Software Architecture for Generic Image Processing Tools

Roland Levillain

Laboratoire de Recherche et Développement de l'EPITA (LRDE)

Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge (LIGM)
Équipe A3SI, ESIEE Paris

Séminaire LRDE
28 avril 2010

A Generic and Reusable Foreword Slide

What is the problem? Most image processing frameworks not generic and reusable enough.

Why is it interesting? Genericity = effective reusability.

How can we address this? Using a paradigm of static generic programming.

What are the benefits? Design, implement and reuse without usual constraints.

A Generic and Reusable Foreword Slide

What is the problem?	Most image processing frameworks not generic and reusable enough.
Why is it interesting?	Genericity = effective reusability.
How can we address this?	Using a paradigm of static generic programming.
What are the benefits?	Design, implement and reuse without usual constraints.

A Generic and Reusable Foreword Slide

What is the problem?	Most image processing frameworks not generic and reusable enough.
Why is it interesting?	Genericity = effective reusability.
How can we address this?	Using a paradigm of static generic programming.
What are the benefits?	Design, implement and reuse without usual constraints.

A Generic and Reusable Foreword Slide

What is the problem?	Most image processing frameworks not generic and reusable enough.
Why is it interesting?	Genericity = effective reusability.
How can we address this?	Using a paradigm of static generic programming.
What are the benefits?	Design, implement and reuse without usual constraints.

A Generic and Reusable Foreword Slide

What is the problem?	Most image processing frameworks not generic and reusable enough.
Why is it interesting?	Genericity = effective reusability.
How can we address this?	Using a paradigm of static generic programming.
What are the benefits?	Design, implement and reuse without usual constraints.

A Few Examples from the Real Life™

Value Type

Issue I must process images with 12-bit values, but my algorithm only handle 8-bit inputs.

Solution 1 Resample 12-bit data onto 8-bit data?

→ Deterioration.

Solution 2 Rewrite all algorithms using the biggest floating-point value type (e.g., `double` or `long double`)?

→ Time and space cost.

→ Does not handle all value types (non-scalars, etc.)

→ No type-checking: e.g. one can mix up binary images with floating-point value images.

A Few Examples from the Real Life™

Value Type

Issue I must process images with 12-bit values, but my algorithm only handle 8-bit inputs.

Solution 1 Resample 12-bit data onto 8-bit data?

→ Deterioration.

Solution 2 Rewrite all algorithms using the biggest floating-point value type (e.g., `double` or `long double`)?

→ Time and space cost.

→ Does not handle all value types (non-scalars, etc.)

→ No type-checking: e.g. one can mix up binary images with floating-point value images.

A Few Examples from the Real Life™

Value Type

Issue: I must process images with 12-bit values, but my algorithm only handle 8-bit inputs.

Solution 1: Resample 12-bit data onto 8-bit data?

→ Deterioration.

Solution 2: Rewrite all algorithms using the biggest floating-point value type (e.g., `double` or `long double`)?

→ Time and space cost.

→ Does not handle all value types (non-scalars, etc.)

→ No type-checking: e.g. one can mix up binary images with floating-point value images.

A Few Examples from the Real Life™(cont.)

Image Domain

Issue: I need to process a subset s of an image (e.g., a region).

Solution 1: Create a new input image (cropping)?

→ May not fit if s is not a box.

→ Image copy (time and space cost).

Solution 2: Rewrite the algorithm to have it take an additional mask (region of interest)?

→ Clutters code with details related to a specific use case.

→ Only address the case of the considered algorithm.

A Few Examples from the Real Life™(cont.)

Image Domain

Issue	I need to process a subset *s* of an image (e.g., a region).
Solution 1	Create a new input image (cropping)?
→	May not fit if *s* is not a box.
→	Image copy (time and space cost).
Solution 2	Rewrite the algorithm to have it take an additional mask (region of interest)?
→	Clutters code with details related to a specific use case.
→	Only address the case of the considered algorithm.

A Few Examples from the Real Life™(cont.)

Image Domain

Issue	I need to process a subset *s* of an image (e.g., a region).
Solution 1	Create a new input image (cropping)?
→	May not fit if *s* is not a box.
→	Image copy (time and space cost).
Solution 2	Rewrite the algorithm to have it take an additional mask (region of interest)?
→	Clutters code with details related to a specific use case.
→	Only address the case of the considered algorithm.

A Few Examples from the Real Life™(cont.)

Large Input

Issue My application must process inputs of 10 GB, but my computer has only 4 GB RAM.

Solution 1 Downsize the input?

→ No longer the same data.

Solution 2 Split the input into several images?

→ Must collate/merge the outputs.

→ The application may not support this.

Solution 3 Change the algorithm to process the input piecewise?

→ Clutters code with details related to a specific use case.

→ Only address the case of the considered algorithm.

Solution 4 Buy more RAM? ;-)

→ But what if the address space is limited to 4 GB?

A Few Examples from the Real Life™ (cont.)

Large Input

Issue My application must process inputs of 10 GB, but my computer has only 4 GB RAM.

Solution 1 Downsize the input?

→ No longer the same data.

Solution 2 Split the input into several images?

→ Must collate/merge the outputs.

→ The application may not support this.

Solution 3 Change the algorithm to process the input piecewise?

→ Clutters code with details related to a specific use case.

→ Only address the case of the considered algorithm.

Solution 4 Buy more RAM? ;-)

→ But what if the address space is limited to 4 GB?

A Few Examples from the Real Life™(cont.)

Large Input

Issue: My application must process inputs of 10 GB, but my computer has only 4 GB RAM.

Solution 1: Downsize the input?

$\rightarrow$ No longer the same data.

Solution 2: Split the input into several images?

$\rightarrow$ Must collate/merge the outputs.

$\rightarrow$ The application may not support this.

Solution 3: Change the algorithm to process the input piecewise?

$\rightarrow$ Clutters code with details related to a specific use case.

$\rightarrow$ Only address the case of the considered algorithm.

Solution 4: Buy more RAM? ;-)

$\rightarrow$ But what if the address space is limited to 4 GB?

A Few Examples from the Real Life™ (cont.)

Large Input

Issue My application must process inputs of 10 GB, but my computer has only 4 GB RAM.

Solution 1 Downsize the input?

→ No longer the same data.

Solution 2 Split the input into several images?

→ Must collate/merge the outputs.

→ The application may not support this.

Solution 3 Change the algorithm to process the input piecewise?

→ Clutters code with details related to a specific use case.

→ Only address the case of the considered algorithm.

Solution 4 Buy more RAM? ;-)

→ But what if the address space is limited to 4 GB?

A Few Examples from the Real Life™(cont.)

Large Input

Issue	My application must process inputs of 10 GB, but my computer has only 4 GB RAM.
Solution 1	Downsize the input?
→	No longer the same data.
Solution 2	Split the input into several images?
→	Must collate/merge the outputs.
→	The application may not support this.
Solution 3	Change the algorithm to process the input piecewise?
→	Clutters code with details related to a specific use case.
→	Only address the case of the considered algorithm.
Solution 4	Buy more RAM? ;-)
→	But what if the address space is limited to 4 GB?

A Few Facts

There are many different images types:

- 2D images on regular discrete grids
- 3D images on regular discrete grids
- Graph-based (n-dimensional) images
- Histograms
- Arbitrary data in n-dimensional space (why not!)
- Sequences
- etc.

A Few Facts

There are many different images types:

- 2D images on regular discrete grids
- 3D images on regular discrete grids
- Graph-based (n-dimensional) images
- Histograms
- Arbitrary data in n-dimensional space (why not!)
- Sequences
- etc.

A Few Facts

There are many different images types:

- 2D images on regular discrete grids
- 3D images on regular discrete grids
- Graph-based (n-dimensional) images
- Histograms
- Arbitrary data in n-dimensional space (why not!)
- Sequences
- etc.

A Few Facts

There are many different images types:

- 2D images on regular discrete grids
- 3D images on regular discrete grids
- Graph-based (*n*-dimensional) images
- Histograms
- Arbitrary data in *n*-dimensional space (why not!)
- Sequences
- etc.

A Few Facts

There are many different images types:

- 2D images on regular discrete grids
- 3D images on regular discrete grids
- Graph-based (*n*-dimensional) images
- Histograms
- Arbitrary data in *n*-dimensional space (why not!)
- Sequences
- etc.

A Few Facts

There are many different images types:

- 2D images on regular discrete grids
- 3D images on regular discrete grids
- Graph-based (*n*-dimensional) images
- Histograms
- Arbitrary data in *n*-dimensional space (why not!)
- Sequences
- etc.

A Few Facts

There are many different images types:

- 2D images on regular discrete grids
- 3D images on regular discrete grids
- Graph-based (n-dimensional) images
- Histograms
- Arbitrary data in n-dimensional space (why not!)
- Sequences
- etc.

A Few Facts (cont.)

Likewise, value types are numerous:

- Boolean
- Integer values, which are not. . .
- Gray levels, which are not
- Labels
- Floating-point values
- Complex values
- Colors
- Points
- Matrices, vectors, tensors, etc.

A Few Facts (cont.)

Likewise, value types are numerous:

- Boolean
- Integer values, which are not. . .
- Gray levels, which are not
- Labels
- Floating-point values
- Complex values
- Colors
- Points
- Matrices, vectors, tensors, etc.

A Few Facts (cont.)

Likewise, value types are numerous:

- Boolean
- Integer values, which are not…
- Gray levels, which are not…
- Labels
- Floating-point values
- Complex values
- Colors
- Points
- Matrices, vectors, tensors, etc.

A Few Facts (cont.)

Likewise, value types are numerous:

- Boolean
- Integer values, which are not…
- Gray levels, which are not…
- Labels
- Floating-point values
- Complex values
- Colors
- Points
- Matrices, vectors, tensors, etc.

A Few Facts (cont.)

Likewise, value types are numerous:

- Boolean
- Integer values, which are not...
- Gray levels, which are not...
- Labels
- Floating-point values
- Complex values
- Colors
- Points
- Matrices, vectors, tensors, etc.

A Few Facts (cont.)

Likewise, value types are numerous:

- Boolean
- Integer values, which are not…
- Gray levels, which are not…
- Labels
- Floating-point values
- Complex values
- Colors
- Points
- Matrices, vectors, tensors, etc.

A Few Facts (cont.)

Likewise, value types are numerous:

- Boolean
- Integer values, which are not…
- Gray levels, which are not…
- Labels
- Floating-point values
- Complex values
- Colors
- Points
- Matrices, vectors, tensors, etc.

A Few Facts (cont.)

Likewise, value types are numerous:

- Boolean
- Integer values, which are not. . .
- Gray levels, which are not. . .
- Labels
- Floating-point values
- Complex values
- Colors
- Points
- Matrices, vectors, tensors, etc.

A Few Facts (cont.)

Likewise, value types are numerous:

- Boolean
- Integer values, which are not…
- Gray levels, which are not…
- Labels
- Floating-point values
- Complex values
- Colors
- Points
- Matrices, vectors, tensors, etc.

A Few Facts (cont.)

Likewise, value types are numerous:

- Boolean
- Integer values, which are not…
- Gray levels, which are not…
- Labels
- Floating-point values
- Complex values
- Colors
- Points
- Matrices, vectors, tensors, etc.

A Few Facts (cont.)

Likewise, value types are numerous:

- Boolean
- Integer values, which are not...
- Gray levels, which are not...
- Labels
- Floating-point values
- Complex values
- Colors
- Points
- Matrices, vectors, tensors, etc.

A Few Facts (cont.)

By the way,

- What is a point?
- How can one define relationships such as
  - adjacency or neighborhood (between values)?
  - order (between values)?
- How one can take act on
  - the domain of an image
  - its dimension
  - a region of interest

  in any algorithm?

A Few Facts (cont.)

- Many image processing software tools available corresponding to various use cases:
  - Graphical User Interfaces (GUIs),
  - Programming libraries
  - Interpreters
  - MATLAB toolboxes
  - Online (Web) services,
  - etc.
- Can we design a unique tool to embrace this diversity?

More Facts

- An image processing practitioner is not necessarily a computer scientist: its tools should be easy to use and helpful.
- Research issues are long-time problems. Will this program/language/tool be still supported in 5, 10, 15 years? Or even be available?
- Many tools are more machine- than user-friendly.
  - Implementation details.
  - Disconnected from the theoretical background.

We need effective solutions.

Our Proposal

Architecture based on:

A Generic C++ Library: Generic, efficient, standard and portable core.

Satellite components based on this library: Command-line tools, interpreters (interactive shells), GUIs, etc.

Some In-Between Glue: Preserving the benefits of the core (genericity and efficiency).

Our Proposal: The Olena Platform

Architecture based on:

A Generic C++ Library: The Milena Library
: Generic, efficient, standard and portable core.

Satellite components based on this library
: Command-line tools, interpreters (interactive shells), GUIs, etc.

Some In-Between Glue
: Preserving the benefits of the core (genericity and efficiency).

Genericity in C++

A non generic algorithm

```
void fill(const image& ima, unsigned char v)
{
  for (unsigned int r = 0; r < ima.nrows(); ++r)
    for (unsigned int c = 0; c < ima.ncols(); ++c)
      ima(r, c) = v;
}
```

This code makes a few hypotheses:

1. 2D Image.
2. Point with nonnegative integers coordinates starting at 0.
3. Values compatible with `unsigned char`.

A non generic algorithm

```
void fill(const image& ima, unsigned char v)
{
  for (unsigned int r = 0; r < ima.nrows(); ++r)
    for (unsigned int c = 0; c < ima.ncols(); ++c)
      ima(r, c) = v;
}
```

This code makes a few hypotheses:

1. 2D Image.
2. Point with nonnegative integers coordinates starting at 0.
3. Values compatible with `unsigned char`.

A non generic algorithm

```
void fill(const image& ima, unsigned char v)
{
  for (unsigned int r = 0; r < ima.nrows(); ++r)
    for (unsigned int c = 0; c < ima.ncols(); ++c)
      ima(r, c) = v;
}
```

This code makes a few hypotheses:

1. 2D Image.
2. Point with nonnegative integers coordinates starting at 0.
3. Values compatible with `unsigned char`.

A non generic algorithm

```
void fill(const image& ima, unsigned char v)
{
  for (unsigned int r = 0; r < ima.nrows(); ++r)
    for (unsigned int c = 0; c < ima.ncols(); ++c)
      ima(r, c) = v;
}
```

This code makes a few hypotheses:

1. 2D Image.
2. Point with nonnegative integers coordinates starting at 0.
3. Values compatible with `unsigned char`.

A non generic algorithm

```
void fill(const image& ima, unsigned char v)
{
  for (unsigned int r = 0; r < ima.nrows(); ++r)
    for (unsigned int c = 0; c < ima.ncols(); ++c)
      ima(r, c) = v;
}
```

This code makes a few hypotheses:

1. 2D Image.
2. Point with nonnegative integers coordinates starting at 0.
3. Values compatible with `unsigned char`.

Limits of this first non generic algorithm

This code cannot handle (as-is) any of the following variations:

1. 3D Image
2. Negative coordinates.
3. Floating-point coordinates.
4. 12-bit integer values.
5. Floating-point values.
6. Multivalued image (e.g., a color or n-channel image).

Limits of this first non generic algorithm

This code cannot handle (as-is) any of the following variations:

1. 3D Image
2. Negative coordinates.
3. Floating-point coordinates.
4. 12-bit integer values.
5. Floating-point values.
6. Multivalued image (e.g., a color or *n*-channel image).

Limits of this first non generic algorithm

This code cannot handle (as-is) any of the following variations:

1. 3D Image
2. Negative coordinates.
3. Floating-point coordinates.
4. 12-bit integer values.
5. Floating-point values.
6. Multivalued image (e.g., a color or *n*-channel image).

Limits of this first non generic algorithm

This code cannot handle (as-is) any of the following variations:

1. 3D Image
2. Negative coordinates.
3. Floating-point coordinates.
4. 12-bit integer values.
5. Floating-point values.
6. Multivalued image (e.g., a color or *n*-channel image).

Limits of this first non generic algorithm

This code cannot handle (as-is) any of the following variations:

1. 3D Image
2. Negative coordinates.
3. Floating-point coordinates.
4. 12-bit integer values.
5. Floating-point values.
6. Multivalued image (e.g., a color or *n*-channel image).

Limits of this first non generic algorithm

This code cannot handle (as-is) any of the following variations:

1. 3D Image
2. Negative coordinates.
3. Floating-point coordinates.
4. 12-bit integer values.
5. Floating-point values.
6. Multivalued image (e.g., a color or *n*-channel image).

Limits of this first non generic algorithm

This code cannot handle (as-is) any of the following variations:

1. 3D Image
2. Negative coordinates.
3. Floating-point coordinates.
4. 12-bit integer values.
5. Floating-point values.
6. Multivalued image (e.g., a color or *n*-channel image).

Rephrasing the issue

- With symbolic notations:

$$\forall p \in D \quad ima(p) \leftarrow v$$

  where D is the domain of *ima*.
- That is, in pseudocode :

```
for_all(p) ima(p) = v;
```

  - Where `p` is an object traversing `ima`'s domain.
  - And a few more details due to C++'s idiosyncrasies.

Rephrasing the issue

- With symbolic notations:

$$\forall p \in D \quad ima(p) \leftarrow v$$

  where D is the domain of *ima*.
- That is, in pseudocode :

```
for_all(p) ima(p) = v;
```

  - Where `p` is an object traversing `ima`'s domain.
  - And a few more details due to C++'s idiosyncrasies.

Rephrasing the issue

- With symbolic notations:

$$\forall p \in D \quad ima(p) \leftarrow v$$

  where D is the domain of *ima*.

- That is, in code :

```
for_all(p) ima(p) = v;
```

  - Where `p` is an object traversing `ima`'s domain.
  - And a few more details due to C++'s idiosyncrasies.

Rephrasing the issue

- With symbolic notations:

$$\forall p \in D \quad ima(p) \leftarrow v$$

  where D is the domain of *ima*.
- That is, in code :

```
for_all(p) ima(p) = v;
```

  - Where `p` is an object traversing `ima`'s domain.
  - And a few more details due to C++'s idiosyncrasies.

A generic version

```
template <typename I, typename V>
void fill(Image<I>& ima_, const V& v)
{
  I& ima = exact(ima_);
  mln_piter(I) p(ima.domain());
  for_all(p)
    ima(p) = v;
}
```

- Not dependency regarding characteristics of the input image type.
- Small yet readable.
- Compatible with all previously mentioned cases.
- A generic implementation

A generic version

```
template <typename I, typename V>
void fill(Image<I>& ima_, const V& v)
{
  I& ima = exact(ima_);
  mln_piter(I) p(ima.domain());
  for_all(p)
    ima(p) = v;
}
```

- Not dependency regarding characteristics of the input image type.
- Small yet readable.
- Compatible with all previously mentioned cases.

→ A generic implementation.

A generic version

```
template <typename I, typename V>
void fill(Image<I>& ima_, const V& v)
{
  I& ima = exact(ima_);
  mln_piter(I) p(ima.domain());
  for_all(p)
    ima(p) = v;
}
```

- Not dependency regarding characteristics of the input image type.
- Small yet readable.
- Compatible with all previously mentioned cases.

→ A generic implementation.

A generic version

```
template <typename I, typename V>
void fill(Image<I>& ima_, const V& v)
{
  I& ima = exact(ima_);
  mln_piter(I) p(ima.domain());
  for_all(p)
    ima(p) = v;
}
```

- Not dependency regarding characteristics of the input image type.
- Small yet readable.
- Compatible with all previously mentioned cases.

→ A generic implementation.

A generic version

```
template <typename I, typename V>
void fill(Image<I>& ima_, const V& v)
{
  I& ima = exact(ima_);
  mln_piter(I) p(ima.domain());
  for_all(p)
    ima(p) = v;
}
```

- Not dependency regarding characteristics of the input image type.
- Small yet readable.
- Compatible with all previously mentioned cases.

→ A generic implementation.

Generic Programming in a Nutshell

- Types can become parameters of data structures and routines:
  - `template <typename T>`
    `class image2d<T>`
  - `template <typename I, typename V>`
    `void fill(I& ima, const V& v)`
- Static (compile-time) mechanism: no run-time overhead.

Generic Programming in a Nutshell (cont.)

- Templates can be seen as generators:

`template <typename T> T f(T x) { ... }` $\equiv$ $f_T : \begin{cases} T & \rightarrow & T \\ x & \mapsto & \ldots \end{cases}$

- Actually, C++ compilers implement them by generating the code of given routine or data structure for each used combination of parameters ("template instantiation").

$\rightarrow$ Compile-time (and space) cost, but...

$\rightarrow$ Dedicated code, enabling optimizations!

Generic-Aware Solutions to Previous Problems

Value Type

Issue I must process images with 12-bit values, but my algorithm only handle 8-bit inputs.

Solution Write algorithms not bound to a specific a value type.

Example

```
template <typename I, typename V>
void fill(I& ima, const V& val)
{
  mln_piter(I) p(ima.domain());
  for_all(p)
    ima(p) = val;
}
```

Generic-Aware Solutions to Previous Problems (cont.)

Image Domain

Issue I need to process a subset *s* of an image (e.g., a region).

Solution Create a “proxy” image type (a “view”) altering the domain of the underlying image.

Example

```
masked = ima | s;    // Masked input.
fill(masked, 51);    // Fill 'ima' on 's' only.
```

Generic-Aware Solutions to Previous Problems (cont.)

Large Input

Issue My application must process inputs of 10 GB, but my computer has only 4 GB RAM.

Solution Create a tiled image type storing its data in the file system and hiding the task of loading and writing data.

Example

```
tiled_image2d<int_u8> ima;
load(ima, "input.dicom");  // No data loaded here.
process(ima);  // Data loaded/stored piecewise.
```

Beyond C++ Genericity: Abstractions and Interfaces

With Generic Programming (GP):

- Algorithms are no longer defined in terms of features specific to an image type.

```
for (unsigned int r = 0; r < input.nrows(); ++r)
  for (unsigned int c = 0; c < input.ncols(); ++c)
    ...
```

- Instead, abstractions are used.

```
mln_piter(I) p(input.domain()); // 'p' is a site iterator.
for_all(p)                      // ∀p ...
  ...
```

The Image Abstraction

The interface of an image type includes:

- Associated types.

```
typedef domain_t;  // Type of the domain (site set).
typedef site;      // Type of a site.
typedef piter;     // Associated iterator type.
typedef value;     // Type of a value.
typedef vset;      // Type of the set of values.
```

- Methods (services provided by the image).

```
value operator()(site p);   // 'ima(p)' → value
bool has(site p);           // Does 'p' belongs to 'ima'?
vset values();              // Return the domain (D).
domain_t domain();          // Return the value set (V).
```

Other Abstractions

`Site_Set` Sets of sites must respect this interface.

```
typedef site;        // The type of the sites.
typedef fwd_piter;   // Forward iterator on the set's sites.
typedef bkd_piter;   // Backward iterator on the set's sites.

bool has(psite p);   // Does 'p' belongs to this set?
```

Also: `Point_Site`, `Delta_Point_Site`, `Site_Iterator`, `Value`, `Value_Set`, `Value_Iterator`, `Neighborhood`, `Window`, `Weighted_Window`, `Accumulator`, `Function`, ...

Constrained Genericity

- Adding constraints on parameters.

```
template<typename I>
void fill(Image<I>& ima_);  // ‘I’ must be an image type.
```

- Accessing specific features of `I`.

```
// ...
I& ima = exact(ima_);
unsigned nr = ima.nrows();  // ‘I’ provides ‘nrows’.
// ...
```

Beyond C++ Genericity: Efficiency

- Compiled Code (C++)
  - Fast
  - Safe
- Specialization mechanism and static dispatch based on properties attached to each type.
  - More expressive than bare overloading.
  - More efficient than (dynamic) polymorphic methods.

A Generic Algorithm

```
namespace generic
{
  template <typename I, typename V>
  void fill(Image<I>& ima_, const V& v)
  {
    I& ima = exact(ima_);
    mln_piter(I) p(ima.domain());
    for_all(p)
      ima(p) = v;
  }
}
```

A Specialized Algorithm

```
template <typename I, typename V>
inline
void fill_one_block(Image<I>& ima_, const V& v)
{
  I& ima = exact(ima_);
  data::memset_(ima, ima.point_at_index(0), v,
                opt::nelements(ima));
}
```

Property-Based Selection

How one can help the compiler find the best (a better) algorithm?

1. Introducing a function (“facade”) checking the input type's properties and delegating to the best version based on them.

```
// Facade.
template <typename I, typename V>
inline
void fill(Image<I>& ima, const V& val)
{
  // Dispatch following the ``value storage'' property.
  fill_dispatch(mln_trait_image_value_access(I)(),
                ima, val);
}
```

Property-Based Selection (cont.)

2. Providing a default delegation calling the generic version.

```
// Generic, slow version.
template <typename I, typename V>
void fill_dispatch(trait::image::value_access::any,
                   Image<I>& ima, const V& val)
{
  generic::fill(ima, val);
}
```

Property-Based Selection (cont.)

3. Introducing delegations for images having certain properties.

```
// Fast version (for images with direct access to values).
template <typename I, typename V>
void fill_dispatch(trait::image::value_access::direct,
                   Image<I>& ima, const V& val)
{
  fill_one_block(ima, val);
}
```

Beyond C++ Genericity: Morphers

- Morphers: lightweight objects producing an image from an image (or from several images).
- Example: filling an image:

```
fill(ima, 42);
```

- Likewise, but restricting the domain of `ima` to the subset `s`:

```
fill(ima | s, 42);
```

- Filling (only) the red channel of an RGB color image:

```
fill(red << rgb_ima, 42);
```

Beyond C++ Genericity: Morphers

- Morphers: lightweight objects producing an image from an image (or from several images).
- Example: filling an image:

```
fill(ima, 42);
```

- Likewise, but restricting the domain of `ima` to the subset `s`:

```
fill(ima | s, 42);
```

- Filling (only) the red channel of an RGB color image:

```
fill(red << rgb_ima, 42);
```

Beyond C++ Genericity: Morphers

- Morphers: lightweight objects producing an image from an image (or from several images).
- Example: filling an image:

```
fill(ima, 42);
```

- Likewise, but restricting the domain of `ima` to the subset `s`:

```
fill(ima | s, 42);
```

- Filling (only) the red channel of an RGB color image:

```
fill(red << rgb_ima, 42);
```

Beyond C++ Genericity: Morphers

- Morphers: lightweight objects producing an image from an image (or from several images).
- Example: filling an image:

```
fill(ima, 42);
```

- Likewise, but restricting the domain of `ima` to the subset `s`:

```
fill(ima | s, 42);
```

- Filling (only) the red channel of an RGB color image:

```
fill(red << rgb_ima, 42);
```

More Morphers

- Many morphers provided by Milena:
  - Wrapping an image (cylinder, torus, etc.).
  - Stacking several images.
  - Taking a slice from a 3D volume and seeing it as a 2D image.
  - Applying a geometrical transformation.
  - Adding tracing, logging or profiling mechanisms.
  - “Synthetic” images computed on-the-fly.
  - ...
- These morphers are themselves generic.
- Compose and reuse at will.

More Morphers

- Many morphers provided by Milena:
  - Wrapping an image (cylinder, torus, etc.).
  - Stacking several images.
  - Taking a slice from a 3D volume and seeing it as a 2D image.
  - Applying a geometrical transformation.
  - Adding tracing, logging or profiling mechanisms.
  - “Synthetic” images computed on-the-fly.
  - . . .
- These morphers are themselves generic.
- Compose and reuse at will.

Beyond C++ Genericity: SCOOP

→ A new programming paradigm: Static C++ Object Oriented Programming (SCOOP) based on template metaprogramming (i.e., “programs” executed by the C++ compiler) [Burrus et al., 2003, Géraud and Levillain, 2008].

Alternatives to Static Genericity

Hand-made Code Duplication	Error prone, does not scale.
Dynamic Genericity	Using polymorphic methods (virtual functions): run-time cost, many limitations.
Dangerous Genericity	Using `void*` instead of `T` (type erasure): error prone, no possible specialization.
No genericity	If you only need algorithms working on a single data structure (doubtful!).

Illustrations

A Simple Milena Processing Chain

- A generic code [Levillain et al., 2009]:

```
closed = morpho::closing::area(ima, nbh, lambda);
wshed  = morpho::watershed::flooding(closed, nbh, nb);
```

Go to full code

- Inputs:
  - `ima` Input image (e.g, `image2d<int>`, `image3d<float>`, `graph_image`, etc.).
  - `nbh` Neighborhood (e.g., `c4`, `c26`, `adjacent_vertices_neighborhood`, etc.).
  - `lambda` Value of the criterion (integer).
- Applicable to many different image types as-is.

Results (2D Image)

Figure: "Classical" image, with 4-connectivity.

Figure: Magnitude of the gradient.

Figure: Result of the image processing chain on the magnitude of the gradient.

Results (3D Mesh)

Figure: Triangle mesh, seen as a simplicial 2-complex.

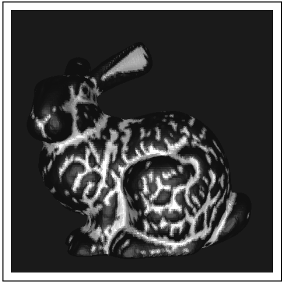


Figure: Curvature computed on the edges.

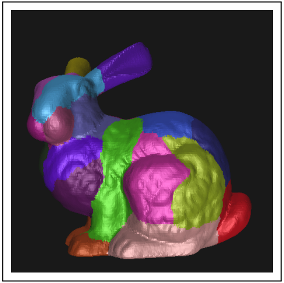


Figure: Result of the image processing chain on the curvature computed on the edges.

Results (Graph)

Example of data clustering using mathematical morphology methods.

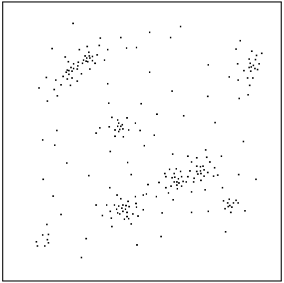


Figure: Vertices of a graph.

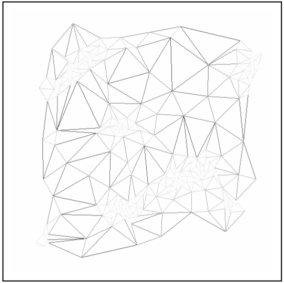


Figure: Distance-based magnitude computed on the edges of the triangulation of the graph.

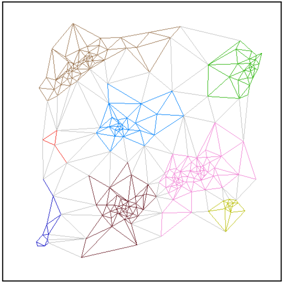


Figure: Result of the image processing chain on this magnitude.

Leveraging genericity outside C++

Limitations of the C++ Library Model

- To benefit from this expressive power of genericity, one has to write C++ code.
- Not necessarily easy.
- Each change requires recompiling.
- Each use case (combination of parameters) requires recompiling.
- Applying a filter to an image ⇒ Writing and compiling (and possibly debugging) a (small) C++ program each time.
- Constraints acceptable for a big application, but not for a small prototype.

Using Milena Outside the C++ World: Issues

- Principle: building GUI, command-line tools, interpreters on top of the Milena library.
- Classical approaches: based on library linking or dynamic module loading.
- But... Milena does not provide a classical library or dynamic modules (files ending in '`.a`', '`.so`', '`.lib`', '`.dll`', '`.dylib`', '`.bundle`', etc.)!
- Milena is composed of headers only ('`.hh`' files), since the compilation of templates is dependent on their use.

Using Milena Outside the C++ World: Issues

- Principle: building GUI, command-line tools, interpreters on top of the Milena library.
- Classical approaches: based on library linking or dynamic module loading.
- But... Milena does not provide a classical library or dynamic modules (files ending in '`.a`', '`.so`', '`.lib`', '`.dll`', '`.dylib`', '`.bundle`', etc.)!
- Milena is composed of headers only ('`.hh`' files), since the compilation of templates is dependent on their use.

Using Milena Outside the C++ World: Solutions

Instantiated Genericity Generating a set of all interesting combinations to produce a library, and build tools on them.

- Might be costly: with A algorithms, I image types and V values types
  $\Rightarrow$ Instantiating and compiling $A \times I \times V$ templates!
- Not all these combinations might be interesting.
- Limited Genericity: does not grow beyond the initial set of chosen parameters.

$\rightarrow$ Still a static approach, with compile-time limitations.

The Moving "IN"

Static approaches based on compiled languages (like C++) are

- **Efficient** Many checks and optimizations are performed at compile time.
- **But INflexible** Once compiled, the code cannot change.

Dynamic approaches based on interpreted languages (like Python) are

- **Flexible** E.g., class introspection, `eval` keyword, etc.
- **But INefficient** Run-time checks are costly and prevent optimizations.

Yet, we want an efficient and flexible solution.

Using Milena Outside the C++ World: Solutions (cont.)

Concealed Genericity General idea: produce the desired code on-the-fly.

- Hide parameterized routines and classes behind opaque types (proxys): the latter delegates to the former.
- On-the-fly generation, instantiation, compiling and loading of C++ code.
- Only the required (interesting) code is instantiated and compiled.
- Compilation costs can be amortized by using a cache.
- The use of a proxy introduces a very small run-time overhead.

→ A static/dynamic bridge based on C++ Just-In-Time (JIT) compiling [Duret-Lutz, 2000, Pouillard and Thivolle, 2006].

Static/Dynamic Bridge: Just-In-Time (JIT) Compiling and Cache

- Only template code used in the program is compiled.
- Each compiled function is stored into a repository (or cache).
- Each time a function is needed, it is looked up in the repository (to avoid recompiling).
- Compilation costs become negligible in the long run.

Static/Dynamic Bridge: Example

```
dyn::include("mln/core/image/image2d.hh");
dyn::include("mln/data/fill.hh");

ctor mk_image2d_int("mln::image2d<int>");
fun fill("mln::data::fill");

var ima = mk_image2d_int(3, 3);
fill(ima, 0);
```

- Declarative approach, but hand-made wrapping (currently).
- Each routine and data structure is represented by an object.
- Calling a wrapped routine triggers the JIT compiling, and caches the products.
- Not tied to Milena nor Olena: reusable technology.

Static/Dynamic Bridge: Applications

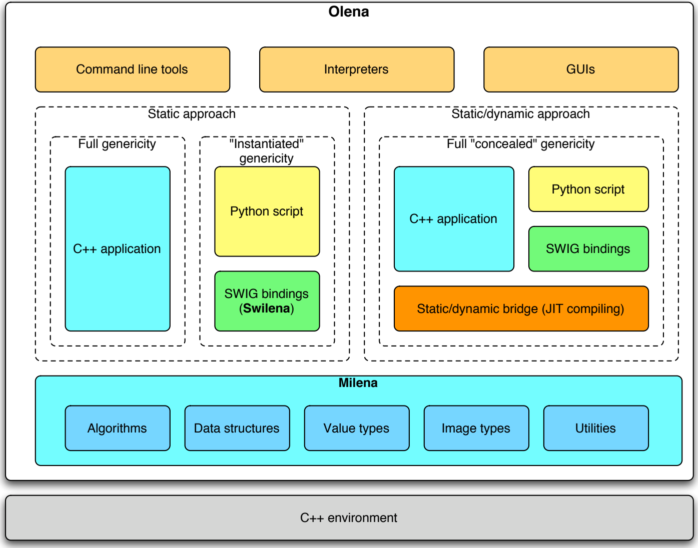
- Can be used from C++ (e.g., “using Milena without seeing the templates”).
- Or from an application linked to it.
- Bindings to other languages (for instance Python) can also benefit from this approach.

```
dyn.include("mln/core/image/image2d.hh")
dyn.include("mln/data/fill.hh")

mk_image2d_int = dyn.ctor("mln::image2d<int>")
fill = dyn.fun("mln::data::fill")

ima = mk_image2d_int(dyn.data(3), dyn.data(3))
fill(ima, dyn.data(0))
```

The Olena Platform



Conclusion and Future Work

The State of the Project

Milena

- Much of the efforts have been put into the Milena library.
- The most advanced component of the platform so far.
- Needs some polishing, but usable.
- Need more documentation.
- Fairly portable: GNU/Linux, Mac OS X, and should compile fine under Cygwin and MinGW. ;-)

The State of the Project (cont.)

Swilena

- We provide a few Python bindings covering a small fraction of Milena, and for a few combinations only.
- Uses the Simplified Wrapper and Interface Generator (SWIG).
- Relies on pre-instantiated templates.
- Can be used from the Python interactive interpreter (Swilena Python Shell)
- Easily extensible.
- Still, bound by the limits of the static world.

The State of the Project (cont.)

- Milena and Swilena have been released June 14, 2009 within the Olena 1.0 platform.

  `http://olena.lrde.epita.fr`

- We invite you to download and try it.
- Olena is Free Software released under the GNU General Public License (GNU GPL).
- There is much more to say about Milena, in particular about efficiency.
- You can send questions and comments to: olena@lrde.epita.fr.

The State of the Project (cont.)

The Dynamic/Static Bridge

- The dynamic/static bridge is still a prototype.
- Tested successfully on Debian GNU/Linux 5.0 and Mac OS X 10.5 on IA-32.
- A foundation for satellite components.
- Goal: writing simple GUIs or image processing tools.
- A promising way towards more C++ dynamic services.

An Improved Dynamic/Static Bridge

Wrappers Generation

- Replace wrapping of each routine and data structure by a list of declaration (e.g., as annotations in the wrapped code).
- Better: generate these annotations from the code itself.
- Fully automated tool.
- Tricky task: requires to parse C++ code.

An improved dynamic/static bridge

Beyond Routines Wrapping

- The dynamic/static bridge really wraps only routines: the interfaces of classes is no taken into account.
- Wrapping classes paves the way for powerful features [Vollmann, 2000].
  - Class introspection/reflexion.
  - Dynamic class generation and more C++ JIT compiling.
  - Meta-Object Protocols (MOPs).

A Last Word

- An interesting paradox of the Olena platform:

  Executing programs at compile time
  : Template metaprograms at the heart of Milena (generating efficient code).

  Compiling programs at run time
  : JIT compiling of Milena routine by the static/dynamic bridge (to address dynamic needs).

- Unusual but effective uses of the C++ language.

Olena Contributors

Alexandre Abraham
Alexis Angelidis
Nicolas Ballas
Christophe Berger
Sylvain Berlemont
Vincent Berruchon
Frédéric Bour
Nicolas Burrus
Edwin Carlinet
Jean Chalard
Rémi Coupet
Tristan Croiset
Jérôme Darbon
Réda Dehak
Akim Demaille
Guillaume Duhamel
Alexandre Duret-Lutz
Yoann Fabre
Étienne Folio
Renaud François
Fabien Freling
Matthieu Garrigues
Ignacy Gawedzki
Thierry Géraud
Quentin Hocquet
Yann Jacquelet
Ugo Jardonnet
Guillaume Lazzara
David Lesage
Roland Levillain
Julien Marquegnies
Thomas Moulard
Jean-Sébastien Mouret
Simon Nivault
Simon Odou
Giovanni Palma
Dimitri Papadopoulos-Orfanos
Ludovic Perrine
Quôc Peyrot
Anthony Pinagot
Raphaël Poss
Nicolas Pouillard
Yann Régis-Gianas
Michaël Strauss
Pierre-Yves Strub
Damien Thivolle
Emmanuel Turquin
Niels Van Vliet
Astrid Wang
Nicolas Widynski
Heru Xue

Thank You For Your Attention!

Software Architecture for Generic Image Processing Tools

References I

Burrus, N., Duret-Lutz, A., Géraud, Th., Lesage, D., and Poss, R. (2003).
A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming.
In *Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL)*, Anaheim, CA, USA.

Duret-Lutz, A. (2000).
Olena: a component-based platform for image processing, mixing generic, generative and OO programming.
In *Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE)—Young Researchers Workshop; published in "Net.ObjectDays2000"*, pages 653–659, Erfurt, Germany.

References II

Géraud, Th. and Levillain, R. (2008).
Semantics-driven genericity: A sequel to the static C++ object-oriented programming paradigm (SCOOP 2).
In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL'08)*, Paphos, Cyprus.

Levillain, R., Géraud, Th., and Najman, L. (2009).
Milena: Write generic morphological algorithms once, run on many kinds of images.
In Springer-Verlag, editor, *Proceedings of the Ninth International Symposium on Mathematical Morphology (ISMM)*, Lecture Notes in Computer Science Series, Groningen, The Netherlands.

References III

- Pouillard, N. and Thivolle, D. (2006).
  Dynamization of C++ static libraries.
  Technical Report 0602, EPITA Research and Development Laboratory (LRDE).

- Vollmann, D. (2000).
  Metaclasses and reflection in C++.
  http://www.vollmann.com/pubs/meta/meta/meta.html.

Actual Code of the Illustrations

```
template <typename L, typename I, typename N>
mln_ch_value(I, L)
chain(const I& ima, const N& nbh, int lambda, L& nb)
{
  mln_concrete(I) closed =
    morpho::closing::area(ima, nbh, lambda);
  return morpho::watershed::flooding(closed, nbh, nb);
}
```

Go to simplified code