Image taxonomy in Milena

Nicolas Ballas

Technical Report n^o0812, March 2008
revision 1876

Milena is the generic image processing library of the Olena platform. The library aims at remaining simple while providing high performance. The introduction of new image types based on graphs has revealed some design problems limiting its genericity. For instance, we have always considered that images are based on a set of points; yet some images have sites that are not points (but edges, facets, and even sets of points). Another error was to consider that sites are localized by a vector (e.g., $\vec{xy}$ in the 2D plane), which cannot be true when sites are not point-wise. Therefore there was a need to reconsider the image types and their underlying image properties. In this report, we present a new image taxonomy that solves those issues.

Milena est la bibliothèque de traitement d'images générique de la plate-forme Olena. Cette bibliothèque a pour but d'être performante tout en restant simple. L'introduction dans Milena de nouveau types d'images basés sur des graphes a mis en évidence des problèmes de modélisation qui sont un obstacle pour sa généricité. Par exemple, nous avons toujours considéré que les images repose sur un ensemble de points. Néanmoins, certain types d'images possèdent des sites qui ne sont pas des points (mais des arrêtes, faces, ou même des ensemble de points). Une autre supposition incorrecte était de considérer que les sites étaient toujours localisés par un vecteur (cad, $\vec{xy}$ dans le plan 2D). Cette supposition est fausse lorsque l'on manipule des sites qui ne sont pas basés sur des points. Il etait donc nécessaire de modifier les types d'images utilisés Milena et les propriétées qui leur sont associés. Dans ce rapport, nous présentons une nouvelle classification d'images permettant de resoudre ces problèmes.

Keywords
Software engineering, Image processing, Taxonomy, Milena, Olena

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
nicolas.ballas@lrde.epita.fr – www.lrde.epita.fr

Copying this document

Copyright © 2008 LRDE.

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**
- 1.1 The Milena library . . . 4
- 1.2 Need to reconsider the image taxonomy . . . 4

2 Milena paradigm **6**
- 2.1 Image data access: Site and Psite concept: . . . 6
- 2.2 Image properties and Milena . . . 7
  - 2.2.1 Definition . . . 7
  - 2.2.2 Purposes of the properties . . . 7
  - 2.2.3 Implementation . . . 8

3 Image properties **12**
- 3.1 Notation . . . 12
- 3.2 Properties defining the image . . . 12
  - 3.2.1 category . . . 12
  - 3.2.2 size . . . 13
  - 3.2.3 data . . . 13
- 3.3 Properties related to the image localization . . . 14
  - 3.3.1 localization . . . 14
  - 3.3.2 space . . . 14
- 3.4 Property related to the data access . . . 15
  - 3.4.1 value_io . . . 15
  - 3.4.2 value_access . . . 16
  - 3.4.3 speed . . . 16
- 3.5 Properties related to the extended domain . . . 17
  - 3.5.1 ext_domain . . . 17
  - 3.5.2 ext_value . . . 17
  - 3.5.3 ext_io . . . 18

4 Images taxonomy: primary image **19**
- 4.1 Image nD . . . 19
- 4.2 Images based on function . . . 20
- 4.3 Images encoded by runs . . . 20
  - 4.3.1 RLE Image . . . 20
  - 4.3.2 Sparse Image . . . 21
  - 4.3.3 Value encoded image . . . 21
  - 4.3.4 Properties . . . 22
- 4.4 graph_image . . . 22

4.5 pkey_image . . . 23
4.6 Flat_image . . . 23

5 Images taxonomy: morphers 25
5.1 Value morphers . . . 25
5.1.1 f_image . . . 25
5.1.2 f_images . . . 26
5.1.3 lut_image . . . 26
5.2 Domain morphers . . . 26
5.2.1 sub_image . . . 26
5.2.2 hexa . . . 26
5.2.3 image_if . . . 27
5.2.4 translate_image . . . 27

6 Conclusion 28
6.1 Acknowledgment . . . 28

Chapter 1

Introduction

In 2007, the Olena project went through some major changes. The generic image processing library Olena abandoned the SCOOP2 programming paradigm and now uses a simpler paradigm. Due to this change, the library was renamed Milena. Milena aims at being both generic and simple. The introduction of new image types in Milena reveals some design problems. This report presents a new image taxonomy, which solves these design issues.

1.1 The Milena library

Milena is the generic image processing library of the Olena platform [1] developed at the LRDE [2] (EPITA [3] 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.

1.2 Need to reconsider the image taxonomy

In Milena, the addition of new image types reveals some design problems. We always consider are based on a set of points. However, this is not always true. For instance, an image representing a delauney triangulation is accessible through facets rather than points (see Figure 1.1).

This design error implies the modification of the Milena concepts, therefore a lot of code refactoring. Thus, there was a need to reconsider the Milena image types. Establishing all the

1. http://www.lrde.epita.fr/cgi-bin/twiki/view/Olena/WebHome
2. www.lrde.epita.fr
3. www.epita.fr

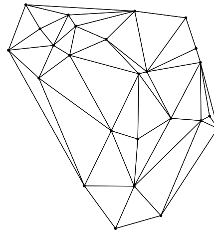

Figure 1.1 – Delauney triangulation

image types' specificities can help us deducing correct concepts, thus avoid a lot of code refactoring, so save some time.

Taxonomy is a proper way to list all the image types. Taxonomy is the classification of organisms in an ordered system that indicates the natural relationship. So, with taxonomy, we can regroup the image types according to their particularities. Some studies have already been made on this subject. In 2006, Christophe Berger presents a taxonomy of the image types in its report **?**. However, the presented taxonomy is related to the SCOOP2 paradigm and assumes that "images have points".

The image taxonomy presented in this report will handle the new design problems. We will also describe the image properties associated with each image type. Indeed, properties are a major part of Milena. Thus, before presenting the Milena image taxonomy, we need to understand the Milena programming paradigm and the importance of the properties.

Therefore, we will review the Milena programming paradigm. Then, we will describe the images properties. Finally, we will present the taxonomy of the Milena image types.

Chapter 2

Milena paradigm

This chapter gives a quick overview of the Milena paradigm. First, it describes the image data access (through the site and the psite concept). This chapter also explains the importance of the properties in Milena.

2.1 Image data access: Site and Psite concept:

From a mathematical point of view, an image is just a function: $point$ to $value$. Thus, it seems logical to say that the image values are accessible through points. However, this is not always the case. In introduction, we have seen that delauney triangulation images provide value access with facets rather than points. For these reasons, Milena introduces the concept of site. The site concept is a generalization of the point concept. A site is a localized object in space. It can be a point, but also an edge, a facet or a segment, etc. With this concept, Milena images can provide value access with facets.

Withal, sites only have a localization aspect. Hence, accessing a value, with a site, in a constant time complexity is not always possible (see the following example). That's why the psite concept was introduced. A psite is a generalization of the site concept. It was introduced to assure that the value access time is in a $O(1)$ complexity. A psite is a site with new information that deals with an image type implementation details. So, a psite has a localization aspect. It can be casted into a site. Milena image types provide value access through psites. When a site type allows an image type to provide a value access in a constant time complexity, the psite type associated to the image is the image site. So, most of the time, the psites are just points. But generalization is a requirement for handling special cases (like image compressed in memory, etc). The name "psite" means "point site". It came from the previous version of the Milena library. In these versions, the psite were just used to speed up the access of types. Now, the psites speed up the value access from any site types.

For instance, consider a 2d image compressed in memory with runs. A run is a succession of points that share the same value. It is represented by a point (the start of the run), and an integer (the length of the run). Here, we want to access the image 2d through its points, so the site type associated to this image type is point2d. A compressed image doesn't store its points in memory, it just has a vector of runs (see Figure 2.1). Therefore, a site must find its corresponding run to access a value. So, we must compare the site to each run composing the image. If the site is included into a run, we can get its corresponding value. However, this access is not in a $O(1)$ complexity. To access to the compressed image values in a constant time complexity, we must

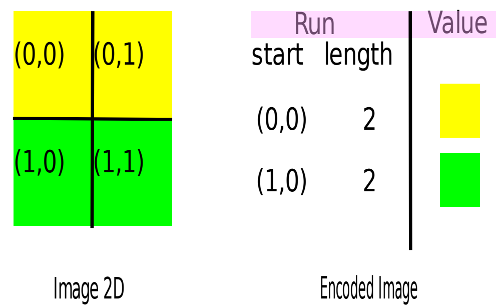

Figure 2.1 – Rle image example

have extra information. Consider these two integers:

1. the index of the run inside the image.
2. the index of the point inside the run.

With these indexes, we can directly access to a run and get its values. So, a psite of this compressed image type is a point2d with these two integers.

Milena image types also provide iterators which go through all the sites composing in the image. The set of sites (also called site set) composing the image can be assimilated to the definition domain of the image.

2.2 Image properties and Milena

Properties are a way to classify classes. They are a major part of the library. Indeed, the Milena library is property driven; they provide a way to do static checking, to specialize an algorithm and even to retrieve automatically some implementations. The goal of this report is to establish the taxonomy of the Milena image types, so we will only consider the properties related to the images.

2.2.1 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. However image types can have different values for a property. For instance, the data access in an image can be $slow$, $fast$ or the $fastest$. All of these values merge into the property $speed$, which informs us about the image data access efficiency.

The values of a property form a hierarchy. In the previous example, an image $fastest$ is an image $fast$. Hence, there is an inheritance relation between $fast$ and $fastest$.

2.2.2 Purposes of the properties

Properties provide three functionalities which are static checking, algorithm specialization and interface or implementation inheritance.

- 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 types 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 `space` 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 which work with all the image types and one specialized version which is optimized for a specific image type. 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.

- Interface or implementation inheritance
  Milena is property driven. Indeed, image types can recover some piece of interface or implementation depending on their properties.
  For example, the dimension property gives information about the image dimension. `image2d` has its `dimension` property set to *two_d*. All its values are localized on a 2d plan. So, `image2d` is able to provide a value access with two coordinates representing a row and a column. Thus, `image2d` automatically gets the method `Value at(coord x, coord y)` which returns the value localized at the point (x, y). This behavior is different for an `image3d`. An `image3d` has its dimension property set to *three_d*. So, an `image3d` automatically gets the method `Value at(coord x, coord y, coord z)` which returns the value localized at the point (x, y, z).

2.2.3 Implementation

Property definition

From an implementation point of view, properties are an extension of the traits mechanism. Every image type has a 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.
The Table 2.1 describes the declaration of the properties for the `image2d`. The meaning of these properties will be explained in the next chapter.

Static Checking

To do static checking, we must check that a property is set to value at compile time.
The Table 2.2 verifies that the `space` property of the image type `I` is set to the value `three_d` with the help of `mlc_equal`. Indeed, `mln_trait_image_space(I)` returns the values of the `space` property associated to the type `I`. Furthermore, `mlc_equal` checks that `mln_trait_image_space` is equal to the type `mln::trait::image::space::three_d`. Hence, it checks that the input image type of the function process is in three dimensions.

```
template <typename T> // T is the image value type
struct image_< image2d<T> >
{
  typedef trait::image::category::primary          category;
  typedef trait::image::size::regular              size;
  typedef trait::image::data::raw                  data;
  typedef trait::image::localization::regular_grid localization;
  typedef trait::image::space::two_d               space;
  typedef trait::image::value_io::read_write       value_io;
  typedef trait::image::value_access::none         value_access;
  typedef trait::image::speed::fastest             speed;
  typedef trait::image::ext_domain:fixed           ext_domain;
  typedef trait::image::ext_value:multiple         ext_value;
  typedef trait::image::ext_io:read_write          ext_io;
};
```

Table 2.1 – Definition of the image2d properties

```
template <typename I>
void process(I& ima)
{
  mlc_equal<mln_trait_image_space(I),
  mln::trait::image::space::three_d >::check()
  // Process
}
```

Table 2.2 – Static checking example

Algorithm specialization

Algorithm specialization allows users to write a specific version of an algorithm which is more efficient on an image type. For instance, we want to write a function iter which iterates over all the sites of an image type. So, we must use the Milena sites iterators. Milena provides several kinds of sites iterators.

`piter` is the basic site iterator. It is provided by all the image types. Hence, with `piter`, we can write a generic version of the iter algorithm. Table 2.3 describes its implementation. The `iter_` function takes an argument which has `trait::image::speed::any` as type. `trait::image::speed::any` is the speed property value. It means that the algorithm works with all the image types. `mln_piter` is a macro function which returns the iterator type associated to the input image type. `ima.domain()` return the set of sites composing ima.

If the input image type has its `speed` property set to `fastest`, it provides pixel iterators which are faster than site iterators. Indeed, pixel iterators iterate directly on the image value in memory instead of iterating on the image sites. However, pixels iterators are not provided by all the image types.

Table 2.4 describes an implementation of `iter` for the `fastest` images. The input image must have their speed property set to `trait::image::speed::fastest`, to use this implementation.

Now, we need a function to choose automatically the correct algorithm version Indeed, a Milena user must not do the selection by hand. Table 2.5 details this function.

```
template <typename I>
void
iter_(trait::image::speed::any, I& ima)
{
  mln_piter(I) p(ima.domain()); // Create an iterator over the ima
                                // sites
  do_something(p);
}
```

Table 2.3 – Generic iter example

```
template <typename I, typename W>
void
iter_(trait::image::speed::fastest, I& input);
{
  mln_pixter(I) p(ima); // Create an iterator over the ima pixels
  do_something(p);
}
```

Table 2.4 – Fast iter example

This function acts like a facade pattern, and hides the implementation details to the user. It automatically calls the right `iter_` method depending on the value of the image speed property. `mln_trait_image_speed(I)()` retrieves the `speed` property value, and pass it as the first argument of the `iter_` function.

```
template <typename I, typename W>
void
iter(const I& input)
{
  iter_(mln_trait_image_speed(I)(), input);
}
```

Table 2.5 – Facade function example

Interface or implementation inheritance

Interface inheritance and implementation inheritance are implemented in Milena through selector. A selector is a small C++ meta-code which will choose from which class it inherits at compile time depending on its associated properties. The Figure 2.2 describes the UML schema associated to a selector. In this example, the selector will inherit from `inter_image_2d` if its space property is set to `two_d` (the image is in two dimensions). In the other case, it will inherit from `inter_image_3d`.

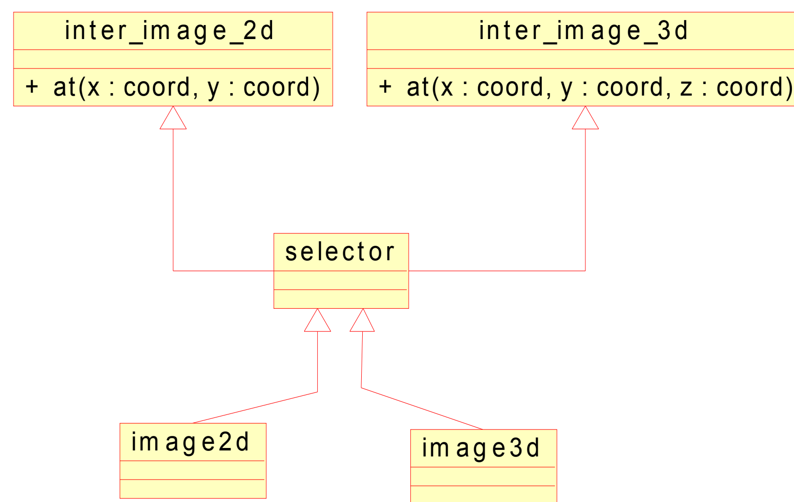

Figure 2.2 – Selector example

Chapter 3

Image properties

Before describing the taxonomy, we need to define the image properties. Indeed, properties are important factors to classify the images. So, this chapter gives an overview of all the properties used in Milena. We will present and define each property, and, we will describe all their related values.

3.1 Notation

As we saw before, a property has several values. Furthermore the values of a property can be seen as a hierarchy. Hence, we need to define a notation to represent these hierarchies. We will use the following notation.

– `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`

3.2 Properties defining the image

3.2.1 category

Property Values

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

```
                    |
                    +--- identity_morpher
```

Definition

Milena has two main image categories: **primary** and **morpher**.

Primary images are instantiated in the first place; they don't need any prior image type definition. `image2d` is an example of primary image (see section 4.1).

Morpher images transform an image type into another one. Morphers are a non-intrusive way to add or modify some behaviors in an existing class. Thus, to be instantiated, a morpher requires an input image. The `f_image` is an example of morpher type (see subsection 5.1.1). 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 don't modify either the definition domain or the image values.

3.2.2 size

Property Values

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

Definition

The size property 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.

So, with this property, we can handle huge images in a different way in the algorithms. For instance, with huge image, it is faster to do in place modification rather than create a temporary image.

3.2.3 data

Property Values

```
data: any
       |
       +--- stored
       |       |
       |       +--- linear
       |       |
       |       +--- raw
       |
       +--- computed
```

Definition

Image values can either be computed on the fly by a function or stored in memory. If the image values are stored in memory, it is possible to take a reference of them.

This property indicates how the image computes its values. If all the values are stored in RAM (using a linear buffer), this property is set to *raw*. If the values are stored both in RAM and on some other storage devices (like a hard drive), this property is set to *linear*. If the values are stored in memory, but not in a linear way, this property is set to *stored*. It is possible to take a reference of all the values stored in memory. If the image uses a function to compute the value this property is set to *computed*.

3.3 Properties related to the image localization

3.3.1 localization

Property Values

```
localization: any
               |
               +--- none
               |
               +--- space
                     |
                     +--- grid
                            |
                            +--- regular_grid
```

Definition

This property defines the underlying support of the images. The support of an image describes the image geometry and the relationship between its sites (see Figure 3.1). 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 grid (which is not regular) has its property set to *grid*. An image based on a aligned and orthogonal grid has its property set to *regular_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.

3.3.2 space

Property Values

```
dimension:  any
             |
             +--- none
             |
             +--- some
                   |
                   +--- one_d
```

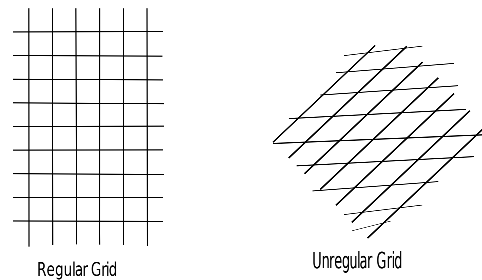

Figure 3.1 – Grid examples

```
|
+--- two_d
|
+--- three_d
```

Definition

This property specifies the dimension of the image. An image in Milena can either be in one dimension, two dimensions or three dimensions. An image can also have no dimension specified.

3.4 Property related to the data access

3.4.1 value_io

Property Value

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

Definition

This property describes the read/write accessibility of the image values. An image can be read only (`value_io` is set to *read*) or readable and writable (`value_io` is set to *read_write*). Image based on function (`data` property is set to *computed*) are forced to be read only. Indeed, the values are computed on the fly by a function. They are not stored in memory, hence we can't modify them.

3.4.2 value_access

Property Value

```
io:  any
      |
      +--- none
      |
      +--- value_wise
      |         |
      |         +-------------+
      |                       |
      +--- cell_wise          +--- value_and_cell_wise
                |             |
                +-------------+
```

Definition

All the image values are accessible through its set of points (or sites) composing the image. Furthermore, some image types provide an access to the values domain (values that the image can possibly take) or even to the cells (values currently taken by the image pixels).

For instance, *cell_wise* accessible images provide the method `T& cell(T val)`. This method allows a user to change a value shared by a set of pixels in a constant time complexity. `cell(red) = blue` will set all the red pixels to the color blue in a $O(1)$ complexity.

This property defines the different accesses provided by an image . *none* means that the image is psite wise; we can access to the image values through its psites. *value_wise* means that the image is psite wise and, we can iterate on the values domain (on all the values that the image can possibly take). *cell_wise* means that the image is psite wise and, we can iterate on all the values taken by the image (the values shared by the image pixels). *value_and_cell_wise* means that the image is psite wise, value wise and cell wise.

3.4.3 speed

Property Value

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

Definition

This property gives some information about the time needed to access to a value from a psite. The value access is provided by `operator(p : psite) : value`. Hence, if `ima` is an image, `ima(p)` gives the value localized at the site p.

If the speed property is equal to $slow$, $ima(p)$ complexity is greater than $O(1)$. If this property is equal to $fast$, $ima(p)$ complexity is in $O(1)$ If this property is equal to $fastest$, $ima(p)$ complexity is in $O(1)$. Furthermore, in this case, ima has an extended domain, and ima values have a pointer semantics (it is possible to directly iterate on the image pixels).

3.5 Properties related to the extended domain

3.5.1 ext_domain

Property Value

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

Definition

This property indicates if the image has an extended domain. An extended domain grows the image domain by adding dummy values on the image border (see Table 3.1). An extended domain increases the performances of the image processing algorithms. Indeed, image processing algorithms often use the neighbors of the image sites. With an extended border it isn't necessary to test if the neighbors of an image site are included in the image domain. The size of the extended domain can differ (fixed, infinite or extendable).

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 3.1 – Image with an extended domain of size 2.

3.5.2 ext_value

Property Value

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

```
           |
           +--- single
           |
           +--- multiple
```

Definition

If all the extended domain sites share a 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”.

3.5.3 ext_io

Property Value

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

Definition

This property describes the read/write accessibility of the image extended border values.

Chapter 4

Images taxonomy: primary image

This chapter presents the major image families of Milena. We will also list the common properties of each image family. Here, we will only describe the primary image types. In the following chapter, we will see the morphers which are another important part of the Milena library.

Primary images are not based on another image type. They are sufficient to define themselves. Thus, a primary image types holds its data. A property of a primary image type can either be declared or automatically deduced with the help of the image associated types (type of site, type of site set...).

4.1 Image nD

An image nD is an image based on a regular grid (aligned and orthogonal). Furthermore, the nodes of the image grid are points. Milena provides three types of image nD: `image1d`, `image2d` and `image3d`. Depending on the dimension, the grid (and the image) can either be linear (in 1D), rectangular (in 2D) or cubic (in 3D). All these types are parameterized by `T`, the value type. And, all the image values are stored in a linear buffer (in the RAM).

As these image types are based on a regular grid, and have their values stored in a linear buffer in RAM, we can directly access an image value from an image points. So for a point p the read access at the point p is $ima(p) \rightarrow data[p]$ and write access is $ima(p) \leftarrow data[p]$ [1].

These image types are regularly used. So, they have to be fast. That's why an extended domain are added to them (see Table 4.1). An extended domain adds dummy value on an image

1. $data$ represents the image values buffer.

Name	value
`size`	$regular$
`data`	raw
`localization`	$regular_grid$
`value_io`	$read_write$
`speed`	$fastest$
`ext_domain`	$some$

Table 4.1 – Properties characteristics of the image$n$D types

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 4.2 – Image in two dimensions with an extended domain.

border (see Table 4.2). This way, a user doesn't have to check if a neighbor of a border point is in the image domain. The extended domain is very useful for algorithms which use structuring elements. So, the `ext_domain` property is set to *some*. Furthermore, the image values type have pointer semantics (there are stored in a linear buffer). So we can access to pixter (iterator over image pixels). That's why the `speed` property is set to $fastest$.

4.2 Images based on function

It is possible to represent an image with just a set of sites and a function. For instance, consider Ps, a site set composed by the following points in two dimensions: $(0,0)$, $(0,1)$, $(0,2)$, $(1,0)$, $(1,1)$, $(1,2)$, $(2,0)$, $(2,1)$, $(2,2)$.

Let's define f, a function:

$$\begin{array}{rcl} Point2d & \rightarrow & Integer \\ p & \rightarrow & (p.row + p.col) \bmod 2 \end{array}$$

The resulting image (Ps, f) is:

0	1	0
1	0	1
0	1	0

The `Fun_image` type has been designed to define an image from a function site to value (called $p2v$ function).

The `Fun_image` type is parameterized by `F`, a $p2v$ function and `S`, a Site Set type. Image values are computed on the fly with the help of the associated function: $ima(p) \rightarrow f(p)$ where f is the image associated function.

`Fun_Image` represents constant image since we can't modify its values. Indeed all the values are computed on the fly. So, the property `data` is set to *computed* and the property `value_io` is set to *read_only*.

4.3 Images encoded by runs

4.3.1 RLE Image

A `rle_image` can be defined as couples (r, v) where r is a run composed by points of type `P` and, v is a value of type `T`. A value v is associated with each run r. A run is a "continuous" succession of **points**. It is encoded by a point representing the beginning of the run and an integer which is the length of the run. For example, the run `((1, 2), 4)` represents the points $(1,2)$,

$(1,3)$, $(1,4)$ and $(1,5)$. In `rle_image`, every point of a run has the same value. From an implementation point of view, a `rle_image` is composed of two data structures: `runs`, a vector of run, and `values`, a vector of values. A run at the index i of the `runs` vector is associated to the value at the index i of the `values` vector. Table 4.3 represents the rle encoding of an `image2d`.

Image2d:

1	1	2
2	2	2
1	1	1

rle_image:

runs	values
(0, 0), 2	**1**
(0, 2), 1	**2**
(1, 0), 3	**2**
(2, 0), 3	**1**

Table 4.3 – Image2d and its corresponding RLE encoded image

Due to the run encoding, we cannot access directly to the image values with points. That's why we use a psite to access to the image values. The rle psites are just a wrapper on two integers which are the position of the run inside the `runs` vector (`out_run_index`), and the position of the point inside the run (`in_run_index`).

Using this psite, the image data accesses are:
$ima(psite) \rightarrow values[psite.out_run_index]$
$ima(psite) \leftarrow values[psite.out_run_index]$

Furthermore, all images encoded by runs are "one shot" in Milena, we cannot add new points to them, once they are created. This image type allows users to create image with a few data and free form domain. Indeed, the runs composing the image are not necessary continuous.

4.3.2 Sparse Image

A `sparse_image` can be defined as couples $(r, [v])$ where r is a run composed by points of type `P` and v a vector of values of type `T`. 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`). The Table 4.4 describes the encoding of an `image2d`. As for the `rle_image`, we must use a psite to access to the image data. The `sparse_image` psite is the same than the `rle_image` psite. The data accesses are:
$ima(psite) \rightarrow values[psite.out_run_index][psite.in_run_index]$
$ima(psite) \leftarrow values[psite.out_run_index][psite.in_run_index]$
Like `rle_image`, `sparse_image` allows the user to create image with few data and free form domain.

4.3.3 Value encoded image

The `value_encoded_image` types are value-driven images. In these images, a value is associated to a set of runs. Thus, a `value_encoded_image` can be seen as couples $(v, [runs])$

image2d:

1	2	3			
			2	2	2
			4	5	8

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

sparse_image:

run	values
`(0, 0), 3`	$1, 2, 3$
`(1, 3), 3`	$2, 2, 2$
`(2, 3), 3`	$4, 5, 6$

Table 4.4 – Image2d and its corresponding sparse encoded image

where v is a value and $[runs]$ is a run vector. This image type is value and cell wise accessible. Indeed, all the runs which shared the same value can be retrieved in a $O(1)$ complexity. The Table 4.5 describes the structure of a `value_encoded_image` .

To recover all the sites that share the value 2, we just have to access to the index 2 in the $[runs]$ vector. The result will be the points composing the runs $((1, 3), 3)$ and $((2, 3), 3)$. Like `rle_image`, `value_encoded_image` allows the user to create image with few data and free form domain.

image2d:

1	1	1			
			2	2	2
			2	2	2
3	3	3	4	4	4

value_encoded_image:

value	[runs]
1	$((0, 0), 3)$
2	$((1, 3), 3), ((2, 3), 3)$
3	$((3, 0), 3)$
4	$((3, 3), 3)$

Table 4.5 – Image2d and its corresponding value encoded image

4.3.4 Properties

All the images encoded by runs have common properties values. (see Table 4.6). There are all based on a regular grid. Furthermore, the domain (set of sites) associated to these image types is const in the Milena library, we can't add or modify a run once the image has been created. So, the `value_io` property is set to *read*.

4.4 graph_image

`graph_image` is an image type which uses a non-directed graph to compute the neighborhood relationships. This image is parameterized by `S`, a site type and `T`, a value type. Two sites

Name	value
data	*stored*
localization	*regular_grid*
value_io	*read*
speed	*fast*

Table 4.6 – Properties characteristics of image encoded by runs

p and q are neighbor in a `graph_image` if a relation between p and q exists in the associated non-directed graph. These images also have a vector of values. Indeed a value is associated to each site of the image.

`S`, the site type, is not necessarily point wise. Indeed, a site can be an edge, a facet, or even a set of points. For instance, an image encoded, using a delauney triangulation, provides a data access through facets rather than points. So, this kind of image isn't point wise.

Since the user defines the neighborhood relationships, we can't assume that the graph images are on a grid and these images can't have an extended domain. So, its `localization` property is set to *space*, and its `ext_domain` is set to *none*.

4.5 pkey_image

The `pkey_image` images use a map to store their values. The `pkey_image` type is parameterized by two parameters: `T`, a value type and `P`, a site type. A `pkey_image` image stores in memory a std::map m which uses `P` as key type and `V` as value type. To access to the image values, we just have to access to the std::map:
$ima(p) \rightarrow m[p]$ (read access),
$ima(p) \leftarrow m[p]$ (write access).

The data access are not in $O(1)$ since the image uses an underlying std::map. So, the `speed` property of `pkey_image` is set to `slow`.

This image type is related to `sparse_image`; it allows the user to create image with a few data and free form domain.

Name	value
data	*stored*
value_io	*read_write*
speed	*slow*

Table 4.7 – Properties characteristics of the pkey_image

4.6 Flat_image

A `Flat_Image` is an image type where all the sites share the same value. It can be seen as an image based on a constant function:

$$\begin{aligned} Psite_Type &\rightarrow Value_Type \\ p &\rightarrow v \end{aligned}$$

where v is the value shared by all the sites. The `Flat_image` images stored the "flat value" in memory. Thus, it is allowed to modify the value shared by all the sites.

Name	value
`size`	$regular$
`data`	$stored$
`value_io`	$read_write$
`value_access`	$value_wise$

Table 4.8 – Properties characteristics of the Flat image

Chapter 5

Images taxonomy: 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. In our case, we will only consider the image morphers. Thus, image morphers rest upon another image (the input image(s) which will be transformed).

Since, it extends an image, an image morpher is also an image type. Hence, image morphers have the same properties than the image types. Most of the morphers' properties keep the same values than the input types properties. However, some of them are transformed.

In this chapter we describe the major morphers present in the Milena library. For each morpher, we list the transformed properties. First, we describe value morphers (morphers which modify the values of the input image(s)). Then, we look at the domain morpher (morphers which modify the definition domain of the input image.)

5.1 Value morphers

In the following, we will use `ref` to refer to the transformed image.

5.1.1 f_image

The `f_image` morpher transforms the input image values through a "function" f. This morpher type is parameterized by `F` a function type and `I` the input image type. This morpher apply f to an input image value when a user tries to access it ($\forall p\ ima(p) \rightarrow f(ref(p))$).

`f_image` has its property `data` set to *computed* since we have to compute the image values. If `I` is writable and the function f bijective, then `value_io` is set to *read_write*. In the other cases `values_io` is set to *read_only*.

For instance, we have a `f_image` composed by a function $f : v \rightarrow (v+1)$ and an input image:

1	2	3
4	5	6

This `f_image` is equivalent to the following image:

2	3	4
5	6	7

5.1.2 f_images

The `f_images` morpher is a generalization of the `f_image` morpher. This type is parameterized by three parameters:

- `F` a function type
- `n` an integer representing the number of the input image(s).
- `I` an image type.

The `f_images` values are vectors of `n` values, viewed through a function. If the `f_images` function is the identity, then the `f_images` acts like a stack of `n` images. `f_images` type has the same properties than the `f_image`.

5.1.3 lut_image

A `lut_image` is an image which uses a look-up table to bind transformed values to the image input values. A `lut_image` is parameterized by `I`, the input image type and `T` a value type. As this morpher uses a look-up table, it has its `value_access` property set to *value_and_cell_wise*. Indeed, we can easily modify a value shared by a set of sites, by changing it in the lookup table.

Consider the image:

1	3	1
2	1	3

and the lookup table: $1 \rightarrow 21$, $2 \rightarrow 51$, $3 \rightarrow 42$

The corresponding `lut_image` is:

21	42	21
51	21	42

Now if we change a value in the lookup-table: $1 \rightarrow$ **12**, $2 \rightarrow 51$, $3 \rightarrow 42$

All the sites sharing the value 21 in the image have their values changed:

12	42	**12**
51	**12**	42

5.2 Domain morphers

5.2.1 sub_image

The `sub_image` morpher restrict an image to a given sites set. This morpher is parameterized by `I`, the input image type and `S` the sites set type. Due to the domain restriction, the `sub_image` has its `ext_domain` property set to *none*.

For instance, consider this 2d image:

21	**42**	**21**
51	21	42

If we want to restricts this image to the points: $(0,1)$, $(0,2)$. The resulting `sub_image` will be:

42	**21**

5.2.2 hexa

The `hexa` morpher makes a hexagonal mesh of the input image. This morpher is parameterized by `I`, the input image type. The `dimension` property of the input image must be set to *two_d*. Indeed `hexa` morpher only works with image in two dimensions. As this morpher add

a hexagonal grid to the input image, it has its `localization` property set to $grid$. Hence, this morphers change the neighborhood relationships of the input image sites since it changes of the image underlying grid.

5.2.3 image_if

The `image_if` morpher restricts the input image domain to all the image sites which satisfy a condition expressed by a function "site to boolean". It is parameterized 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 of the 2d image. Hence, we define the function $f : \forall p$ $f(p) \to (p.row + p.col) \bmod 2$ 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:

21		21
	21	

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.

5.2.4 translate_image

The `translate_image` is parameterized by `T`, the input image type. It applies a translation of dp (a given delta-point) to all the input images sites. So, with this morpher, the input image do a translation of a given delta point.

Chapter 6

Conclusion

This report was the opportunity to give a quick overview of the Milena programming paradigm. The generic image processing library Milena is property driven. Proprieties provide a way to do static checking, algorithms specialization and to retrieve automatically some implementations or interface.

In this report we also presented some image types that we want to integrate in Milena. These image types are not based on points anymore. Indeed, Milena images have sites. A site is a localized object in space; it can be a point, a segment or even a set of points. Thus, the image data can be accessed through points or segments. . .

We also tried to define the relevant properties of each image type, in order to draw some categories of images (images nD, images encoded by runs, image based on function, morphers. . .). This work can help to redefine the Milena image concepts, and be a base to rethink the design of the Milena library.

However, the taxonomy described in this report is not complete. Some new image types will be added in the future. Thus, it would be useful to maintain the taxonomy of the Milena image type.

6.1 Acknowledgment

Thanks to Thierry Géraud and Matthieu Garrigues for their help.

Chapter 7

Bibliography

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 Multiparadigm Programming with Object-Oriented Languages*.

Index

Domain morphers, 26

f_image, 25
f_images, 26
Flat Image, 23
Fun Image, 20

hexa, 26

Image nD, 19
image_if, 27

lut_image, 26

Morpher, 25

Pkey Image, 23
Property, 7
Property: category, 12
Property: data, 13
Property: ext_domain, 17
Property: ext_io, 18
Property: ext_value, 17
Property: localization, 14
Property: size, 13
Property: space, 14
Property: speed, 16
Property: value_access, 16
Property: value_io, 15
Psite, 6

Run Image, 20

Site, 6
sub_image, 26

taxonomy, 19
translate_image, 27

Value morphers, 25