Developing Morphological Building Blocks: From Design to Implementation

MARCOS CORDEIRO D'ORNELLAS[1], REIN VAN DEN BOOMGAARD[1]

[1]ISIS - Intelligent Sensory Information Systems
University of Amsterdam, Faculty WINS
Kruislaan, 403 - 1098 SJ
Amsterdam, The Netherlands
{ornellas,rein}@wins.uva.nl

Abstract. Mathematical morphology has become a widely used technique for image processing and computer vision. Initially designed as a set theory, it was generalized to the set of grayscale images and to the complete lattice framework. Despite this accredited theoretical evolution, contemporary practice in morphological algorithm development still lacks of a standardized, mathematically rigorous algebraic structure that is specifically designed for image handling. The purpose of this paper is twofold: first, it is intended to give rise to a new morphological framework that overcomes the combinatorial explosion of algorithms needed to deal with all possible types of lattices and structuring functions. Second, it provides us with the essential tools in order to design and implement generic building blocks for morphological image operators.

1 Introduction

Many times, people involved in software development for image processing and computer vision are confronted with questions concerning the design and implementation of algorithms. The reason for these questions is that the developers as well as end users rarely communicate in the same language and in turn, what an algorithm precisely does is somewhat unclear. Particular features and pixel configurations, not taken into account in the original implementation, are some of the problems that often occur. Moreover, very often the answer for such questions are not straightforward and what developers do is to show how to *program around* that particular feature.

No doubt, that much of these problems is showed up because in the past no one paid attention to specify and design the method of working of the algorithms. Unfortunately, an algorithm that is hacked this way is nearly impossible to define afterwards. One of the reasons is that program developers still do not make the most of the functionality provided by the available programming tools.

For instance, current released morphological software packages like `Micromorph` [1] and the `MMach` Toolbox [2] for the `KHOROS` environment are still facing these problems which has led to a entire reformulation in such systems and in some cases, to the development of a complete new software, with the intention to continue standing as an useful tool. Sometimes, slightly different formulations for one particular issue in one of the aforementioned systems arrive at a different algorithm representation, which in turn leads to a distinct implementation. Moreover, having distinct implementations in practice for the same morphological operator in theory, generates more documentation, make the software difficult to understand, and to use in an appropriate way.

Serra [18] was the first to observe that a general framework for morphology was necessary. He noticed that it could be achieved if one starts from the assumption that the object space is a complete lattice. This idea has been carried further by various people, in particular Matheron [12], Heijmans [9], and Heijmans and Ronse [10]. In the last few years, this idea has been extended to multi-valued morphology (e.g. color images or image sequences) in the works of Serra [19] and Goutsias et al. [8].

In this paper, the essential algebraic theory from image algebra and lattice representation is used in order to build a morphological framework that overcomes the combinatorial explosion of algorithms needed to deal with all possible types of lattices (image types) and structuring functions. The important point with respect to this theory is that it gives a concrete system for the representation of translation-invariant image processing algorithms. Even if an algorithm such as erosion is formulated in terms of minima and subtractions, these binary operations must be implemented in a proper way.

In addition, we propose useful and elegant algorithm implementations for morphological image operators based on the building block design approach. We explore the functionality presented in the `Horus` library to model the building blocks within the proposed morphological framework.

The organization of the rest of this paper is as fol-

[1] ©Centre de Morphologie Mathématique, École Nationale Supérieure des Mines de Paris.
[2] ©Khoral Research, Inc.

lows: Section 2 introduce the essentials of mathematical morphology and image algebra. In section 3, we present the modern way of constructing a morphological framework based on pixel lattice, image lattice, and the operator lattice. Section 4, addresses the design of morphological building blocks and the set of rules attached to them. In section 5, we move the building blocks design into practice by using the `Horus` library to represent template and neighborhood operations. Section 6, shows some of our experimental results and evaluation and we conclude with section 7, summarizing the results and further research.

2 Mathematical Morphology within Image Algebra

The development of algebraic frameworks in the context of algorithm development for image processing has been strongly influenced by image algebra. This term was first used in reference to mathematical morphology, whose binary theory had been developed by Serra [17] in a precise an well-designed algebra for image analysis.

Let us recall briefly the algebraic framework in which a general characterization for mathematical morphology and its elementary operators and properties are possible. We refer to Heijmans and Ronse [10], Ronse and Heijmans [16], Heijmans [9], Ronse [15], and Dougherty and Sinha [5] [6].

We take a set $\mathcal{L}$ on which we have a partial order relation $\preccurlyeq$ and $\mathcal{V}$ as the set of all values on $\mathcal{L}$. Then $\mathcal{L}$ is called a partially ordered set, or in brief a poset. For any subset $\mathcal{S}$ of $\mathcal{L}$, we define its supremum $\curlyvee$ and infimum $\curlywedge$ as respectively the least upper bound and the greatest lower bound of $\mathcal{S}$ and $\mathcal{L}$. It is clear that the supremum and infimum of $\mathcal{S}$ are unique, provided that they exist.

Definition 2.1 (Complete Lattice) *A complete lattice is a set $\mathcal{L} = (\mathcal{V}, \preccurlyeq)$, if every non-void subset $\mathcal{S}$ of $\mathcal{L}$ (even a finite one) has a supremum $\curlyvee\mathcal{S}$ and infimum $\curlywedge\mathcal{S}$ defined in $\mathcal{L}$. Then it has a least and greatest element written by $\curlyvee_{\mathcal{L}}$ and $\curlywedge_{\mathcal{L}}$.*

Definition 2.2 (Dilation and Erosion) *Let $\mathcal{L}$ and $\mathcal{M}$ be complete lattices. A dilation is an operator $\delta : \mathcal{M} \Rightarrow \mathcal{L}$ such that for any family of signals $\{y_i \mid i \in \mathcal{S}\} \subseteq \mathcal{M}$:*

$$\varepsilon(\bigvee_{i\in\mathcal{S}} y_i) \quad = \quad \bigvee_{i\in\mathcal{S}} \varepsilon(y_i). \tag{1}$$

By duality, an erosion is an operator $\varepsilon : \mathcal{L} \Rightarrow \mathcal{M}$ such that for any family of signals $\{x_i \mid i \in \mathcal{S}\} \subseteq \mathcal{L}$:

$$\varepsilon(\bigwedge_{i\in\mathcal{S}} x_i) \quad = \quad \bigwedge_{i\in\mathcal{S}} \varepsilon(x_i). \tag{2}$$

The recognition that the original binary and extended grayscale morphological theories are properly developed in the context of complete lattices is essential to the algebraic theory of image operators as proposed by Sternberg [20]. If we view algebraic operator theory as representation of operators between lattices, then, a complete theory has been developed by Banon and Barrera [1] [2]. Since $\preceq$ is the only embedded operation on lattices, it seems natural to expect that other lattices operations can be expressed in terms of $\preceq$.

Image algebra is essentially acceptable as a mathematical necessary circumstance in which to represent algorithms. These algorithms should appear as a sequence of operators and images into a well designed framework, where each operator can finally be expressed as a sequence composed of some collection of elementary operators. One of the main issues of image algebra is to produce statements in terms of low-level operations that are tied to the algebraic representation of the fundamental structures upon which images and image operators are designed as proposed by Ritter and Wilson [13].

Therefore, our goal is to obtain direct expressions for lattice operators and relate these expressions to efficient algorithm implementations. This can be accomplished by supporting a collection of fundamental operators that can be grouped together, leading to more elaborated compositions.

3 Constructing the Morphological Framework

The modern way of constructing the morphological framework is a three-step process based on pixel lattice, image lattice, and the operator lattice. It is indeed possible to make a distinction between scalar and non-scalar pixel lattices whether or not the ordering relation is implied or imposed. In this section, we will explain the scalar pixel lattice. We refer to Ornellas et al. [4] for the details about non-scalar pixel lattices and its use with color images.

3.1 The Pixel Lattice

At the first level, we do not need to work with images at all and only look at a complete lattice $\mathcal{L}$ that represents the set of all values that a pixel can have ($\mathcal{V}$), together with a partial ordering relation $\preccurlyeq$. For computational reasons we also assume that the infimum $\curlywedge$ and supremum $\curlyvee$ operators are explicitly defined (Implicitly, the ordering defines the notions of supremum and infimum). Furthermore, we assume that the infimum $\curlywedge_{\mathcal{L}}$ and supremum $\curlyvee_{\mathcal{L}}$ of $\mathcal{L}$ are known explicitly.

Definition 3.1 (Pixel Lattice) *A pixel valued lattice is the tuple: $\mathcal{L} = (\mathcal{V}, \preccurlyeq, \curlyvee, \curlywedge, \curlyvee_{\mathcal{L}}, \curlywedge_{\mathcal{L}})$ where:*

- *$\mathcal{V}$ (value set): the set of all values;*
- *$\preccurlyeq$ (ordering): the orderings relation $\mathcal{L} \times \mathcal{L} \rightarrow$ (true,false);*

$\curlyvee$ *(supremum):* $\mathcal{L} \times \mathcal{L} \times \cdots \times \mathcal{L} \to \mathcal{L}$ *(for discrete images we only need a finite number of arguments);*

$\curlywedge$ *(infimum):* $\mathcal{L} \times \mathcal{L} \times \cdots \times \mathcal{L} \to \mathcal{L}$*;*

$\curlyvee_{\mathcal{L}}$ *(lattice supremum): the largest value in the value set;*

$\curlywedge_{\mathcal{L}}$ *(lattice infimum): the smallest value in the value set.*

The operations on and between pixel elements of a given value set $\mathcal{V}$ are the usual elementary operations associated with $\mathcal{V}$. $\mathcal{V}$ may vary according to the application. Possible examples for $\mathcal{V}$ are $\overline{R} = R \cup \{\Leftrightarrow\infty, +\infty\}$, $\overline{Z} = Z \cup \{\Leftrightarrow\infty, +\infty\}$, a closed segment $[v_o, v_i]$, or a discrete interval $\{0, 1, \cdots, N\}$. We can also take $\mathcal{V}$ not contained in $\overline{R}$ as in the case of color images, represented by tuples (R,G,B).

A dilation in the context of the pixel lattice is any mapping $\delta : \mathcal{L} \to \mathcal{L}$ that distributes over the supremum and preserves the lattice supremum (i.e. $\delta(\vee_{\mathcal{L}}) = \vee_{\mathcal{L}}$). Equivalently, an erosion is any mapping $\varepsilon : \mathcal{L} \to \mathcal{L}$ that distributes over the infimum and preserves the lattice infimum (i.e. $\varepsilon(\wedge_{\mathcal{L}}) = \wedge_{\mathcal{L}}$).

For example, consider the lattice $\mathcal{L} = (R \cup \{\Leftrightarrow\infty, +\infty\}, \leqslant, \vee, \wedge, \Leftrightarrow\infty, +\infty)$. This is evidently the example that explains the choice of the symbols in the abstract case. Infinitely many dilations and erosions can be defined on this lattice. For example, let us consider the erosion:

$$\varepsilon(x) \quad = \quad x \Leftrightarrow a \tag{3}$$

where a is also an element from $\bar{R} = R \cup \{\Leftrightarrow\infty, +\infty\}$. This is obviously an erosion since this operation distributes over the infimum and $\Leftrightarrow\infty \Leftrightarrow a = \Leftrightarrow\infty$ for all a. Strangely enough the above operation is also dilation.

A dilation is defined as:

$$\delta(x) \quad = \quad x + a \tag{4}$$

where a is the same scalar used in the erosion. Note that dilation and erosion are inverse operators in this case, something that is very rare in morphological image processing. Nevertheless the above defined dilation and erosion form the basis of the classical dilations and erosions on image lattices.

In the previous example, we defined a parameterized family of dilations and erosions, where the parameter is the value a. Once we have defined them in a complete lattice, many tools that are often used (like openings, closings, granulometries, alternating sequential filters, etc.) follow automatically from this framework.

Again, we would like to stress the fact that even in the familiar lattice $\mathcal{L} = (R \cup \{\Leftrightarrow\infty, +\infty\}, \leqslant, \vee, \wedge, \Leftrightarrow\infty, +\infty)$ the above definitions of dilations and erosions are just examples and we can define many more. For instance, consider the operator:

$$\varepsilon(x) \quad = \quad \frac{x}{a} \tag{5}$$

where a is a *positive* real value. It is indeed an erosion. The dilation in this case is:

$$\delta(x) \quad = \quad ax. \tag{6}$$

3.2 The Image Lattice

Let an image be defined as a map from some domain E onto the complete lattice $\mathcal{L}$. The space of all these images will be denoted as $E^{\mathcal{L}}$. We construct a complete lattice on this set of values and embed the ordering relation from $\mathcal{L}$ into $E^{\mathcal{L}}$. Let f and g be two images (i.e. elements from $E^{\mathcal{L}}$), then we define an ordering relation $\preccurlyeq$ on the new complete lattice $\mathcal{L}' = E^{\mathcal{L}}$ as:

$$f \preccurlyeq g \Leftrightarrow \forall y \in E : f(y) \preccurlyeq g(y). \tag{7}$$

We also define the embedded operator Γ that takes a value a from $\mathcal{L}$ and makes an image that has the value a elsewhere: $\Gamma_a(y) = a, \forall y \in E$. In addition, the infimum and supremum operators are embedded in $E^{\mathcal{L}}$ using a pixelwise definition. For instance for the infimum operator we have:

$$(f \curlywedge g)(y) \quad = \quad f(y) \curlywedge g(y). \tag{8}$$

The meaning of the symbols $\preccurlyeq$, $\curlyvee$ and $\curlywedge$ is overloaded since it depends on the context whether we are referring to the operators on the pixel values or to the operator working on the entire image. We refer to Ronse and Heijmans [16] for the proof that

$$\mathcal{L}' \quad = \quad (E^L, \preccurlyeq, \curlyvee, \curlywedge, \Gamma_{\curlyvee_{\mathcal{L}}}, \Gamma_{\curlywedge_{\mathcal{L}}}) \tag{9}$$

is indeed a complete lattice.

Defining dilations and erosions on the image lattice is particularly simple. The following theorem proposed by Goutsias et al. [8] provides us with a simple and elegant construction for image dilations and erosions. Let $\delta_{x,y}$ and $\varepsilon_{x,y}$ be a parameterized family of dilations and erosions in the pixel lattice respectively. Any dilation on the image lattice can now be written as the supremum over dilations in the pixel lattice. In turn, any erosion on the image lattice can be written as the infimum over erosions in the pixel lattice:

$$\delta(f)(y) \quad = \quad \bigvee_{x \in E} \delta_{x,y}(f(x)) \tag{10}$$

$$\varepsilon(f)(y) \quad = \quad \bigwedge_{x \in E} \varepsilon_{x,y}(f(x)) \tag{11}$$

In case we take the simplest dilation and erosion in the pixel lattice, that is translation invariant, we come with:

$$\delta_{x,y}(f(x)) = f(x) + g(x \Leftrightarrow y) \quad (12)$$
$$\varepsilon_{x,y}(f(x)) = f(x) \Leftrightarrow g(x \Leftrightarrow y) \quad (13)$$

The notion of template as used in image algebra unifies and generalizes the usual concepts of templates, masks, windows, and neighborhood operations into one general mathematical entity. Furthermore, templates generalize the notion of structuring functions used in mathematical morphology as proposed by Ritter et al. [14]. In this way, it seems natural to represent the respective dilation and erosion definitions in terms of templates having B as a support.

$$\delta_{x,y}(f(x)) = f(x) + t_y(x), \quad (14)$$
$$t_y(x) = \begin{cases} g(x-y) & \text{if } (x-y) \in B \\ -\infty & \text{otherwise} \end{cases}$$
$$\varepsilon_{x,y}(f(x)) = f(x) + t^*_y(x), \quad (15)$$
$$t^*_y(x) = \begin{cases} -g(x-y) & \text{if } (x-y) \in B \\ +\infty & \text{otherwise} \end{cases}$$

Note that several types of template operations might be more easily implemented in terms of neighborhood operations. Typically, neighborhood operations replace template operations whenever the values in the support of a template consist only of the unit elements of the value set associated with the template.

By using the template representation, we arrive at:

$$\delta(f_g)(y) = \bigvee_{x \in E} f(x) + t_y(x) \quad (16)$$
$$\varepsilon(f_g)(y) = \bigwedge_{x \in E} f(x) + t^*_y(x). \quad (17)$$

This leads to the classical dilation and erosion of an image f with respect to the structuring function g:

$$\delta(f_g)(y) = \bigvee_{x \in E} f(x) + g(x \Leftrightarrow y) \quad (18)$$

$$\varepsilon(f_g)(y) = \bigwedge_{x \in E} f(x) \Leftrightarrow g(x \Leftrightarrow y). \quad (19)$$

Take due notice of the fact that the general definition of dilations in equation 10 and erosions in equation 11 are not dependent on the classical image border problem. Only elements from the image domain E are used. The translation invariant formulation is more error prone. As it is stated in equations 18 and 19, the translation invariant dilation and erosion are not dependent on the domain E. Classically the function $g(x \Leftrightarrow y)$ is interpreted as an image, and in that case it is evident that $x \Leftrightarrow y$ is not necessarily in the domain E. It is only that $x \Leftrightarrow y$ *visits* all points in E for any constant value x as y *visits* E, when E is an infinite domain. Because in practice E must be bounded, we are bound to run in all sorts of conceptual problems when we insist on interpreting E as the domain of f and g.

3.3 The Operator Lattice

The third and last step in the morphological theoretical framework is to consider the set of all increasing image operators. Indeed, it can be shown that the partial ordering relation introduced for images can be extended to work on operators as well. This last step is perhaps the most interesting one because it allows us to deal with the property of image operators, regardless of what images they actually work. Consider an image I and its corresponding dilation δ_I and erosion ε_I outputs. It is clear that these images adhere to the $\varepsilon_I \leq I \leq \delta_I$ ordering relation in the operator lattice.

4 The Design of Morphological Building Blocks

Design methods vary in how they ease abstraction and reuse to make complexity and change manageable in a given situation. The notion of scale will give us a deeper understanding of these differences. The discussion of scale will be the basis for the fundamental definitions of a building block that is the main concept presented in this section.

4.1 Scales and Building Blocks

The notion of scale in software development has received considerable attention recently. It was resulted that many serious software mistakes have been made because the relationship between scale of the problem and the scale of the solution was poorly understood or applied. Cockburn [3] defines scale as a means of hiding things that are not relevant at the current stage of development. As if a group of code lines will become an algorithm at a coarser scale, a group of related classes might become a module or a component.

Scales are classified by the nature of the abstractions that are typically implemented at each level:

- **Algorithms and Data Structures:** This is the lowest software level. Consider an image: when the image is implemented so that it can hold any value set, its operations can be easily described. Of course, efficiency is a major concern at this level, because performance given away here can not be retrieved at higher levels. Generic programming approach is

especially suited to achieve flexibility and speed simultaneously.

- **Modules and Packages:** It is used to set up boundaries between different parts of the system. Boundaries divide the system into clearly identifiable subsystems that can be handled and used separately. In this sense, modules have been the origin of the interface concept as a means to cross subsystem boundaries in an appropriate manner.

- **Architecture:** The architecture determines how the modules are put together. It communicates the structure of the application to the developers and maintainers so that changes and extensions can be made without destroying the underlying abstractions. Architecture thus acts as a design center that supports system evolution by defining a stable core that preserves system integrity.

It is interesting to observe in the above discussion of scales the need for a balance between flexibility and encapsulation, accomplished by the module level. Therefore, it seems a natural conclusion to generalize the notion of a module to any scale level. This is the purpose of building blocks.

A building block establishes the necessary boundaries that facilitate reuse. It encapsulates some reusable functionality and provides well-defined mechanisms to adapt this functionality to the requirements of a larger whole. A building block containing a detailed functionality will be easier to learn, to maintain, and to transfer into a new context. Moreover, if changes are required, only a single building block must be modified or exchanged.

In this paper, we will restrict ourselves to the design of morphological building blocks for scalar lattices at the algorithms and data structures (lowest level). These basic modules are constructed based on the morphological framework already stated.

4.2 Design Details of Building Blocks

The design of the morphological building blocks may comply with a small set of rules. These rules are intended to specify the kind of lattice we are working with, the support type related to the shape of the structuring function, the reduce and lattice operation within pixel lattice respectively, and the support size.

4.2.1 Lattice Type

The lattice type can be one of the lattices associated with the pixel types in the image. The lattice type determines the lattice values in the structure. In this way, we have a list of scalar lattices associated with byte, short, int, float, and double types. The same holds for two-dimensional and three-dimensional vector representation of scalar lattices like vec2int and vec3double.

Lattice type set in the concepts of lattice supremum and infimum. They can be defined by the user or by default. When defined by default, it follows the pre-fixed limits included in the programming language definition. As an example, a scalar lattice of integers has its infimum assigned to INT_MIN + 1 and its supremum assigned to INT_MAX.

4.2.2 Support Type

The Support type can be specified for flat and non-flat operations and is associated to the shape of the support. Support types are usually two-dimensional and flat. When referring to two-dimensional flat operations, the support acts as a characteristic function in that it do not weigh a pixel but simply note which pixels are in its support and which are not.

Flat support types can have one of the following shapes:

- Square - square shape with dimensions defined by the support size; If support size $= 3$, we have the classical 3×3 representation;

- Cross - cross shape with dimensions defined by support size;

- Disk - disk shape with dimensions defined in support size and disk metric. Here, disk metric assumes one of the metrics: city-block, chessboard, Euclidean, and round;

- Line - line shape with length and angle defined by line length and line angle;

Non-flat support is more complex since we are not dealing with characteristic functions anymore. From that account, it may be possible for the user to define a constant value for all pixels in the support, a set of values for the pixels in the support, or to make use of a function that generates the values within the support.

4.2.3 Support Size

The support size defines the size of the support type and helps allocate a scratch image with a border, handling the border problem.

4.2.4 Reduce Operation Type

The Reduce operation type is represented as the pixel lattice operation of infimum or supremum for flat or non-flat dilations and erosions respectively. These operations should be defined for every pixel lattice.

4.2.5 Lattice Operation Type

The lattice operation type is meant to be one of the trivial operations performed in the pixel lattice that gives rise to dilations and erosions in the pixel lattice. The set of operations (e.g., $+, \Leftrightarrow, *, /, \cdots$), should embed a range checking test routine in order to bound the values within the lattice extrema. The lattice operation type is only necessary when a non-flat operation is performed, using the template representation from image algebra.

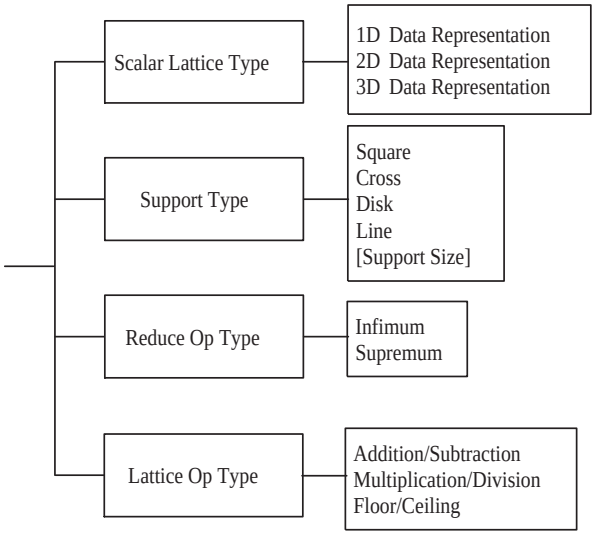

Figure 1: Basic Building Blocks Representation.

Figure 1 gives the building block representation and its components.

5 From Design to Implementation

The design of morphological building blocks is firmly established in the structure provided by the `Horus` library as proposed by Koelma et al. [11]. `Horus` is a multi-platform library, including a concise separation between semantics, representation, and implementation of `Horus` objects. Semantics, representation, and implementation are defined and implemented by different classes. The most important class in the `Horus` library is HxImageRep which defines images and image processing functionality grounded on image algebra. HxImageRep is an abstract data type in that its interface hides all details of the implementation (data storage and manipulation) from the class user. The methods defined in the HxImageRep interface become the only access path to the actual pixels in an image. The functionality is actually implemented by HxImageData. Binary, grayscale, and color images are used to associate semantics with the pixel values, i.e. what does the numerical value of a pixel actually represent.

The implementation of the HxImageData aims at efficiency and makes use of the template mechanism of C++ to implement operations independent of the lattice type. Furthermore, the value set $\mathcal{V}$ may assume different types according to the application in the `Horus` library and determines the respective values for $\curlyvee, \curlywedge, \curlyvee_{\mathcal{L}}, and\ \curlywedge_{\mathcal{L}}$.

With the intention to use the functionality provided by the building blocks representation, we apply function objects by means of template (non-flat) and neighborhood (flat) operations:

- **Template operations**: The result is obtained by sliding the support type over this image and, at each position, combining the values of the support type with the underlying values of this image through the lattice operation type and reducing this set of values to a single one by means of the reduce operation type function object f.
- **Neighborhood operations**: The result is obtained by sliding the support type over this image and, at each position, reducing the underlying values of this image by means of the reduce operation type function object g.

The inner loops for these two operations are implemented through a non-mutating sequence algorithm for_each included in the Standard Template Library (STL), which traverses a sequence of relevant elements contained in the support type. The algorithmic representations for template and neighborhood operations are shown in figure 2 and 3. Dilations and erosions are mapped depending on whether the operation is non-flat or flat.

It is also meaningful to give the reader a graphical overview of the design of morphological building blocks used to implement flat operations within the `Horus` library. For this purpose, we decided to use the Unified Modeling Language (UML) mentioned by Fowler and Scott [7].

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems and consolidates a set of core modeling concepts that are generally accepted across many current methods and modeling tools.

Whenever an operation is called, HxMorphFunctor searches for a function object key in HxMorphFunctorTable with its respective parameters. Once a key has been found, it goes through the derived classes HxFlatMorphLatticeFunctor and HxFlatMorphLatticeFuncSpec in the hierarchical structure with the intention to

find the appropriate method that will actually perform the operation. This is done through the pure virtual function mechanism, included in C++. The UML class representation is expressed in figure 5.

```
Get LatticeType information from the input image;
Call _support.setSize(supportSize);

Allocate scratch image (border handling);
For every pixel (p in this, result) {
    Set result to ReduceOpType::initval;
    For every pixel (q in the SupportType) {
        result = ReduceOpType(LatticeOpType(this(p + q),
                 q.value()),result);
    }
}
Copy result to the output;
Delete scratch image;
```

Figure 2: Non-flat Operation Pseudo-code.

```
Get LatticeType information from the input image;
Call _support.setSize(supportSize);

Allocate scratch image (border handling);
For every pixel (p in this, result) {
    Set result to ReduceOpType::initval;
    For every pixel (q in the SupportType) {
        If (q.value() == 1)
            result = ReduceOpType(this(p + q),result);
    }
}
Copy result to the output;
Delete scratch image;
```

Figure 3: flat Operation Pseudo-code.

6 Experimental Results and Evaluation

The design and implementation of morphological building blocks put its main emphasize on flexible algorithms, because algorithms represent the principle expertise of this field. Algorithm Implementations are consequently built using generic programming and STL. By writing adapters (image iterators and accessors), one can use the algorithms on top of his/her own data structures, within his/her own environment. Alternatively, one can also use the data structures provided by `Horus`, which can be easily adapted to a wide range of applications. Algorithm flexibility and functionality come almost for free since the design uses compile-time polymorphism (templates).

Table 1 shows the processing time for flat dilations with respect to the support type when applied to a grayscale and color image of size 512×512 with lattice type set to double and vec3double respectively. The support size was set to 5 (e.g. 5×5 mask). All the algorithms were tested with a Pentium-II 300 MHz, running Windows NT 4.01, and using Visual C++ compiler 5.0 with optimizations enabled. Figure 5 shows some of the outputs obtained when processing both the test image in its grayscale and color versions.

	square	cross	Euclidean disk	line
Dilation	0.206	0.158	0.238	0.116
Color Dilation	0.529	0.412	0.623	0.307

Table 1: Processing Time (seconds).

It is important to mention that the algorithms proposed in this paper do not exploit machine dependent properties, which speed up computations. By using those properties, algorithms are hard-coded and consequently, the required generality can not be fulfilled anymore.

7 Conclusions and Further Research

In this paper, we described a modern way of constructing the morphological framework based on image algebra representations whose family of operators can be extended in accordance with the application needs. It is important to stress the strong relationship between theory and practice, since the available functionality presented in theory is freely mapped into the design and implementation of the morphological building blocks. The proposed Building blocks are computationally simple, producing very close representations within the lattice theory.

In order to map mathematical representations from the morphological framework into the design level, we made use of the `Horus` environment. `Horus` provides a set of image processing functionality firmly established on image algebra concepts and gives the possibility to combine the fundamental operators into more specialized ones. The morphological framework proved very useful in reducing the time lag between the formulation of an algorithm and its implementation, relieving the developer from many time-consuming programming tasks.

We have attempted to design the morphological building blocks in such a way that only those responsibilities that definitely must be taken on by the morphological framework and the `Horus` classes are imposed upon them. Likewise, the underlying representation of most objects is abstract. This allows the building blocks to be expanded with the intention to support new and possibly more efficient representations. However, we may have to balance these design criteria against the need for efficiency.

The subject of our research is in an early but very interesting stage. The subject is important and in need of

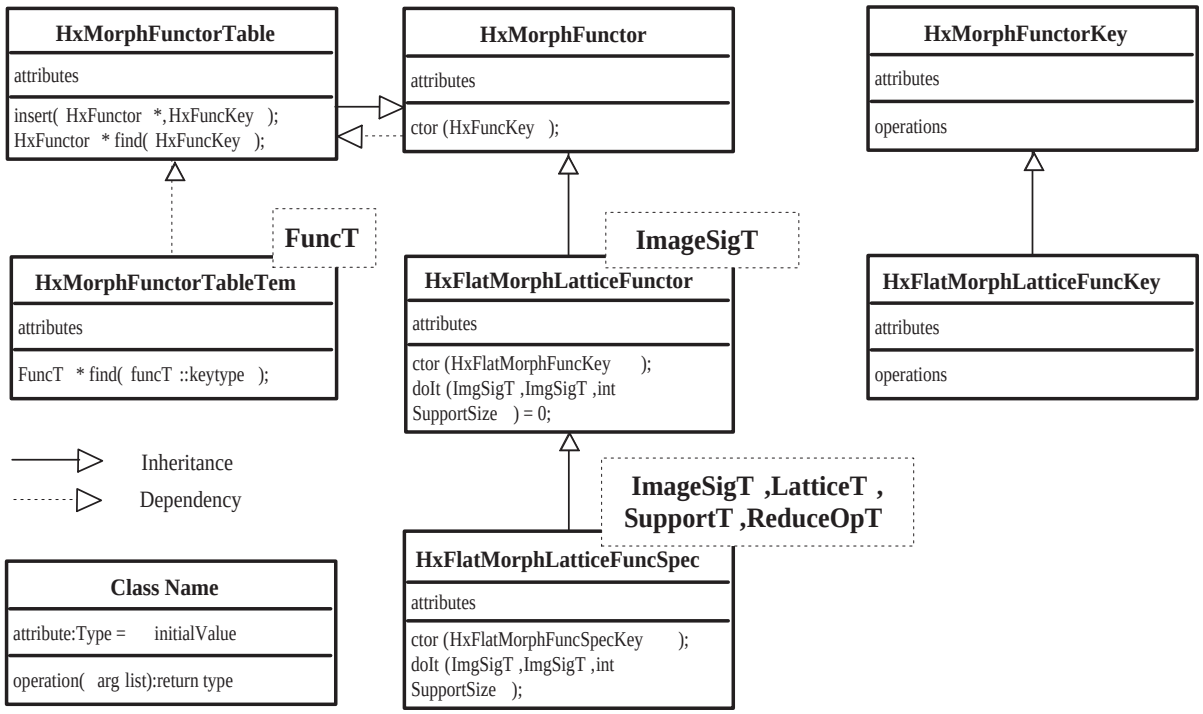

Figure 4: UML Class Representation for Flat Operations.

additional work. Extra care should be reserved to speed up computations by designing more elegant and efficient non-mutating sequence algorithms for structuring function decomposition wherever possible.

Acknowledgements

The authors would like to thank Edo Poll for his valuable suggestions and expertise with respect to the low-level implementation of `Horus` library. This work is partially supported by CAPES Foundation under grant BEX2780/95-0.

References

[1] G. J. F. Banon and J. Barrera. Minimal representations for translation invariant set mappings by mathematical morphology. *SIAM Journal of Applied Mathematics*, 51(6):1782–1798, 1991.

[2] G. J. F. Banon and J. Barrera. Decomposition of mappings between complete lattices by mathematical morphology part 1: General lattices. *Signal Processing*, 30:299–327, 1993.

[3] A. Cockburn. *Surviving Object-Oriented Projects*. Addison-Wesley, London, 1998.

[4] M. C. d'Ornellas, R. v.d. Boomgaard, and J. Geusebroek. Morphological algorithms for color images based on a generic-programming approach. In *Proceedings of the Brazilian Conference on Computer Graphics and Image Processing (SIBGRAPI'98)*, pages 323–330, Rio de Janeiro, 1998. IEEE Press.

[5] E. R. Dougherty and D. Sinha. Computational gray-scale mathematical morphology on lattices (a comparator-based image algebra) part 1: Architecture. *Real-Time Imaging*, 1(1):69–85, 1995.

[6] E. R. Dougherty and D. Sinha. Computational gray-scale mathematical morphology on lattices (a comparator-based image algebra) part 2: Image operators. *Real-Time Imaging*, 1:283–295, 1995.

[7] M. Fowler and K. Scott. *UML Distilled - Applying the Standard Object Modeling Language*. Addison-Wesley Object Technology Series, New York, 1997.

[8] J. Goutsias, H. J. A. M. Heijmans, and K. Sivakumar. Morphological operators for image sequences. *Computer Vision and Image Understanding*, 62:326–346, 1995.

[9] H. J. A. M. Heijmans. *Morphological Image Operators*. Academic Press, Boston, 1994.

[10] H. J. A. M. Heijmans and C. Ronse. The algebraic basis of mathematical morphology – part I: Dilations and erosions. *Computer Vision, Graphics and Image Processing*, 50:245–295, 1990.

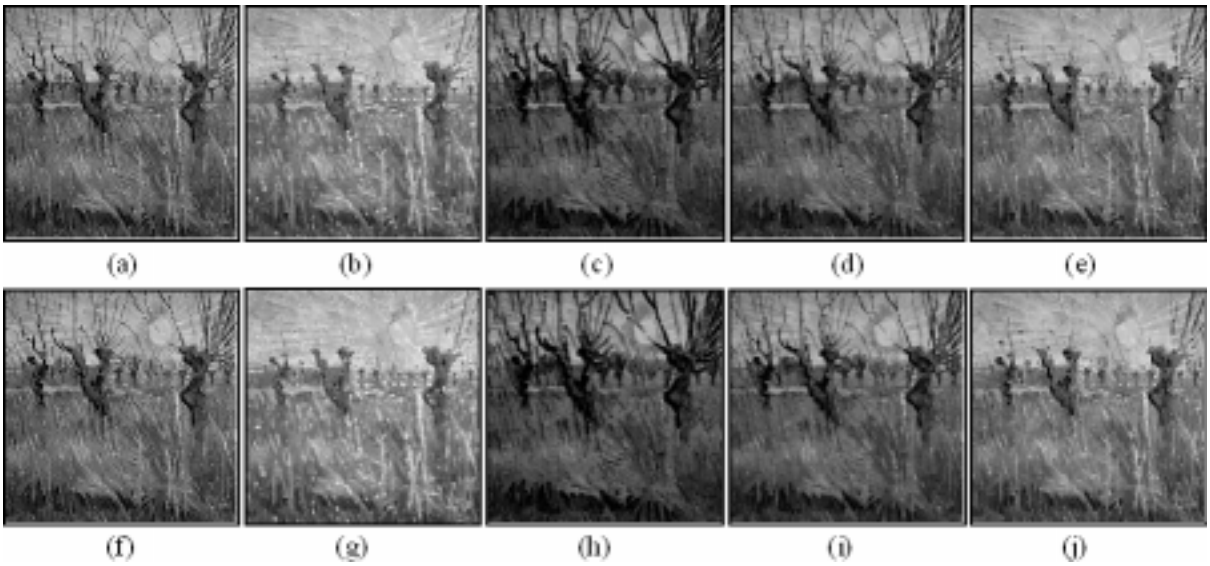

Figure 5: (a) greyscale input, (b) dilation, (c) erosion, (d) opening, (e) closing, (f) color input, (g) color dilation, (h) color erosion, (i) color opening, and (j) color closing. *Pollard Willows With Setting Sun*, Vincent van Gogh, 1888, Kroller-Müller Museum - The Netherlands.

[11] D. Koelma, E. Poll, and F. Seinstra. Horus release 0.6. Research report, University of Amsterdam, Amsterdam, 1998.

[12] G. Matheron. Filters and lattices. In J. Serra, editor, *Image Analysis and Mathematical Morphology, Vol. 2: Theoretical Advances*, chapter 6. Academic Press, London, 1988.

[13] G. X. Ritter and J. N. Wilson. *Handbook of Computer Vision Algorithms in Image Algebra*. CRC Press, New York, 1996.

[14] G. X. Ritter, J. N. Wilson, and J. L. Davidson. Image algebra: An overview. *Computer Vision, Graphics and Image Processing*, 49:297–331, 1990.

[15] C. Ronse. Why mathematical morphology needs complete lattices. *Signal Processing*, 21:129–154, 1990.

[16] C. Ronse and H. J. A. M. Heijmans. The algebraic basis of mathematical morphology – part II: Openings and closings. *Computer Vision, Graphics and Image Processing: Image Understanding*, 54:74–97, 1991.

[17] J. Serra. *Image Analysis and Mathematical Morphology*. Academic Press, London, 1982.

[18] J. Serra, editor. *Image Analysis and Mathematical Morphology. II: Theoretical Advances*. Academic Press, London, 1988.

[19] J. Serra. Anamorphoses and function lattices (multivalued morphology). In E. R. Dougherty, editor, *Mathematical Morphology in Image Processing*, chapter 13, pages 483–523. Marcel Dekker, New York, 1993.

[20] S. R. Sternberg. Grayscale morphology. *Computer Vision, Graphics and Image Processing*, 35:333–355, 1986.