access the pixel in a uniform manner independently of the actual index representation. The semantics of the imaging operation is defined within the local operation specification and the neighborhood objects. The example of general convolution over rectangular neighborhood have been realized.

4.3. Communication objects

To demonstrate the basic functionality of communication objects, two simple prototype user interfaces have been implemented to show the flexibility of the chosen approach:

—a command language user interface (CLUI) has been implemented in Prolog to give access to every object and method of IKS-PFH. The CLUI is designed to follow the external control paradigm, i.e. the communication interface only has to provide a logical view of the IKS-PFH to the CLUI. The basic idea behind this approach is that one general message predicate is sufficient for completely representing the communication interface.

—a menu system has been implemented in Common Loops as an independent process so that communication objects have to provide inter-process communication facilities. We have developed two kinds of communication objects—the low-level and the high-level ones. The low-level communication objects (LLCO) define the inter-process communication, whereas the
high-level communication objects (HLCO) provide the mapping of the UIF components onto the IKS-PFH functionality.

The inter-process communication is handled by mailbox services, with the LLCO defining mailboxes for sending and receiving messages from the IKS-PFH. The VMS-specific details are hidden within these LLCO. The HLCO have to map the components of the user interface to IKS-PFH components and vice versa, e.g. the operator communication object defines the correspondence between an IKS operator object and a menu object containing the possible selections.

The menu system does not know anything about the IKS-PFH—it does not even know of its existence! The IKS-PFH only holds knowledge about the existence of a UIS (via the UIS mailbox). The IKS-PFH does not make any assumptions about the internal structure of the user interface. So, the important encapsulation and separation requirements are met. Further, it is possible to employ user interface components and to adapt them to the IKS-PFH needs.

4.4. Interrelationship between different object categories

In the context of the averaging operation we want to describe the interrelationship between the different components, i.e. the image objects, operator objects, and user interface objects involved. We prefer a user’s perspective to an imaging system as a starting point of our discussion.

First, the user initiates an imaging operation by selecting an image and the desired operation. An image object is represented at the user interface level in different ways as described in the previous section. It may be stored in a peripheral storage device, it may be contained in an image directory system, which itself may be represented as a scrollable list at the user interface level, or it may be available in main memory to be displayed in a window or represented by an icon. The image selected by the user for further processing is highlighted in some way (e.g. by a colored frame). But, a standard window object in a UIS does not (and should not) “know” anything about the internal structure of image objects. It only displays the contents of an image, i.e. it holds some kind of pixel data whose structure may be different from the internal data representation used in the image object, and provides some “hooks” for it to be coupled to communication objects. Such a hook may be a remote procedure call of a user-interface-specific procedure with the title of the user-selected window passed as a parameter. The corresponding communication object knows everything about the image object needed by the window object and vice versa and manages the exchange of information.
needed, e.g. the ratios of the display image size related to the original image size, information related to the color depth such as look up tables, and status information.

The same mechanism is used for coupling operator objects to corresponding user interface objects. An operator object may be represented as a control panel containing basic user interface elements (e.g. sliders and buttons) which are used to control the operators parameters. When the execution of the operation is initiated, the communication object sends a "doIt" message with the selected image object as a parameter to the embedded operator object "averageOperator" (see fig. 14):

[averageOperator doIt: selectedImage!].

In our example, the operator object "averageOperator" now receives the message "doIt" together with parameters describing the input and output image objects. If no output image is provided, a new one will be instantiated or the input image will be overwritten (according to a corresponding parameter held by the average operator object).

Next, the "averageOperator" object checks whether a specialized average operation is available for the selected image object type:

[selectedImage respondsTo: average!].

If such a specialized method is available, it will be executed. Such a method
may be available as an externally coded function which is accessed via an alien
operator object (fig. 7)

\[\text{outputImage setContents: [selectedImage average: parameters]}\].

If not, the corresponding IKS operator object “averageIKS” will be used. This IKS operator object “averageIKS” (fig. 15) includes a control flow strategy and holds references to a local operation object “localAverageOperation” and to a neighborhood object “rectangularNeighbourhood”:

\[\text{outputImage setContents: [averageIKS doit: selectedImage]}\].

The “doIt” method of the “averageIKS” operator object activates the control flow strategy “loopOver”, which sequentially loops over the image matrix in a row-by-row fashion:

\[\text{self loopOver: inputImage}].\)

Within this method, the local function “localAverage” is invoked for each pixel in turn:

\[\text{localAverage process: image : neighbourhood : x_coord : y_coord}.\]

The “process” method of the “localAverage” object uses the neighborhood object “rectangularNeighbourhood” to calculate the result pixel

\[\text{rectangularNeighbourhood getSurroundingPixels: image : x_coord : y_coord}].\]

In this way, the result image is calculated pixel by pixel using the pipeline of operations described above.

Our prototype implementation showed the feasibility of the two different approaches:

—incorporation of alien operations (reusing existing software retaining high execution speed); and
—exploitation of the complete “repertoire” of the object-oriented design methodology (sacrificing speed for flexibility and safe software components).

5. Conclusion

We have proposed an Iconic Kernel System as a framework for a flexible
Object-oriented framework for image processing

Operator Object 'averageIKS'

```plaintext
id localAverage;
id neighbourhood;
```

```plaintext
doIt: inputImage { ...
  [self loopOver: inputImage]
  ...
}
loopOver: image {
  (local Average
    process: image
      neighbourhood
      : x_coord
      : y_coord
      ...
    getSurroundingPixels: image
      : x_coord
      : y_coord
      ...
    calculate and return
    surrounding pixels
  }
}
```

Local Average Operation 'localAverage'

```plaintext
process: image
  : neighbourhood
  : x_coord
  : y_coord

  getSurroundingPixels: image
    : x_coord
    : y_coord
    ...
  calculate result pixels.
  ...
```

Rectangular 3 x 3 Neighbourhood

Fig. 15. IKS operator object "averageIKS".

software environment for image processing. Its strict object-oriented design provides a high degree of modularity and flexibility. Its four major components—the image, operator, peripheral and communication objects—serve as reusable components for a large variety of applications. The hierarchical
description of the data objects and the related operations provide a clearly
arranged overall structure of the IKS-PFH prototype. A separate UIS allows
the connection of the IKS-PFH to a multitude of user interfaces. The
peripheral objects have not been worked out in detail, although the concepts
have already been shown to meet the requirements of hardware and system
software independence.

The first IKS-PFH prototype implementation has shown the feasibility of
the chosen approach. We deliberately limited the implementation to the crucial
aspects of object-oriented design and reusability. Although we implemented
only a limited number of possible components we were able to demonstrate
that

—by carefully adapting object-oriented design and implementation to various
levels of “efficiency versus flexibility” requirements, it is indeed possible to
design a flexible yet efficient object-oriented environment,
— it is possible to integrate conventionally designed software into object-
oriented environments and to access them in a uniform manner without
significantly sacrificing their original performance and thus to make use of
large amounts of already existing well-proven software,
— it is possible—and with respect to the currently available implementation
languages also advisable—to use different implementation languages.

During the design process an evolutionary approach was possible owing to the
unsurpassed degree of flexibility and extendibility inherent to object-oriented
concepts (i.e. inheritance and genericness). IKS-PFH as a whole was designed
in a way which allowed an independent development of its individual com-
ponents. This rapid prototyping approach was only possible using the dynamic
binding capabilities of Objective C in combination with the interpreted
languages Prolog and CommonLoops.

Future research is needed with respect to

—more sophisticated user interfaces providing dedicated development tools
for image processing and specific help and status information,
— attaching user interfaces based on X-windows, e.g. OSF Motif and Open
Look,
— the improvement of the calling templates for externally coded software
modules,
— the management and analysis of the history objects to provide powerful
undo/redo strategies.
REFERENCES


20) A. Goldberg and D. Robson, Smalltalk-80—The Language and its Implementation, Addison Wesley, Reading MA, 1983.


