

Properties in Milena

Nicolas Ballas

Technical Report *n°0829*, December 2008
revision 1974

Getting both high performance and genericity at the same time is one of the major research field at the LRDE. Milena, the Olena platform library, faces that issue in the context of image processing. Furthermore Milena has for extra objective to remain simple to use for a practitioner. A solution, experimented since several years, is based on properties. Properties are a set of features statically bound to a type. For instance, image types in Milena have a property named speed that gives information at compile time about the value access time. In this report, we focus on the image properties. We detail the definition of those properties and justify them. We show how those properties help in improving efficiency while maintaining genericity. For that, we take as illustration the implementation of low level routines in the library.

Avoir de hautes performances tout en conservant la généricité est un des domaines de recherche prépondérant au sein du LRDE. Milena, la bibliothèque de la plate-forme Olena, confronte ce problème au domaine du traitement d'image. De plus, Milena a aussi pour objectif de rester simple à utiliser. Une solution à ces problèmes, utilisée depuis plusieurs années, repose sur les propriétés. Les propriétés sont un ensemble de caractéristiques associées statiquement à un type particulier. Par exemple, les types d'images de Milena possèdent une propriété speed qui indique les temps d'accès aux valeurs des images. Durant ce rapport, nous nous intéresserons aux propriétés des types d'images. Nous détaillerons les définitions de ces propriétés. Nous montrerons aussi comment les propriétés aident à améliorer les performances tout en maintenant la généricité. Pour cela, nous prendrons en illustration l'implémentation des routines bas niveau dans la bibliothèque.

Keywords

Olena, Milena, Image processing library, Software engineering



Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
ballas@lrde.epita.fr – www.lrde.epita.fr

Copying this document

Copyright © 2008 LORIE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just “Copying this document”, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.

Contents

1	Introduction	4
2	Properties	6
2.1	Properties in Milena	6
2.2	Image properties	7
2.2.1	Properties related to the value access	7
2.2.2	Properties related to the image organization in memory	10
2.2.3	Properties related to the image geometry	12
2.2.4	Properties related to the image extended domain	14
3	Images in Milena	16
3.1	Images in Milena	16
3.2	Primary images	16
3.3	Morphers	18
4	Routines	20
4.1	Properties implementation	20
4.1.1	Properties definitions	20
4.1.2	Properties and optimization	21
4.2	Routines	22
4.2.1	Fill with value	23
4.2.2	Paste	24
4.2.3	Fill with image	26
4.2.4	Clone	26
4.2.5	Transform	27
4.2.6	Dependencies between low level routines	29
4.2.7	Low level routines performances	29
5	Conclusion	31
6	Bibliography	32

Chapter 1

Introduction

The Milena library

Milena is the generic image processing library of the Olena platform¹ developed at the LRDE² (EPITA³ Research and development laboratory). The library is designed to fulfill three objectives.

- Simplicity : Milena is easy to use and to extend. A non C++ expert user must be able to add new algorithms to the library.
- Genericity : Milena users can write an image processing algorithm once and use it with all the image types present in the library.
- High performance : Genericity must not have a negative impact on efficiency. Milena provides specialized versions of generic algorithms which are more efficient on some specific image types. The appropriate version of an algorithm is automatically chosen at compile time depending on the input image types.

Genericity and high performance

In 2007, the Olena project went through some major changes. The generic image processing library Olena left aside the SCOOP2 programming paradigm ([Thierry Géraud, Roland Levilain, 2008](#)) and now uses a simpler paradigm. This new paradigm is based on properties, a set of information that is statically bound to the library types. Due to these changes, the library was renamed Milena.

The Milena library aims at being generic while remaining efficient. Indeed, Milena provides tools that allow a user to write an image processing algorithm compatible with all the Milena image types. But, the properties also provide the possibility to optimize an algorithm.

For instance, most of the Milena images have their values directly stored in RAM. When we paste an image into another one, it is possible to iterate directly over the value stored in memory instead of iterating over the image points. Of course, it is faster to dereference a pointer in memory, than accessing an image value through a point. But, obviously, this optimized version of the paste algorithm does not work with images that do not store their values in RAM.

1. <http://olena.lrde.epita.fr>

2. www.lrde.epita.fr

3. www.epita.fr

Fortunately, the properties give us information on the images. Thus, they allow us to choose the good implementation of the paste algorithm, the generic (which iterates over the image points) or the optimized (which iterates over the image values in memory).

This report describes the property mechanism, and more specifically the properties related to the Milena image types. First, this report defines the property concept. Then, it focuses on the image properties. It details the definition of those properties and justifies them. In a second time, this report highlights the main image types present in the library. And finally this report shows how properties help in improving efficiency while maintaining genericity. For that, we take as illustration the implementation of low level routines in the library.

Chapter 2

Properties

Properties are a way to classify the Milena types. They are a major part of the library. Indeed, the Milena library is property driven; they provide a way to do static checking. And they also allow users to specialize an algorithm. This chapter defines the concept of property. Then, it lists and details the properties associated to the Milena image type.

2.1 Properties in Milena

Definition

A property is a trait, a characteristic of an image. This characteristic is shared by all the image types, and brings some information about a precise issue. Image types can have different values for a property. For instance, an image in the Milena library can either be in one dimension, two dimensions, three dimensions or more than three dimensions. Hence, there is a property `dimension` that specifies what the dimension of an image is. This property is composed by the values `one_d`, `two_d` and `three_d`. Note that this property can also be equal to the value `none`. It means `dimension` property does not provide any information. This value is useful to deal with image with more than three dimensions.

Purposes of the properties

Properties provide different functionalities like static checking or algorithm specialization.

- Static Checking:

Some image processing operators are only defined to work with specific image types. With the properties, we can check that the operator input image type respect the operator requirements at compile time.

For instance, consider an algorithm which takes a slice of an image in three dimensions, and interprets it as an image in two dimensions. The input image must be in three dimensions. Therefore, the `dimension` property of the input image must be set to the `three_d` value.

- Specialization of an Algorithm

Some image processing algorithms can be optimized to be much more efficient. However the optimizations often imply a loss of genericity. The optimized algorithms will not work with all the image types anymore.

A developer can write two versions of the same algorithm, one generic version which work with all the image types and one specialized version which is optimized for specific image types. As properties provide information known at compile time, the right version of the algorithm is automatically chosen at compile time according to the input image types.

Properties and interface

In Milena, all the image types share a common interface. However, the image types have specificities. And these specificities are very important from an optimization point of view. Thus there are procedures that handle these specificities. These procedures are not defined inside the image class. They are externalized in a function.

For instance, if an image is in two dimensions, we can access an image value through the row and column coordinates. Thus, we define the function `at(Image ima, index row, index col)` that accesses a value of `ima` localized at the point, which has `(row, col)` as coordinates. This function statically checks that the input image is in two dimensions with the help of the `dimension` property. With an image in one dimension, the function `at(Image ima, index ind)` is defined. This function returns the value of `ima` associated with the point, which has `(ind)` as coordinate.

2.2 Image properties

Notation

A property has several values. The values of a property can be seen as a hierarchy. Hence, we need to define a notation to represent these hierarchies.

- category: primary means that primary is a value of the property category.
- category: any
 - |
 - +--- primary
 - |
 - +--- morpher

It means that `primary` and `morpher` inherit from `any`

2.2.1 Properties related to the value access

```
sw_io:  any
      |
      +--- read
      |
      +--- read_write
```

Table 2.1 – `sw_io` property values

The Milena images types have different kind of value access. Most of the time, the users access to the image values through the image sites. A *site* is a localized object in space. Points `1D`, `2D` or `3D` are the sites objects commonly used in the library. Every image type provides a

read value access through $\text{ima}(s)$ where s is a site. However, some images do not provide a write access through $\text{ima}(s)$. For instance, we can not modify the values of a constant image. This characteristic is expressed by the property sw_io that describes the read/write accessibility of the site wise access. So, sw_io (which stands for site wise io), can take the value *read* and *read_write* as expressed in [Table 2.1](#).

```

speed:  any
        |
        +--- slow
        |
        +--- fast
        |
        +--- fastest

```

Table 2.2 – speed property values

The efficiency of the site wise access is also an important information. The speed property gives some information about the time needed to access a value from a site through $\text{ima}(s)$ where ima is an image and s a site. If the speed property value is equal to *slow*, $\text{ima}(s)$ complexity is greater than $O(1)$. If this property is equal to *fast*, $\text{ima}(p)$ complexity is in $O(1)$. If this property is equal to *fastest*, $\text{ima}(p)$ complexity is in $O(1)$. Furthermore, in this case, ima has an extended domain (that notion is defined later), and ima values have a pointer semantics (it is possible to iterate directly on the image pixels). From a property point of view, if the speed property is set to *fastest*, this implies the $\text{value_organization}$ property to be set at *oneblock*, and the ext_domain property to be set at *some*. These two properties are described afterwards in this section.

```

vw_io:  any
        |
        +--- some
        |   |
        |   +--- read
        |   |
        |   +--- read_write
        |   |
        |   +--- global_read_write
        +--- none

```

Table 2.3 – vw_io property values

Some Milena images type are also accessible through their values. For instance consider the labeled image ([Figure 2.1](#)). This image is segmented into regions. All the points in a same region share the same value. As a result, there are fewer values than points in this image. So, if the users want to access to the image values, it is faster to access them directly (by iterating over the image values set) rather than access them through the sites of the image. However, the Milena images types are not always value wise accessible. It depends on the image implementation

details (how the values are stored in memory).

Thus, a property is needed to inform the users that an image is value wise accessible. This property, called `vw_io`, is described in Table 2.3. If `vw_io` is set to *none*, the associated image type does not provides any value access. If the image provides a readable (or writable) value access, its property `vw_io` is set to *read* (or *read_write*).

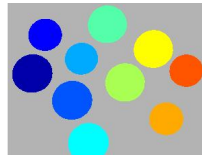


Figure 2.1 – Labeled image example

If the image provide a value wise access that allows the user to change all the image values through a function *value to value*, this property is set to *global_read_write*.

When an image is *value_wise* accessible, it can provide an access to two kinds of value set: the values taken set and the cells list.

The values taken set constitutes a mathematical set that contains the values currently taken by an image. So for all s , site in the image domain, $ima(s)$ belongs to the values taken set. For instance, the values taken set of the image described in Table 2.4 is **(R, G, B)**.

		R	
	G	G	
			B
B			

Table 2.4 – Image in two dimension

An image value wise accessible can also provide the cells list that composed the image. A cell is a location in RAM or in a file that stores an image value. For instance, the cells list of the image described in Table 2.4 is **(R, G, G, B, B)**. The cell list is not a mathematical set. Indeed, in cells lists a member can be duplicated. In the previous example, the **B** value is present twice in the cells list.

`vw_set` (Table 2.5) property precises what kind of value access is provided by an image type. If an image type does not provide any value access (`vw_io` set to *none*), `vw_set` is set to *none*. Otherwise, `vw_set` is set to *uni* if the image provides an access to value taken set. This set is accessible through the function `values_taken(ima : Image) : value_taken_set`. If `vw_set` is set to *multi*, the image provides an access to the values taken in memory (cells). The cells list is accessible through the function `cells(ima : Image) : cells_list`.

```

vw_set: any
|
+---- some
|   |
|   +---- uni
|   |
|   +---- multi
+---- none

```

Table 2.5 – vw_set property values

2.2.2 Properties related to the image organization in memory

Depending on their types, Milena images are stored in different ways in memory. Some images can be stored in a raw buffer in RAM whereas some images compress the size occupied by their values in memory. Hence we need properties that define the image memory organization.

```

value_access: any
|
+---- direct
|
+---- indirect
|
|
+---- computed

```

Table 2.6 – value_access property values

Image values can either be computed on the fly by a function or stored in memory. If an image type has a direct access on the values in memory, it is possible to take the address of the values in RAM. In this case, the `value_access` property (Table 2.6) is set to *direct*. In the other case, this property is set to *indirect*. When an image computes its values on the fly, this property is refined to *computed*.

```

value_storage: any
|
+---- memory
|   |
|   +---- singleton
|   |
|   +---- one_block
|   |
|   +---- piecewise
+---- disrupted

```

Table 2.7 – value_storage property values

Milena provides several ways to iterate over the image values. All the Milena image types

provide `piter` (Point ITERator). `piters` allow a user to run over all the points of an image (or more generally sites) and use `ima(p)` to access to the image values. However, if an image has a direct value access and is stored in a linear buffer in RAM, the user can use a `pixter` (PIXel ITERator) to access the values. The `pixters` iterate directly on memory by taking the address of the image values. Hence, they are faster than `piter`.

The property `value_organization` (Table 2.7) indicates how the values are stored in memory. If the image values are stored in a linear buffer, this property is set to `one_block`. If this property is set to `singleton`, only one value is stored in memory for the whole image (this is the case of the flat images). And, an image is *piecewise*, when its values are stored in several buffers in memory. If the values are not stored in the *memory*, this property is set to `disrupted`. Figure 2.2 illustrates this property.

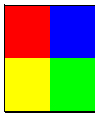

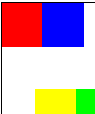



value_storage	Image	Storage in memory
one_block		
piecewise		
singleton		

Figure 2.2 – Values organization in memory

```
size: any
      |
      +--- regular
      |
      +--- huge
```

Table 2.8 – size property values

The `size` property (Table 2.8) gives an indication about the memory needed by an image. A *huge* image is an image which cannot be fully stored in the RAM. Accessing a *huge* image is slower than accessing a *regular* image. Indeed, to access to a *huge* image, we first need to load in the RAM a part of the image from another storage device.

```

category: any
  |
  +--- primary
  |
  +--- morpher
      |
      +--- domain_morpher
      |
      +--- value_morpher
      |
      +--- identity_morpher

```

Table 2.9 – category property values

Milena contains two categories of images, primary and morpher. Primary images are instantiated in the first place; they do not need any prior image type definition. Morpher images transform an image type into another one. So, a morpher image lies on another image (a morpher has a reference image in its attributes). Morphers are a non-intrusive way to add or modify some behaviors in an existing class. Primary images directly have their value data, while morphers use the values of their references images. Domain morphers only modify the domain (the set of points/sites composing the image) of the input image. Value morphers only change the input image values (cast the values into another type...) Identity morphers do not modify either the definition domain or the image values.

2.2.3 Properties related to the image geometry

```

dimension: any
  |
  +--- none
  |
  +--- some
      |
      +--- one_d
      |
      +--- two_d
      |
      +--- three_d

```

Table 2.10 – Dimension property value

The `dimension` (Table 2.10) property specifies the dimension of an image. An image in

Milena can either be in one dimension, two dimensions or three dimensions. An image can also have no dimension specified. This property is useful for image processing operator that only handles images in a specific dimension. It allows us to statically check that the input image satisfies the operator requirement.

```

localization:  any
               |
               + -- none
               |
               + -- space
                   |
                   + -- grid
                       |
                       + -- isotropic_grid
                       |
                       |   + -- basic_grid
                       |
                       + -- anisotropic_grid
  
```

Table 2.11 – localization property value

The `localization` (Table 2.11) property defines the underlying support of the images. The support of an image describes the image geometry and the relationship between its sites. An image on a non-localized space has this property set to *none*. An image on a localized space has this property set to *space*. An image based on a isotropic grid has this property set to *isotropic_grid*. An image based on a anisotropic grid has this property set to *anisotropic_grid*. An image based on an aligned and orthogonal grid has its property set to *basic_grid*.

If an image has its `localization` set to *space*, the user must define the neighborhood relationships between the sites. Otherwise, the image underlying grid defines the neighborhood relationships.

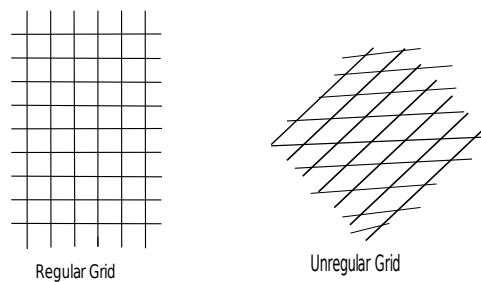


Figure 2.3 – Grid examples

The `value_alignment` (Table 2.12) property is between the properties defining the image

```

value_alignment: any
|
+---- with_grid
|
+---- not_aligned
|
+---- irrelevant

```

Table 2.12 – value_alignment property values

organization in memory and the image geometry. This property indicates if the image values are ordered in the same way in memory than when we iterate over them with an iterator on the image sites (`piter`). Hence, iteration on the image sites or on the image cells will give the same result. In this case, the image values are aligned with the image underlying grid, this property is set to *with_grid*. If the values are not ordered the same way in memory than the `piter` order, this property is set to *not_aligned*. If the image values are not stored in memory or the image does not have an underlying grid, this property is set to *irrelevant*.

2.2.4 Properties related to the image extended domain

```

ext_domain: any
|
+---- none
|
+---- some
|
+---- fixed
|
+---- infinite
|
+---- extendable

```

Table 2.13 – ext_domain property values

An extended domain grows the image domain by adding dummy values on the image border (see [Table 2.14](#)). It can increase the performances of some image processing algorithms. Indeed, a lot of image processing algorithms iterate on all the image sites, and then iterate over the neighbors of the current site. However, we need to check that the neighbors are included in the image domain (especially for sites which are located at the extremity of the image domain). With an extended border it is not necessary to test if the neighbors of an image site are included in the image domain. It speeds up algorithms using structuring elements. The size of the extended domain can differ (*fixed*, *infinite* or *extendable*). The property `ext_domain` ([Table 2.13](#)) indicates if the image has an extended domain.

As for the image values, we need properties to provide some information about the extended values organization in memory and the values access. `ext_value` property ([Table 2.15](#)) gives

1	1	1	1	1	1
1	1	1	1	1	1
1	1	12	2	1	1
1	1	2	15	1	1
1	1	1	1	1	1
1	1	1	1	1	1

The values equal to 1 represent the extended domain.

Table 2.14 – Image 2d with an extended domain of size 2.

some information about the extended values organization in memory. If all the extended domain sites share the same value (in memory), this property is set to *single*. If each site of the image extended domain has its proper value, this property is set to *multiple*. Otherwise, this property is set to *undefined*. Furthermore, `ext_io` property (Table 2.16) indicates the read/write accessibility of the image extended border values.

```
ext_value:  any
           |
           +--- undefined
           |
           +--- single
           |
           +--- multiple
```

Table 2.15 – `ext_value` property values

```
ext_io:  any
        |
        +--- some
        |   |
        |   +--- read
        |   |
        |   +--- read_write
        +--- none
```

Table 2.16 – `ext_value` property values

Chapter 3

Images in Milena

This chapter presents some of the Milena image types. This chapter does not aim at providing an exhaustive list of the image types. But, by listing some image types, we illustrate the properties defined in the previous chapter. This chapter also highlights the interface related to the properties by detailing the images types specificities.

3.1 Images in Milena

In the Milena library, an image can be seen as an application *site* to *value*. A *site* is a localized object in space. An image is composed by a set of localized objects (*sites*), which forms the definition domain of an image. A value is associated to each site of an image. This is the destination domain of the image. As we saw before, to access a value localized at the site s in an image named ima , we just use the mathematic notation: $ima(s)$.

All the image types are parameterized by different static parameters. In this report, we will use the following naming conventions for the image types parameters:

- T: represents an image *value* type.
- S: represents a type of a *sites set*.
- F: is a type of a function *site* to *value*.
- P: represents a *site* type.
- I: represents an *image* type.

3.2 Primary images

flat_image

`flat_image<S, T>` is defined by a domain d and a value v . All the sites included in the flat image domain d share the value v . Thus, a flat image stores only one value in memory for the whole image. Its `value_storage` property is set to *singleton*.

Since the flat images store only one value in memory, we can use the function

`value(ima : Image) : Value`. This function takes a flat image as argument and returns its associated singleton value. This function allows us to directly retrieve the value shared by the sites of a flat image, and update it.

image2d

`image2d<T>` is a rectangular image based on a 2d grid. The nodes of the image grid are points in two dimensions. `image2d<T>` is parameterized by `T`, the value type. All the image values are stored in a linear buffer (in the RAM). An extended domain is added to the image domain.

1	1	1	1	1
1	3	2	5	1
1	6	8	7	1
1	12	15	9	1
1	1	1	1	1

The values equal to **1** represent the extended domain.

Table 3.1 – Image in two dimensions

Consider `ima` which is an `image2d<T>`. `ima` is obviously in two dimensions (it has its `dimension` property set to `two_d`). Hence, to access to the value of `ima` localized at the point $p: (i, j)$, users can use `ima(p)`. But Milena also provides the routine `at(ima, i, j)` to access to the value localized at the coordinates (i, j) .

Furthermore, since all the values are stored in a linear buffer in RAM (`value_storage` and `value_access` properties are set to `one_block` and `direct`), it is possible to take the address of an image value in memory. Thus, several functions are defined to deal with the address aspect of the values. `buffer(ima)` returns the address at the beginning of the values buffer. `nelements(ima)` returns the number of elements stored in the buffer and `element(ima, i)` returns the value localized at the buffer `i` in the `ima` buffer. `ima` also defines pixels iterators that iterate directly on the values in memory. Therefore, pixel iterators are faster than site iterators.

Finally, this image has an extended domain (`ext_domain` is set to `some`). So some functions are also defined to handle the image border. For instance `border::size(ima)` returns the size of the extended domain. `border::fill(ima, v)` fills the image border with a value `v`.

`image1d<T>` and `image3d<T>` types are similar to `image2d<T>`. Only the dimension changes between these types.

sparse_image

A `sparse_image<P, T>` is composed by couples (r, v) where r is a run composed by points of type `P` and v a vector of values of type `T`. A run is a succession of points that share the same value. It is encoded by a point (the beginning of the run), and an integer (the length of the run). A value is associated to **each point** composing a run. A `sparse_image` has two data structures: a vector of run (`runs`), and a vector of “values vector” (`values`). [Table 3.2](#) describes the encoding of an `image2d`. `sparse_image` allows users to create image with few data and free form domain.

A vector of values is associated to each run. Due to this organization, `value_storage` and `values_access` properties of sparse images are respectively set to `piecewise` and `direct`. Several functions are defined to provide a direct access on the values vector composing a sparse image. `nruns(ima)` returns the number of values vector composing `ima`. `run(ima, i)` returns a pointer on the beginning of the i th values vector in `ima`. `run_size(ima, i)` returns the size of the i th value vector in `ima`.

image2d:	<table style="border-collapse: collapse; text-align: center;"> <tr> <td style="border: 1px solid black; padding: 2px 10px;">1</td> <td style="border: 1px solid black; padding: 2px 10px;">2</td> <td style="border: 1px solid black; padding: 2px 10px;">3</td> <td style="border: 1px solid black; padding: 2px 10px;"></td> <td style="border: 1px solid black; padding: 2px 10px;"></td> <td style="border: 1px solid black; padding: 2px 10px;"></td> </tr> <tr> <td style="border: 1px solid black; padding: 2px 10px;"></td> <td style="border: 1px solid black; padding: 2px 10px;"></td> <td style="border: 1px solid black; padding: 2px 10px;"></td> <td style="border: 1px solid black; padding: 2px 10px;">2</td> <td style="border: 1px solid black; padding: 2px 10px;">2</td> <td style="border: 1px solid black; padding: 2px 10px;">2</td> </tr> <tr> <td style="border: 1px solid black; padding: 2px 10px;"></td> <td style="border: 1px solid black; padding: 2px 10px;"></td> <td style="border: 1px solid black; padding: 2px 10px;"></td> <td style="border: 1px solid black; padding: 2px 10px;">4</td> <td style="border: 1px solid black; padding: 2px 10px;">5</td> <td style="border: 1px solid black; padding: 2px 10px;">8</td> </tr> </table>	1	2	3							2	2	2				4	5	8
1	2	3																	
			2	2	2														
			4	5	8														

The empty cases indicate that there is no data at these coordinates.

sparse_image:	<table style="border-collapse: collapse; text-align: center;"> <tr> <td style="border: 1px solid black; padding: 2px 10px;"></td> <td style="border: 1px solid black; padding: 2px 10px;">run</td> <td style="border: 1px solid black; padding: 2px 10px;">values</td> </tr> <tr> <td style="border: 1px solid black; padding: 2px 10px;">(0, 0),</td> <td style="border: 1px solid black; padding: 2px 10px;">3</td> <td style="border: 1px solid black; padding: 2px 10px;">1,2,3</td> </tr> <tr> <td style="border: 1px solid black; padding: 2px 10px;">(1, 3),</td> <td style="border: 1px solid black; padding: 2px 10px;">3</td> <td style="border: 1px solid black; padding: 2px 10px;">2,2,2</td> </tr> <tr> <td style="border: 1px solid black; padding: 2px 10px;">(2, 3),</td> <td style="border: 1px solid black; padding: 2px 10px;">3</td> <td style="border: 1px solid black; padding: 2px 10px;">4,5,6</td> </tr> </table>		run	values	(0, 0),	3	1,2,3	(1, 3),	3	2,2,2	(2, 3),	3	4,5,6
	run	values											
(0, 0),	3	1,2,3											
(1, 3),	3	2,2,2											
(2, 3),	3	4,5,6											

Table 3.2 – Image2d and its corresponding sparse encoded image

pvkeys_image

In `pvkey_image<P, T>` a site s is associated to a value v . The `pvkey_image` type is parameterized by two parameters: T , a value type and P , a site type. This kind of image uses in memory a `std::map` m with P as key type and V as value type. To access to the image values, we just have to access to the `std::map` ($ima(p) \rightarrow m[p]$).

At the same time, the `pvkey_image` coherently maintains another `std::map`. This map maintains couples $(v, \text{list of } p)$ where the value v is a key and p represents a site. This second map provides a direct access on the image values (`vw_io` is set to `read_write`). A key can have just one entry in a map, so a value can't be present twice in the second map table. As a consequence, `vw_set` is set to `uni`. Due to these properties, `pvkey_image` can use the function `values_taken`. `values_taken(ima)` returns the mathematical set of the value currently taken by `ima`.

For instance the value taken set of the following image is **(R, G, B)**.

R	R	R	R
R	G	G	R
R	R	R	B
B	R	G	G

3.3 Morphers

A morpher transforms a type into another one. It can be seen as an extension of the decorator design pattern. Morphers are a non-intrusive way to add or modify some behaviors in an existing class. This section only describes the `image_if` morpher.

image_if

The `image_if<I, F>` morpher restricts the input image domain to all the image sites which satisfy a condition expressed by a function “site to boolean”. It is parametrized by I , the image input type and F , the function type.

For instance, consider this 2d image:

21	42	21
51	21	42

We want to keep only one site over two in the 2d image. Hence, we define the function $f : \forall p$
 $f(p) \rightarrow (p.row + p.col) \bmod 2$ that express our condition. f is a function site to boolean. Thus,
 we keep only the sites that verify the f condition. The resulting `image_if` is:

$$\begin{array}{|c|c|c|} \hline 21 & & 21 \\ \hline & 21 & \\ \hline \end{array}$$

This morpher is useful to define masks on an image. A mask allows a user to consider only a specific region (a specific set of sites) of an image. All the morphers provide the `delegatee_()` that returns a reference on the referee image (the “unmorphed” image). That method is used in Milena internal functions.

Chapter 4

Routines

Low level routines perform basic operations on images. They provide the possibility to fill an image with a value, to transform the image values by applying a function. . . Furthermore, low level routines compose a significant part of the image processing chains. Indeed, they are used to initialize images data and to exploit results of image processing algorithms. Since, low level routines are frequently called, they must be as efficient as possible.

This chapter shows how the properties help in the routines optimization. It defines some of the Milena low level routines. And it presents their generic implementation and optimized specializations.

4.1 Properties implementation

4.1.1 Properties definitions

From an implementation point of view, properties are an extension of the traits mechanism. Every image type has an associated virtual type, which contains its properties declarations. This virtual type is called `image_`. So, every image type specializes the `image_` class in order to define its properties. [Table 4.1](#) details the properties declarations of the `image2d` class. As we saw before, an `image2d` is an image in two dimensions, based on a regular grid, where all its values are stored in the RAM.

```

template <typename T> // T is the image value type
struct image_< image2d<T> >
{
    // Value access related properties
    typedef trait::image::pw_io::read_write      pw_io;
    typedef trait::image::speed::fastest        speed;
    typedef trait::image::vw_io::none           vw_io;
    typedef trait::image::vw_set::none          vw_set;

    // Memory organization related property
    typedef trait::image::category::primary      category;
    typedef trait::image::size::regular          size;
    typedef trait::image::value_access::direct   value_access;
    typedef trait::image::value_storage::one_block value_storage;

    // Geometry related property
    typedef trait::image::localization::basic_grid localization;
    typedef trait::image::dimension::two_d      dimension;
    typedef trait::image::value_alignement::with_grid value_alignement;

    // Extended domain related property
    typedef trait::image::ext_domain::extendable ext_domain;
    typedef trait::image::ext_value::multiple   ext_value;
    typedef trait::image::ext_io::read_write    ext_io;
};

```

Table 4.1 – Definition of the image2d properties

4.1.2 Properties and optimization

Depending on their properties, images can provide different behaviors. For instance, an image with its `value_storage` property set to *singleton*, stores only one value for the whole image. This flat image can use the function `value` to retrieve its singleton value. However, the `value` function will not work with images that have a different `value_storage` property value. Indeed, specific behaviors depend on the implementation details of the image types.

These specific behaviors allow us to optimize a routine. For example, the `fill_with_value` routine sets all the values of an image to a given value. In the case of a flat image (`value_storage` set to *singleton*), it is faster to use the `value` function in order to update the "flat value", instead of iterating over all the image sites, and update the image value associated to each sites. Properties give information to know the specific behaviors provided by an image type.

Hence, several versions of a same routine are usually written to handle the different images specific behaviors. Some versions of the routine are highly efficient on a specific image type. Nevertheless, these versions do not work with all the image types. The image types do not necessarily provide the behavior required (`value` function can not work with the non flat images).

Users don't have to choose a specific version of a routine depending on the image types they used. A dispatch is done automatically at compile time to choose which version of the routine to call. This dispatch uses the facade design pattern and the overloading polymorphism.

For instance, [Table 4.2](#), [Table 4.3](#) and [Table 4.4](#) detail a part the implementation of the `fill_with_value`

routine.

```

template <typename I, typename V>
void
fill_with_value(Image<I>& ima, V v)
{
    fill_with_value_impl(mln_trait_image_value_storage(I)(), ima, v);
}

```

Table 4.2 – Fill with value facade

```

template <typename I, typename V>
void
fill_with_value_impl(trait::image::value_storage::singleton,
                    Image<I>& ima, V v)
{
    value(ima) = v;
}

```

Table 4.3 – Fill with value for flat images

```

template <typename I, typename V>
void
fill_with_value_impl(trait::image::value_storage::any,
                    Image<I>& ima, V v)
{
    mln_piter(I) p(ima.domain());
    for_all(p)
        ima(p) = v;
}

```

Table 4.4 – Generic fill with value

The dispatch function (Table 4.2) acts like a facade pattern and hides the implementation details to the users. It automatically calls the right `fill_with_value_impl` function depending on the input image type. Indeed, if the input image type `I` has its `value_storage` property set to `singleton`, it will automatically call the fill with value implementation specialized for flat images (Table 4.3). In the other case, it will call the fill with value generic implementation (Table 4.4). This dispatch uses the overloading polymorphism. `mln_trait_image_value_storage(I)()` retrieves the `value_storage` property value and passes it as the first argument of the `fill_with_value_impl` function. This dispatch is automatically done at compile time. Users just have to call the `fill_with_value` facade function.

4.2 Routines

This section details the different implementation provided for the low level routines in Milena. It presents the generic implementation of these routines, but also the optimized versions. This

section does not detail the dispatch process since it uses the mechanism described in the previous section.

4.2.1 Fill with value

Prototype: `fill_with_value(ima : Image, v : Value) : void`

As we saw before, `fill_with_value` aims at filling an image with a value `v`. This algorithm takes an image `ima` and a value `v` as arguments. It associates the value `v` to all the `ima`'s sites.

```
mln_piter p(ima.domain());
for_all(p)
  ima(p) = v;
```

Table 4.5 – Pseudo code of the `fill_with_value` routine (generic version)

The generic version, [Table 4.5](#), just iterates on the `ima` sites and affects the value `v` to each of them. Of course, the image must be site wise writable (`pw_io` must be set to `read_write`). However, depending on the `ima` properties, more efficient versions of the `fill_with_value` routine can be used.

```
memory_set(buffer(ima), v, size(ima));
```

Table 4.6 – Pseudo code of the `fill_with_value` routine (block version)

If the `value_storage` property of `ima` is set to `one_block`, `ima` stores its values in a linear buffer. Furthermore, if its property `value_access` is set to `direct`, we can take the addresses in RAM of the values. Hence, we can use `buffer(ima)` to get the beginning address of the `ima` buffer. With this address, we can directly use a memory set to fill the image value. `size(buffer)` sends back the size of the `ima` buffer. Of course, `ima` must be writable (`pw_io` or `vw_io` set to `read_write`).

```
value(ima) = v;
```

Table 4.7 – Pseudo code of the `fill_with_value` routine (singleton version)

In case of flat images, all the sites share the same value. This kind of image has the `value_storage` property set to `singleton`, only one memory cell is used to store the value shared by all the sites. So, we can use the `value` routine that returns a reference on the flat value cell. Hence, with flat image the `fill_with_value` routine just requires one affectation. Note that `ima` must be writable.

In case of images with `vw_io` set to `read_write`, we can directly iterates on the image values rather than on the image sites. Iterate on the image values is faster than iterate on the image sites, they are less value than sites (see [Figure 2.1](#) for instance). Hence, in case of value wise

```

mln_viter viter(values_taken(ima));
for_all(viter)
  viter.change_to(v)

```

Table 4.8 – Pseudo code of the fill_with_value routine (cell wise version)

accessible image we declare an iterator on values `viter`, and we set all the values to `v` through this `viter`.

4.2.2 Paste

Prototype: `paste(input : Image, output : Image) : void`

The paste routine takes two images as argument, input and output. It pastes the image values of input into the output image. Thus, the domain of input must be included in the domain of output.

Consider these two images:

	1	2	3	4
1	R	R	R	B
2	G	G	G	B
3	B	B	B	B
4	B	G	G	G

ima1:

	1	2	3	4	5	6
1	R	R	R	G	B	B
2	R	R	R	G	B	B
3	R	R	R	G	B	B
4	R	R	R	G	G	G
5	R	R	R	G	G	G

ima2:

If we want to paste `ima1` into `ima2`, we will obtain:

	1	2	3	4	5	6
1	R	R	R	B	B	B
2	G	G	G	B	B	B
3	B	B	B	B	B	B
4	B	G	G	G	G	G
5	R	R	R	G	G	G

ima2:

However, it is not possible to paste the value `ima2` into `ima1`. Indeed, the domain of `ima2` is larger than the domain of `ima1`.

```

piter p(input.domain())
for_all(p)
  output(p) = input(p)

```

Table 4.9 – Pseudo code of the paste routine (generic version)

The generic version (Table 4.9) iterates over the domain of the input image. For each sites `p` of the input image, it copies the values `input(p)` into `output(p)`. Hence, the output image must be site wise writable (`pw_io` set to `read_write`).

In order to increase performances, the pixter version of the paste routine (Table 4.13) uses pixel iterators instead of site iterators. But to provide pixel iterator, input and output must

```

mIn_pixter pi(input);
mIn_pixter po(output);
for_all2(pi, po)
    po.val() = pi.val();

```

Table 4.10 – Pseudo code of the paste routine (pixter version)

satisfied requirements. Their `value_access` and `value_storage` properties must be respectively set to `direct` and `one_block`. Furthermore, the values of the images `input` and `output` must be ordered the same way in memory. Indeed, `po` and `pi` must represent values which share the same location at each iteration step. So the `value_alignement` property of the two images must be set to `with_grid`. Finally, the output image must be writable.

```

memory_copy(buffer(input), buffer(output), input.size())

```

Table 4.11 – Pseudo code of the paste routine (memory_copy version)

The `memory_copy` paste specialization (Table 4.11) is even faster than the `pixter` version. Indeed, the underlying implementation uses `std::memcpy`. But `input` and `output` images must satisfy stronger requirements. First, as in the `pixter` version, the `memory_copy` version accesses to the images values cells (address in memory). Hence, `value_access`, `value_alignement` and `value_storage` of `input` and `output` must be respectively set to `direct`, `with_grid` and `one_block`. Furthermore, the size taken in memory by the input value type must be the same than the output value type. The domain of `input` must be equal to the domain of `output`. Finally, `output` must be writable.

```

mIn_line_piter p(input.domain());
for_all(p)
    memory_copy(pixel(output, p), pixel(input, p), p.run_length());

```

Table 4.12 – Pseudo code of the paste routine (copy per line)

If the input domain is not equal but inferior than the output domain, it is not possible to use the `memory_copy` paste version. But, since the value are organized the same way in memory (`value_organization` is set to `with_grid`), we can do memory copy line per line. In the copy per line version, Table 4.13, we declare a `line_piter`. A `line_piter` is a special iterator that goes on a point representing the beginning of a new line at each iteration step. With this iterator, we do a memory copy from the pixel which is at the beginning of the line (`pixel(input, p)`), to the pixel at end of the line (the length of the line is given by `p.run_length()`).

To use this specialization, the `input` and `output` must satisfy the same requirements than for the `memory` paste version, except for the images domain equality.

When the `input` is a flat image (`value_storage` set to `singleton`), the `paste` routine is equivalent to a `fill_with_value` routine. Indeed, it just needs to fill `output` with the flat value of `input`. `output | input.domain()` restricts the output images to the domain of `input`.

```
level :: fill_with_value(output | input.domain(), value(input));
```

Table 4.13 – Pseudo code of the paste routine (singleton version)

4.2.3 Fill with image

Prototype: `fill_with_image(ima : Image, data : Image) : void`

The `fill_with_image` routine takes two images as argument, `ima` and `data`. It fills the image values of `ima` using the `data` image.

For instance, consider these two images:

	1	2	3	4
1	R	R	R	B
2	G	G	G	B
3	B	B	B	B
4	B	G	G	G

	1	2	3	4	5	6
1	R	R	R	G	B	B
2	R	R	R	G	B	B
3	R	R	R	G	B	B
4	R	R	R	G	G	G
5	R	R	R	G	G	G

If we want to fill `ima` from `data`, we will obtain:

	1	2	3	4
1	R	R	R	G
2	R	R	R	G
3	R	R	R	G
4	R	R	R	G

```
mIn_piter p(ima.domain())
for_all(p)
  ima(p) = data(p)
```

Table 4.14 – Pseudo code of the paste routine (generic version)

However, it is not possible to fill `data` with `ima`. Indeed, the domain of `data` is larger than the domain of `ima`. The generic implementation (Table 4.14) of the `fill_with_image` routine is really close to the paste routine. In this implementation, a site iterator is declared on `ima`. Then this iterator is used to affect to `ima` the value of `data`. The optimizations used for the `fill_with_image` routine are the same than the paste routine. Hence, this section does not detail these optimizations.

4.2.4 Clone

Prototype: `clone(ima : Image) : Image`

The clone routine takes an image `ima` as input, and returns a clone of `ima`. The generic version of the algorithms described at Table 4.15, first declare the clone image. Then it initializes it, the clone image must have the same dimensions than `ima`. Finally, it pastes `ima` content in the clone

```

ima_result_type clone;

// Initialize clone
initialize(clone, ima);
// Clone ima values
paste(ima, clone);

```

Table 4.15 – Pseudo code of the clone routine

image through the `paste` routine. Since all the optimizations are done in the `paste` routine, we do not need to optimize the clone routine.

4.2.5 Transform

Prototype: `transform(input : Image, f : Function) : Image`

The transform routine takes an image input and a function `f` as argument. It returns an output image. Transform routine uses the function `f` on the input image to transform it, output image is the resulting transformation. Therefore, `f` must be a function *value to value*, to transform the image input image values into the values of the output images.

```

mln_piter p(input.domain());
for_all(p)
    output(p) = f(input(p))

```

Table 4.16 – Pseudo code of the transform routine (generic version)

In the generic transform algorithm (Table 4.16), we simply apply the function `f` to all the values of the input image, in order to construct the output image. So, for each site of the input image we apply the function `f`. The output image has to be site wise writable. The transformation function `f` can be CPU time consuming. Hence, limits the calls to the `f` function can improve the efficiency of the transform routine.

```

lut_table lut; // Look-up table
v = values(input); // Set of values that input can possibly take
for_all(v)
    lut(v) = f(v) // Stores the result of f(v) in a temporary table
mln_piter p(input.domain());
for_all(p)
    output(p) = lut(input(p)) // Use the lut table to construct output

```

Table 4.17 – Pseudo code of the transform routine (low quantified version)

The Milena library provide value types like `rgb` types, gray level types. . . All these types have a `quant` property. This property informs us about the number of bits needed to code a value. By extension, we will consider that images have a `quant` property which is the `quant` property of their associated value type.

If an image has its `quant` property set to `low_q`, its image values are coded on less than 12 bits. In this case, the function `f` is called on every value that the image can possibly take and the result is stored in a look-up table. The look-up table is a temporary table that associates to each value `v` its associated transformed value `f(v)`. This look-up table is then used to transform the input values into the output image (Table 4.17).

```
lut_table lut; // Look-up table
v = values_taken(input); // Values currently taken by input image
for_all(v)
    lut(v) = f(v); // Stores the result of f(v) in the lut table
mIn_piter p(input.domain());
for_all(p)
    output(p) = lut(input(p)) // Use the lut table to construct output
```

Table 4.18 – Pseudo code of the transform routine (values taken version)

The low quantified version of the transform routine uses a look-up table to store the result of the application of `f` on every value that input can possibly take. However, if input has its `vw_io` property set to `read` or `read_write` and its `vw_set` property set to `uni`, we can access to the values currently taken by input through the method `values_taken`. Indeed, `values_taken` function returns all the values currently taken by the image. So, in Table 4.18, the `lut` table stores only the result of the application of `f` on the values currently taken by input (and not for the values that input can possibly take).

```
v = f(value(input))
fill_with_value(output, v)
```

Table 4.19 – Pseudo code of the transform routine (singleton version)

If input has its `value_storage` property set to `singleton`, input is composed by only one value. So, the transform routine just transforms, with `f`, the flat value of the input image. Then, it fills the output image with the transformed value (Table 4.19).

```
pixter pi(input);
pixter po(output);
for_all2(pi, po)
    po.val() = f(pi.val);
```

Table 4.20 – Pseudo code of the transform routine (fast version)

As usual, if the input image has its `value_storage` property set to `one_block` and its `value_access` set to `direct`, we can use a pixel iterator in rather than a site iterator in order to improve the routine efficiency (Table 4.20).

```

lut:                // Look-up table
v = values(input); // Set of values that input can possibly take
for_all(v)
  lut(v) = f(v)
pixter pi(input);
pixter po(output);
for_all2(pi, po)
  po.val() = lut(pi.val);

```

Table 4.21 – Pseudo code of the transform routine (fast low quantified version)

We can also use a look-up table in the fast transform version, if the input image has its quant property set to *low_q* (Table 4.21).

4.2.6 Dependencies between low level routines

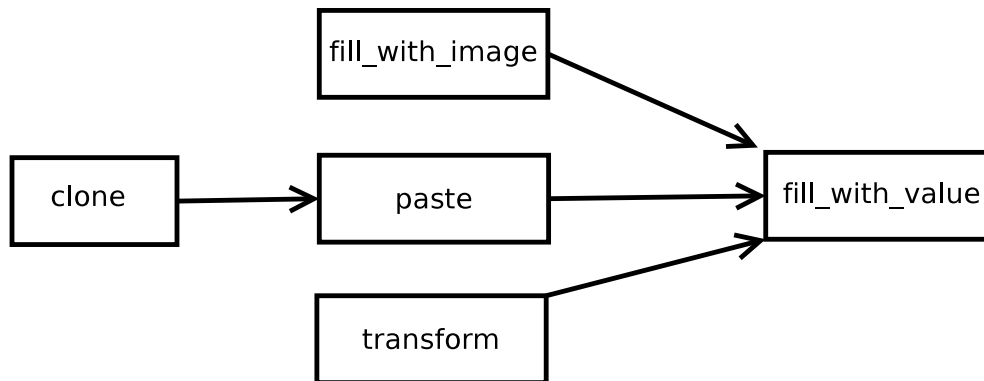


Figure 4.1 – Dependencies between low level routines

Figure 4.1 resumes the dependencies between the different low level routines. The arrow relation means that a routine depends on another routine. Thus, here we can see that *fill_with_image*, *paste* and *transform* depends on *fill_with_value*. Each routine, *fill_with_image*, *paste* and *transform* routines, has a version that calls the *fill_with_image*. Furthermore, since the *clone* generic implementation calls the *paste* routine, *clone* directly depends on *paste*.

Since the *clone* routine directly depends on *paste*, there is no need to optimize it. All the optimizations are done in the *paste* or *fill_with_value* routine. In a same way, some versions of *fill_with_image* and *transform* rely on optimizations present in the *fill_with_value* routine.

4.2.7 Low level routines performances

The paradigm used in Milena for writing routines implies an overhead cost from the developer point of view. Indeed, the developer needs to write a facade, dispatch functions and

specialized versions of a routine, depending on the input image properties.

However, this paradigm also implies some improvements in the routine execution time (compared to the generic version). Routines benefit from the Milena paradigm and are more performant than the generic versions (for some image types).

Figure 4.2 compares the execution time between a generic version of a routine, and the facade that uses a dispatch to automatically execute the code which adapted to the input image type. This test has been performed on a 1.66GHz with some `image2d` of size 10000 x 10000. Since `image2d` is a type frequently used, it is important to have algorithms as efficient as possible on this type.

Algorithm	Version	Total Time (second)
<code>fill_with_value</code>	generic	1.83
<code>fill_with_value</code>	facade	0.02
<code>transform</code>	generic	4.08
<code>transform</code>	facade	2.12
<code>paste</code>	generic	3.25
<code>paste</code>	facade	0.38

Figure 4.2 – Low level routines performances

Chapter 5

Conclusion

This report was the opportunity to give an overview of the properties used in Milena. Through this report we presented the different properties relative to the Milena image types. We saw that the properties allow users to do static checking, and to optimize some algorithms. We illustrated the properties by presenting the implementation of the low level routines in Milena.

Properties are an interesting way to provide performance and genericity. With their declarative aspect, they inform us about the requirements needed by the input type of an algorithm, at the compile time. Furthermore, they also provide information on the image types specificities and implementation details at the compile time.

Thus, properties provide a way to deal with genericity problems at the compile time and avoid a loss of performance.

Acknowledgment

Thanks to Thierry Géraud, Roland Levillain, Guillaume Lazzara, Florian Lessaint and Matthieu Garrigues for their help.

Chapter 6

Bibliography

Ballas, N. (2008). Image taxonomy in milena. *LRDE report*.

Berger, C. (2006). Image taxonomy in olena. *LRDE Seminar*.

Garrigues, M. (2008). Stage de traitement d'image au lrde. *LRDE report*.

Odou, S. (2005). An introduction to image modeling. *LRDE Seminar*.

Thierry Géraud, Roland Levillain (2008). Semantics-driven genericity: A sequel to the static c++ object-oriented programming paradigm (scoop 2). *6th International Workshop on Multi-paradigm Programming with Object-Oriented Languages*.

Index

Image: flat_image, 16
Image: image2d, 17
Image: image_if, 18
Image: pvkeys_image, 18
Image: sparse_image, 17

Milena, 4

Property, 6
Property: category, 10
Property: dimension, 12
Property: ext_domain, 14
Property: ext_io, 14
Property: ext_value, 14
Property: localization, 13
Property: size, 11
Property: speed, 8
Property: sw_io, 7
Property: value_access, 12
Property: value_alignment, 13
Property: value_storage, 10
Property: vw_io, 8
Property: vw_set, 9

Routine, 20
Routine: clone, 26
Routine: fill with image, 26
Routine: fill with value, 23
Routine: paste, 24
Routine: transform, 27