Towards a software architecture for generic image processing

Roland Levillain

Université Paris-Est

École Doctorale MSTIC

THÈSE

pour obtenir le grade de Docteur de l'Université Paris-Est

spécialité Informatique

présentée et soutenue publiquement par

ROLAND LEVILLAIN

le 15 novembre 2011

VERS UNE ARCHITECTURE LOGICIELLE POUR LE TRAITEMENT D'IMAGES GÉNÉRIQUE

TOWARDS A SOFTWARE ARCHITECTURE FOR GENERIC IMAGE PROCESSING

Revision 14f6947 (2012-02-16 01:48:52 +0100)

CONTENTS

ABSTRACT

In the context of software engineering for image processing (IP), we consider the notion of reusability of algorithms. In many software tools, an algorithm's implementation often depends on the type of processed data. In a broad definition, discrete digital images may have various forms—classical 2D images, 3D volumes, non-regular graphs, cell complexes, and so on—thus leading to a combinatorial explosion of the theoretical number of implementations.

Generic programming (GP) is a framework suited to the development of reusable software tools. We present a programming paradigm based on GP designed for the creation of scientific software such as IP tools. This approach combines the benefits of reusability, expressive power, extensibility, and efficiency.

We then propose a software architecture for IP using this programming paradigm based on a generic IP library. The foundations of this framework define essential IP concepts, enabling the development of algorithms compatible with many image types.

We finally present a strategy to build high-level tools on top of this library, such as bridges to dynamic languages or graphical user interfaces. This mechanism has been designed to preserve the genericity and efficiency of the underlying software tools, while making them simpler to use and more flexible.

RÉSUMÉ

Dans le cadre du génie logiciel en traitement d'images (TDI), nous nous intéressons à la notion de réutilisabilité des algorithmes. Dans de nombreux outils logiciels, l'implémentation d'un algorithme est souvent dépendante du type des données traitées. Au sens le plus général, les formes que peuvent prendre les images numériques discrètes sont nombreuses (image 2D classiques, volumes 3D, graphes non réguliers, complexes cellulaires, etc.) conduisant à une explosion combinatoire du nombre théorique d'implémentations.

La programmation générique (PG) est un cadre adapté au développement d'outils logiciels réutilisables. Nous présentons un paradigme de programmation basé sur la PG conçu pour la création de logiciels scientifiques tels ceux dédiés au TDI. Cette approche concilie réutilisabilité, puissance d'expression, extensibilité et performance.

Nous proposons ensuite une architecture logicielle pour le TDI basée sur ce paradigme de programmation, s'appuyant sur une bibliothèque générique de TDI. Les fondations de ce cadre définissent des concepts fondamentaux du TDI, qui permettent l'écriture d'algorithmes réutilisables sur de nombreux types d'images.

Nous présentons enfin une stratégie pour construire des outils haut niveau au dessus de cette bibliothèque tels que des ponts vers des langages dynamiques ou des interfaces graphiques. Ce mécanisme est conçu pour préserver la généricité et la performance des outils logiciels sous-jacents, tout en permettant un usage plus simple et plus flexible de ceux-ci.

ACKNOWLEDGMENTS

I would like to first thank Jacques-Olivier Lachaud, David Coeurjolly and Ullrich Köthe, who accepted to review this thesis. I am grateful to Karl Tombre, president of the jury, and to Étienne Decencière and Gilles Roussel, who agreed to be member of my jury.

I am deeply grateful to my two advisors, Laurent Najman and Thierry Géraud. During the four years of this thesis, we have spend numerous hours together working at ESIEE and EPITA. They supervised my work in different ways and in the end their approaches were revealed to be complementary.

Laurent Najman has done a remarkable work at supervising all the steps of this work. As my time was shared between this Ph.D. thesis and my position at EPITA, it has been sometimes hard to find a balance between these two activities, but Laurent was always keeping tracks of events and giving previous advice.

Thierry Géraud was the first to encourage me to start this thesis. He was also the one who introduced me to the world of computer science research. For this and for many other reasons, I am endlessly grateful to him.

I thank all my colleagues and friends from LRDE, and especially Akim Demaille, who encouraged and helped me during this thesis. I cannot imagine this period of my life in another setting than the one of this laboratory, which has become a kind of second home for me for many years now.

I am grateful to all the members of the A3SI laboratory, who hosted me during these four years in a friendly atmosphere, especially among Ph.D. candidates.

And of course, I thank my friends and my family, for their friendship and love, their support and their patience when I was not available. None of this could have been made without their enduring support

FOREWORD

This work is in line with the Olena project, a Free Software platform for generic and efficient Image Processing (IP). Olena has been started by Thierry Géraud at the end of the 1990s. It is the result of more than ten years of work with contributions from more than 50 participants.

The Olena project is structured around a generic and efficient IP library written in the C++ programming language called Milena. Milena proposes many data structures (images, sets of points, values types, etc.) and algorithms. The library is said to be generic since these data structures and algorithms have a unique definition, and each algorithm can be used with any data structure, with no intrinsic limitation, as long as the combination is consistent. The project promotes a "Write Once, Reuse Many" strategy, were algorithms are not written for a specific data type, but have a general definition compatible with many (and possibly not yet written) inputs.

The contributions presented in this thesis are located in several places in Olena, and I share credit with Thierry Géraud for many of them. In particular, the work on the Static C++ Object-Oriented Programming (SCOOP) paradigm (see Chapter 3) and the design of the Milena library (see Chapter 4) has started before I joined the Olena project, and my involvement in terms of ideas, experimentation, design and implementation in Olena are visible from versions 0.11, 1.0 and 2.0 of the project (respectively released in 2007, 2009 and 2011).

The development and experimentation in Olena of data structures such as graphs, simplicial complexes and cubical complexes, as well as the introduction of some discrete geometry elements and the extension of mathematical morphology features was prompted by Laurent Najman. These ideas showed us that the design of the project was general enough to integrate new algorithms and data structures easily, and helped us to identify minor design issues that were easily fixed.

The idea and first implementation of a dynamic-static bridge to harness generic C++ code out of the traditional C++ compilation model in the project was first proposed by Alexandre Duret-Lutz and Thierry Géraud, and later reimplemented by Nicolas Pouillard and Damien Thivolle. Likewise, some bindings for scripting languages were already provided by previous releases of the project: Python (Olena 0.7), Ruby (Olena 0.10). The dynamic use of Olena proposed in this thesis (see Chapter 6) is an extension of this prior work.

1 INTRODUCTION

1.1 CONTEXT

Image Processing (IP) is a science dedicated to the acquisition, analysis, transformation and production of (discrete) digital images. A digital image can be defined as a numerical description of a real-world multidimensional (most often 2-dimensional) signal. Even if bi-dimensional signals have been studied since the beginning of the 20th century, IP really started with the introduction of computers to the field in the 1960s, becoming powerful enough to process digital images. In this thesis we use the term "image" to mean "digital image".

Most of the time, an image is acquired by sampling a signal (visible light reflected by a scene, radio waves, X-rays passing through an object, etc.) using a sensor (camera, radar, Computed Tomography (CT) scanner, etc.). The image produced is a discrete object that is composed of elements containing information on the signal, which is almost always arranged as a regular structure, such as a hyper-rectangular subspace (or box) of a discrete orthogonal space (such as $\mathbb{Z}^2$). For instance, most 2D images are made of a rectangular set of identical square elements arranged in rows and columns, called pixels (picture elements). Likewise, many 3D images are composed of cubical elements following the regular organization of an orthogonal 3D grid, called voxels (volume elements). However, we may consider other data as "images" in a more general definition. For instance, a discrete surface, represented by a triangular mesh built from a cloud of points acquired with a laser, is an example of a non-regular data structure that can be considered as an image. We can also mention that instead of producing discrete images, other reconstruction techniques may be used to generate continuous or vectorial representations of the original signal, but we do not consider them in this thesis.

Processing an image is the action of analyzing its contents to produce a result. The output of this process can be a modified version of the initial image to improve it in some way (removing noise, suppressing blur, correcting its orientation, etc.), a new image computed from the original input (a segmentation into regions, a Fast Fourier Transform (FFT), etc.) or some kind of information from this input (characteristics, parameters, statistics, classes, etc.) that may be required to perform another task later. Some IP techniques involve several images, e.g. a registration process, where the expected output is a geometric transformation applied to a first image so that it can be aligned to match a second image.

The IP literature lists many algorithms to perform these tasks. An algorithm can be defined as a set of more or less complex instructions to be performed by a computer to realize an action, taking some input, and modifying it or producing an output as result. Algorithms are usually defined in an abstract manner and must be translated into a program expressed in a programming language to be actually manipulated by a computer. This translation is called an *implementation* of an algorithm. By assembling algorithms, one can create processing chains performing tasks of a level higher than a sole algorithm, often dedicated to an application, like detecting a particular object or restoring damaged data.

1.1.1 *Diversity of Image Types*

Digital Image Processing is used in many domains nowadays: biomedical imaging, remote sensing, photography, document image analysis, material analysis, security and video surveillance, etc. The types of images produced and processed also reflects this variety: there is not a unique type manipulated across all domains, or even within a given domain. For example if we consider applications of IP in biomedical imaging, sensors may produces 2D or 3D images or sequences of such images; these images may contain color values, gray-level (intensity) values, or even matrices (diffusion tensors).

More broadly, we can characterize an image type by two aspects: its *structure* and its *values*. The structure of an image represents the spatial organization of its data, along with its combinatorial, topological and geometric properties. There are many possible image structures, such as the following ones.

- The 2D image on a regular discrete square grids is one of the most common cases, and is the one that often comes to mind first when the term "digital image" is mentioned.

- There are other 2D regular cases using triangular or hexagonal grids, where pixels are respectively hexagons and triangles. Though these structures are less common than square-grid images, they have interesting properties that make them useful to IP.

- We have also already mentioned the case of 3D images (on a cubic grid), that are used to represent volumes, in particular in biomedical imaging. Note that the tiling used in the tessellation of the space may not be a cube, but a box (also called rectangular cuboid), the sides of which have different lengths.

- 1-dimensional discrete data can also be considered as images, although they are usually considered as discrete *signals*. Many IP techniques initially come from the science of signal processing.

- The notion of *graphs* is also present in IP. A graph may represent a regularly arranged structure (such as a 2D image on a square grid), but generally they are used to represent data with no regular structure. For instance, from a segmentation of a 2D image, one can deduce a (planar) Region Adjacency Graph (RAG). Each vertex of this graph represents a region, and each of its edges represents the link (boundary) between two regions. Such a graph may also be considered as an image, and be processed by IP techniques. Graphs are not limited to 2D representations: they may be used in 3D spaces, or even in spaces of higher dimensions. RAGs are not the only graph-based image type; we can also mention Binary Space Partitioning (BSP) trees as an example of graph structures used in IP.

- A *mesh* is a structure composed of geometric primitives such a points, lines segments, polygons and polyhedrons representing digital 3D data. They may be used to represent models in Computer-Aided Design (CAD), objects reconstructed by a 3D scanner (statues, sculptures, etc.), and so on. Even if meshes are usually associated with computer graphics or computational geometry, they may be processed with IP techniques.

- Finally, we may also consider *sequences* of several images of the same type as an image structure. A video or an animation can be considered as a special case of 3D image, with two spatial dimensions (the width and the height of the frames) and a spatial dimension (frame axis), referenced to as a "2D+t" (time) image. 3D+t images are likewise made of a sequence of 3D images.

The values of an image are the data stored within each of its elements (pixel, voxels, etc.). As for structures, there are various types of values.

- *Binary* or *Boolean values*, for instance to represent black and white images; or images where a value represents the foreground (usually, the value "true") and the other value ("false"), the background.

- Gray-level images, where each value is a gray (scalar) level between a lower and an upper bounds representing respectively the darkest value (black) and the lightest value (white). *Gray-level values* are usually taken on a linear scale (i.e. regularly distributed between the bounds) but other scales (e.g. logarithmic) may be used as well. A gray level often represents an intensity level, such as the brightness of a color. Though gray-level values are often expressed as integer in a range delimited by an upper (black) and lower (white) value (e.g. 0 and 255), we shall distinguish gray-level values from *integer values*. Gray-level should indeed be interpreted as a ratio between 0 (white) and 1 (black). Let us consider an 8-bit gray-level value v_1 (represented by an integer in the range $[0, 255]$), and a 16-bit gray-level value v_2 (represented by an integer in the range $[0, 65535]$)[1]. If $v_1 = 127$ and $v_2 = 32639$, both value represent the same (medium) gray-level, since $\frac{127}{255} = \frac{32639}{65535}$. On the other hand, integer values represent actual integer numbers. Integer values 127 and 32639 are different values whatever their widths.

- In many IP issues, such as segmentation or classification, we need *labels* to identify the different parts (e.g. region or classes) of an image. Labels are enumerated value types found in IP. Such labels can be symbols, names, or numbers. In the latter case, they should not be mixed up with gray-level or integer values. For example it would make no sense to add two labels 42 and 51.

- Integer values are the way the $\mathbb{N}$ and $\mathbb{Z}$ sets are approximated in computers. Likewise, *floating-point values* ("floats" for short) are an approximation of $\mathbb{R}$, implemented as a number (significand or mantissa) scaled to an exponent.

- Likewise, *complex values* can be implemented as a pair of two floating point values, standing for the real and imaginary part of the number (or the magnitude and argument of the number).

- *Colors values* can be represented in many ways. Classic color spaces include RGB (red, green blue), HSV (hue, saturation, value), HSL (hue, saturation, lightness), CMYK (cyan, magenta, yellow, key), YUV, CIE XYZ, and CIELAB.

- The pixels of a 2D image on a grid square are spatially located thanks to 2D points, composed of a row and a column numbers. Some algorithms depend on images where each pixel contains such a location. Thus *points* can be used a values.

- Algebraic objects such as *vectors*, *matrices*, *tensors* may also be used as value types.

1 An 8-bit integer encodes up to $2^8 = 256$ different values; a 16-bit integer may represent up to $2^{16} = 65536$ values.

These two aspects are *orthogonal*: they constitute two independent axes of the theoretical space of image types. Put differently, the set of image types based on the previous structures and value types can be seen as the Cartesian product of these two dimensions. Therefore the potential number of images types is very large.

Moreover some recent developments in IP are based on an inter-pixel approach, where data are not only stored on the elements (pixels, voxels) on an image, but also *between* these elements. Examples include Mathematical Morphology (MM) on graphs [39] and complexes [45] and digital geometry on pseudomanifolds [37]. These new approaches call for additional image types as well.

As a side note, let us make a distinction between an image *type* and an image (file) *format*. The former describes the structural properties of a digital image, while the latter is a specification to represent a digital image in a form suitable for Input/Output (I/O) operations (using a file, a stream, etc.).

1.1.2 *Diversity of Users*

Usually scientific software can be roughly decomposed as a set of data structures—e.g. images, finite state machines, graphs, arrays, matrices, or geometric structures— and a set of algorithms operating on them. Let us consider an extensible scientific software package, offering many tools to solve problems of the corresponding field, but where new features can also be implemented to augment its possibilities. Such a tool may be a programming library with a well-defined Application Program Interface (API) for a compiled language, such as a C++ object-oriented library; a package for a dynamic language that may be used interactively such as Python; or an integrated specialized environment for scientific work such as MATLAB. The majority of people interacting with this tool are its *end users*: they solely use the feature provided by this tool. The other, smaller involved population can be qualified as *developers*, either as direct authors of the tool or because their enrich it by adding new algorithms, and sometimes, new data structures.

We can apply this vision to IP software and broadly decompose users in three categories [65, 94]:

END USERS This category (maybe 80% of the audience) experiments and evaluates IP tools, and realizes prototypes and small applications. These users apply and assemble existing algorithms from their tool to create processing chains and solve IP problems.

DESIGNERS OF ALGORITHMS These developers (maybe 15% of the audience) invent new algorithms. These algorithms may be implementations of new techniques, extensions of existing features to address more use cases or handle more data structure or optimized versions performing better with respect to execution times, memory usage, I/O operations, etc.

PROVIDERS OF DATA STRUCTURES The last category of users is interested in adding new data structures, in particular new image types. As this operation is both rarely needed and often difficult, very few people do it (probably less than 5% of the audience). However, as we have seen earlier more and more IP works need non-classic image types. The possibility to extend the tool with respect to other aspects such as value types (e.g. a new color type) or adjacency relationships (e.g. in graph-based images) is also interesting.

IP software tools do not always targets all these categories. They almost always address the needs of end users, as most tools provide at least a minimal set of features that can be used immediately, with no need to develop algorithms. However, not all tools allow their users to extend the set of algorithms easily or even at all, for instance tools based on a Graphical User Interface (GUI) or made of small programs, each corresponding to an operation of a Command Line Interface (CLI). Lastly, very few tools address the needs of the third category of users, as algorithms are often bound to one particular data structure.

1.1.3 *Diversity of Use Cases*

In addition to targeting various categories of users, an IP tool supports various *use cases*, often related to the type of user, but not always. In particular, more experienced users may want to make use of their tool in various manner, depending on the task. The following list shows examples of various uses case of IP software.

- End users are generally more at ease with a GUI program to assemble and experiment image processing chains. Such User Interfaces (UIs) are simpler, do not require programming skills and are a faster way to develop simple programs: there is no compiling phase; a chain can be developed and run incrementally instead of being recompiled each time; inputs, intermediate results and outputs (including images, plots, histograms, simple values, etc.) can be displayed immediately; etc. Graphical interfaces may for instance propose features such as:

  - Visual programming, where each input/output and each routine of the image processing chain is represented by a graphical element (icon), connected to other collaborating elements.
  - Graphical display of images or other data (histograms, plots, values, graphs, etc.), either on-demand or automatically, when an object is updated.
  - An interactive scripting window, allowing more experienced end users to invoke routines as well as objects' methods, and possibly build and run a processing chain using a (simple) scripting interface.

- Some users may want to run their algorithms as small programs (e.g., one program per algorithm) from a CLI (e.g., a command line interpreter or "shell"), in the Unix philosophy, as small tools manipulating data (like `grep`, `sed`, `awk`, etc.) using files, pipes and shell variables to communicate. The possibility to launch a processing task in a single line may be indeed very convenient. Non trivial chains may be also saved as shell scripts for a later (re)use.

- More experienced users may want to program their processing chains in a scripting language (for example Python, Ruby or Tcl), as it is probably faster than manipulating a GUI for people knowing this language. Moreover,
  - Scripting improves the reusability of some of the work, as a script routine may be stored in its own file, and shared among users and/or uses cases.
  - Processing chains written as scripts may also use elements from the packages or libraries of the host language or platform, and even of external packages. For instance, many C or C++ libraries are wrapped into a thin layer to be made available within a dynamic language.

- Users with some knowledge of a compiled language such as C, C++, Fortran, or Ada may want to use a native programming library for this language, to benefit from a higher expressive power (as library and client code share the same programming idioms) and also because it leads to faster, compiled programs. This lower-level "UI" may be compulsory if the aim is to create a standalone program (with or without a GUI on top) or a component for an embedded platform, where resources (Central Processing Unit (CPU) time, memory, storage) may be scarce and programming facilities (available languages and libraries) limited.

Another aspect is the ability to reemploy previous work done using a given UI, with another one. For instance, a user may want to run an algorithm that they have written as a C++ routine within a visual programming environment. Or reciprocally, they may want to record a visual programming workflow as a C++ code, to be modified and compiled later—for improved performances and/or standalone use. Other scenarios include the extension of the C++ core framework in a dynamic language such as Python or Ruby, by providing new algorithms, but also data structures (images types).

The notion of use case also covers situations specific to the application domain, namely IP. We can mention the following cases as examples of useful features.

- It is sometimes interesting to process a subset of an image, expressed as a Region of Interest (ROI) box drawn in a GUI, as a user-defined mask, as a region computed by a previous algorithm, or a set defined by a predicate on the pixel values (such as a threshold).
- Likewise, one may want to process a single slice of a 3D volume instead of the whole set.
- Users may want to define their own neighborhood relationship for any image type, so as to change the behavior of their algorithm.
- For some value types, in particular color types, there is no natural order relation, although many algorithms (e.g. in mathematical morphology) require one. Several propositions to define such a relation however exist. We may want to be able to integrate one of them into our algorithms.

These features are orthogonal to algorithms. A naive implementation would either require changes in every routines—a practice that does not scale well from a software engineering point of view—or creating new inputs, which implies data duplication having an impact on performances and memory usage. Very few IP tools address many of these use cases elegantly and efficiently.

1.1.4 *Diversity of Tools*

Consequently there are plenty of software tools for IP, targeting a specific domain, a category of users, determined uses cases, and/or providing support for given data types. We have enumerated more than 80 "tools" in a broad sense during our study of existing IP software. Only some of them are mentioned hereafter.

We can roughly classify IP software tools in the following categories. This categorization is admittedly limited, as some projects may belong to several categories. Likewise some tools have not been specifically designed to process images—though they can be used to solve IP issues.

COMMAND LINE UTILITIES These packages are made of several programs (or "binaries") using a Command Line Interface (CLI) each performing one operation or more. These programs can either be invoked from a text-mode terminal or run as a shell script. Data are passed to programs as arguments. Programs may accept options passed as arguments as well, for instance to select an operation among the available ones, set the value of a parameter, choose the name of an output file, etc. Command lines utilities are notably proposed by projects such as ImageMagick [128], GraphicsMagick [71] and MegaWave [61].

GRAPHIC EDITORS This category contains programs focusing on a GUI offering image visualization and manipulation services. IP operations can be applied by selecting an item in a pull-down menu or by typing simple commands in a small interpreter window. These tools usually support a limited set of image types (such as 2D and 3D regular images) though they often support many file formats. They provide services to perform image analysis and in-place processing tasks. This set of operations can sometimes be extended thanks to macro-commands (or simply "macros"), with a dynamic language such as Python, or more generally with compiled "plug-ins" written in C or Java. Tools in this category include applications such ImLab [121], Imview [135], ImageJ [3, 31] and MICROMORPH [33].

VISUAL PROGRAMMING ENVIRONMENTS A visual programming environment is a platform where a user can graphically assemble IP operations as a graph connecting filters and data flows. The obtained workflow can be easily modified or extended, run interactively, and told to display intermediate images. The resulting flexibility and ease of use of this approach makes visual programming environments convenient tools for rapid experimentation and prototyping. This category includes tools such as VisiQuest (formerly Khoros) and XIP [8].

INTEGRATED ENVIRONMENTS This category features platforms for scientific computing and numerical analysis, providing both a programming language and a graphical environment to execute commands, run scripts, and visualize data (arrays, matrices, signals, images, etc.). MATLAB [102], Mathematica [151, 152], Octave [55, 49] and Scilab [7] are well-known examples of this kind of tools.

PACKAGES FOR DYNAMIC LANGUAGES There is a growing trend towards using dynamic language such as Python [118] or Tcl [114] for scientific computing. Several packages have been developed to add scientific computing facilities to these languages, including image-related features. Regarding Python, we can mention projects such as PIL [5], Mamba [21] or Sage [6].

PROGRAMMING LIBRARIES This category is probably the largest of our classification. For a long time, IP services have been implemented as programming libraries for compiled languages such as Fortran [85], C [86], or C++ [87], because these languages have been traditionally well-known, widely available and efficient. Libraries for compiled languages continue to be developed and maintained, as they offer very good performances. Examples of such libraries include CImg [137], DGtal [1], Gandalf [2], GENIAL, GIL [12], ImaGene [54], ImLib3D [26], ITK [113, 75], Leptonica [23], Morph-M [59], OpenCV [27], Pink [53], Qgar [100], VIGRA [56, 91], VSIPL++ [68] and Yayi [51].

DOMAIN SPECIFIC LANGUAGES In some cases, developers have considered the library-based approach insufficient to properly implement IP software, because the host language was too distant from IP considerations, or not powerful enough to express efficient solutions. Thus they have chosen the solution of creating a new language dedicated to IP, known as a Domain Specific Language (DSL). Such languages feature image idioms and may be optimized to process data efficiently. This approach offers much more possibilities, often at the price of an important development effort: in addition to a compiler or interpreter for the new DSL, components such as debuggers, profiling tools, libraries for general-purpose tasks or bindings to third-party languages way be necessary. DSL for IP include the Apply [74] and Neon [73] languages.

1.2 TOPIC OF THE THESIS

In the previous section, we have emphasized the polymorphic nature of the IP software domain: the multiplicity of data types, users and use cases is such that we cannot initially envision a single tool to address them all. What is more, new IP techniques or image types are rarely implemented using an existing framework; instead, new tools are constantly being developed. For instance, many research teams in IP, pattern recognition or computer vision have developed their own library to implement their algorithms. IP software is thus characterized by a lack of *reusability*.

This thesis tries to address the question of reusable IP software. Can we design a tool not dedicated to an image type, an application domain or an IP technique, able to absorb new data types, algorithms and UIs?

To answer this question we must first choose an implementation context (programming language(s), programming paradigm(s), run-time environment, etc.) to host our potential solution, with the goal of reusable software in mind. This choice may involve an existing framework, the extension of such a framework, or even the creation of a new one. This work could possibly include the extension of an existing language, the creation of a new DSL, the definition of a new programming paradigm, etc.

Another issue is related to the diversity of image types. Handling *all* of these types in a naive way normally requires dedicated implementations for each type. Alternative implementations might introduce degradation of data or performances due to a simplified framework (for instance if the design includes a single value type such as the `double` floating-point type) and/or run-time overheads introduced by the implementation framework. The quality of the proposal shall not be evaluated by the number of available image types, but by the ability to add new types to the framework with minimal work.

The reusability of the framework also depends on the generality of algorithms. Firstly, for each algorithm there should be a single implementation compatible with all eligible data structures. Secondly, the addition of a new image type shall not invalidate existing algorithms nor require a new implementation. This issue raises the question of algorithms *genericity*: instead of featuring specific implementations for each image type, our ideal solution should provide a single, generic algorithm implementation. In addition, a reusable solution should be flexible enough to support classic IP idioms mentioned previously (restricting an input to a ROI, changing the neighborhood relation in an algorithm, defining an order on a value type, etc.), with not impact whatsoever on algorithms.

Performance issues should also be addressed. As in most domains of scientific computing, IP software tools are expected to deliver efficient solutions. The need for fast IP tools may be driven by large data sets (either numerous or voluminous) or real-time processing constraints. Note that even IP prototyping tasks may require a solution with good run-time performances. Consider for example the case of very large images such as the one manipulated in astronomy (of several gigabytes). To be practically usable, a prototyping environment should process these images as fast as possible. Processing a sub-sample of these inputs to reduce processing times cannot be considered as a correct alternative, as input data have been *changed* and the obtained results may not be representative of an application to the whole images.

Finally, in addition to its reusability, we shall consider the *usability* of our proposal. This is especially relevant in this context since all IP practitioners are not Computer Science (CS) experts. They would likely favor a limited and non-reusable but user-friendly solution over a powerful and efficient tool requiring a deep knowledge of an obscure programming language, or failing to provide accurate error messages.

This thesis examines these issues and proposes a solution trying to address all of the above aspects. The goal is to ultimately propose a reusable software architecture for IP.

1.3 CONTRIBUTIONS

This section briefly presents the key elements of our proposal of a reusable software tool, along with our contributions. We start with the implementation context. Following the example of many modern scientific computing frameworks, we have chosen a solution based on a *generic C++ library*, forming the core of our architecture proposal.

Our work starts with an analysis of the Generic Programming (GP) paradigm. We outline the benefits of this approach in the context of scientific applications in general and IP in particular. We compare GP to another popular programming paradigm, Object-Oriented Programming (OOP), which is widely used in problems with an emphasis on data structures such as ours. We then propose a new programming paradigm, SCOOP (Static C++ Object-Oriented Programming), mixing the benefits of GP and OOP, that is well suited to the implementation of scientific software. SCOOP encourages framework developers to define orthogonal *abstractions* representing fundamental concepts of their application domain, as well as *properties* characterizing concrete realizations of these abstractions.

We then explain the choice of C++ as implementation language for IP software, and the decision to base our architecture on a generic library. Using the SCOOP paradigm, we propose an organization of IP concepts in order to define the abstractions mentioned previously. In IP, such abstractions represent essential notions such as images, points, domains, neighborhoods, structuring elements, values or functions. We also highlight the connections between these elements and their role in the creation of actual IP routines.

To demonstrate the suitability of this approach, we present several generic implementations of image data types, ranging from classical regular structures to less common mathematical objects. We then address the question of IP algorithms for the generic and efficiency points of view. In particular, we present the concept of generic optimizations to preserve performances in generic software.

Finally, we present some examples of generic image algorithms and processing chains in the fields of mathematical morphology.

The ideas presented in this thesis have been implemented in a generic C++ library, *Milena*. This library is part of the latest release of a platform for generic and efficient IP, *Olena* [52]. This project is Free Software distributed under the terms of the GNU General Public License (GNU GPL).

Although this work is placed in the context of IP, this thesis emphasizes its software engineering contributions. The approach proposed here is indeed not tied to IP. Other scientific domains may adopt our strategy to create reusable software, including

- A generic design.
- The use of the SCOOP paradigm.
- The formalization of orthogonal concepts of the target domain with their properties.
- The construction of data types based on these abstractions.
- The development of generic algorithms.
- The introduction of generic optimizations to address performance issues.

1.4 CONTENTS OF THE THESIS

This thesis is structured as follows:

- Chapter 2 covers the Generic Programming (GP) paradigm. The principles and origins GP are presented. We also present the Standard Template Library (STL), a generic library that pioneered essential ideas, such as the notion of *concepts*. This chapter also contains a comparison of GP and OOP. It closes with an explanation of *static metaprogramming*, a technique to implement compile-time programs used in the SCOOP paradigm and in the implementation of the Milena library in general.
- The Static C++ Object-Oriented Programming (SCOOP) paradigm is presented in Chapter 3. The chapter starts with an explanation of a programming idiom at the root of SCOOP, and then presents key elements of the paradigm: concepts, implementation classes, generic algorithms, compile-time

verifications, properties and *morphers*. This last item proposes to implement "object transformations" modifying the behavior of algorithms (e.g. restricting an image to a ROI, applying a rotation to an image, thresholding its values, etc.—before passing this image to an algorithm).

- Chapter 4 shows how GP can be used in the design and implementation of IP software. We first present the motivation for creating generic IP software. We also show why C++ and SCOOP are relevant choices in this context. We then present the organization of our proposal, based on a generic core library. The elements of this library are then detailed: concepts, data structures, algorithms and morphers. While the previous chapter is essentially software engineering, this chapter is more IP-oriented.

- Chapter 5 presents applications of generic IP algorithms used with various data types.

- The last (6th) chapter concludes this thesis by summarizing and analyzing the solutions proposed in this work. We also present future perspectives regarding the evolution of the whole architecture. In particular, we present a solution to use a compile-time library such as Milena in dynamic environments.

2 GENERIC PROGRAMMING

This chapter presents the GP paradigm, its goals and uses. We compare it with OOP and study their pros and cons. We advocate the use of GP as a basis to implement an efficient and reusable software IP framework. We present GP in the C++ context, featured by the `template` keyword. In addition to using GP, we present another technique called static metaprogramming, based on C++ templates, which is a complement to GP that enables us to perform computations at compile time (e.g, functions on C++ types, static computations on integral values, etc.).

Many (if not most) tools presented in the Introduction fail to handle the many different kinds of data in Image Processing (IP) because they have not been designed with the goal of being *reusable software* in mind. Reusing software is a long-standing software engineering issue, which can be addressed by the use of *programming libraries*. In a broad sense, a library is collection of reusable software units, ranging from small entities like functions, data types, or classes; to larger elements like modules, parameterized modules, templates or functors. The actual definition, size and scope of a library varies with the language; whether the library addresses low- (e.g. system or hardware-related) or high-level (e.g. application-related) issues; if it is specific or general, etc. Depending on the context, the content of a library is made of code (compiled sources, sources to be compiled, sources to be interpreted) and may contain documentation and even meta-data providing information on the library's components (e.g., for documentation or introspection purpose).

Examples of successful libraries are the standard C library, the Unix Application Program Interfaces (APIs) or the GTK+ GUI toolkit. All provide reusable software elements which are the result of factoring effort with respect to design, implementation, optimization, testing and distribution. Creating and using reusable components is not only a gain of time as far as development is concerned; it brings a shared knowledge among their users by establishing common practices, terminologies, and idioms. In this respect, many libraries are more than implementation facilities, as they provide a *language* for developers, much as design patterns [62] are a language for designers.

However, many libraries, despite their usefulness, are limited by their interface—the set of provided services—as they support only a pre-determined set of types [44]. Data having a type that was not taken into account at the time of design and implementation of such a library cannot be reliably used: the library may work as expected but it is more likely that it will be unable to handle such data or produce degraded or erroneous results. For instance, the family of power functions from the C99 and C11 standards [80, 86] (resp. `powf`, `pow`, `powl`, `cpowf`, `cpow` and `cpowl`) works with built-in types (resp. `float`, `double`, `long double`, `float complex`, `double complex` and `long double complex`), but cannot handle a user-defined type (e.g. a fixed-precision number).

This example illustrates a need for *reuse without modification*: many libraries are not flexible enough to cope with new contexts of use. From a very general point of view, programs can be decomposed into algorithms and data types. However, the former are often tied to the later, thus preventing any other combinations of existing algorithms with new data types. Separating algorithms from data structures and keeping them as orthogonal as possible is one of the incentives behind programming paradigms such as Generic Programming (GP) and Object-Oriented Programming (OOP).

In this chapter, we present the GP paradigm and see how it can be useful in the design and implementation of reusable software libraries, especially in scientific applications like IP. We compare it with OOP and show why we consider it a better strategy for efficient and reusable software design. Most examples of this chapter make use of the C++ programming language.

2.1 ELEMENTS OF GENERIC PROGRAMMING

The limitations of traditional libraries mentioned above are due to a rigid interface that cannot be adapted to new input data types. In compiled programming languages, were libraries are composed of a compiled implementation (e.g. '`lib.a`', '`lib.so`', '`lib.dll`', etc.) and an interface for the compiled language (e.g., `lib.h` in C), the emitted code (bound to the target hardware architecture) is specific to the data types involved in the interface (e.g., the types of a routine's arguments). This code is not designed for other data types.

A solution to overcome this restriction is to create programming entities not strictly bound to their data types, by turning them into *adjustable parameters*. This strategy is made possible by the fact that the code of many algorithms, data structures and other language elements is not fundamentally bound to the data types they manipulate. For example, consider the following C implementation of the function `sqrf` computing the square of a floating point value.

```
float sqrf(float x) { return x * x; }
```

If we were to implement this function for other types such as `int` or `double`, we would reuse the same code, except for types. This could be achieved by turning the type `float` used in this function into a parameter τ, and making this definition valid for any type τ:

```
∀τ,  τ sqr(τ x) { return x * x; }
```

This new implementation is no longer bound to a specific type: `sqr` can be considered a *generic function*.

This idea of *parameterization* of elements such as functions or data structures is a central notion of Generic Programming. GP is a programming language paradigm enabling the definition of generic entities through the use of parameters. Several kinds of programming language notions can be parameterized: routines (functions, procedures, methods), data types, modules, etc. From a functional point of view, a parameterized entity can be seen a mapping from a set of parameters (e.g., types) to a concrete entity (e.g. a function or a type).

2.1.1 *Generic Algorithms*

The definition of `sqr` above is not actually valid C++ syntax. In C++, generic entities are expressed as *templates*. A parameterized function is introduced with the `template` keyword together with a list of parameters between angle brackets ('<' and '>'). The `typename` (or `class`) keyword before a parameter means this parameter stands for a type. With these notations, the `sqr` functions reads in actual C++ syntax as

```
template <typename T>
T sqr(T x) { return x * x; }
```

Using a function template is similar to using a classic function:

```
sqr(3.14f);    // Calls srq<float>.
sqr(42);       // Calls srq<int>.
```

The template parameter `T` is deduced (inferred) from the type of the argument passed to `sqr`. In the first line, `3.14f`, which is a literal value of type `float`, propagates its type to `sqr`. Since `sqr` is not a true function, but a function template, the compiler needs to create the code of the function `sqr` where the formal parameters (`T`) are replaced by their actual values (`int`, in the case of the first call). This generation mechanism is called *template instantiation*[1]. In this respect, templates works a bit like macros of the C++ preprocessor, although there are many differences between the two features.The function template is said to be

1 Template instantiation should not be mixed up with *class instantiation*, which is the creation of objects (instance) from classes. See also Section 2.5.1 (p. 63)

(fully) *specialized* when (all) its parameters are given an actual value. The name of a specialized function template is composed of the name of the template followed by the list of the actual parameters in angle brackets. The names of the specializations of `sqr` in the previous example are respectively `srq<float>` and `sqr<int>`.

In this thesis, we always reserve the name *parameter* for template parameters, while *argument* refers to the arguments of a function (either defined or called). We will qualify a parameter (resp. an argument) as *formal* if it is part of a template definition (resp. a function definition) and as *effective* if it is part of a template instantiation (resp. a function call).

In the previous definition of `sqr`, `T` is a formal parameter while `x` is a formal argument, since they belong to the definition of the function template. In the first instantiation of `sqr`, `float` is an effective (implicit) parameter, while `3.14f` is an effective argument.

An algorithm written as a function template is said to be generic if it can be applied to any "meaningful" input, i.e. which complies with the requirements of an abstract definition of this algorithms. For instance, a sorting algorithms may require its input to define comparison operations (e.g. an operator <) on the sorted elements. Generic algorithms, in particular generic IP algorithms, are studied throughout this thesis. The notion of requirements of templates is addressed particularly in Section 2.4 (p. 52) and in Section 3.3 (p. 79).

2.1.2 *Generic Data Types*

In addition to functions templates, C++ supports the notion of generic data types trough *class templates*. The syntax is similar to function templates: the `template` keywords followed by the list of parameters is placed in front of a `class` (or `struct`) definition, and parameters may be used within this definition.

Classes templates are especially useful to implement generic containers. For instance, the following lines implement a very simple singly linked list:

```
template <typename T>
struct list
{
  T data;
  list* next;
};
```

Unlike function templates, class templates cannot be instantiated by inferring the parameters from the context of use: the full name of the template must be supplied together with its parameters.

To create a list of three elements, we can create three values of type `list<int>`.

```
list<int> l1 = {1, nullptr};
list<int> l2 = {2, &l1};
list<int> l3 = {3, &l2};
```

The instantiation of the `list<int>` type is triggered the first time the compiler sees it. Note that `list` alone does not name a type, but a (class) template, whereas `list<int>` (as well as `list<float>`, `list<double>`, etc.) is a type: `list` and `list<int>` belong to different *kinds*.

Like generic functions, generic containers are reusable units of code: a single definition may be used many times with different parameters. Moreover, the type of the data stored in such a container is known a compile time, enabling the compiler to detect errors early and possibly to perform optimizations. Containers based on dynamic types and typeless containers are more error prone and less efficient.

2.1.3 *Instantiation of C++ Templates*

Templates can be seen as *generators*. For instance, the following function template `f` having a type parameter `T`

```
template <typename T>
T f(T x) { return x * x; }
```

can be interpreted as function:

$$f_T : \left\{ \begin{array}{ccc} T & \rightarrow & T \\ x & \mapsto & x \times x \end{array} \right.$$

The resolution of C++ templates is *static*: all template-related features are handled at compile time. Therefore, an instruction such as

```
f(51);
```

is two-fold:

- at *compile time*, it first triggers the instantiation of the code of the specialized function `f<int>` (if it has not yet been instantiated);
- at *run time*, it executes the statement `f<int>(51)`.

This two-level mechanism adds expressive power to the language, but also restricts the scope of templates: one cannot instantiate templates past compile time. This may be a problem when the context of use of a template is only known at run time. Some solutions to this issue are proposed in Section 6.3.1 (p. 174).

As for compile- and run-time performances, each unique specialization of templates leads to code generation and compilation, possibly increasing the generated code and compilation times. However, since the compiler generates dedicated code for each instantiation, it has the opportunity to optimize it.

2.1.4 *Generic Libraries*

In the same way libraries are collections of reusable code, it is possible to create *generic libraries* made of reusable generic code. In C++ such libraries are composed of function and class templates. But unlike traditional libraries, which contains *compiled code* (e.g., files ending in '`.a`', '`.so`', '`.lib`', '`.dll`', etc.), generic libraries are merely collection or source code containing templates. Indeed generic algorithms and data types cannot be turned into compiled code unless they are specialized. Thus to preserve their most general (abstract) form, they must be distributed and installed as source files.

The most famous generic library is probably the Standard Template Library (STL). The STL is a collection of generic containers (character strings, vectors, lists, dictionaries, queues, etc.) and generic algorithms for these data structures (for sorting, searching, copying, applying transformations, etc.).

STL data structures and algorithms are *orthogonal*: any STL algorithm can be applied to any STL container, as long as the combination is valid (for instance, a random shuffling algorithm cannot be applied to a container that does not support random accesses). Therefore, data structures and algorithms from the STL are *loosely coupled*: the former can be extended irrespective of the later, and vice versa. This powerful property is due to the use of *iterators*, acting as abstract handles on the values of a container. The concept of iterator belongs to the set of *design patterns* presented by Gamma et al. [62]. STL algorithms do not take containers as input, but iterators: therefore, they are not tied to a particular data structure, or even to an abstract interface (e.g., the interface of a sequential or random-access container). The only interface between containers and algorithms are iterators: they are provided by the former and use by the latter. The STL algorithms have been designed to be also compatible with ordinary C arrays, as C pointers can be used as iterators as well. Most of the STL has been integrated into the ISO C++ standard library [79].

We can illustrate the possibilities of the STL with the following example: a dynamic array (`std::vector<int>`) of three integers is created (`push_back()` calls), sorted (`std::sort`), and copied (`std::copy`) into a linked list of floating point values (std::list<float>), with on-the-fly conversion of int values into float ones.

```
std::vector<int> ints;
ints.push_back(2);
ints.push_back(3);
ints.push_back(1);
std::sort(ints.begin(), ints.end());

std::list<float> floats;
std::copy(ints.begin(), ints.end(),
          std::back_inserter(floats));
```

Method calls *c*.begin() and *c*.end() return iterators pointing to the beginning and (past) the end of the container *c*. std::back_inserter creates a special iterator inserting data at the end of its container.

In addition to containers, algorithms and iterators, the STL defines a set of *concepts* defining syntactic and semantic requirements over types. Concepts are addressed in Section 2.4 (p. 52). Section 2.2.3 (p. 38) provides additional information on the STL. Besides the STL, there are many successful generic libraries in C++. We present some of them in Section 2.3 (p. 49).

Our proposal for a generic image processing platform is centered on a generic image processing library. We reuse and extend ideas that have been made popular by the STL and other libraries, including data structure/algorithm uncoupling, iterator-based traversals, etc.

2.2 HISTORY OF GENERIC PROGRAMMING

2.2.1 *CLU*

The first ideas of generic programming before it was named like this date back to the invention of the CLU language in 1974 by Barbara Liskov and her students [98]. Among many programming concepts including data abstraction (encapsulation) and iterators, the language features *parameterized modules*. In CLU, modules are implemented as *clusters* which are programming unit grouping a data type and its operations. CLU's procedures, iterators and clusters can be parameterized. This feature introduced the notion of *parametric polymorphism*, that is the ability to define a generic function or data type having always the same behavior, whatever the types of the values handled, while preserving full static type-safety.

Initially, parameters were checked at run time in CLU. Then, where-clauses were introduced to specify requirements over parameters. Only the operations of the type parameter listed in the where-clause may be used, enabling complete compile-time check of parameterized modules, as well as the generation of a single code. For instance, the following declaration announces a cluster 'set' parameterized by a type t which is expected to provide an equal operation. Inside set, the only valid operation on t values is equal.

```
set = cluster [t: type] is
  create, member, size, insert, delete,
  elements
where
  t has equal: proctype (t, t) returns (bool)
```

set[t] represents a set of values such as $\forall x \in$ set[t], x is of type t.

The syntax *module*[*parameter*] binds a module and its parameter, and is called *instantiation* [16]. Instantiations in CLU are dynamic: a new object module is created once for each distinct set of parameter values. Each type used as a parameter is represented by a type descriptor. Instantiated modules are created from a non-instantiated module object, where the type descriptor representing the type parameter is replaced by a type descriptor of an actual type. In this respect, CLU differs from C++, where all instantiations are done at compile-time.

Atkinson et al. discussed the pros and cons of compile-time versus load- and run-time binding strategies [16]. Compile-time binding has the drawback of creating an instantiated module code per set of parameters, which may lead to a combinatorial explosion if many different instantiations are done, or worse if a comprehensive set of instantiations is required (the topic of combinatorial explosion of instantiation is discussed in Section 4.1, p. 113). The compile-time binding scheme is akin to macro expansion: the code is generated by substituting actual values in a template. Therefore, the compiler benefits from a concrete context for each specific instantiation, and may be able to generate optimized code, at the expense of longer compilation times (as in C++). On the other hand the load- or run-time strategy requires a single compiled code for each parameterized module, independent of the parameters' values, and shared by all instantiations. Parameters are passed at run-time as objects (type descriptors).

2.2.2 *Ada*

Ada is a programming language which was started in 1977 and appeared in 1980. It was standardized in 1983 (then 1995 and 2005 [82]). Ada features *generic packages* and *generic subprograms* (routines) through the generic keyword. The following example shows a generic implementation of a routine swapping its arguments.

```
generic
  type T is private;
procedure swap (x, y : in out T) is
  t : T
begin
  t := x; x := y; y := t;
end swap;
```

Parameterized packages and routines must be instantiated explicitly: they cannot be generated implicitly from the context of use, as in C++. For instance, the following two lines instantiate `swap` for integers and character strings, giving an explicit name to each instantiation.

```
procedure int_swap is new swap (INTEGER);
procedure str_swap is new swap (STRING);
```

Unlike in C++, compilation of generics can be independent of use in Ada, leading to "shared generics".

Ada supports syntactic constraints on parameters like CLU. For instance, a generic minimum function requires an order relation. In the following generic function definition, this requirement is enforced by the `with` clause requiring a binary function '<=' on the parameter's (`T`'s) values:

```
generic
  type T is private;
  with function "<=" (a, b : T) return BOOLEAN
    is <>;
function minimum (x, y : T) return T is
  begin
    if x <= y then
    return x;
  else
    return y;
  end if;
end minimum;
```

Instantiation of constrained generics can either be fully qualified:

```
-- Here 'T1' is a type and 'T1_le' is an order
-- relation on 'T1'.
function T1_minimum is new minimum (T1, T1_le);
```

or take advantage of implicit names (here, the comparison function is already known as "<=" on `INTEGER` values):

```
function int_minimum is new minimum (INTEGER);
```

The idea of Generic Programming was introduced by Musser and Stepanov in Ada [108]. They pioneered the use of abstractions to defined generic algorithms independent of the processed data. The Ada standard library was augmented in the 2005 ISO standard with a library of generic containers inspired by the STL.

2.2.3 *C++, Templates and the Standard Template Library*

Bjarne Stroustrup, the initial designer and implementer of C++ considered parameterized types in the first design of C++. However, their introduction was postponed because of time constraints and complexity reasons.

Before Templates

The initial incentive to implement templates in C++ was parameterized containers, as in CLU [130]; before, macros were used. For instance, a generic vector class may be implemented with the following `GEN_VECTOR` macro:

```
#define VECTOR(T) vector_ ## T

#define GEN_VECTOR(T)                          \
  class VECTOR(T) {                            \
  public:                                      \
    typedef T value_type;                      \
                                               \
    /* Constructors.  */                       \
    VECTOR(T)() { /* ... */ }                  \
    VECTOR(T)(size_t n) { /* ... */ }          \
                                               \
    value_type&                                \
    operator[](size_t n) { /* ... */ }         \
                                               \
    /* ... */                                  \
  }
```

When `GEN_VECTOR` is invoked with a type *T* (e.g. `int`) it expands into a definition of class `vector_`*T* (`vector_int`). The `VECTOR(`*T*`)` macro is a shortcut for `vector_`*T*[2]. Such a macro must be invoked ahead of any use for a given set of parameters: this macro calls is similar to Ada's explicit instantiation of generics. The following lines triggers the instantiations of `int` and `long` vectors.

```
GEN_VECTOR(int);  // Instantiate VECTOR(int).
GEN_VECTOR(long); // Instantiate VECTOR(long).
```

2 "##" is the concatenation operator of the C++ preprocessor, used to join two text words.

Once instantiated these macro-based generic data types can be used similarly to templates:

```
VECTOR(int) vi;  // Create a VECTOR(int).
VECTOR(long) vl; // Create a VECTOR(long).
```

However, macro-based generic containers have many limitations. Most of them come from the fact that macros are handled by the C++ preprocessor before the actual compilation of C++ code. The preprocessor merely acts as a code generator and is indeed not aware of the semantics of the C++ language. Among the drawbacks of macros, we may mention the fact that they are not aware of types; they do not obey any scoping mechanism; they do not support recursion; and they are not well handled by tools (e.g., debuggers), as most of them work with preprocessed source code, after macros have been expanded. Because of these limitations, a solution for actual and sound generic containers was sought.

Addition of Templates to C++

Two paths to address the issue of generic containers were considered during the evolution of the C++ language:

- The Smalltalk approach, based on dynamic typing and inheritance (dynamic polymorphism);
- The CLU approach, based on static typing and type parameters (static polymorphism).

The latter is more complex and less flexible than the former but is more efficient and safer with respect to types, and was eventually chosen.

Stroustrup presented in 1988 a first design for *templates* [129], a feature enabling the creation of generic containers (presented in Section 2.1.2, p. 32). The first implementation of templates in Cfront, the first C++ compiler from AT&T Bell Laboratories, supported only class templates, but was later expanded to function templates. A first minimal template mechanism was described in *The Annotated C++ Reference Manual* (ARM) [50]. Templates were accepted by the ANSI C++ committee for inclusion in the upcoming standard; templates or "fake templates" were already used in many projects, which facilitated their adoption by the committee [130].

Features of C++ templates

This paragraph enumerates a few features of C++ templates; for more details about templates, see the book by Vandevoorde and Josuttis [139][3].

3 We follow the conventions of Vandevoorde and Josuttis and use the terms *class template* and *function template* instead of *template class* and *template function*.

INSTANTIATION TIME Instantiation of C++ templates is based on code generation at compile- or link-time (the latter is supported by very few compilers, despite being a part of the current standard [81]);

IMPLICIT INSTANTIATION By default, if the implementation (body) of a template is available, it is implicitly (automatically) instantiated, for each set of parameters used. This is the dominant instantiation model, which requires generic libraries to provide the full implementation of their templates together with their declarations in headers[4].

EXPLICIT INSTANTIATION In addition to the implicit instantiation model, C++ features an explicit (manual) template instantiation mechanism. This scheme is more complex than implicit instantiation, but enables a finer control of templates, and may be required if design constraints impose a limited, specific set of template instantiations (for instance in the context of embedded systems).

EXPLICIT SPECIALIZATION C++ features an original mechanism allowing users to provide their own specialization of a given template, in addition to the generic definition. This mechanism is useful to implement dedicated implementations for some parameters, which may perform better than the generic definition. Specializations are also preceded by the `typename` keyword, but *bound* (i.e. fixed) parameters are removed from the list of parameter, and the name of the specialized class or function is followed by the list of parameters (possibly containing both bound and unbound parameters). For instance, in the case of a container of values of type `T`, there may be a better implementation when `T` = `bool`. Such an implementation can be realized as an explicit specialization:

```cpp
template <typename T>
class container
{
  // Generic implementation.
  // ...
};

template <>
class container<bool>
{
  // Dedicated implementation for T = bool.
  // ...
```

4 Headers of non generic libraries are usually made of declarations only and do not include definitions (statements).

```
};
```

When there are more than one parameter, it is possible to provide partial specializations. For instance, the following dictionary class, mapping values of type `T` to values of type `V`, can be partially specialized for `V` = `bool`.

```
template <typename T, typename V>
class dictionary
{
  // Generic implementation.
  // ...
};

template <typename T>
class dictionary<T, bool>
{
  // Dedicated implementation for V = bool.
  // ...
};
```

The compiler always picks the most suitable definition of a template[5]. In the previous example, the second definition would be chosen over the first one to generate the code of `dictionary<char, bool>` as it is a more appropriate choice.

Explicit template specialization is especially useful in static metaprogramming, a powerful feature of C++ (see Section 2.6.1, p. 69).

NON-TYPE TEMPLATE PARAMETERS In addition to type parameters, preceded by the `typename` (or `class` keyword), C++ allows non-type parameters, such as `int`. Such parameters act as constants in the definition of the template. For instance, the following function template implements the parameterized function $f_a : x \in \mathbb{R} \rightarrow sin(ax), a \in \mathbb{Z}$:

```
template <int a>
float f(float x) { return sin(a * x); }
```

The integer `a` is a parameter bound at compile time, while `x` is an argument, the value of which is known at run time only. This feature was also present in CLU [98].

Lack of Explicit Constraint Mechanism

A non-feature of C++ templates is the lack of a syntax to express constraints on parameters, as CLU and Ada do. The lack of constrained genericity is not a strong deficiency, as each template

5 Unless there are ambiguities due to concurrent specializations.

instantiation triggers code generation (after parameter substitution); if an actual parameter does not fit the (implicit) requirements imposed by the template, an error will be produced *inside* the template, which will halt the compilation. This mechanism may however be cumbersome, since the user is warned indirectly: the compiler will report the error(s) triggered by the attempt to generate the code of the template with the given parameter(s), instead of precisely identifying an invalid parameter. Moreover, error messages may be long and complex, since template instantiations may result from the instantiation of other templates. This is especially true in the context of generic libraries, were most data types and routines are parameterized. Instantiation error messages from the compiler then show the "stack" of successive template instantiations up to the point where the error was detected. For an example, see Section 2.4.2 (p. 58).

Mechanisms enforcing named (or nominal) conformance (e.g. through derivation) and structural conformance (e.g. through constraint clauses, as in CLU or Ada) of template parameters have been considered during the design of C++, but none has been integrated to the standard yet. However, despite not enforced by tools, constraints on template parameters have been formalized in the context of generic libraries by the introduction of *concepts* (discussed in Section 2.4, p. 52) in the STL, reused by many other projects. Concepts are conventions expressed as documentation that are intended for the programmers, not the tools, although they may be expressed and checked to some extent (see Section 2.4.2, p. 58).

The Standard Template Library

The Standard Template Library (STL) previously mentioned in Section 2.1.4 (p. 34) is the first library of generic algorithms and data structures created for C++. It was initially designed and implemented by Alexander Stepanov, with the help of Meng Lee and later David Musser. The ideas behind the STL originates from the work on GP in Ada. According to Musser et al.,

> [the STL was created] with four ideas in mind: generic programming, abstractness without loss of efficiency, the Von Neumann computation model, and value semantics [110].

COMPONENTS The STL is a collection of template components for C++. This paragraphs presents the different elements of the library.

Containers The STL contains *generic data structures* such as doubly linked lists (`std::list<T>`), dynamic arrays (`std::vector<T>`), sets (`std::set<T>`), storing elements of a given type (`T`). Also provided are associative containers mapping key objects (of type `K`) to data objects (of type `T`), such as dictionaries (`std::map<T, K>`). In addition to these core containers, the STL comes with container adapters, that is containers built on top of other containers. For instance, LIFO (Last In, First Out) stacks (`std::stack<T, S>`) and FIFO (First In, First Out) queues (`std::queue<T, S>`) are containers based on a Sequence container (of type `S`). Sequence represent a set of constraints on the type used to built a stack or a queue, called a *concept* (see below).

Algorithms The library also provides *generic algorithms*, working with the containers. Instead of being tied to specific input (and output) data types, these algorithms are implemented as template functions. Moreover, they do not take container objects as input, but *iterators* (see below). Algorithms may express requirements over the processed data; e.g., the `std::sort` algorithm only works with random-accessible containers. To express such constraints, these algorithms impose their input iterators some concepts to satisfy (e.g. RandomAccessIterator), as container adapters do with respect to their underlying container (see above).

Iterators An iterator is an object acting as a *location*, *handle* or *pointer* to some data (most often in a container). In the STL, access to the data pointed by an iterator mimics the C++ pointer dereferencing operation (using operator '`*`'). If `i` points to some element of an STL vector of `int`s (`std::vector<int>`), `*i` returns this element. In addition to accessing values, iterators provide traversal (or *iteration*) services. These services vary with the capabilities of the iterator, depending on the container's capabilities (if applicable). For instance, an iterator on a container that supports one-way (also called *forward*) traversal provides an operator '++' which advances the iterator, making it point at the next element. In the previous example, `i++` would make `i` point at the next element. If the container support two-way (or *bidirectional*) iteration, then it proposes the converse operation, namely operator '`--`'. Finally, an iterator on a random-accessible containers can advance (resp. move back) several steps at once. Such an iterator supplies an operator '+=' (resp. '-=') taking an integer as argument. The iterator of the previous example can be moved forward (resp. backward) by five cells using `i += 5` (resp. `i -= 5`). This syntax mimics C++ pointer arithmetic on purpose. Iterators extends the notion of pointer on a classic (built-in) array to new containers while reusing the same syntax. Moreover, pointers themselves can be used as iterators within STL containers, making C++ arrays containers compatible with STL algorithms. Iterators can be obtained from containers, most often using its methods `begin()` and `end()`; the former returns an iterator pointing to the beginning of the container, while the latter returns an iterator pointing just after the last element of the container (called *past-the-end* iterator). Therefore, traversing a container `c` of type `C` boils down to a simple loop:

```
for (C::iterator i = c.begin();
     i != c.end(); ++i)
  // ...
```

where `C::iterator` is a `typedef` (type alias) of the iterator's type. STL algorithms make use of similar algorithm constructs, and therefore often take two iterators (or more) as input. For instance, the sorting algorithm has the following signature (or declaration):

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first,
          RandomAccessIterator last);
```

The name of the template parameter has been chosen to remind the user that the iterator is expected to fulfill the requirements of a RandomAccessIterator. In addition to containers' iterators, the library proposes some iterators working as adapters, i.e. adapting the interface of another object, such as a stream, to the interface of an iterator, or having a different behavior. This is the case of the `std::back_insert_iterator` (returned by the `std::back_inserter` routine) used in Section 2.1.4: each time this iterator is used to write data, it actually appends an element at the end of the associated container (not to be confused with the iterator returned by the container's `end()` method, which returns a location that is not write- (nor read-) accessible.

Functors The STL also promotes some elements of Functional Programming (FP) style. FP is about making functions first-order objects, that can be manipulated like any other value. In C++, such functions are represented by function objects or *functors*. Functors are implemented as objects having an operator '`()`' (providing a syntax similar to the one of a function call). The following class defines a functor adding a pre-defined quantity to and integer.

```
struct adder
{
  // Initialize ‘a_’.
  adder(int a) : a_(a) {}

  // Add ‘a_’ to ‘x’.
  int operator()(int x) { return x + a_; };

  int a_;
};
```

Functors are instantiated like other objects:

```
// Create a x ↦ x+5 functor.
adder add_five(5);
```

and can be "invoked" like functions.

```
// Add 5 to 3.
int i = add_five(3);
```

Several STL algorithms make use of functors, as they imitate the functional flavor of FP routines found in many languages, in particular the ones that are referred to as *functional programming languages*, such as members of the Lisp family (Common Lisp, Scheme, etc.), descendants of ML (Standard ML, Objective Caml, etc.), or Haskell. For example the `std::for_each` algorithm takes two iterators on a container `first` and `last`, as well as a function `f`, and applies `f` to all elements of the container in the range [`first`,`last`):

```
template <class InputIterator,
          class UnaryFunction>
UnaryFunction
for_each(InputIterator first,
         InputIterator last,
         UnaryFunction f);
```

The argument `f` can be a standard function or a functor. The advantage of functors is that they can record a `state` and make side effects, whereas traditional functions are expected to be pure (i.e. they should only produce a computation while modifying nothing). Using a functor is especially useful to inject information into the computation (like the previous `add_five` example), or to extract information from within the computation. For instance, one can write a functor summing all the elements passed as argument to its operator '`()`'; using this functor with `std::for_each` and iterators on an `std::vector<int>` will compute the sum of this vector's elements. Functors are an extra step towards component orthogonalization: they help to separate the tasks of algorithms into reusable and orthogonal units. In the previous example, `std::for_each`'s work is to traverse a container and apply some function, while the applied function is a variable element (e.g., an accumulator computing a sum).

Concepts The use of templates within a library makes generic components very open, to the point that some parameters would not fit, either syntactically (e.g., because they are lacking a method) or semantically (because one of their method does not behave as expected by the component) or both. A classical example of syntactic incompatibility is the lack of an order on the elements of a container to be sorted. To order values, `std::sort` expects a comparison function (or functor) passed as argument, or relies on the existence of an operator '<' for the type of these values. If none is available, then the sorting operation on this container is not defined, and the compiler must exit with an error. The signature of the algorithm, however, does not enforce these constraints. Such a feature is missing in C++ and as a consequence many type-constraints related error messages are long and complex (see above, as well as Section 2.4.2, p. 58).

Even though the language cannot materialize constraints syntactically, the designers of the STL have designed abstract entities gathering syntactic, semantic and complexity constraints called *concepts*. A concept describes a type by listing the minimal interface it has to provide (e.g. the methods that must be provided by an iterator) and characterizing its behavior in defined contexts (e.g. how the iterator responds to method calls). The analysis of data types and algorithms have led the STL's authors to design concepts for containers, iterators and functors, and define algorithms with respect to these concepts. As said earlier, these entities does not exist as language constructs; however, they are omnipresent in the design of the STL and they are an essential part of its documentation (for instance in SGI's guide [122]). Users of the library are expected to carefully read and follow the indications of the involved concepts. To remind them of concepts, the STL documentation (and code) uses concepts to name template parameters. Concepts are one of the biggest contributions of the STL though they have no concrete materialization within the library's code yet. They are covered in depth in Section 2.4 (p. 52).

ORIGINS The first experiments on GP that will later give birth to the STL date back from the 1970s, when Alexander Stepanov was working with Deepak Kapur and David Musser on a programming language implementing the idea of abstract algorithms preserving efficiency [9]. Stepanov observed algorithms and data structures can be made orthogonal, because the former do not depend on the implementation of the latter, but only on "a few fundamental semantic properties of the structure". This work put an emphasis on efficiency and efficiency requirements in abstract definitions of algorithms, to the point that an abstract definition of an algorithm was as efficient as a dedicated one. It led to the design of the Tecton programming language also making use of a pure functional programming style (i.e. free of side effects).

This style of programming later known as Generic Programming would later be applied to graph algorithms, in the form of a toolbox implemented in the Scheme programming language. Later, Stepanov developed a generic library in Ada with David Musser benefiting from the notion of strong typing (compared to

Scheme) to catch errors and design issues. The first experiments of GP with C++ came after, but the language was not practicable at this time (because of the lack of templates). However, C++ was more flexible compared to Ada, especially because of pointers, which are a key feature to achieve performance.

GP really started in C++ when templates were introduced by Bjarne Stroustrup (mentioned earlier in this section). Stepanov together with Meng Lee created the roots of the STL at Hewlett-Packard (HP). They defined algorithms that were as generic as possible while still being very efficient (with no abstraction penalty according to Stepanov). The design of the first STL was also the occasion to build the first theoretical foundations of GP. Stepanov stated that despite C++ was a very lax language with respect to the semantics of user-defined operations, library design should observe some soundness rules, such as operator '==' always being an equality operator being reflexive ($\forall x, x == x$), symmetric ($\forall x \forall y, x == y \Leftrightarrow y == x$) and transitive ($\forall x \forall y \forall z, x == y \wedge y == z \Rightarrow x == z$). In other words, design rules should observe standard mathematical axioms since these properties are essential to build other operations (algorithms) [9].

The work of Stepanov and Lee was promoted by Andrew Koenig and Bjarne Stroustrup of Bell Labs to the ANSI/ISO committee for C++ standardization in 1993. They sent a draft proposal to the C++ committee in 1994. Eventually, a later proposal was accepted to the C++ draft standard, and STL was partially included in the C++ standard library. The STL implementation from HP (written by Stepanov, Lee and Musser) was made freely available on the World Wide Web (WWW) in 1994 and used as the basis of many other STL implementations (including SGI's version [122]). The STL is also described in an HP technical report by Stepanov and Lee [127]; books on the library have been published later [115].

LEGACY The STL is a cornerstone work and had many implications on the future of C++, GP and even other programming languages, including the one that inspired it at first (Ada). The library has a great legacy among modern programming languages. The STL style (concepts, iterator-based algorithms, use of functors, etc.) has inspired many generic libraries and generic programming in general.

First and foremost, the library provided useful generic containers replacing data structures that used to be written and rewritten in the old days, because the language lacked reusable, safe and efficient containers. It is hard to imagine C++ programming nowadays without `std::vector`, `std::map`, etc. Likewise, STL algorithms, iterators, functors and auxiliary tools have become a fundamental part of the language.

The design of the STL is also of prime importance. While many libraries focuses on generic data structures (only), the STL put the emphasis on generic algorithms, as advocated by Stepanov [120]. Other generic libraries later followed this strategy (see Section 2.3, p. 49). The form of the algorithms themselves is typical of the STL: most of them are written as generic routines taking iterators on their inputs (and outputs) to minimize the coupling between data structures and algorithms. Iterators play a fundamental role in this orthogonal design.

The STL was also successful because it tried to follow C idioms, just as C++ did with the syntax of the C language. The library extends the semantics of C constructs, by providing iterators that mimics pointers, loop-based iterations using the same operations on pointers and iterators ('`*`' to access pointed data, '`++`' and '`--`' to resp. advance to the next element and move back to the previous element, etc.). Indeed the STL follows the design rules of templates and C++ in general by providing language constructs compatible with C constructs: STL containers can store not only class instances but also Plain Old Data (POD) values (such as `int` values); algorithms expecting iterators derived from containers also work with pointers on C arrays. In this respect, GP and the STL are fully compatible with the C foundations of C++, whereas other approaches (namely OOP) imposes a new structural framework (classes, inheritance, methods) in which existing C types do not fit, and must be adapted (for instance, `int` or `float` objects do not support method calls nor inheritance). A language such as Java [67] illustrates the consequences of a design choice favoring a programming paradigm (OOP) at the expense of the compatibility with base language constructs, although this design was later improved. For instance built-in (atomic) types (such as `int`) have an object counterparts (`java.lang.Integer`) so as to fit in the OOP framework; automatic conversions known as *boxing* and *unboxing* were later introduced in the language to simplify this duality. Likewise, there was a discrepancy between statically type-checked built-in arrays (e.g. `java.lang.String[]`) and Object-Oriented (OO) containers (e.g. `java.util.ArrayList`), "losing" the type of their elements. The subsequent addition of *generics* to the language fixed this issue by preserving the type of contained elements (e.g. `java.util.ArrayList<String>`) and more generally enabling a Generic Programming style.

Perhaps more importantly, the STL implemented an essential idea in language design: designing abstract entities (together with their properties) which may not map directly to language constructs, such as iterator concepts. On this topic, Stepanov said that "[an iterator] is something which doesn't have a linguistic incarnation in C++" [9]. Concepts (see Section 2.4, p. 52) were helpful in designing the library, in particular because they provide "intellectual" objects supporting the logic of the library. This is even more relevant in the context of a template library, which cannot be truly compiled (apart from its uses), and for which tools (such as compilers) offer a limited help as design checking tools. Tools have improved since, and the language will probably evolve to better support GP and STL idioms, but this is the definition of concepts and axioms that have led down the foundations of the library.

DEVELOPMENT OF GENERIC PROGRAMMING IN C++ Following the design, implementation and popularization of the STL, GP emerged as a field of scientific research and industrial advances in computer science. GP developed a lot in C++, where it was experimented and defined. Musser and Stepanov proposed to design generic libraries centered on algorithms following the STL model [109]. Dehnert and Stepanov presented fundamental properties required to built a sound generic programming framework [44]. Many work on GP was conducted during in the 1990s in parallel with the C++ standardization process, and continued in the early 2000s. Jazayeri et al. introduced in 1998 a concise definition of GP [88] summarized by Figure 1. It introduces the idea of *type subsets* and *specializations* (or *variants*) that we pursue in this work (see Section 4.6.3, p. 139).

The development of GP in C++ influenced other programming languages among which Java [67] and C# [78, 83], which were augmented with generic data classes, interfaces and methods named *generics*. GP is still an active field of research and development in C++, in particular within the standardization process of the language, which recently delivered C++ 2011 [87]. Among the new GP-related features are variadic templates, enabling an arbitrary number of parameters in a template, and template aliases ("template `typedefs`") simplifying the use of long template names. C++ 2011 was initially planned to host a new major feature, concepts as actual syntactic constructs. This addition would have allowed user to define and use concepts as concrete entities, so as to simplify the use of templates and even enable some optimizations (see Section 2.4, p. 52). They were dropped from the standard proposal in 2009 for complexity reasons, but they may still be part of a future C++ standard.

2.3 APPLICATIONS OF GENERIC PROGRAMMING

Like other programming paradigms, GP increases the *expressive power* of languages that implement it. GP lets users define programming constructs (data structures and algorithms) sharing

Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. Key ideas include:

- Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.
- Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.
- When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.
- Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient.

Figure 1: Definition of Generic Programming from Jazayeri *et al.* [88].

a common structure in a factored and concise way. The generic approach to designing software is also *open*: generic components are not restricted *per se*: parameters are not necessarily tied to a named interface (at least in C++). Therefore, one can write generic code with future reuse without modification in mind.

In addition, GP is a static programming paradigm: its scope is limited to compile time, and thus it requires no ad hoc run-time support; hence it does not introduce run-time overhead. In fact, a generic code, once specialized (i.e. used in a concrete context, where all parameters are given values) is as fast as the equivalent hand-written dedicated code. Figure 1 defines the *lifting* of an algorithm as the act of making it as general as possible without losing efficiency.

These aspects (abstraction-based programming, reusability, open design and efficiency preservation) makes GP a good strategy to implement fundamental data structures and algorithms (as the STL does) as well as scientific software. Indeed applications in mathematics, logic, physics, graph theory, image processing and other scientific fields often require intensive computations (because they may process large and/or numerous data); and they may manipulate data of many kinds (e.g. matrices, vectors, images based on various value types). They need *efficient* and *polymorphic* (generic) algorithms, for which GP is actually well suited. In particular, the benefits of using Generic Programming are covered by Chapter 4.

Historically the language used for scientific computation was Fortran. Fortran compilers were known for generating fast programs, in particular because the language enabled some optimizations leading to efficient code. C and C++ compilers also generated efficient programs but were not able to catch up with Fortran, as they did not permit the same optimizations. However the addition of templates to C++ and the development of the GP style in the language has helped C++ gain ground over Fortran in the domain of scientific programming [149], to the point that many scientific software is now developed in C++.

Successful projects using GP include of course the Standard Template Library, which has since become a part of the C++ standard [81]. Many libraries from the Boost project [136] extend the C++ standard library with many general-purpose and dedicated quality libraries. Other noteworthy projects include the Blitz++ library of numeric arrays [145], using advanced C++ programming techniques such as expression templates [144, 141] and template metaprogramming (presented in Section 2.6, p. 69) to generate specialized algorithms [143, 140] competing with Fortran's efficiency; POOMA (Parallel Object-Oriented Methods and Applications) [15], providing a framework for writing parallel Partial Differential Equation (PDE) solvers using finite-difference and particle methods; the Matrix Template Library (MTL) [126, 125] and the Generative Matrix Computation Library (GMCL) [40, 41]; and CGAL (Computational Geometry Algorithms Library) [57], dedicated to computational geometry.

It is interesting to note that it is possible to make an algorithm even more generic, beyond the recipe imposed by the lifting process of Figure 1. This policy may increase the level of abstraction at the expense of run time efficiency. This is often the case when higher and/or more abstractions are introduced in algorithms. This situation exhibits a phenomenon known as *abstraction penalty*, which may appear in programming paradigms providing abstraction mechanisms. Abstraction penalty is hard to avoid in some paradigms such as Object-Oriented Programming (OOP), but it can be controlled in GP (see Section 2.5, p. 62): we have already mentioned that GP is based on compile-time mechanisms that do not impact the run-time behavior of programs intrinsically. Very abstract generic code is not immune to abstraction penalty though (see Section 4.6.3, p. 139). It is however possible to design additional, slightly less generic alternative algorithms to regain the performance of dedicated code. Moreover, in a careful designed software organization based on orthogonal components, generic algorithms have the interesting property of being usable with general-purpose data structures as well as dedicated data structures, as they do not alter their definition. Compilers, however, may take advantage of the features (compile-time information) to generate optimized code, more efficient than code produced when using a general-purpose data structure (see Section 4.6.3).

2.4 CONCEPTS

Generic Programming was shaped in part by the design and the development of the Standard Template Library. The generic algorithms of the library, expressed as function templates, have requirements on their parameters (see Section 2.2.3, p. 38). The authors of the STL discovered that these requirements were shared among the algorithms. More precisely, the set of requirements on the parameters of a given algorithm could be the same as the set of requirements of another algorithm, or even be a subset of the requirements of yet another algorithm. For instance, the STL algorithm performing a sequential search in a container (`std::find`) and the STL algorithm applying a function to all elements of a container (`std::for_each`) both use an iterator to traverse this container[6]; both algorithms place the same requirements on this iterator: it must be able to access the pointed

6 More precisely, `std::find` and `std::for_each` use a range of iterators [`first`, `last`), since STL algorithms are designed to work on ranges of elements within a container.

element (dereferencing operation) and to advance to the next element (incrementation operation). The translation of these constraints into C++ is the need for a dereferencing `operator*` and for a incrementation `operator++`. For each STL algorithms, such requirements have been identified and regrouped as sets called *concepts*.

Dehnert and Stepanov thus define a concept as "the set of axioms satisfied by a data type and a set of operations on it" [44]. When a type satisfies the axioms of a concept, it is said to *model* the concept or to be a model of this concept. The terms *concept* and *models* have been introduced by Austern [17]. The notion of concept is similar to the notion of interface or abstract class in OOP: both represent a type-related abstract entity. Likewise, a concept's model plays the same role as a concrete subclass derived from an interface or an abstract class. For more information on this topic, see the comparison of GP and OOP in Section 2.5 (p. 62).

In the previous example, the iterators passed to `std::find` and `std::for_each` are expected to be models of the concept InputIterator, which is defined by the two operations identified earlier: dereferencing and incrementation. The STL and other libraries define concepts in a formal way, by listing precisely their syntactic, semantic and complexity properties (see Section 2.4.1, p. 54). The code and the documentation often name a parameter of a generic algorithm after the concept it shall model to remind users of this constraint. In the documentation of the STL, the `std::find` algorithm, which seeks an element '`value`' within the range [`first,last`) of a container, has therefore the following signature:

```
template<class InputIterator,
         class EqualityComparable>
InputIterator
find(InputIterator first,
     InputIterator last,
     const EqualityComparable& value);
```

This example also shows that concepts are not limited to iterators, or even to objects. They can describe requirements that may be fulfilled by any type. For example, the third argument of `std::find` (the searched `value`) must be a model of EqualityComparable. This concept requires `value` to support comparison through operators '==' and '`!=`'. Many types qualify as EqualityComparable, among which all C++ atomic (built-in) types such as `int`, `float`, etc. as the language has built-in definitions of operators '==' and '`!=`' for these types.

If we consider another STL algorithm such the algorithm reversing a range of elements in a container (`std::reverse`), we can observe that the STL places more requirements over its inputs: the iterators defining the range must be able to move forward *and* backward[7]. Incrementation (forward move) of an iterator is still performed with operator '++', while decrementation (backward move) is done with operator '--'. Requiring `std::reverse`'s arguments to be InputIterator s is not enough, as the InputIterator concept does not guarantee the existence of an operator '--'. `std::reverse` thus demands that its iterators model another, richer concept, BidirectionalIterator, providing all the required operations (dereferencing, incrementation, decrementation). BidirectionalIterator includes all the requirements of InputIterator, hence a a model of BidirectionalIterator is also a model of InputIterator. BidirectionalIterator is said to be a *refinement* of InputIterator, or to *refine* InputIterator.

2.4.1 *Concept Definition*

The documentation of the STL uses a uniform style to present concepts. A concept definition is composed of its fundamental notions and properties, including the relations with other concepts, the relations between a concept's model and other types, and syntactic and semantic requirements. This section presents these elements. Figures 2 and 3 show an example of concept definition from the STL, BidirectionalIterator.

Description

The first part of a concept definition describes its category (e.g. iterator, container, functor in the case of the STL) and its main traits (features, behavior, additional services provided with respect to parent concepts, etc.).

Refined Concepts

A concept may be a refinement of one or more concept(s). This part lists concepts refined by the concept being defined. For instance, BidirectionalIterator is a refinement of ForwardIterator, being itself a refinement of InputIterator. The refinement relationship creates hierarchies of concepts, the same way the inheritance relationship creates hierarchies of classes in OOP (see also Section 2.5.2, p. 65 for a parallel between the role of concepts in GP and the role of classes an interfaces in OOP).

Unless the defined concept alters them, the elements of its definition that are "inherited" from a parent concept are not repeated. For instance, BidirectionalIterator documents the decre-

7 By placing such a constraint on its iterators, `std::reverse` can be implemented by performing a number of comparisons equal to half the number of elements in the range).

BidirectionalIterator (1/2)
Category: iterators Component type: concept

DESCRIPTION A Bidirectional Iterator is an iterator that can be both incremented and decremented. The requirement that a Bidirectional Iterator can be decremented is the only thing that distinguishes Bidirectional Iterators from Forward Iterators.

REFINEMENT OF ForwardIterator

ASSOCIATED TYPES The same as for ForwardIterator.

NOTATION

`X`	A type that is a model of BidirectionalIterator
`T`	The value type of X
`i, j`	Object of type X
`t`	Object of type T

DEFINITIONS (none)

VALID EXPRESSIONS In addition to the expressions defined in ForwardIterator, the following expressions must be valid.

Name	Expression	Type requirements	Return type
Predecrement	`--i`		`X&`
Postdecrement	`i--`		`X`

EXPRESSION SEMANTICS Semantics of an expression is defined only where it is not defined in Forward Iterator.

Name	Predecrement	Postdecrement
Expression	`--i`	`i--`
Precondition	`i` is dereferenceable or past-the-end. There exists a dereferenceable iterator `j` such that `i == ++j`.	
Semantics	`i` is modified to point to the previous element.	Equivalent to `{ X tmp = i; --i; return tmp; }`
Postcondition	`i` is dereferenceable. `&i = &--i`. If `i == j`, then `--i == --j`. If `j` is dereferenceable and `i == ++j`, then `--i == j`.	

Figure 2: BidirectionalIterator concept definition [122] (part 1/2).

BidirectionalIterator (2/2)
Category: iterators Component type: concept

COMPLEXITY GUARANTEES The complexity of operations on bidirectional iterators is guaranteed to be amortized constant time.

INVARIANTS

Symmetry of increment and decrement: If `i` is dereferenceable, then `++i; --i;` is a null operation. Similarly, `--i; ++i;` is a null operation.

MODELS

- `T*`
- `list<T>::iterator`

Figure 3: BidirectionalIterator concept definition [122] (part 2/2).

mement operation ('`--`'), but not the increment operation ('`++`'), already defined by ForwardIterator.

Associated Types

A concept may define *associated types*. Models of such a concept may indeed require cooperation from other types to have a meaningful definition. For instance, a Container is linked to the type of the values it contains. Hence the definition of the Container concept defines an associated "value type". This type must be defined by all models as an embedded `typedef` (type alias) called `value_type`. Therefore, for each type C modeling the Container concept, the expression `C::value_type` returns the type of the container objects. Likewise Container defines (among others) an associated type named `iterator` corresponding to the iterator type returned by its methods `begin()` and `end()`. All models of a concept must provide the associated types listed in the concept's definition.

Notation

Later sections of the concept definition make use of expressions that must be satisfied by models of the concept. To shorten these expressions, repeated symbols (such as a type `X` modeling the concept, an object `a` of type `X`, etc.) are defined here.

Definitions

Concepts may introduce associated notions. For instance, a Container defines what a *size* is. This section presents these definitions.

Valid Expressions

This section list the syntactical (and semantic) requirements of a concept, i.e. expressions involving one or several model(s) of the concept and/or objects of this (these) model(s), which must be valid from a compilation point of view. The interface of a concept (the minimal set of its methods) is in particular a part of these requirements. Models failing to satisfy the requirements of the valid expressions section may trigger compilation errors.

Expression Semantics

In addition to the previous list of valid expressions, a concept assigns meaning to some expressions and defines requirements on their semantics. For instance, let us consider an object `a` of a model of Container. The Containers commands that the expression `a.empty()` (checking whether `a` is empty or not) be semantically equivalent to the expression `a.size() == 0` (checking that `a` has a size of 0), and possibly be faster.

Most expression semantics cannot be checked at compile-time, as they often characterize run-time behaviors, hence compiler are unable to enforce them. Models failing to satisfy the requirements of the expression semantics section may cause errors or behave unexpectedly at run time.

Complexity Guarantees

As efficiency is also a concern of GP, a concept may impose constraints on the complexity of the services provided by its models. For instance, Container requires the method `size()` to have a linear complexity in the container's size. Complexity constraints give users some guarantees about the global run-time efficiency of an algorithm using a model of a given iterator or container concept.

Invariants

The invariants section completes the expression semantics with additional requirements on the models and their instances with respect to semantics. But instead of being tied to a particular expression, these requirements express general rules of the concept.

Axioms on fundamental properties of data structures are in particular defined as invariants. For example the EqualityComparable concept defines that for a model `X` of EqualityComparable, and for three objects `a`, `b`, `c` of type `X`, the following invariant must hold:

Reflexivity	`x == x`
Symmetry	`x == y` implies `y == x`
Transitivity	`x == y` and `y == z` implies `x == z`

Thus an EqualityComparable type is guaranteed to implement an equality relation. EqualityComparable also lists the following invariant:

Identity	`&x == &y` implies `x == y`

meaning that two objects of the same EqualityComparable model sharing the same memory address[8] must be equal.

Models

A concept may cite some of its models, so as to give examples to readers. The example illustrated by Figure 3 presents two models of the BidirectionalIterator concept: `T*`, the type of a pointer to elements of type `T`; and `list<T>::iterator`, the type of an iterator on an STL list of `T`s.

2.4.2 *Concept Checking*

A primary use of concepts is to express constraints on the parameters of a class or function template, as part of its *interface*. Then, the *implementation* of the template can rely on the concept being satisfied to use its specification. Concepts act as "pivot" in GP between the interface and the implementation of generic types and functions. If a template is instantiated with parameters satisfying the constraints of its concepts, then it can guarantee that no concept-related errors will happen in the compilation of its implementation. For instance, if the interface of a function expects one of its arguments to be a BidirectionalIterator (see Figures 2 and 3, p. 55 and 56), then its implementation can safely use the pre- or postdecrement operator '`--`' of this argument: as long as this function is invoked with models of BidirectionalIterator, this statement will trigger no error.

Concept checking is the action of checking whether a type satisfies the requirements of one (or more) concept(s). As concepts are used to constrain template parameters, such checks should be enforced when the template is specialized for a given (set of) type(s) used as effective parameter(s), and *before* any element that is part of the requirements is used. However, the current C++ standard does not provide any means to perform such ahead-of-use concept checks as concepts have no actual existence in the

8 In C++, the prefix unary operator '`&`' returns the address of a variable.

language. An immediate drawback of this lack of explicit constraint mechanism previously mentioned in Section 2.2.3 (p. 38) is that concept-related errors are reported late in the compilation process. Errors are triggered at the first erroneous use of the template, most often within the *implementation* of the template, instead of showing which part of the concept the type failed to implement. The following program illustrates this issue.

```
1 #include <set>
2
3 struct S {};
4
5 int main()
6 {
7   std::set<S> s;
8   s.insert(S());
9 }
```

This example makes use of the sorted unique associative STL container `std::set<Key, Compare>`[9]. This container stores data of type `Key` according to a strict weak ordering of type `Compare` and holds at most one copy of each element (using the ordering to implement identity between two elements). When it is not provided, the parameter `Compare` is set to a default value, `std::less<Key>`, a functor implementing a strict weak ordering based on operator '<' by default. In the case of `std::set<S>` however, the type `S` does not provide any comparison operator, thus failing to make `std::less<S>` a model of the StrictWeakOrdering concept. This concept-check failure is not detected by the compiler at line 7, where `std::set` is specialized with parameter `Key` set to `S` and parameter `Compare` implicitly set to `std::less<S>`. Instead, the issue is detected as a side effect of the instantiation of the `insert()` method in line 8, giving the following compiler error message[10]:

```
/usr/include/c++/4.2.1/bits/stl_function.h:
In member function
'bool std::less<_Tp>::operator()(const _Tp&,
                                 const _Tp&) const
 [with _Tp = S]':
/usr/include/c++/4.2.1/bits/stl_tree.h:982:
instantiated from
'std::pair<typename std::_Rb_tree<_Key, _Val
                                   _KeyOfValue,
                                  _Compare,
                                  _Alloc>
                      ::iterator,
```

9 The `std::set` class template actually have a third parameter, `Alloc`, which governs the allocation policy of the container. We do not go into further details about STL allocators as this topic is not related to concept checking issues.

10 This output has been obtained with the GNU g++ compiler version 4.2.1. The layout of this message has been modified to make it more legible.

```
13            bool>
14  std::_Rb_tree<_Key, _Val,_KeyOfValue, _Compare,
15                _Alloc>::_M_insert_unique(const _Val&)
16  [with _Key = S, _Val = S,
17        _KeyOfValue = std::_Identity<S>,
18        _Compare = std::less<S>,
19        _Alloc = std::allocator<S>]'
20 /usr/include/c++/4.2.1/bits/stl_set.h:307:
21 instantiated from
22 'std::pair<typename std::_Rb_tree<_Key, _Key,
23                                   std::_Identity<_Key>,
24                                   _Compare,
25                                   typename
26                                   _Alloc::rebind<_Key>
27                                     ::other>
28                        ::const_iterator,
29            bool>
30  std::set<_Key, _Compare, _Alloc>::insert(const _Key&)
31  [with _Key = S, _Compare = std::less<S>,
32        _Alloc = std::allocator<S>]'
33 invalid-set.cc:8:   instantiated from here
34 /usr/include/c++/4.2.1/bits/stl_function.h:227:
35 error: no match for 'operator<' in '__x < __y'
```

The actual error that produced this long message is shown at the bottom of the trace (lines 34–35): the compiler was unable to find an operator '<' to compare two elements of the set as reported in the beginning of the message (lines 1–5). The next lines shows where this error originates from: it is located in the depth of the STL implementation, in a method called `_M_insert_unique` of a class template `std::_Rb_tree` (lines 6–19). This template, which triggered the error, was instantiated by another template. The compilers shows this list of successive template instantiations in reverse order until it reaches the initial instantiation at the bottom of this "instantiation stack" (line 33) This origin is located in the program shown earlier, the call to the `insert` method.

Had the language and its compiler supported concepts as a language construct, the message would have reported an earlier issue (the invalid instantiation of `std::set<S>`) with a more precise error location (the instantiation of `std::less<S>`, triggered by the instantiation of `std::set<S>`) and a shorter message, similar to the following one[11]:

```
/usr/include/c++/4.3.0/bits/concepts.h:
In function 'int main()':
/usr/include/c++/4.3.0/bits/concepts.h:495:
error: template arguments for
'struct std::set<S, std::less<S> >' do not meet the
requirements of the primary template
/usr/include/c++/4.3.0/bits/concepts.h:495:
note:   no concept map for requirement
```

11 This message is similar to the kind of messages produced by the Concept-GCC compiler [76], which provides some support for concepts as language constructs.

```
9  'std::StrictWeakOrdering<std::less<S>, S>'
10 /usr/include/c++/4.3.0/bits/concepts.h:495:
11 error: aggregate 'std::set<S, std::less<S> > s' has
12 incomplete type and cannot be defined
```

The error is much more explicit than in the previous compiler message: it states (lines 4–6) that one of the template parameters[12] of `std::set< S, std::less<S> >` (namely `std::less<S>`) does not meet the requirements of the `std::StrictWeakOrdering< std::less<S>, S >`[13] concept definition (lines 8–9). This message is also much shorter than the previous one.

Extensions to Support Concept Checking

There have been several proposals to add support for concept checks to C++ as language extensions. Gregor et al. [69] mention the where-clause mechanism found in CLU [99], Theta [43] and Ada [82] (see also Sections 2.2.1 and 2.2.2, p. 35 and 36). Programming languages of the ML family use a similar strategy to constrain parameterized modules. This approach enforces a *structural conformance* of the parameters, as the constraint is expressed by its contents and does not bear a name.

Another technique is to use the inheritance relation to constrain parameters of generic entities. In this approach a parameter `T` is constrained by a class (or an interface, depending on the language) `C` and must inherit from (resp. realize) `C`. The methods expressed in `C` form the set of requirements. This approach expresses a *named conformance*: constrained parameters not only must match the signature of the constraint (`C`), but must also derive from it: if `T` were to derive from another class `D` structurally equivalent to `D`, instead of inheriting from `C`, it would not satisfy the constraint imposed by `C`. The constraint-through-derivation strategy is being used in Eiffel, Java and C#.

A third strategy is to introduce a new kind of entity to capture the a set of constraints on a parameter, often called *signatures* (not to be confused with ML's signatures). Such a set of constraints is given a name and may involve more than one type, as it is not tied to a particular data type (unlike derivation-based constraints that can only be fulfilled by a subtype of the constraint class, i.e. a single type). This idea of a new constraint entity is at the heart of the abandoned "concepts" proposal for the C++ 2011 standard.

12 Called "template arguments" in this message.

13 This example also shows an example of concept definition related to more than one type: `std::StrictWeakOrdering<F, T>` expresses the notion that `F` is a strict weak order relation over values of type `T`. So in the example of the text, it is the *pair* (`std::less<S>`, `S`) that does not model the StrictWeakOrdering concept.

Checking Concepts without Language Constructs

Despite C++'s lack of support for concepts, workarounds have been developed to enforce concept checking. These techniques rely on template metaprogramming (see 2.6) and can be used to check for structural and/or named conformance of template parameters. For example, Siek and Lumsdaine propose a concept-checking framework [124] where a concept's constraints are expressed as a class template exercising all valid expressions listed in the concept [124]. This approach checks the structural conformance of template parameters. The authors also propose the use of *archetype classes* to ensure that the concept(s) used in an algorithm interface covers all the requirements of the algorithm's implementation with respect to its parameters. These ideas have been implemented in the Boost Boost Concept Check Library [123].

McNamara and Smaragdakis proposes another framework called *static interfaces* [103], where requirements of a concept (e.g. `LessThanComparable`) are encapsulated in a class template (e.g. `LessThanComparable<T>`) defining a method pointer (as a member) for each method required by the concept (e.g. `operator<`). Models of the concept shall inherit from this class template, and pass their own name as effective parameter to the concept class (e.g. `struct Foo : LessThanComparable<Foo>`). This technique relies on an application of the Curiously Recurring Template Pattern (CRTP) described in Section 3.1 (p. 76). Static interfaces are primarily used to enforce named conformance, as they require a model to inherit from the concept class to satisfy the concept. The authors however propose a mechanism to support structural conformance, in particular to support non-class types (e.g. `int`) that does not support inheritance, and existing data types that cannot be changed to support static interfaces.

The SCOOP paradigm used by the Milena library and presented later in Chapter 3, uses a technique similar to static interfaces, based on an extended CRTP (see Section 3.2, p. 79). This choice enables named conformance to concepts and concept-based overloading.

2.5 object-oriented programming vs generic programming

Object-Oriented Programming (OOP) and Generic Programming (GP) propose two different approaches to design, implement, extend and maintain software, the former with an emphasis on dynamic behavior and the latter with a focus on static resolution. However, they share some similarities in their organization.

OOP is a programming paradigm that is provided in different flavors, depending on the language. Some object-oriented features may or may not be present in a given Object-Oriented Language (OOL). In particular, some OOLs are class-less (or prototype-based). In this section, we only consider the OOP paradigm as implemented in C++, which is also present in most compiled and statically checked programming language, such as Java, C#, D, and Eiffel. Before comparing OOP and GP, we recall some important definitions about OOP.

2.5.1 *Elements of Object-Oriented Programming*

In (class-based) OOP, data types are implemented as classes, entities regrouping data and services acting on them, named respectively *attributes* and *methods*. In C++, attributes are called *members*, while methods are called *member functions*. Like many programming entities (functions, modules, etc.), a class can be decomposed in two parts: firstly, its *interface* or *signature* (called *declaration* in C++), which is made of its name, the names and types of its attributes, and the names of methods together with the name of their arguments and types thereof; and secondly, its *implementation* (called *definition* in C++), which contains the code of its methods.

A value having the type of a given class is called an *instance* or an *object* of that class. The action of creating such an object is called *(class) instantiation*. Class instantiation should not be confused with template instantiation, which is the process of creating an actual type or function from a template, presented in Section 2.1.1 (p. 31).

A class `B` may be *derived* from another class `A`, therefore acquiring `A`'s attributes and methods, in addition to its own attributes. `B` is said to *inherit* from `A`. `A` is called a *base class* of `B`, while `B` is a *subclass* of `A`. This kind of relation between two classes is called *implementation inheritance*. `B` may provide its own implementation for a methods already present in `A`. This action is called *(method) overriding*. The overridden method is said to be *polymorphic*, as the code executed at run-time depends on the exact type of the object "owning" the method. Polymorphic methods are called *virtual (member) functions* in C++.

A class may also represent an abstraction (e.g. an `animal`) , i.e. an entity that is not representable as a concrete data structure with a given set of attributes and methods (such as a `cat` or a `dolphin`). Such a class is named an *abstract class* or an *interface* (not to be confused with the previous acceptation of the word). The differences between the terms "abstract class" and "interface" depend on the language that provide them. To simplify the explanations of this section and to avoid the confusion with the previous meaning of "interface" (a class declaration, as opposed to its implementation or class definition), we consider that these terms are equivalent, and only use the term "abstract class" thereafter and reserve "interface" for "class declaration". A concrete class (e.g. `cat`) may be derived from an abstract class (e.g. `animal`). This relation represent the "IS A relation" ("a `cat` IS AN `animal`"). Later, instances of the concrete type may be used wherever values of the abstract types are expected (for example a list of `animals` may be composed of `cats`, `dolphins` and other instances of concrete subclasses of `animal`).

An abstract class may declare methods, that is, announce them without providing an actual implementation[14]. Concrete classes may be derived from this abstract class, and are expected to provide a definition (implementation) for such methods, using the overriding mechanism mentioned previously. Such methods are also polymorphic, as different implementations may be provided by different classes. A polymorphic method declared in an abstract class is also described as *abstract*. In C++, abstract polymorphic methods are called *pure virtual (member) functions*.

An abstract class cannot be instantiated as a consequence of the nature of this type (one cannot picture an `animal` *per se*). This is even more relevant when the class has abstract methods: its instances would lack the code of these methods. However, a concrete class (having no abstract methods by definition) derived from this abstract class may be instantiated (an `animal` cannot be instantiated, but a `cat` can).

An abstract class `Y` (e.g., a `vertebrate`) may be derived from another abstract class `X` (e.g., an `animal`), therefore representing a *sub-abstraction* of `X`. The interface of `Y` includes the interface of `X` (methods and attributes). Likewise, concrete classes derived from `X` (resp. `Y`) include the interface of `X` (resp. `Y`).

A set of classes linked (directly or indirectly) by an inheritance relation is called a *class hierarchy*. The property for a language to allow at most one (direct) base class for each class is called *single inheritance*. If on the contrary a class may have more than one (direct) base class, this is a case of *multiple inheritance*. In single inheritance languages, a class hierarchy form an oriented tree, whereas it forms a Directed Acyclic Graph (DAG) in multiple inheritance languages. Some programming languages define a greatest and a least element as respectively (direct or indirect) ancestor and descendant of all classes. In Eiffel, these classes are called `ANY` and `NONE` [84]. In such languages, all classes are therefore part of the same hierarchy, and form a bounded lattice.

14 Depending on the language, abstract classes may also *define* methods, as well as attributes.

2.5.2 *Comparison of OOP and GP*

Object-Oriented Programming (OOP) and Generic Programming (GP) are two strategies to organize software design and development. Both of them propose their own conception: interfaces, abstract and concrete classes, inheritance and polymorphic methods on the one hand; concepts, parameterized containers and parameterized routines on the other hand. Both programming paradigms, however, share qualities from the software engineering point of view and exhibit similar ideas in their organizations.

In OOP, data abstractions of a given domain are implemented as abstract classes, while data implementations are provided by concrete classes that inherit (or are derived) from these abstract classes. In GP, data abstractions are represented as concepts (see Section 2.4, p. 52) while data implementations are models of these concepts.

This parallel between OOP and GP is not limited to the separation of interfaces and implementations. There is an actual symmetry between the two paradigms [104, 32]. The remainder of this section shows a comparison of different notions present in both OOP and GP, though implemented in a different manner. Table 1 summarizes this comparison. It is important to note that in the case of GP, some of these notions may not be actually present in the language (e.g., concepts and modeling relationships are not expressed in standard C++); they are nonetheless of prime importance in the organization of the paradigm.

Abstraction mechanism

As said above, abstract classes serve as abstractions in OOP. Abstract methods define the interface of an abstraction, that must be fulfilled by its concrete representatives (concrete subclasses). For instance, an `animal` may have an abstract `eat()` method, the actual code of which depends on the concrete type of animal.

In GP, concepts are used to convey abstractions. The interface of a concept is shaped by the various elements of its definition, as presented in Section 2.4.1 (p. 54). Among these, associated types and valid expressions (which include signatures of methods) express syntactic constraints on the models of a concept. As such, they are the first and the stronger requirements, as they can be checked at compile time. Concept definitions however encompass other requirements, mostly of semantic nature, such as expression semantics, complexity guarantees and invariants. Some of them may be checked at compile time if the constraint does not depend on run-time behavior and if the language is able to express it; otherwise, run-time checking may be used.

Notion	OOP	GP
Abstraction mechanism	Abstract classes	Concepts
Sub-abstractions	(Abstract) subclasses	Refined concepts
Implementations mechanism	(Concrete) subclasses	Parameterized classes (models)
Relation between implementation and abstraction	Inheritance	Modeling
General definition of an algorithm	Polymorphic methods	Parameterized routines
Type of polymorphism	Inclusion polymorphism	Parametric polymorphism
Binding time of a routine	Run time	Compile time
Number of times a routine is compiled	Once	Once per specialization (unique combination of parameters)

Table 1: Comparison of OOP and GP notions.

Sub-abstractions

A sub-abstraction is a refined abstraction, acquiring and extending the interface of one (or several, depending on the language) abstraction(s). A type conforming to a sub-abstraction also conforms to its upper abstractions (e.g., a of `vertebrate` is also an `animal`).

In OOP, a sub-abstraction `B` of an abstract class `A` is implemented as an (abstract) subclass of `A`. The inheritance relation between the two classes is explicit. In GP, a sub-abstraction is implemented as refined concept. According to the language and context, the link between the two concepts may be implicit or explicit. GP in standard C++ is an example of implicit relation between a concept and its refinement, as a direct consequence of the lack of language constructs to express concepts. The "concepts" proposal for C++ offered the choice between implicit (structural) or explicit (named) relation.

Implementation mechanism

Classes serve as implementation facility in both OOP and GP. In OOP, a (concrete) class must satisfy the interface expected by its (abstract) base classes. In GP, implementation classes (models) are parameterized (i.e., class templates in C++) and both classes and their parameters may be constrained by concepts.

Relation between implementation and abstraction

Similarly to sub-abstractions, the nature of the link between an implementation (e.g. a `cat`) and its abstraction(s) (e.g., an `animal`) depends on the paradigm. In OOP, the link is inheritance and is always explicit. In GP the link may be implicit, as in standard C++ where concepts have no existence in the code and models do not mention them; or as in the "concepts" proposal for C++ [70, 133], using structural conformance (thanks a new `auto concept` statement). This proposal also contained a means to force explicit model-concept relationships using named conformance (using a new construction called `concept_map`).

General definition of an algorithm

Both paradigms are able to express general or abstract definitions of algorithms not tied to a particular input or output type. In OOP, such general algorithms are written using polymorphic methods and inclusion polymorphism. In GP, parameterized routines (e.g. C++ function templates) are used.

Type of polymorphism

OOP is an example of *inclusion polymorphism*: if a class `B` derives from a class `A` (whether one or the other or both is abstract), then `B` is included in `A`: each instance of `B` may be used wherever an `A` is expected. If an instance of `B` is passed to an algorithm taking an `A` as argument, and if this algorithm invokes a method of `A` that happens to be overridden in `B`, then `B`'s version of that method is eventually invoked. This algorithm is therefore polymorphic with respect to the type of its argument, as it accepts instances of subclasses of `A` as arguments, and its behavior depends on the methods of the actual (dynamic) type of this instance.

GP provides *parametric polymorphism*, where a generic algorithm is a function having (at least) one generic *type variable* or parameter to qualify the type of an argument, instead of having a fixed type. A parameterized algorithm is polymorphic with respect to the type of its input, as this type can be freely chosen (though it might be constrained, e.g. through a concept in C++). Moreover the behavior or the algorithm depends on the type of its input, e.g. this algorithm may invoke methods of this argument (the implementation of which depends on the type used as effective parameter); also, the argument may be passed (again) as argument to a call to an overloaded function, the resolution of which depends on the type of the argument as well.

Binding time of a routine

Following the previous item, OOP and GP are different with respect to the moment where a routine is selected. In OOP, a routine implemented as a polymorphic method requires the knowledge of the exact (dynamic) type of its target (the object the method belongs to). This information cannot be obtained until the code is actually executed. Therefore polymorphic methods are bound at run time. This mechanism is known as *dynamic dispatch* or *late binding*. In GP, a generic routine cannot be used unless all its parameters are known. As these parameters are types, a compile-time information, the binding of parameterized routines is done at compile time.

GP's parameterized routines are similar to classic functions, bound at compile time. Hence they do not introduce run-time penalties and they do not prevent optimizations from the compiler. Polymorphic methods, however, cannot be bound at compile time. The selection of a method's implementation requires extra computation compared to a classic function call. In addition, as the compiler does not known which method will be called, it cannot perform some optimizations (inlining of function bodies, constant propagation, etc.). In the case of a call to a polymorphic method executing very few instructions (e.g. an accessor) executed a lot of times (e.g. within a loop), this can lead to major run-time overhead: the accumulated time to perform the dynamic dispatches may be significant compared to the time to execute the body of the method. This pattern is very frequent in image processing, and using polymorphic methods may be an issue if performance is sought.

Number of times a routine is compiled

Though their invocation procedure differs from classic functions, OOP's polymorphic methods are compiled just like them and each of them is compiled once. The situation is different with GP's parameterized functions, at least in C++, where the instantiation model creates a new function per unique (combination of) parameters. Such a routine may be compiled multiple times, depending on its number of calls with different types of arguments.

Another important difference is the scope of each technique. In OOP, every combination of an object and a polymorphic method called with this object may be used at run time, as each method is compiled once, independently of its use. In GP however, generic routines available at run time must have been instantiated previously at compile time. This means that if a parameterized function has not been instantiated (either implicitly or explicitly, see Section 2.2.3, p. 38) with a given (set of) actual parameter(s) at compile time, then this routine will not be available at run time.

Every template–parameter combination must be planned ahead of the execution in GP, while in OOP polymorphic method–object combinations used at run time need not to be known at compile-time. In this regard, OOP offers more run-time flexibility than GP: from the run time point of view, polymorphic methods proposes an open world framework, while parameterized routines and data structures live in a closed world.

2.6 BEYOND GENERIC PROGRAMMING: STATIC METAPROGRAMMING

Templates have been introduced the C++ language to provide parameterized containers and routines in the first place. With the development of template libraries, and in particular the Standard Template Library, templates have helped shape the Generic Programming paradigm. C++ templates, however, have been found to exceed the scope of GP. As they have been designed as a powerful generating facility, they allowed new kinds of programming constructs performing various computations at compile-time known as *static metaprogramming*. This section presents this technique and some of its applications.

2.6.1 *C++ Template Metaprogramming*

To introduce the notion of static metaprogramming, let us consider the following simple example using templates.

```
template <unsigned n>
struct fact
{
  static const unsigned val =
    n * fact<n - 1>::val;
};

template<>
struct fact<0>
{
  static const unsigned val = 1;
};
```

This code defines a class template `fact` containing a single unsigned integer constant (`val`). The `static` qualifier in front of `val`'s definition means that this constant is an attribute belonging to the class, and not to a particular object: it is shared by all instances of the class, and can be accessed directly from the class (i.e., it does not require an object of this class to be accessed). The class `fact` is however a template, and cannot be used without a parameter. Therefore, for a given integer n, `fact<`n`>::val` returns the value of `val` in `fact<`n`>`[15]. `val` has the noticeable property of being defined recursively: the definition of `fact<`n`>::val` is defined by a product a factor of which is `fact<`$n-1$`>::val`. This definition is valid because n is known at compile-time (since it is a template parameter), because C++ template support recursion, and because the recursion terminates. The previous code contains indeed an explicit specialization of `fact<n>` (see Section 2.2.3, p. 38) for the case where n equals 0: in this very case, `fact<0>::val` is equal to 1. For a given integer n,

$$\texttt{fact<}n\texttt{>::val} = \begin{cases} 1 & \text{if } n = 0 \\ n \times \texttt{fact<}n-1\texttt{>::val} & \text{otherwise} \end{cases}$$

In other words,

$$\texttt{fact<}n\texttt{>::val} = n!$$

that is, the factorial of n.

In the previous code, the class template `fact` is not used as a generic data structure, but for *computation* purpose. `fact` acts as a function, but it has a very different nature: it is "evaluated" by the compiler, and produces a *static* (compile-time) result, as a consequence of templates being a compile-time mechanism. In the following expression, `fact<4>::val` is replaced by the compiler with the value of $4!(=24)$.

```
unsigned x = fact<4>::val; // == 4! = 24.
```

This kind or programming style in C++ is known as *static metaprogramming* or *template metaprogramming*. Metaprogramming is the action of writing metaprograms, which are programs that manipulates other programs (or even themselves). In a sense, a compiler can be seen as a metaprogram, since it generates an output in a (usually low-level) target language, from a program expressed in a (usually higher-level) source language. In the previous example, `fact` is a metaprogram computing the factorial of a static integer value. This computation is said to be static, as it occurs during the compilation process.

Using the metaprogramming features of C++ templates, it is possible to write compilation-time "functions" similar to `fact`. Such functions are implemented as class templates. "Arguments" must be constant known at compile-time, "passed" to these functions as template parameters. The "return value" must be implemented as a static member of the class. These functions are said to be *pure*: they do not have side effects, such as modifying a value or performing input/output operations[16]. As C++ templates support recursion, compile-time metaprograms can make use of recursive functions (as `fact` does). The evaluation of such functions is achieved through recursive template instantiations.

15 Operation '`c::m`', where `c` is a class, returns the member `m` of `c`.
16 Except for compiler errors triggered by invalid template code.

Template metaprogramming was discovered almost by accident by Erwin Unruh [138], who had written a program printing out a list of prime numbers at compile-time as an error message. The term "template metaprogramming" was coined by Todd Veldhuizen [142].

C++ template metaprogramming offers a new language within C++. Despite its curious syntax, this language is actually powerful. It has been demonstrated to be Turing-complete [147] and could theoretically be used to perform the same computations as a Turing-complete general purpose programming language, including C++ itself. Template metaprogramming is useful to generate an optimized algorithm code for a given input, thereby trading longer compilation times for better run-time efficiency. Classical examples include compile-time computations of trigonometric functions and Fast Fourier Transform (FFT) for static values [143, 140].

Like GP, static metaprogramming is not tied to C++. Other Programming languages supporting metaprogramming include Curl, D, Eiffel, Haskell, ML and XL.

2.6.2 *Functions on Types and Traits Classes*

Another powerful application of template metaprogramming is computations on types. As template parameters can convey both non-type and type values, template metaprogramming functions "accepting" and "returning" types may be written. "Argument passing" is similar to the case of metaprogramming function on non-type values; "returning" a value works differently though: instead of using a static constant member, a `typedef` within the class template is used.

The following code shows a example of a "function" $T \mapsto T*$ mapping a type T to the corresponding pointer type $T*$.

```
template <typename T>
struct ptr_type
{
  typedef T* type;
};
```

This "function" is used in the same way as in the precedent example of `fact`, using the '`::`' operator.

```
typedef ptr_type<int>::type  t;  // t == int*.
typedef ptr_type<char>::type v;  // v == char*.
typedef ptr_type<v>::type    w;  // w == char**.
```

Functions on types are useful in generic code to attach external properties and behaviors to a type. Consider for instance a generic function `sqr` retuning the square of a value of type `T`. Using `T` as return type of `sqr` is not a wide decision, as it may cause overflows, especially when `T` is a small data type such as `signed char`[17]. For instance, `sqr(100)` (= `10.000`) does not fit on a `signed char`, and would therefore trigger an overflow. Choosing a fixed "larger" type whatever the (input) parameter `T` is no better solution, as it is not robust to changes. If e.g. `double` (double-precision floating point numbers) were chosen as return type, `sqr` would become obsolete on a system supporting quad-precision floating point numbers.

The solution to this problem is to make the return type of `sqr` depend on `T`, by using a function on types. The following code shows a possible solution.

```
// Default case: sqr<T> : T ↦ T
template <typename T>
struct sqr_return_value
{
  typedef T type;
};

// Case of \code{signed char}:
// sqr<signed char> : signed char ↦ signed short
template <>
struct sqr_return_value<signed char>
{
  typedef signed short type;
};

// Case of \code{unsigned char}:
// sqr<unsigned char> : unsigned char ↦ unsigned short
template <>
struct sqr_return_value<unsigned char>
{
  typedef unsigned short type;
};
```

By default the "return type" is the same as the "input type", except for `signed char` and `unsigned char`, which are respectively mapped by this "function on types" to `signed short` and `unsigned short`, which are larger types than the corresponding input ones. We can then use `sqr_return_value` to define `sqr`'s return type as `sqr_return_value<T>::type`. An implementation of `sqr` could therefore be[18]:

```
template <typename T>
typename sqr_return_value<T>::type
```

17 The range of values fitting on a `signed char` is usually $[-128, 127]$.

18 Note that the C++ ISO standard [81] requires the presence of the `typename` keywords before `sqr`'s return type, as it comes from a `typedef` (`type`) within a class template (`sqr_return_value<T>`), and because it is used in a "template context" (the definition of `sqr`).

```
sqr(T x)
{
  return x * x;
}
```

sqr<T> now depends on sqr_return_value<T> regarding its return value type.

```
signed char x = 42;
// Returns 42 * 42 on a ‘signed short’
// (specialized return type).
sqr(x);

int y = 1000;
// Returns 1000 * 1000 on an ‘int’ (the
// default return type is the input type).
sqr(y);
```

The list of specializations of sqr_return_value can be extended as new input types for sqr are considered.

sqr_return_value<T> acts as a policy regarding the behavior of a generic algorithm. This idea can be extended by storing more information on the policy class, for instance typedefs, constants, and even functions. The standard class std::numeric_limits [81] is an example of such a class. For a built-in numeric type T, std::numeric_limits<T> provides routines such as min() and max() (returning the smallest and largest value of T), Boolean values such as is_signed (telling whether a type is signed or not), etc. Such a class is called a *traits class*. Traits class were invented to add support for internationalization in the Standard C++ Library [112, 111], to simplify input/output stream classes.

Traits classes are also used in the STL, in particular for iterators. The class template std::iterator_traits [122, 81] defines useful types associated to a given iterator type passed as parameter (in particular the value type associated to the iterator). Traits can also contain categories or *tags*, i.e. types that are used as labels. Such types may be used to qualify the nature of a type, and help algorithms adjust their behavior with respect to their inputs. This is the case of the std::iterator_traits<I>::iterator_category type, declaring the most specific concept (see Section 2.4, p. 52) modeled by the iterator type I, using a type such as std::input_iterator_tag, std::forward_iterator_tag, std::bidirectional_iterator_tag, etc. (see also Section 2.2.3, p. 38). An algorithm (such as std::reverse) may takes advantage of the fact that I is a RandomAccessibleIterator by checking that std::iterator_traits<I>::iterator_category is std::bidirectional_iterator_tag, to use an implementation faster than the default one used for a BidirectionalIterator. This discussion is carried further in Section 4.6.4 (p. 143).

3 A STATIC C++ OBJECT-ORIENTED PROGRAMMING (SCOOP) PARADIGM

This chapter presents SCOOP, a new programming paradigm mixing the benefits of GP and OOP, designed to provide a framework for scientific software development. This paradigm enables the definition of actual concept classes representing abstractions of the target domain. As it does not rely on dynamic features, SCOOP does not introduce run-time penalties, and is therefore suited to the production of efficient applications. In addition to concepts, properties can be added to describe static characteristics of data types. These properties can be used in compile-time assertions, or to write a static dispatch mechanism for algorithm selection. Finally, SCOOP also proposes the idea of morphers which enable lightweight object transformations. Morphers can be used to change the behavior of an algorithm non-intrusively, by transforming its input beforehand.

Using Generic Programming (GP) to design scientific software is a good strategy (see Section 2.3, p. 49). GP provides abstraction mechanisms required to define reusable data structures and algorithms compatible with the many data types involved in numerical computations (integer, floating-point, complex types; vectors and arrays; matrices and tensors, etc.). Besides, GP implies no run-time penalty *per se*, which makes it a good choice when efficiency is a primary requirement, which is often the case in scientific applications.

Object-Oriented Programming (OOP), on the other hand, offers many useful traits regarding scientific software. Classes can be used to implement abstract data types, hiding their implementation details and used through the type's interface (methods and attributes). Data hiding, provided by keywords `private` and `protected` in C++, also increases the separation between a type's interface and its implementation. Inheritance is useful in two ways. First it can be used as a factoring mechanism (implementation inheritance), where data structures sharing implementation traits inherit from a common base class factoring common code. Then it can be used to create named type constraints (or signatures) known as interfaces in programming languages such as Java or C#[1]. Interfaces contains only abstract methods (having no implementation) expressed as a set of syntactic requirements. A class inheriting from an interface shall implement all the methods of the interface. Interface inheritance thus offers

1 In C++ interfaces are implemented as abstract classes.

an object-oriented abstraction mechanism: in OOP, algorithms express their constraints as interfaces, and types observe these requirements by inheriting from (or *realizing*) these interfaces. However this abstraction mechanism has a cost, as it is based on virtual methods (see Section 2.5, p. 62).

GP and OOP both have desirable properties as far as scientific software is concerned. In this section we study how these programming paradigms can be mixed in an efficient way, so as to fulfill scientific and software engineering requirements. Such a strategy is known as a *multiparadigm* approach: combining two paradigm or more to solve programming problems. The first step towards this unified paradigm relies on a programming pattern mixing templates and inheritance (see Section 3.1). This pattern has been explored and generalized by Burrus et al. [32] (see Section 3.2) to design a Static C++ Object-Oriented Programming (SCOOP) paradigm. This paradigm mixing OOP and GP has evolved over the last ten years. We present in this work its third evolution, implemented in the Olena 1.0 and 2.0 platforms.

3.1 CURIOUSLY RECURRING TEMPLATE PATTERN

The Curiously Recurring Template Pattern (CRTP) is a C++ construction mixing two C++ features representative of OOP and GP, namely inheritance and templates. Let us consider the following two definitions:

```
template <class E> struct A {};
struct B : A<B> {};
```

The class `B` inherits from a template class `A`, the effective parameter of which is set to `B` itself. This (valid) C++ construct is known as the Curiously Recurring Template Pattern (CRTP)[2]. The name of the pattern has been coined by Coplien [35], who noticed it at least in three independent works by Lorraine Juhl (for an Finite State Machine (FSM) implementation); Barton and Nackman [18]; and Tim Budd [30]. This last example demonstrates a use of the CRTP in Leda, a multiparadigm programming language mixing imperative, object-oriented, functional, and logic-based programming styles. The CRTP is indeed a multiparadigm programming product from OOP and GP rather than a language-related feature. The previous Leda example shows that it is not tied to C++. It is also possible to implement it in the D programming language [13].

2 This pattern is sometimes also called the Barton-Nackman trick, although this name refers to a different idiom (the Restricted Template Expansion) [18], which however relies on the CRTP.

The CRTP exhibits a curious circular dependency between the derived and the base class. There is a first dependency from the derived class to the base class via inheritance, and a second dependency from the base class to the derived class via a template parameter. The idea behind the CRTP is to combine templates and inheritance to provide an abstract class with information on its derived concrete class[3].

In classical OOP, it is generally impossible to tell the exact (most derived) type of an object at compile time. An object has indeed two types: a *static type*, which is known at compile time; and a *dynamic type* or *exact type*, which is known at run time, and which is a subtype of the static type (and even possibly the static type itself). An object can always be converted towards the type of (one of) its base class(es), as such a type is more general than its exact type; the conversion does not introduce type checking issues at compile time. Thus the compiler can guarantee that the conversion is valid *statically*. Therefore, a function taking an instance of the base class as input can accept any object of its derived classes: this feature is a trait of the *inclusion polymorphism* provided by the OOP paradigm.

However the downward conversion, from a base class instance to a derived class, cannot be verified at compile time: as the real type of an object is only known at run time, any conversion to a type more precise than the static type requires run-time checks, as it may fail at run time.[4] The compiler cannot guarantee that such a conversion is valid statically.

In the case of the CRTP however, the information of the exact type is propagated to the base class. Put differently, each base class using the CRTP has a name specific to its derived class. In the previous example, `A<B>` is base class of `B` and we can statically guarantee that an instance of `A<B>` can be converted to `B`, as only instances of `B` have `A<B>` objects as base classes. A downward conversion is not implicit, like an upward conversion; the `static_cast` keyword must be used to explicitly *cast* an instance to a subtype.

```
B b;
// Implicit upward conversion.
A<B>& a = b;
// Explicit downward conversion.
B& b2 = static_cast<B&> (a);
```

Note that in the definition of the class template `A`, we have used the name '`E`' for the formal parameter of the base class template holding the name of the derived class, standing for "exact". We stick to this convention in the rest of this thesis.

3 Abstract classes cannot be instantiated, therefore they can only serve as base classes.

4 Such conversions are performed with the `dynamic_cast` operator in C++.

The CRTP affects method calls as well. In classic OOP, calls to polymorphic methods (called virtual function members in C++) are resolved a run-time, as the executed code depends on the exact type of the target (the object the method belongs to). This behavior is also distinctive of inclusion polymorphism. In the CRTP however, the exact type is known at compile-time and a method call does not require any run-time computation: the binding of a method is entirely done at compile-time.

The CRTP acts as static OOP paradigm. A direct benefit of being static is getting rid of OOP run-time overheads. By integrating at compile time information that is usually only available at run time, the CRTP avoids run-time overheads of OOP mechanism. Moreover, it enables static resolution of method calls and opens the way for optimizations relying on static binding such as inlining of function bodies or constant propagation.

As the CRTP trades flexibility for efficiency, its main drawback is its lack of dynamic possibilities. There is no equivalent to polymorphic methods in CRTP, which are especially useful when the exact type of an object cannot be determined statically (for example when it depends on user input). Likewise, base classes generated from a common class template (e.g., `A`) are not compatible for different values of the exact type `E`. Instances of the previous base class `A<B>` cannot be converted to the base class `A<C>` of the following class `C`.

```
struct C : A<C> {};
C c;
A<B>& b = c; // Invalid: ‘c’ is not an ‘A<B>’.
```

In OOP, there would be no problem as the derived class would share the exact same base class. In the CRTP this lack of base class compatibility prevent inclusion polymorphism. A direct consequence is the impossibility to create polymorphic containers, such as lists, accepting instances of subtypes of a CRTP base class. There is indeed no sweet spot between a type such as `std::list<A>` (which is invalid, as `A` is not a type) and `std::list< A<B> >` (which is too restrictive, as it does not accept derived classes other than `B`, such as `C`).

Finally, as a product of template programming, the CRTP suffers from the same issues. It generates as many base classes as there are derived classes, whereas in classical OOP a single base class would be compiled. This behavior may increase compilation times, the size of the compiled unit (library or program) and even impact run time performances, as more code means more memory cache use.

Because of their static nature, CRTP classes are useful in contexts where efficiency is sought. They do not offer the flexibility of inclusion polymorphism of classical OOP classes, but both types of classes can coexist in a single program.

3.2 GENERALIZED CURIOUSLY RECURRING TEMPLATE PATTERN

The initial formulation of the CRTP [35] includes only one class and its base class: it is a two-level hierarchy. The issue of CRTP class hierarchies of more levels is not addressed by Coplien. Veldhuizen has postulated the possibility to generalize the CRTP to a class hierarchy of more than two levels [146]. Burrus et al. have proposed a generalization of the CRTP [32] that we call Generalized Curiously Recurring Template Pattern (GCRTP). The idea is to equip each non leaf class[5] of a hierarchy with a parameter (E) denoting the exact type of the leaf class that is derived from them. The exact type E is then passed bottom-up recursively among the non leaf classes. Such GCRTP class hierarchies are called *static hierarchies* by Burrus et al.. The following lines show an example of a static hierarchy.

```
template <class E> struct A {};
template <class E> struct B : A<E> {};
template <class E> struct C : B<E> {};
struct D : C<D> {};
```

Note that in the previous example, a non leaf class (for instance C) cannot be used as a leaf classes, as it expects an exact type passed as a parameter—here, C itself. However C alone is not a valid type name, hence C<C> is also invalid. The initial proposal of Burrus et al. shows how non-leaf classes support can be added to the CRTP. We do not detail nor use this feature in our work, as it increases the complexity of the pattern without adding much expressive power. Besides, making non-leaf classes of a hierarchy abstract [106]—or put differently, not deriving from concrete classes [134]—is a sound software engineering principle.

3.3 THE SCOOP PARADIGM

Based on the idea of the GCRTP, Burrus et al. have designed a new programming paradigm mixing the benefits of OOP and GP called SCOOP (Static C++ Object-Oriented Programming). The paradigm has been later refined by Géraud and Levillain [65] ("SCOOP 2") to support additional features such as morphers (see Section 3.3.6, p. 95). The third version of the paradigm that we present in this thesis ("SCOOP 3") is a simplified version of this second design, yet providing the same expressive power. It has been used to develop the Milena library from the Olena 1.0 and 2.0 platforms [52]. Despite its name, SCOOP is not tied to C++ *per se*, and can probably be adapted to a similar programming language such as D.

5 We call "leaf class" a class that has no subclass.

3.3.1 *Concepts*

In SCOOP, classes are organized in static hierarchies using the GCRTP. The top-most base class of all these classes is the same class template, used to factor common paradigm-related equipment, named `Object<E>`[6]. This class is defined as follows:

```
template <typename E>
struct Object
{
  typedef E exact_t;
protected:
  Object();
};
```

The `protected` constructor prevents `Object<E>` from being instantiated directly[7], and therefore ensures it is an abstract class. The `exact_t` typedef contains the exact type (`E`) of the concrete class derived from `Object<E>`.

Static class hierarchies deriving from `Object` are split in two parts (see Figure 4). The upper part is related to the type interfaces, and contains classes expressing *concepts* (see Section 2.4, p. 52). Such concepts are related to the domain, and should be designed as orthogonal abstractions to develop "abstract" (generic) algorithms later. In IP, abstractions may be Image, Site_Set (a set of point defining the domain of an image), Neighborhood, Window (sliding window implementing a structural elements from Mathematical Morphology), etc. As a matter of fact, the Milena library features classes named `Image<E>`, `Site_Set<E>`, `Neighborhood<E>`, `Windows<E>`, etc. to implement these very abstractions. It is possible to refine a concept by deriving from it. The `Box<E>` concept, representing an abstract "box" (or hyperrectangle) type in a discrete space of points, is a subclass of `Site_Set<E>`, as the latter is more general than the former (the requirements of the concept `Site_Set<E>` are a subset of the requirements of `Box<E>`).

As in the STL (see Section 2.2.3, p. 38) and similar generic libraries (see Section 2.3, p. 49), SCOOP concepts shall define requirements, to be fulfilled by classes modeling (implementing) them (see Section 2.4, p. 52). Requirements are expressed as syntactic and semantic constraints on the model and its instances in the documentation of the concept. The two main kinds of requirements are the ones that can be checked at compile-time: *associated types* and *services*. Associated types are a set of `typedef`s that the model must define. The documentation may specify additional constraints on an associated type. For instance, an image domain

6 In the initial SCOOP proposals [32, 65], this class was named `Any<Exact>`.
7 A `protected` constructor can still be called by the constructor of a derived class.

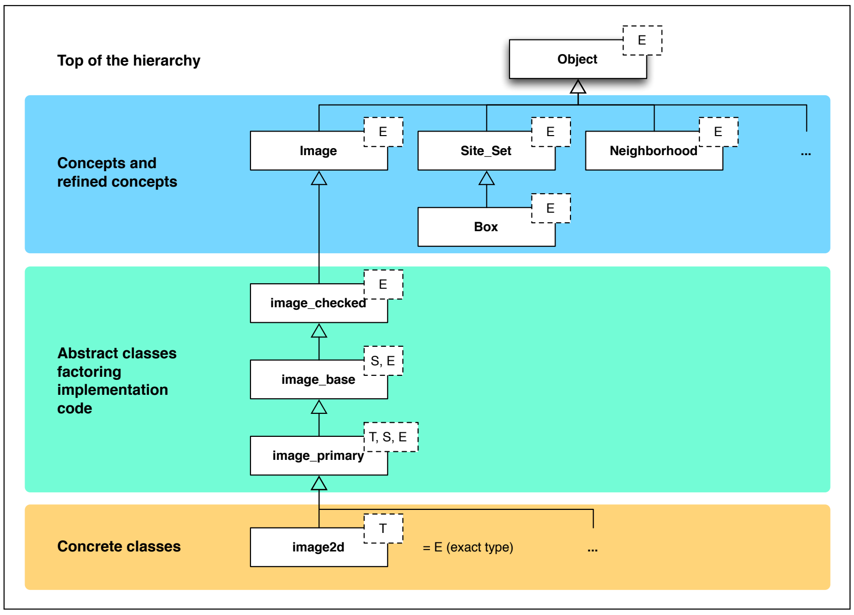


Figure 4: An example of static hierarchy from the Milena library following the design principles of the SCOOP paradigm.

type in Milena, defined by the associated type `domain_t` of the Image concept (see Table 2, p. 82), must be a model of Site_Set. Services are the methods that must be provided by a model of a concept. The constraint of each service is expressed as the signature of a method.

Note that in Milena we use the term *site* instead of *point* for the sake of generality (see Section 4.4, p. 124). Table 2 also mentions *psites*; their difference with sites is also explained in Section 4.4, as well as other IP-related concepts.

The role of a concept is essentially to give a *name* to an abstract notion. Indeed, as the current C++ standard [87] does not provide any means to express the requirements of a concept, concept classes can basically be *empty*. However, it is possible to use template metaprogramming techniques (see Section 2.6.1, p. 69) to implement compile-time concept checking (see Section 2.4.2, p. 58). Although this concept enforcement strategy has many drawbacks, it is helpful to diagnose an error related to an incorrect use of a concept (see Section 3.3.4, p. 86).

3.3.2 *Implementation Classes*

The lower part of the static hierarchy is composed of implementations classes (see Figure 4) containing actual code. Leaf classes represent concrete data types, while abstract (non leaf) classes serve as factoring facilities. For instance, `image2d<T>` in Figure 4 is a concrete image type of a 2-dimensional image, the domain

Image		
Associated types		
Type	Model of	Definition
`domain_t`	Site_Set	Type of the domain
`site`	(Site)	Type of a site
`psite`	Point_Site	Type of a site access
`piter`	Site_Iterator	Default iterator type
`fwd_piter`	Site_Iterator	Forward iterator type
`bkd_piter`	Site_Iterator	Backward iterator type
`value`		Type of a value
`rvalue`		Type of a read-only access
`lvalue`		Type of a read/write access
`vset`	Value_Set	Type of the set of values
Mandatory Services		
Method signature		Definition
`const domain_t& domain() const`		Return the image's domain
`bool has(const psite& p) const`		Site membership test
`bool is_valid() const`		Image validity test
`rvalue operator()(psite& p) const`		Read-only value at `ima(p)`
`lvalue operator()(psite& p)`		Read/write value at `ima(p)`
`const vset& values() const`		Return the value set
Optional Services		
Method signature		Definition
`unsigned nsites() const;`		Return the number of sites

Table 2: Signature of the Image concept.

of which is a box (rectangle) on $\mathbb{Z}^2$, mapping points of this domain to a value T[8]. image2d<T> is derived from classes such as image_primary<T, S, E> and image_base<S, E>, which contains code shared by image2d<T> and other concrete image classes based on a domain of type S. Being a model of Image, image2d<T> implements the requirement of this concept. In particular, it provides definitions for Image's associated types and services (see Table 3), either by defining them directly or by inheriting them from of a base classes such as image_primary<T, S, E> and image_base<S, E>.

The interface of implementation classes may be richer than the interface(s) required by the concept(s) it models. This is especially useful when its specific properties (see Section 3.3.5, p. 89) are used to implement a faster variant of a generic algorithm (see Section 4.6.4, p. 143). For instance, in addition to the mandatory site-based accesses provided by operator(), image2d<T> supports value access through an index (a position relative to the beginning of the data array) thanks to the following element methods:

```
/// Read-only access to the image value
/// located at index \p i.
const T& element(unsigned i) const;
/// Read-write access to the image value
/// located at index \p i.
T& element(unsigned i);
```

Accessing (and therefore browsing) an image with an index is faster than with a site (a point2d in this case), since the former is implemented as a mere memory access, while the latter requires some computation to obtain the value location beforehand.

3.3.3 *Algorithms*

As in GP, SCOOP algorithms are written as standalone (non member) functions, to uncouple data structures and algorithms [107]. Algorithms take their input as abstract data types. For instance, a Milena algorithm algo taking models of Image as input has the following signature:

```
template <typename I>
void algo(Image<I>& input);
```

The implementation of this algorithm however needs to know the exact type of input, that is I. Therefore a routine exact converting an instance of the SCOOP hierarchy to its exact type

8 The parameter T of the concrete image image2d<T> should not be confused with the parameter E of any of its base classes (e.g., Image<E>) : the former is a means to make image2d<T> generic with respect to the type of value contained in the image, while the latter is an artifact of the SCOOP paradigm.

image2d<T>		
Associated types		
Type	Value	Where defined
domain_t	box2d	image_base<E>
site	point2d	image_base<E>
psite	point2d	image_base<E>
piter	box2d::piter	image_base<E>
fwd_piter	box2d::fwd_piter	image_base<E>
bkd_piter	box2d::bkd_piter	image_base<E>
value	T	image2d<T>
rvalue	const T&	image2d<T>
lvalue	T&	image2d<T>
vset	value::set<T>	image2d<T>
Services		
Method signature		Definition
const box2d& domain()		image2d<T>
bool has(const point2d& p) const		image2d<T>
bool is_valid() const		image_base<E>
const T& operator()(point2d& p) const		image2d<T>
T& operator()(point2d& p)		image2d<T>
const value::set<T>& values() const		image2d<T>
Optional Services		
Method signature		Definition
unsigned nsites() const;		image_base<E>

Table 3: image2d<T>, a model of Image.

comes in handy. The implementation is straightforward: it reuses the downward conversion based on the `static_cast` operator from Section 3.1 (p. 76)[9].

```
template<typename E>
E& exact(Object<E>& o)
{
  return static_cast<E&>(o);
}
```

By converting `input` to its exact type, the algorithm is able to use the interface (methods, attributes, typedefs) of the exact type (e.g. `image2d<T>`) that is only *described* in the concept constraining the type of the input (`Image<I>`, where `I = image2d<T>`)[10].

```
template <typename I>
void algo(Image<I>& input_)
{
  // ‘input_’ (of type ‘Image<I>’) does not
  // have a method ‘domain()’.

  // Convert ‘input_’ to its exact type (‘I’).
  I& input = exact(input_);

  // ‘input’ has a method ‘domain()’.
  typedef I::domain_t = input.domain();

  // Perform some computation.
  // ...
}
```

In classical OOP there is no similar issue as abstractions (expressed as abstract base classes) have an actual interface expressed as abstract methods (pure virtual members). Therefore the compiler is aware of the available methods, though their actual implementation is deferred to the exact (dynamic) type of the input.

Beside this particularity, the use of algorithms is very similar to GP. Calls do not require any explicit conversions of inputs to their corresponding abstractions, as upward conversions are automatic. In the following lines, `ima` is implicitly converted to the type `Image< image2d<int> >` by the compiler when passed to `algo`.

9 This simple implementation is yet limited. It cannot handle diamond inheritance (where a class `D` is derived both from classes `B` and `C`, themselves inheriting from a class `A`). The version of `exact` implemented in Milena is more complex and is able to handle diamond hierarchies.

10 As a convention, in the implementation of an algorithm, we add an underscore (‘_’) to the name of an argument that is to be converted to its exact type, so that the original name (without underscore) can be used in the body of the algorithm to refer to the object using its exact type

```
// Create a 512 x 512 image.
image2d<int> ima(512, 512);
// Initialize 'ima' with data.
// ...

algo(ima);
```

3.3.4 *Concept-Checking and Other Compile-Time Constraints*

Section 2.4.2 (p. 58) defines concept checking as the action of checking whether the effective parameters used in a template specialization actually model the concepts of the corresponding formal parameters of the template.

The SCOOP paradigm enforce concepts using named conformance: to model a concept `C`, a type `C` must inherit from `C<T>`. However, `C` does not express requirements on `T` *per se*. With no further equipment, this concept checking strategy is a pure naming convention, and does not enable the compiler to report concept errors ahead of the use of `T` within a routine expecting a parameter modeling the concept `C`.

Therefore, SCOOP encourages concept writers to equip their concept classes with additional compile-time concept checking clauses using C++ metaprogramming techniques (some of them have been evoked in Section 2.6, p. 69) and proposes some tools to simplify the writing of such checks. The technique is similar to the static interfaces proposed by McNamara and Smaragdakis [103]. Constraints are expressed as expressions performing no operations ("no-ops") but that must however still comply with syntactic and semantic rules of the language. If any of these expressions are deemed invalid by the compiler, then the model has failed to fulfill the requirements of the concept. Usually, these constraints are located in the default constructor of the concept class. The following excerpt of the constructor of the `Image<E>` class in Milena shows constraints on required associated types and services:

```
template <typename E>
inline
Image<E>::Image()
{
  // Check associated types.

  typedef typename E::domain domain_t;
  typedef typename E::site   site;
  typedef typename E::psite  psite;

  typedef typename E::piter     piter;
  typedef typename E::fwd_piter fwd_piter;
  typedef typename E::bkd_piter bkd_piter;

15    typedef typename E::value  value;
16    typedef typename E::rvalue rvalue;
17    typedef typename E::lvalue lvalue;
18
19    // ...
20
21
22    // Check services.
23
24    const domain_t& (E::*m1)() const = & E::domain;
25    m1 = 0;
26
27    bool (E::*m2)(const psite& p) const = & E::has;
28    m2 = 0;
29
30    bool (E::*m3)() const = & E::is_valid;
31    m3 = 0;
32
33    rvalue (E::*m4)(const psite& p) const = & E::operator();
34    m4 = 0;
35    lvalue (E::*m5)(const psite& p) = & E::operator();
36    m5 = 0;
37
38    // ...
39  }
```

These constraints represent some of the requirements of the Image concept from Table 2 (p. 82). We recall that E is the exact type inheriting from Image<E>, which also means that E shall be a model of Image. Firstly, associated types required by a concept are defined as typedefs (lines 7–17). For instance Image requires a domain_t associated type from the model (E), which implies that the expression E::domain_t shall represent a valid type, which may be given an alias of the same name in the concept class (line 7). Secondly, services can be checked by creating a pointer on each required method (member pointer) having as type the expected signature (lines 24–36). If the model does not provide a method with the name assigned to the pointer and having the same signature, the compilation will fail to compile the constructor of Image, revealing a lack in the concept enforcement. Line 24 shows how Image ensure that the model E provides a method `const domain_t& domain() const`: it creates a pointer to a member (m1) having the required signature (`const domain_t& (E::*)() const`), and initializes it with the address of the method from the exact type (`& E::domain`). If E does not provide a domain() method, or if this method does not match the expected signature, the compiler will report an error[11]. Although the syntax of this technique is cumbersome, it is effective and it can even distinguish two methods by their

11 Such member pointers are then set to '0' as a means to avoid warnings from the compiler regarding unused variables.

`const` nature (e.g., `rvalue operator()(psite& p) const` versus `lvalue operator()(psite& p)` in lines 33–26).

In addition to these concept checking tools, compile-time constraints can be expressed using *static assertions* implemented using metaprogramming constructs triggering compile-time errors when they are not satisfied. Milena provides metaprogramming constructs (see Section 2.6, p. 69) to perform compile-time computations on types and constants values. For instance, the class template `metal::equal<T1, T2>`[12] is used to compare two types `T1` and `T2`. `metal::equal<T1, T2>` defines a class method (`static` member function) `check()` if and only if `T1` and `T2` are equal. This template can be used for example within an algorithm working only with binary images (e.g., a component labeling algorithm) to ensure that the input image has values of type `bool`. The first lines of such an algorithm could be as follows:

```
template <typename I, typename J>
void
labeling (const Image<I>& input,
          Image<J>& output)
{
  // Static precondition: Ensure the value
  // type of ‘I’ is ‘bool’.
  metal::equal<I::value, bool>::check();

  // ...
}
```

When `I` is not an image of `bool`, the compiler is unable to find the `check()` method, and stops the compilation. This mechanism is similar to the `BOOST_STATIC_ASSERT` macro [34] and to the `static_assert` keyword [90] of the C++ 2011 standard [87].

The idioms presented in this section make up for C++'s lack of proper language constructs to enforce concepts. However, these techniques have several drawbacks. First and foremost, they make use of a verbose and unnatural syntax, which is a classical side effect of template metaprogramming, and templates in general. Secondly, they may not cover the whole spectrum of concept requirements. A simple example is checking for a template member function: as C++ forbids pointers on (non-specialized) functions[13], we cannot use the technique shown above to check the existence of such a member function. Finally, these concept check expressions generally yield long and complex errors from compilers, like any template-related errors.

12 Milena's `metal` namespace is dedicated to metaprogramming.

13 Like any template, a template (member) function is "potential" code. Only when fully specialized (instantiated) does a template function generate actual code and is given a memory location, i.e. an address a pointer can point to.

3.3.5 *Properties*

One of the motivations at the origin of the SCOOP paradigm is to augment generic data structure with semantic information related to the target domain [65]. By attaching extra information to a type, it is possible to let algorithms inspect its characteristics and take advantage of them. Such algorithms may bring improvements with respect to one or several of the following aspects:

EXECUTION SPEED Fast implementations of an algorithm may be known for a structure exhibiting e.g. a regular organization of data or computing results on the fly. See Section 4.6.3 (p. 139) for more information on such implementations and examples in the domain of IP and Section 4.6.4 (p. 143) for details on how to implement the selection of an implementation based on input types' properties.

MEMORY USAGE Likewise, there may be variants of an algorithm using less memory than the default implementation, for a data structure meeting some requirements.

TRACING, DEBUGGING AND ADJUSTING ALGORITHMS Additional information provided by a type's semantic information may be useful to understand the behavior of an algorithm, fix its bugs and possibly improve its implementation.

In SCOOP, semantic information attached to a data structure are called *properties*. A property is a static piece of information written by the implementer of a type describing an aspect of this type. Such information is usually related to the application domain such as IP; and it is non-trivial, meaning that it cannot in general be inferred by the compiler (e.g. to perform some optimization). So this programming approach is considered *declarative*. As properties are compile-time information, they may be used in a template metaprogram (see Section 2.6, p. 69) to generate a dedicated code within an algorithm with zero run-time overhead.

A property is related to an abstraction. In a IP context, designers may want to define properties for abstractions of the domain such as Image, Site_Set (image domain), Value (image value type), Window (sliding window or morphological structuring element), etc. An abstraction's properties form a set that is first *declared*. This declaration introduces the name of all properties, but does not give a value to them—they are abstract. The documentation should describe the meaning and the intent of each property. Table 4 shows the properties of the Image concept in Milena.

A concrete class modeling an abstraction shall *define* each of the abstraction's properties. Meaningful values may be provided by the designer of the abstraction, so that implementers of its

Name	Meaning
	Miscellaneous properties
`category`	Image nature: primary or morpher (see Section 3.3.6, p. 95)
`speed`	Ability to provide fact access/browsing
`size`	Size of the image: regular or very large
	Values properties
`vw_io`	Read-only or read/write values (data)
`vw_set`	Data stored in zero, one or several sets
`value_access`	Directed or indirect (e.g. computed) access
`value_storage`	Layout in memory: unique value, single block, piece-wise or disrupted data
`value_alignment`	Values aligned w.r.t. a grid or not
	Domain and geometry properties
`pw_io`	The image provides a read-write `operator(const psite\& p)` or not
`localization`	Site located on a grid (isotropic or not), some other space, or no location at all
`dimension`	Dimension of the space (if any)
	Extended domain properties
`ext_domain`	Presence and nature of an extended domain (fixed, infinite, extensible)
`ext_value`	Number of values constituting the domain (one or many)
`ext_io`	Input/output operations in the extended domain: none, read-only, read/write
	Data properties
`kind`	Kind of values: color, gray-level, binary, etc.
`nature`	Nature of values: scalar, vectorial, etc.
`quant`	Quantification: low or high

Table 4: Properties of the Image concept in Milena. Prefixes "pw" and "vw" stand for "point-wise" and "value-wise" respectively.

models use the same convention, which is useful when these properties are later used to implement algorithms variants.

Properties are implemented as traits classes[14] (see Section 2.6.2, p. 71). In addition, they are organized as a class hierarchy ordered like the data structure hierarchy. A set of properties is a list of (key, value) pairs implemented as `typedef`s: the key is the newly introduced type alias, while the "value" is the type upon which the `typedef` is defined. The latter type is often a type dedicated to the definition of a property, which is a part of a class hierarchy. The top of this hierarchy (`any`) is the most general value of the property.

Following the OOP paradigm, an inherited property may either be left unaltered (keeping the value of the base class' property) or given a new value to reflect a change in the semantic of the class with respect to its base class.

To simplify the work of data structure implementers, the designers of properties should provide default values for each set of properties. For instance, Milena has a `trait::undefined_image_<I>` class template "setting" all the properties listed in Table 4 to the "value" (type) `undef`:

```
namespace trait
{
  template <typename I>
  struct undefined_image_
  {
    typedef undef category;
    typedef undef speed;
    typedef undef size;
    // ...
  };
}
```

Actual values of properties are themselves organized as separate hierarchies. For instance, the `speed` property of an image should be defined as one of the types of the following `trait::image::speed` hierarchy:

```
namespace trait
{
  namespace image
  {
    struct speed
    {
      struct any {};
      struct slow : any {};
      struct fast : any {};
```

14 For that matter, the classes enclosing the definition of properties are named "traits" in Milena.

```
        struct fastest : fast {};
      };
   }
}
```

A `speed::fast` image is an image that can be processed with a pixel iterator, a small object faster than a site iterator since it browses the *memory* where the image data is stored instead of browsing the *domain* of the image, which is made of sites (e.g., 2D points). A `speed::fastest` image is an image that can be processed with even lower-level (and faster) tools, such as pointers or routines like `memcpy()` (byte-level memory copying). A `speed::slow` image does not support such access mechanisms: it can only be browsed with a site iterator delivering a site (e.g. `p`) corresponding to a location in the image's domain. Such a site can be used as input of the image's `operator()` to access the corresponding value (`v = image(p)`). This operation is slower than using a pixel iterator (`speed::fast`) or a pointer (`speed::fastest`), since it implies a computation to obtain the address of the value (`v`), whereas the two previous access types have a direct knowledge of this address.

The inheritance relation between two values represent the inclusion. Here, a `speed::fastest` image is also a `speed::fast` image (and also a `speed::any` image), meaning that an algorithm working on `speed::fast` images will also be able to process `speed::fastest` images.

Many image types store their data in a single buffer[15], which is the sole requirement a `speed::fast` image shall fulfill. Therefore, Milena provides a `trait::default_image_<T, I>` class template derived from `trait::undefined_image_<I>`, where the `speed` property of an image type `I` having values of type `T` is set to `speed::fast`. The rest of the class defines default values for properties `kind` and `quant` computed from the properties of the value type `T`:

```
namespace trait
{
  template <typename T, typename I>
  struct default_image_ : undefined_image_<I>
  {
    // Miscellaneous properties.
  public:
    // Speed is fast by default.
    typedef speed::fast speed;

    // Data properties.
  private:
    typedef mlc_equal(mln_trait_value_quant(T),
                      value::quant::high)
```

15 This buffer may take into account the space needed to store the values of a potential extended domain (e.g. a fixed-width image border).

```
                                     is_high_quant_;
  public:
    typedef mln_trait_value_kind(T)   kind;
    typedef mln_trait_value_nature(T) nature;
    typedef mlc_if(is_high_quant_,
                   image::quant::high,
                   image::quant::low) quant;
  };
}
```

In the previous code, `mlc_equal` and `mlc_if` are shortcut macros performing metaprogramming operations (resp. compile-time type equality and `if`-based test). `mlc_equal` is in fact a short alias for the `metal::equal` metaprogramming construct seen in Section 3.3.4 (p. 86), defined as this:

```
#define mlc_equal(T1, T2) metal::equal< T1, T2 >
```

Macros invocations `mln_trait_value_quant(T)`, `mln_trait_value_-kind(T)` and `mln_trait_value_nature` are used to query properties associated to a value type, stored in the traits class `trait::value_`:

```
#define mln_trait_value_quant(V) \
  typename trait::value_< V >::quant
#define mln_trait_value_kind(V) \
  typename trait::value_< V >::kind
#define mln_trait_value_nature(V) \
  typename mln::trait::value_< V >::nature
```

`trait::default_image_<T, I>` factors values of properties which are shared among several image types, and hence shortens the property list of these classes. `image2d<T>` is an example of such a class. It inherits from `trait::default_image_<T, image2d<I> >` and only defines the remaining properties:

```
namespace trait
{
  template <typename T>
  struct image_< image2d<T> >
    : default_image_< T, image2d<T> >
  {
    // Miscellaneous properties.
    typedef category::primary category;
    typedef speed::fastest    speed;
    typedef size::regular     size;

    // Values properties.
    typedef vw_io::none                 vw_io;
    typedef vw_set::none                vw_set;
    typedef value_access::direct        value_access;
    typedef value_storage::one_block    value_storage;
    typedef value_alignment::with_grid value_alignment;

    // Domain and geometry properties.
    typedef pw_io::read_write        pw_io;
    typedef localization::basic_grid localization;
    typedef dimension::two_d         dimension;
    // Extended domain properties.
    typedef ext_domain::extendable ext_domain;
    typedef ext_value::multiple    ext_value;
    typedef ext_io::read_write     ext_io;
  };
}
```

Constraints on Properties

In order to constitute a consistent and meaningful set of information regarding the values assigned to the properties of a type, it may be useful to develop a set of constraints on the properties of every concerned abstraction. Such constraints shall be part of the documentation, and may be enforced along with concept-checking statements, either in implementation classes (e.g. within abstract implementation classes where such checks can be factored), in algorithms making use of said properties, or even in external, stand-alone tests.

For instance, the documentation of Milena regarding Image properties states that a "fastest" image shall store all its values in a contiguous block of data and shall have an extended domain to accommodate browsing around sites (with respect to a neighborhood or sliding window) located on the image's border, without a prior membership test. Indeed in many cases it is faster to allow an algorithm to browse sites outside the official domain of an image by supplying an extended domain, instead of systematically checking whether accessed sites belong to the initial domain (which requires extra computation and above all, probable memory access, that are even more time consuming). This constraint can be expressed as this:

$$
\begin{aligned}
&\forall I\ \texttt{trait::image\_<}I\texttt{>::speed} = \texttt{speed::fastest}\\
&\Rightarrow (\texttt{trait::image\_<}I\texttt{>::value\_storage}\\
&\qquad\quad = \texttt{value\_storage::oneblock}\\
&\qquad \wedge\ \texttt{trait::image\_<}I\texttt{>::ext\_domain}\\
&\qquad\quad = \texttt{ext\_domain::some})
\end{aligned}
$$

Specific Interface

A type's properties convey some of the semantics of this type, with the intent to help authors of algorithms to leverage this information. Taking advantage of such a characteristic offered by a data structure (reflected by one or several of its properties) may require an augmented interface (additional associated types and services) with respect to the minimal interface imposed by the concepts modeled by the structure. This extra interface shall be mentioned in the documentation of the properties and may be enforced with concept-checking statements.

We have seen earlier that `speed::fastest` image types must store their data in a single buffer (`value_storage::oneblock`) and provide an extended domain (`ext_domain::some`). These two property values come with extra interface requirements. Milena's documentation states that `value_storage::oneblock` images shall provide the following extra interface[16] in addition to the one required by Image (see Table 2, p. 82)[17]:

Extra interface for `value_storage::oneblock` image types.

Associated types		
Type	Model of	Definition
`dpoint`	Dpoint	Delta-point type.

Services	
Method signature	Definition
`unsigned nelements_()`	Number of elements in the buffer
`psite point_at_index_(unsigned i)`	Psite corresponding to the index `i`
`unsigned delta_index_(dpoint dp)`	Delta-index related to delta-point `dp`.

Likewise, `ext_domain::some` images must features the following additional service:

Extra interface for `ext_domain::some` image types.

Services	
Method signature	Definition
`unsigned border_()`	Return the border thickness.

Algorithms providing an implementation variant for `speed::fastest` may make use of the `typedefs` and methods shown previously. For instance, they may browse the values of an image `ima` by iterating directly on its memory buffer, returned by `ima.point_at_index_(0)` and comprising `ima.nelements_()`. The class `image2d<T>`, being a `speed::fastest` image type, features the above interface and can therefore benefit from faster implementations.

3.3.6 *Morphers*

The second proposal of the SCOOP paradigm [65] introduces the idea that generic data types in a reusable programming framework should be decomposed into

16 Note that methods which are part of the specific interface often have a trailing underscore in their name to remind users that they are not part of the concept's requirements and to discourage their use in generic implementations. They may however be used in algorithms variants optimized for types having certain properties; see Section 4.6.3 (p. 139) on generic optimizations for more details.

17 The notions of delta-point and delta-index are presented in Sections 4.4.2 (p. 129) and 4.4.5 (p. 133).

- *primary data structures*, i.e. standalone data types storing actual data such as images, vectors, graphs, etc.; and
- *light* data types with no or little actual data build on top of another type[18] and acting like a *type transformations*, by adding new services, modifying the behavior of the underlying type, changing its interface to adapt it to another interface, etc. We call such data types *morphers*.

By light, we mean that morpher data types are not meant to carry actual data, like primary data structure do. Instead, as they are built on top of another data structure (either a primary or a morpher type), they are expected to use the data of that type instead of duplicating them. Morphers may thus *delegate* some or all of their operations to the structure they are based on.

Definition

A morpher `M` is a generic type parameterized by a type `T` ("the underlying type"), built on top of this type. `M<T>` realizes a type transformation of `T` and should observe the following rules:

- An instance of `M<T>` must hold a reference on an instance of `T`. There shall be not data duplication: a morpher instance is a light object that accesses the data of the underlying type through a reference, not a copy.
- `M<T>` must provide an interface similar to `T`'s, or derived from `T`'s, according to the nature of the transformation. For instance, a morpher adding logging capabilities to a type to record a trace of each method called on an object does not change the interface the underlying type; the morpher only alters the behavior of the object at run time. On the other hand, a morpher turning a 3D image into a 2D image, by offering a view of one of its slices along a given plane, introduces important change in the interface of the initial type: instead of using 3D points to access the values as in the case of the initial image, the morpher object uses 2D points, and shall refuse 3D points as argument of its `operator()` (see the signature of the Image concept in Table 2, p. 82).
- By default, an instance `M<T>` delegates its operations (method calls) to the aggregated `T` instance, unless `M<T>` provides a replacement for it or if the transformation does not allow this service. In the previous example of the morpher logging read and write accesses in an image, `M<T>` delegates every method call to the `T` object, except for `M<T>::operator()`:

18 These data types may also be based on more than one type, but we do not address this case in this thesis.

the morpher provides its own implementation of this routine, logging each call (e.g. into a file), and eventually delegates the actual access to data to `T::operator()`. In the case of the second morpher example above, most of the methods of the morpher supersede the methods of the underlying type to match the deep changes of the interface, although all of them will contain calls to the methods of the initial type *in fine*.

- The coupling between the morpher type and the underlying type should be minimal, to ensure a maximal reusability of the morpher. To make `M` and `T` as orthogonal as possible, `M<T>` should use `T`'s associated types instead of specific types. The goal is to make morphers as generic as possible, similarly to primary data structures.
- The properties of `M<T>` should be defined by taking into account the properties of `T` and the transformation induced by `M`.

Morphers use an architecture based on *aggregation*, to model the HAS A relation ("an `M<T>` HAS A `T`"); and *delegation*, as some or all of the calls to `M<T>`'s method may be direct or indirect calls to `T`'s methods). There should be no inheritance link between `M<T>` and `T` (i.e., `M<T>` shall not be a subclass of `T`). Indeed, as far as interfaces are concerned, inheritance models the IS A relation in OOP. However, some morphers may alter the interface (and consequently the implementation) of their underlying type so much that we can no longer say that "`M<T>` IS A `T`". This is the case of the slice morpher mentioned previously, which turns a 3D image into a 2D one: a 2D image is *not* a 3D image.

Purpose and Illustrations

The idea of morphers is a continuation of the idea of lifting from Figure 1 (p. 50). Even in a generic programming context, it is not uncommon to find a set of algorithms sharing a common structure, but having slight differences. For instance, many IP algorithms can be implemented with a "mask" passed as extra argument, limiting the processed area of the input image(s). To illustrate this issue, let us consider a simple algorithm filling the values of an image `ima` with a value `v`, named `fill`. A definition of `fill` is shown below. As the full listing corresponding to this example (as well as the next ones) introduces notions of generic IP from Milena that are beyond the scope of this chapter and that are not directly related to the topic of morphers, we only briefly present some of the features being used. More details are given later in Sections 4.6 (p. 137) and 4.7 (p. 146).

```
template <typename I, typename V>
```

```
void
fill(Image<I> ima_, const V& val)
{
  I& ima = exact(ima_);

  // ‘p’ browses the sites of ima’s domain.
  typename I::piter p(ima.domain());
  for(p.start(); p.is_valid(); p.next())
    ima(p) = v;
}
```

I::piter is a site iterator type associated to the image type I, much like C::iterator is an STL iterator type associated to the STL container C, where C may be std::vector<int>, std::list<float>, etc. (see p. 43). p is a site iterator object, and is passed the ima's domain as argument (ima.domain()) at its construction. Milena's site iterators does not work like STL iterators. The latter work similarly to pointers, while the former are object attached to a site set (here, the domain of an image) and behaving like a site. The call to p.start() positions p at the beginning of the domain; p.is_valid() indicate whether p is within the boundaries of the site set; and p.next() advances p to the next item.

Some patterns are so frequently used in Milena that the library features shortcuts for accessing associated types and iterating on an image using a site iterator. Obtaining the site iterator type associated to a type T is shorter to write with the mln_piter macro defined as this:

```
#define mln_piter(T) typename T::piter
```

Likewise, the for-loop site iteration above is so common that Milena provides a for_all macro:

```
#define for_all(x) \
  for(x.start(); x.is_valid(); x.next())
```

With the previous definitions, we can rewrite the fill routine as a shorter and clearer function:

```
template <typename I, typename V>
void
fill(Image<I> ima_, const V& val)
{
  I& ima = exact(ima_);

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

We can define a variant of `fill` taking a `mask` object implemented as a (binary) image[19] as an extra argument. The implementation is similar to the previous one, with the addition of an extra condition on `mask(p)`.

```
template <typename I, typename V, typename J>
void
fill_within_mask(Image<I> ima_, const V& val,
                 const Image<J>& mask_)
{
  I& ima = exact(ima_);
  const J& ima = exact(mask_); // A binary image.

  mln_piter(I) p(ima.domain());
  for_all(p)
    if (mask(p))
      ima(p) = v;
}
```

Despite their similarity, the two previous algorithms, though generic, require *two different* implementations. Likewise, each additional variant, as small as it may be, introduces new functions with redundant parts. As a last example, if `fill` were to process only the red channel of a red-green-blue (RGB) color image, yet another implementation would be needed:

```
template <typename I, typename V>
void
fill_red_channel(Image<I> ima_, const V& val)
{
  I& ima = exact(ima_); // An RGB image.

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

In the previous example, `ima(p)` returns the location of the value (an RGB triplet) in `ima` associated to `p`, and `ima(p).red()` returns only the red component of this value. This routine is again very similar to the previous two others, and proceeds with the code duplication phenomenon that we have precisely tried to avoid by using GP in the first place.

An alternative strategy to code duplication is to identify the shared structure of the routines to write a single algorithm (`fill`), and act on the input to change the behavior of the algorithm *from the outside* using a morpher data type. In the case of `fill_within_mask`, this amounts to embed the mask *into* the input

19 Where `mask(p) = true` (resp. `false`) means that `p` belongs (resp. does not belong) to the mask.

object, and have it present a different domain taking the mask into account. Such a type could be written like this:

```
class masked_image2d<T>
  : ... // Some class derived from
        // Image< masked_image2d<T> >.
{
public:
  // Instead of a box2d, the domain is now
  // a mask.
  typedef mask2d domain_t;
  // Likewise, the site iterator is now
  // related to a mask.
  typedef mask2d::piter piter;

  // Construction of a 2D masked image.
  masked_image2d(image2d<T>& image,
                 const mask2d& mask)
    : image_(image), mask_(mask)
  {
  }

  // Return the mask as domain.
  const mask2d& domain() const
  {
    return mask_;
  }

  // Query the mask for site membership.
  bool has(const point2d& p) const
  {
    return mask_.has(p);
  }

private
  /// The actual image.
  image2d<T>& image_;
  /// The domain implemented as a 2D mask.
  mask2d mask_;
};
```

where `mask2d` is a site set type implementing a binary mask, possibly containing itself an instance of `image2d<bool>` (but other implementations are possible). Likewise, `mask2d::piter` is a new site iterator type browsing a `mask2d` (instead of a `box2d::piter` traversing a `box2d`). We postpone the actual definition of `masked_image2d<T>`'s base class to a later section (see Section 4.7, p. 146) as this matter is beyond the topic of morphers in general.

We can observe that the initial implementation of `fill` is actually already generic enough to accept this new image type, and therefore it is perfectly possible to reuse it to process a `masked_image2d`:

```
// An 2D initial image.
image2d<int> ima1;
// Initialize 'ima' with data.
// ...

// A 2D mask (compatible with 'ima').
mask2d mask;
// Initialize 'mask' with data.
// ...

// Create a masked image.
masked_image2d<int> masked_ima(ima, mask);
// Apply the original 'fill' to it.
fill(masked_ima, 42);
```

The invocation of `fill` on the last line triggers the specialization of the `fill< masked_image2d<int>, int >`, where the domain returned by `ima.domain()` is no longer a `box2d`, but a `mask2d`. The associated `piter` type returned by `mln_piter(I)` is a `mask2d::piter` initialized with the input's mask (domain). This iterator browses only the sites that are member of the mask, therefore the `for_all(p) ima(p) = v` assigns only the values in the mask. In other words, this specialization of `fill` performs the exact same operations as the loop of the `fill_within_mask`, but does not require a new implementation. The behavior of the algorithm is different because its input is different.

We can push this idea further by turning `masked_image2d` into a type compatible with *any* image type, not just `image2d<T>`. To do so, this new type must be generic with respect to both the image type and the mask type. Milena's `image_if<I, F>` (see Section 4.7, p. 146) implements this idea. This image morpher type is built on top of an Image of type `I` and a functor of type `F`. In Milena, functors must model the Function concept or one of its refined concepts like Function_v2b, representing a function from a value (v) to a Boolean (b) (a unary predicate). A unary predicate `f` where the value is a site is indeed more general that a mask: its implementation may be based on a binary image `ima` (`f(p)` = `ima(p)`), a site set `s` (`f(p)` = `s.has(p)`) or based on any function `g` (`f(p)` = `g(p)`). The user code is very similar to the previous example, except that a functor is now used as a mask:

```
// An image.
I ima;
// Initialize 'ima' with data.
// ...

// Some predicate functor on sites
// implementing a mask.
F mask;
// ...

// Create a masked image using the 'image_if'
// morpher.
image_if<I, F> masked_ima(ima, mask);
// Apply the original 'fill' to it.
fill(masked_ima, 42);
```

In Milena, an instance of a morpher type is called a *morphed image*. Section 4.7 (p. 146) shows examples of morphers on images, including `image_if`.

The implementation of `image_if<I, F>` is not tied to a specific image type `I` or functor type `F`: it is a generic and reusable data type. Such a type is called a *generic morpher* and acts as a *type transformation*. `image_if` can indeed be seen as function from the Cartesian product of the set of image types and the set of unary predicates on sites, to the set of images types, transforming the domain of the input image according to the predicate:

$$\texttt{image\_if} : \left\{ \begin{array}{ccccc} \mathsf{Image} & \times & \mathsf{Function\_v2b} & \to & \mathsf{Image} \\ (\texttt{I} & , & \texttt{F}) & \mapsto & \texttt{image\_if<I, F>} \end{array} \right.$$

Following the example of `fill_within_mask`, `fill` can be used to replace the `fill_red_channel` routine, by passing as input an instance of a morpher (`component_image`) acting as a *view* on the initial RGB image, presenting only one of its channels (e.g. the red one) when reading data from and writing data to this morphed image[20]:

```
// Shortcut.
typedef image2d<int> I;
// An 2D initial image.
I ima1;
// Initialize 'ima' with data.
// ...

// Create a view showing only the component
// number 0 (red) of 'ima'.
component_image<I> ima_red_channel(ima, 0);
// Set the level of red to 255 in 'ima'
fill(ima_red_channel, 255);
```

20 Note that `component_image` is not an actual type of Milena. Instead, the library proposes a more general morpher to apply a function f after reading a value in the underlying image, and—if applicable—to apply its inverse f^{-1} before writing a value (see Section 4.7, p. 146)

The call `fill(ima_red_channel, 0)` on the last line has an impact on `ima` as well: as a morphed image based on `ima`, `ima_red_channel` does not hold any data *per se*. instead, `ima_red_channel` keeps a reference on the initial image. Each time a value is read in `ima_red_channel` (e.g. `v = ima_red_channel(p)`), the morphed image delegates the read operation to `ima` and select the red component (`ima(p).red()`). Similarly, every write operation (e.g. `ima_red_channel(p) = 127`) is translated as a write operation on the red component of `ima` (`ima(p).red() = 127`).

In conclusion, with the addition of morphers (e.g. `image_if`, `component_image`), a *single* generic implementation of an algorithm (`fill`) may be used to perform *different* operations (filling an image with a value, filling a subset of this image, filling only a component of this image, etc.).

Applications of Morphers

Morphers can be used to implement various services. We present here broad categories of morpher uses in this section. It is difficult to establish a precise classification of morpher types for two reasons. First, the boundaries between categories used to classify morphers types may be a bit blurry and may show some overlap. It may be for instance difficult to determine the relation between an adapter and a view: according to the definition and the context, one can be included in the other, they may overlap, or be distinct notions. Then, some morphers are often closely related to their application domain, and some category may not be transposed easily to another domain. Actual image-related examples are detailed in Section 4.7 (p. 146).

ADDING DATA OR OPERATIONS To extend a data type with new data (attributes) or operations (methods), one may adopt several strategies. Firstly, creating a whole new type offering the same interface as the initial data type, plus some new attributes and/or methods. We do not recommend this code duplication approach, as it is error-prone (when fixing a bug in duplicated code, it is not uncommon to forget one of the copies), cumbersome (modifying duplicated code requires multiple editions), and it does not scale (each new extension augments the size of the code). Moreover, the original and the extended types are unrelated and therefore incompatible.

A second strategy is to derive the new type from the initial type, using inheritance as a factoring means. No code is duplicated and types are compatibles, provided inclusion polymorphism (virtual member functions) is used (instances of the extended type can be assigned to variables of the initial type). This approach has nevertheless two drawbacks: first, inclusion polymorphism may induce run-time overhead (see Section 2.5, p. 62); then, the extension is tied to a unique type: the inheritance relationship cannot be changed to extend another type.

The third strategy is to implement the extension as a (generic) morpher. There is no code redundancy and the extension is not tied to a particular type. Instead, it forms a reusable piece of code, acting as a type transformation function (here, an extension).

In IP, useful extension morphers may be used to add non-standard operations to an image. For instance, Mathematical Morphology algorithms require that values of the image be organized as a lattice, defining for each pair a, b of values a supremum ($a \vee b$) and an infimum ($a \wedge b$). For value types that do not define a total order on their values, these two operations do not usually have an obvious definition. This is the case of the RGB color space, where there is no natural order between values. It is however possible to define a lattice for RGB values within an image [14]. The integration of such a lattice structure within an image type can be implemented as a morpher defining a supremum and an infimum operation on the image's values. One of the advantages of this solution is that more than one lattice can be implemented and "attached" to an image, as each of them are defined inside a separate morpher instance that can be used with any compatible image.

Another useful morpher is a type extending the domain of an image, either by providing actual data corresponding to sites outside the initial domain, or a function computing these values from the image. Extending the domain of an image allows users to control the behavior of an algorithm using a neighborhood or a window (see Section 4.4.5, p. 133) on the borders of an image.

adapting a type to another interface Some morphers act as *adapters* between a type and a client (most often, an algorithm), where the type does not match the interface expected by the client: the morpher exposes the required interface and delegates requests (with possible adjustments) to the underlying type.

The `slice_image` mentioned earlier is an adapter of a 3D regular image to the interface of a 2D regular image, where "regular" means that the image has a box as domain and that it is built on an orthonormal grid. In the following example, `slice_image` is used to adapt the interface of a 3D image of integers (`image3d<int>`) to the interface of a pretty-printing routine displaying 2D images on the terminal's standard output (`println`):

```
// The initial data type.
typedef image3d<int> I;
// An instance of this type.
image3d<int> ima3d;
// Initialize 'ima3d' with data.
// ...

// The slice number 2 of ‘ima3d’.
slice_image<I> slice(ima3d, 2);

// Print ‘slice’ on the standard output.
println(slice);
```

The `slice` object above adapts the interface of `ima3d` to the interface (a 2D image) expected by the client, `println`. The domain of `slice` is a `box2d`, and its size is deduced from the domain of `ima3d` (a `box3`). When `println` browses `slice`'s domain, it browses a set of `point2d`s that are translated by `slice_image::operator()` into corresponding `point3d`s in `ima3d`. This mechanism also works when writing data into the morpher: values are actually written into the underlying type. A call to `fill(slice, 0)` would set all values of the slice number 2 of `ima3d` to 0 (but not the values of the other slices).

Using a morpher as a "glue" between the type and its client has some advantages:

- By having access to actual (non-copied) data, morphers offer a light solution to the issue of adapting a type to another interface. There is no duplication of the initial data in the morpher (not any allocation of data of similar size): read and written values are instead computed on-the-fly. Therefore morphers save memory space, and as a consequence some computation time as well, as memory allocation and initialization have a cost in terms of execution time.

- Instead of writing a new *ad hoc* client (algorithm) able to process data of the initial type, or the converse (developing a new type compatible with the client), writing a morpher is usually a shorter and safer solution, minimizing code duplication.

- The morpher is a piece of reusable code. In addition to generic data structures and generic algorithms, a generic framework may provide a collection of generic morphers, notably usable as glue code.

In some use cases, an adapter provide the service of a view: an object presenting some data in another form. The `slice_image` morpher is an example of view: it shows a subset (a slice) of a 3D image as a 2D image. Data read from and written to the view are read from and written to the original image.

CHANGING THE BEHAVIOR OF A TYPE Another use of morphers is to change the behavior of the underlying type so that initial data can be processed differently. The example of the `image_if` morpher illustrates this application. For an image type `I` and a predicate `f` (having type `F`) on `I`'s sites, `image_if<I, F>` changes the behavior of the iterators: instead of traversing the whole domain `d` of `I`, they are now limited to the subset of `d` verifying the predicate `f`. This morpher is especially useful to limit the values processed by an algorithm to a region characterized by `f`.

Not that since it creates a restriction of an image's domain, the `image_if` morpher also qualifies as a view.

CREATING A NEW TYPE BASED ON ANOTHER ONE A morpher can also be used as a building block for creating a new type based on another one. Such a type remains a light object as it does not create nor copy data *per se*.

For instance, a `stack_image` morpher based on a 2D image type `I` may stack several instances of `I` "vertically" to create a 3D image, provided they have the same size. Each image, stored as a reference in an array of the morpher, corresponds to a slice of the 3D volume. In a sense, such a morpher is the inverse of the `slice_image` morpher seen previously. More generally, such a morpher may be used to create an $(n+1)$-dimensional image from a set of n-dimensional images.

Let us consider again a set of images of type `I` sharing the same size, having values of type `V`. One could imagine another morpher to arrange this set of images "horizontally', by creating an image having a type similar to `I`'s, but where the value type is an array of `V` values, instead of `V`. This morpher does not change the dimension of the image; it only affects `I`'s value type. For instance, one could implement an RGB color image by using this morpher to combine three gray-level images corresponding to the intensity level in each channel (red, green, blue).

Using a morpher to create a new type is usually different from using a morpher as an adapter, as the new type is not necessarily designed to provide an interface expected by some client (algorithm). Morpher-based type creation shares some similarities with morpher-based adaptation when it is used as a view. The `stack_image` morpher is an example of view, for instance when it is used to represent a set of 2D images (e.g. the slices acquired by a 3D imaging device or the frames of a sequence) as a 3D volume.

LAZY FUNCTION APPLICATION Finally, as a type transformation on a type `T`, a morpher can also be used to implement a function on the instances of `T`. There is however an important difference with a traditional function or even a functor: instead of representing the function itself, a morpher represent the yet-to-be-computed result of its application. The values of the actual result are not computed when the morpher object is created; instead, values are computed on-the-fly, in a lazy fashion.

Milena provides a morpher type illustrating this idea, named `fun_image<F, I>`. This morpher is built atop an image `ima` of type `I` having values of type `V`, and a functor `f` of type `F`, mapping values of type `V` to another type `W`. An instance of this morpher type represents the result of `f(ima)`. `fun_image<F, I>` presents the same interface as `I`, except for its value type, which is `W` instead of `V`. The function application occurs *only* when values are accessed in the morpher. If we note `m = f(ima)`, then for a given site `p`, `m(p)` triggers the computation of `f(ima(p))`.

Such a morpher could also be extended to support bijective functions: instead of taking a single functor `f` of type `F`, it may require a second functor `g` of type `G` such that `g` = $\texttt{f}^{-1}$. Writing data to `m` uses `g` prior to writing data into `ima`: `m(p) = 42` is translated as `ima(p) = g(42)`.

Implementing functions on data structures as morphers has again the interesting property of creating no data. This may be really efficient when several operations are applied successfully to an object. For instance, implementing the computation `f3(f2(f1(ima)))` as the successive application of three functions `f1`, `f2`, `f3` on image `ima` creates three images, requiring as many memory allocation. On the other hand, implementing these function applications using three morpher `fun_image<F, I>` objects creates no image data in itself. Only if the result of the whole computation is stored in another image (e.g., `out = f3(f2(f1(ima)))`, does the code allocate memory for a new image, `out`[21]. When `ima` is large, avoiding the allocation cost induced by the function-based approached may be a real gain in performance, or simply a necessity (if all the images were not to fit in memory).

Note that the previous illustration showed an example of morpher representing a lazy application of a function on the *values* of an image. It is also possible to apply a function on the *sites* (i.e. the domain) of an image as well, for instance to implement on-the-fly geometrical transformations.

Morphers and Design Patterns

Morphers can be seen as a generalization of some design patterns, with a static flavor. Design patterns are software engineering solutions to a recurring software design issue, often in the context of OOP. They have been introduced by Gamma et al. in their seminal book [62]. We study in this section the relationships between morphers and design patterns.

21 Note that even in this case, the allocation is not triggered by the use of the morphers, but by the construction of (or assignment to) `out`

A morpher may be used to implement design patterns such as Adapter (adapting the interface of a type to another one), Decorator (dynamically add behavior to an existing object), Proxy (creating an object acting as an interface for another object), and Observer (an object subscribed to another object, and receiving notifications from it).

Morphers can also be part of an implementation of the Strategy design pattern (used to define a family of algorithms as a set of interchangeable objects), as morphers can play a role to implement variants of an algorithm, by affecting the behavior of one or more of its element(s). Likewise, morphers can be used to implement a Template Method (defining an algorithmic skeleton, where the definition of some of the steps may be deferred and possibly overridden to implement various algorithms). Note that a more general approach to implement an "algorithm skeleton" is to implement it as a *canvas* of algorithms or pattern of algorithms. In both the case of Strategy and the case of Template Method, the action of the morpher(s) is "external", as they alter the input (and possibly the output) of an algorithm.

Finally, the architecture of primary data structures and morphers is similar in its organization to the Composite design pattern, which mixes in a tree structure leaf (standalone) objects and "composite" objects (aggregating other "child" objects). In our case, primary data structures are the leaves of a tree, while morphers are composites. Morpher hierarchies are usually simple and not deep: many use cases involve only a primary data structure (e.g. `image3d<int>`) as the (sole) child of a morpher (e.g. `slice_image<I>`, with `I = image3d<int>`).

Note that generic versions of existing design patterns have been proposed by Duret-Lutz et al. [64, 48]. Some use cases of morphers resemble some of these design patterns, namely Generic Bridge , Generic Template Method and Generic Decorator. In the case of the Generic Bridge and the Generic Template Method, this similarity includes the use of the CRTP (see Section 3.1, p. 76)

Constraints on Morphers

A morpher may place constraints on its underlying type, the same way a generic algorithm may express constraints on the type(s) of its argument(s). From a design point of view, these constraints should be expressed as concepts (see Section 2.4, p. 52). However, morpher-related concept checking faces the same issues as algorithm-related concept checking (see Section 2.4.2, p. 58). Fortunately, SCOOP techniques for enforcing compile-time verifications (see Section 3.3.4, p. 86) can be applied to morphers as well.

Static assertions are usually expressed at the beginning of an algorithm. In the case of a morpher, such assertions may be placed inside the constructor(s) of the class. For instance, Milena's `slice_image<I>` morpher creating a 2D view from the slice of a 3D image of type `I`, contains the following static precondition in its constructor to ensure that `I`'s domain is a 3D box:

```
mlc_equal(typename I::domain_t, box3d)::check();
```

Morphers and Properties

As it may modify the behavior of a data type, a morpher may also change its semantics. Properties of the new type must be adjusted accordingly. In many cases, a property of a morphed type has the same value as the corresponding property in the underlying type. For such a property, the morpher type does not need to give a definition of its own, but can instead *delegate* the definition of this property to the underlying type.

In practice, the number of delegated properties in a morpher depends on the nature of the morpher: the more the semantic and the interface of the morphed type differs from the original type, the less delegations there are. A morpher adding logging facilities to some methods of a type does not really change the nature of the underlying type, and most (if not all) of its properties will be delegated. On the other hand, a morpher type like `fun_image`, transforming the values of the initial image, has to provide fresh definitions for value- and data-related properties (see below). Other properties may however be delegated.

To shorten the definitions of morphers' properties, it is useful to provide for each concerned abstraction a base class containing default values for relevant properties, i.e. delegations to the corresponding property values in the underlying type. In Milena, the traits class `trait::default_image_morpher` plays this role for Image morpher types, the same way `trait::default_image_` defines default properties for primary Image data types (see Section 3.3.5, p. 89).

`default_image_morpher<D, T, I>` provides values for the image properties (see Table 4, p. 90) of the image morpher type `I` built atop the image type `D` (the delegation) having values of type `T`, by using delegations to `D`'s properties:

```
namespace trait
{
  template <typename D, typename T, typename I>
  struct default_image_morpher : default_image_<T, I>
  {
    // Miscellaneous properties: delegations (except
    // for ‘category’).
    typedef mln_internal_trait_image_speed_from(D)
                                                  speed;
    typedef typename image_<D>::size              size;

    // Values properties: delegations.
    typedef mln_internal_trait_image_vw_io_from(D)
                                vw_io;
    typedef typename image_<D>::vw_set vw_set;
    typedef typename image_<D>::value_access
                                value_access;
    typedef typename image_<D>::value_storage
                                value_storage;
    typedef typename image_<D>::value_alignment
                                value_alignment;

    // Domain and geometry properties: delegations.
    typedef mln_internal_trait_image_pw_io_from(D)
                                   pw_io;
    typedef typename image_<D>::localization
                                   localization;
    typedef typename image_<D>::dimension dimension;

    // Extended domain properties: delegations.
    typedef typename image_<D>::ext_domain ext_domain;
    typedef typename image_<D>::ext_value  ext_value;
    typedef typename image_<D>::ext_io     ext_io;

    // Data properties: delegations.
    typedef typename image_<D>::nature nature;
    typedef typename image_<D>::kind   kind;
    typedef typename image_<D>::quant  quant;
  };
}
```

The only exception is the `category` property, reflecting the very nature of the morpher type, for which a default value cannot be decently provided: it is the responsibility of the morpher's set of properties to define the `category` property.

Most delegations are direct: they are defined as aliases of the property of the same name within the underlying type. E.g., for a property *p*:

```
typedef typename image_<D>::p p;
```

Some properties are implemented as indirect delegations: this happens when the definition of the delegation depends on a feature of the initial type (`D`). In the previous example, the `speed` property is computed from `D`'s `speed` using the `mln_internal_trait_image_speed_from` macro:

```
#define mln_internal_trait_image_speed_from(I)       \
  mlc_if( mlc_equal( mln_trait_image_speed(I),       \
                     trait::image::speed::fastest ), \
          trait::image::speed::fast,                 \
          mln_trait_image_speed(I) )
```

`mln_trait_image_speed` is a shortcut macro to fetch the `speed` property of an image type defined as this:

```
#define mln_trait_image_speed(I) \
  typename trait::image_< I >::speed
```

The mln_internal_trait_image_speed_from macro prevents an image morpher type from having speed::fastest as default value for the speed property (if so, the speed is changed to speed::fast by default). This is because most morphers introduce extra computations and/or change the organization of the image data, therefore suppressing the speed::fastest behavior of their underlying type.[22]

Likewise, the vw_io and pw_io properties related to input/output accesses (see Table 4, p. 90) cannot be given immediate default values in default_image_morpher<D, T, I>. Their definitions must take into account whether the initial image type is constant or mutable (resp.) by checking whether the type passed as effective parameter for D has a const qualifier in its definition, to define by default the input/output value- and point-wise accesses as read-only or read/write (resp.) This is the role of the mln_internal_trait_image_pw_io_from and mln_internal_trait_image_vw_io_from macros, defined as this:

```
#define mln_internal_trait_image_vw_io_from(I) \
  mlc_if( mlc_is_const(I),                     \
          vw_io::read,                         \
          mln_trait_image_vw_io(I) )

#define mln_internal_trait_image_pw_io_from(I) \
  mlc_if( mlc_is_const(I),                     \
          pw_io::read,                         \
          mln_trait_image_pw_io(I) )
```

The mlc_is_const macro is a shortcut for a metaprogramming construct determining whether the type passed as argument has a const qualifier or not. It it used as a condition of the mlc_if macro to choose respectively for the vw_io and pw_io properties between the read-only policy (vw_io::read, pw_io::read) and the initial property value, obtained with mln_trait_image_vw_io and mln_trait_image_pw_io. These two macros are shortcuts to get the vw_io and vw_io properties of an image type:

```
#define mln_trait_image_pw_io(I) \
  typename mln::trait::image_< I >::pw_io
#define mln_trait_image_vw_io(I) \
  typename mln::trait::image_< I >::vw_io
```

The definition of default_image_morpher<D, T, I> above is very effective to factor the definitions of image morpher types properties. The following lines show for example how Milena defines the set of properties image_< fun_image<F, I> >, related to the morpher type fun_image<F, I> representing the application of a function of type F to an image of type I (mentioned in Section 3.3.6, p. 95):

```
namespace trait
```

22 This is just a default value; if the morpher does not degrade the performance of the initial image type, it may define speed as speed::fastest.

```
{
  template <typename F, typename I>
  struct image_< fun_image<F, I> >
    : default_image_morpher< I,
                             mln_result(F),
                             fun_image<F, I> >
  {
    // Miscellaneous properties.
    typedef image::category::value_morpher  category;

    // Values properties.
    typedef image::value_access::computed
                                          value_access;
    typedef image::value_storage::disrupted
                                          value_storage;

    // Data properties.
  private:
    typedef
      mlc_equal(mln_trait_value_quant(mln_result(F)),
                value::quant::high)     is_high_quant_;

  public:
    typedef mln_trait_value_kind(mln_result(F)) kind;
    typedef mln_trait_value_nature(mln_result(F))
                                                  nature;
    typedef mlc_if(is_high_quant_,
                   image::quant::high,
                   image::quant::low)           quant;
  };
}
```

The previous listing shows that `image_< fun_image<F, I> >` gets some of its definitions of properties from `default_image_morpher < I, mln_result(F), fun_image<F, I> >` while it provide its own for the others. The `mln_result(F)` macro retrieves the result type of the functor type `F`, e.g., if `F` represents a function `float` $\rightarrow$ `int`, then `mln_result(F)` = `int`. It is actually defined as an alias for `F`'s associated type named `result`, which is part of the signature of every model of the Function concept:

```
#define mln_result(T) typename T::result
```

Thus `default_image_morpher` is made aware of the new value type introduced by the morpher.

Properties `category`, `value_access`, `value_storage` are given values by `image_< fun_image<F, I> >`, while data-related properties (`kind`, `nature` and `quant`) are defined as in `default_image_` (see Section 3.3.5, p. 89), but this time using `mln_result(F)` as value type, instead of the underlying image's value type.

4 GENERICITY IN IMAGE PROCESSING

We propose in this chapter an organization of generic software components for Image Processing (IP), ordered by their categories and properties. We show this design is well suited for the creation of reusable software. We then study the impact of genericity on run-time performances, and show how to retain efficiency in a generic fashion (without resorting to specific or non reusable solutions). Finally, we present generic object transformations or morphers*, applied to IP and used to create new data types from existing ones and to alter the behavior of algorithms externally.*

We believe that one of the goals of an IP framework is to provide its IP practitioners with a collection of useful, reusable and efficient algorithms. This idea was already proposed by Stepanov [9]:

> "I mentioned before the dream of programmers having standard repositories of abstract components with interfaces that are well understood and that conform to common paradigms."

This principle is developed in the context of IP throughout this chapter.

4.1 MOTIVATION FOR GENERIC IMAGE PROCESSING SOFTWARE

In IP as in many scientific computing domain, software can often be decomposed in three parts:

DATA STRUCTURES The different kinds of containers offered to represent data sets processed by the framework. In IP, such data structures are mainly images. But an IP framework may also provide—depending on its complexity—graphs (see Section 4.5.2, p. 135), cell complexes (see Section 4.5.3, p. 135), topological maps, etc. More generally, scientific software data structures encompass mathematical notions such as vectors or matrices and computer science entities such as automata, decision diagrams, etc.

VALUES The different types of elements that can be stored in data structures. Such elements can be scalar types, such as Boolean, integer, floating-point or complex value types, but also more complex values such as tuples, vectors, matrices, or points (especially in IP). As a matter of fact, it is also possible to use a data structure as the value of another data structure; for instance, a 3D image acquired using Diffusion Tensor Imaging (DTI)—a Magnetic Resonance Imaging (MRI) technique—may have tensor values, where each voxel contains a symmetric positive definite 3×3 matrix describing the 3-dimensional shape of the diffusion.

ALGORITHMS Non-elementary operations to be applied to data sets. An algorithm may not necessarily be tied to a specific data structure in theory and may therefore be compatible with various input types. In practice, this versatility requires multiple (specific) implementations (one per data type) or a generic implementation (as seen in Section 2.1.1, p. 31).

One of the motivations behind generic IP software is to avoid code redundancy (in particular, several implementations of an algorithm for various input types) while maintaining run-time efficiency. Object-Oriented Programming (OOP) might be seen as a potential solution at first to factor the code of similar algorithms. Instead of writing several specific algorithms for the various compatible input types (e.g. `binary_image`, `gray_level_image`, etc.), one may write a single implementation accepting inputs of an abstract data type (e.g. an abstract class `image`). This algorithm would therefore accept as input any instance of a class deriving from this abstract data type (e.g., all image classes inheriting from `image`). However, OOP does not deliver the performances expected in a scientific numerical context, because of its run-time mechanisms, and thus cannot be used in practice. GP, on the other hand, does not suffer from similar issues.

Let S, V and A be respectively the number of theoretic data structures, value types and algorithms provided by a scientific framework. If this framework is not generic, then there would be at most $R = A \times S \times V$[1] different routines to accommodate all the variants [66]. Likewise, the total number of actual data structures is the combination of every type of data structure with every value type: $D = S \times V$. So in the non-generic case, the number of entities (the total number of algorithms, data structures and value types implementations) to maintain is $E = A \times S \times V + S \times V + V = ((A+1) \times S + 1) \times V$. The numbers R, S, and E are prone to a combinatorial explosion in a non-generic context. If the framework is generic however, the number of (generic) routines R equals A, as generic algorithms are orthogonal to data structures and value types: R does not grow with S nor V. Neither does $D = S$ grow with V, as data structures are themselves orthogonal

1 For simplicity's sake, we make the hypothesis that all algorithms are compatible with all data structures and that the latter are themselves compatible with all value types.

to their value types. The total number of entities to maintain is consequently $E = A + S + V$.

In addition, the framework may support transformed types, or morphers (as seen in Section 3.3.6, p. 95), thus increasing the number of entities to maintain [65]. Let M be the number of these type transformations. If we take into account the case that corresponds to a single application of a morpher (i.e. if morphers were only applicable to primary data structures only, not to morphed types), the number of routines to implement in a non generic framework is:

$$\begin{aligned} R &= \underbrace{A \times S \times V}_{\text{Non-morphed case}} + \underbrace{A \times S \times V \times M}_{\text{Uses of single morpher}} \\ &= A \times S \times V \times (1 + M) \end{aligned}$$

But morphers can also be applied to types that have already been transformed, which increases the number of potential combinations. If we limit ourselves to n applications of morphers, the number of algorithm variants rises again: $R = A \times S \times V \times (1 + M)^n$, which tends towards ∞ when n tends towards ∞, thus adding to the combinatorial explosion problem.

There is no similar issue in the case of the generic approach. The number of (generic) routines is the same ($R = A$), and the total number of elements is just increased by M, as morphers are orthogonal to data structures, and the total number of elements to maintain is $E = A + S + V + M$.

In conclusion, the generic approach scales with the extension of the framework in whatever dimension (A, S, V, M). For this reason, and because it does not introduce run-time penalty *per se*, a generic strategy is suitable for scientific software and in particular for IP software.

4.2 DESIGN CHOICES

The first section of this chapter exposed the motivations to create IP software in a generic fashion. The present section presents additional design choices for a generic scientific framework, that we believe are especially relevant in the context of IP, both from a technical and a practical point of view.

4.2.1 *The Choice of SCOOP*

The generic approach is indeed well suited to design and implement reusable and efficient IP software. However, the SCOOP paradigm, which can be seen as an advanced GP paradigm with some OOP flavor, seems much more appropriate to implement a generic IP framework. The incentive to use SCOOP has been presented in the previous chapter, but in the context of IP, this choice is especially relevant for the following reasons.

COMPILE-TIME CHECKING Thanks to both static (metaprogramming) assertions and SCOOP' concept- (see Section 3.3.4, p. 86) and property-based checks (see Section 3.3.5, p. 89), many errors can be detected at compile-time. Though C++ compiler messages may be long and difficult to read (see Section 2.4.2, p. 58), compile-time errors are nevertheless easier to understand and fix than most run-time errors (for instance, an invalid memory access). Moreover, as such errors can be detected earlier (before users run any code), some of them may be detected by authors of the code, instead of users.

NO RUN-TIME OVERHEAD SCOOP is a compile-time paradigm: it does not rely on dynamic behavior (such as polymorphic methods) and does not introduces run-time costs *per se*.

CONCEPT-BASED POLYMORPHISM As SCOOP is able to express concepts, it can be used to create overloaded routines that can be distinguished by the concept of their inputs. This is especially useful in the case of binary operators overloaded with respect to the concepts of their operands. For instance, there may be two generic implementations of `operator==`, one for Images and another for Values:

```
template <typename I1, typename I2>
bool operator==(const Image<I1>& ima1,
                const Image<I2>& ima2);

template <typename V1, typename V2>
bool operator==(const Value<V1>& val1,
                const Value<V2>& val2);
```

Thanks to the use of concepts in the signatures of these two overloaded operators, the addition of two Images (resp. two Values) is not ambiguous and calls the former (resp. the latter) routine:

```
image2d<int> ima1 = ... // Initialization.
image2d<int> ima2 = ... // Initialization.
if (ima1 == ima2) // Equality on ‘Images’.
  { /* ... */ }

// ‘int_u8’ and ‘int_u16’ (8- and 16-bit
// unsigned integer types) are models of
// the ‘Value’ concept.
int_u8  i = 42;
int_u16 j = 51;
```

```
if (i == j) // Equality on 'Values'.
  { /* ... */ }
```

Concept-based overloading improves the expressive power of the language. Indeed, overloading (and template specialization) are somehow limited, as they enable the definition of an alternative routines with respect to a given (tuple of) *type(s)*. This is especially frustrating in the context of templates, where explicit specialization contrasts with the generality and extensibility of the template mechanism. With concept-based overloading, one may defined variants with respect to a (tuple of) *concept(s)*, which is much more general.

PROPERTIES AND PROPERTY-BASED POLYMORPHISM SCOOP's properties enable the characterization of a type (possibly a generic one) with respect to its capabilities. They can be used to create and automatically select a more efficient variant of an algorithm—if its input supports it. Such variants are called generic optimizations (see Section 4.6.3, p. 139): they help improve performances without writing specific, non-reusable code. For instance many IP algorithms can be rewritten to take advantage of images having their values stored in a single linear contiguous memory block (buffer); such values can be manipulated with pointers, which are faster than iterators.

MORPHERS Another important feature brought by SCOOP is type transformations or morphers (see Section 3.3.6, p. 95). Some IP-related morphers are detailed in Section 4.7 (p. 146). Morphers can be useful to implement e.g.,

- Lazy function application to images, e.g. thresholding, color conversion, etc..
- Domain restrictions—for example using masks—and domain extensions—e.g. wrapping a 2D image around one (resp. two) of its dimension(s) to make it have the topology of a cylinder (resp. a torus).
- Image composition—e.g. stacking a set of 2D images to create a 3D volume—and image decomposition—e.g. viewing the slice of a 3D image as 2D image.

Morphers improve the expressive power of the framework and are efficient (they do not create intermediate temporary values that may introduce memory and/or run-time costs).

4.2.2 *The Choice of C++*

The SCOOP paradigm has been initially designed and implemented in the C++ programming language. But as said previously in Section 3.3 (p. 79), SCOOP is not tied to C++ specifically. We consider that this paradigm can be for instance transposed in the D programming language, which supports the programming traits required by SCOOP (generic programming, inheritance, compile-time metaprogramming).

C++ has many drawbacks. Its syntax is complex and sometimes unfriendly; C++ compilers often emit long and hard to understand error messages; the language features dangerous low-level features (mostly inherited from C) such as pointers, casts and unions, that may make debugging a program a long and painful work; finally the language does not feature automatic memory management (through garbage collection), which adds to the work of the developer—although libraries may simplify this task.

Yet C++ seems to be the best option to implement a SCOOP-based IP software framework, and what is more, for reasons which are mostly not related to SCOOP. The following list tries to summarize why C++ appears to be a good compromise for generic IP software.

STANDARDIZED LANGUAGE The C++ language is defined by an ISO/IEC international open standard [87]. A standardized language benefits from the work of hundreds of people coming from different places, in order to create a robust, long-lasting, precisely-defined, and useful tool. As the language does not belong to a single entity, it is more likely to be durable, and it is less prone to incompatible evolutions. C++ is developed in an open fashion, thus all the work of the committee on future versions of the standard is freely available on the Web.

WIDESPREAD LANGUAGE Nowadays, standard-compliant C++ compilers are available on virtually any computing platform. This fact guarantees that almost everybody is able to compile a C++ program in its programming environment. This statement is not only valid today, but also for the future, as C++ is still actively maintained and developed by its ISO standards committee, as are compilers and tools by industrial vendors and open source communities. This is especially relevant in industry and research contexts, where the durability of a technology is fundamental.

WELL KNOWN LANGUAGE Like C [86] and Java [67], C++ is a language taught in a lot a computer science curricula, and in particular in IP courses. Stroustrup estimates the number of C++ users to be more than 3 million in 2004 [132]. Therefore the probability for a new user of an IP framework to know C++ is high (compared to another language). Using a less-known language would probably divide the audience of the framework by a factor of ten, a hundred or even more.

WELL SUPPORTED LANGUAGE C++ benefits from many great tools. There are many compilers, debuggers, libraries, GUI toolkits and Integrated Development Environments (IDEs) for the language on most platforms, both as proprietary and free software. Most of these tools have existed for many years and are robust, reliable, and efficient.

EFFICIENCY Due to its mix between low-level and high-level capabilities, C++ is deemed the fastest language, or one of the fastest languages—or more accurately, producing the fastest or among the fastest programs [4]. Together with genericity, we consider efficiency a fundamental requirement for IP software, since data may be large and/or may need to be processed fast.

VERSATILITY C++ features several programming paradigms, including procedural programming, data abstraction (or encapsulation), Object-Oriented Programming (OOP), Generic Programming (GP), and to some extent, Functional Programming (FP) (in particular with the latest C++ standard [87]). One of the strengths of the language is not to impose a specific programming style: users may choose to use a single programming paradigm or use a mix of several styles. For that matter SCOOP is a programming paradigm taking advantage of both OOP and GP[2]. This freedom adds to the expressiveness of the language. In addition, C++ is one of the language best supporting GP (as for expressiveness and efficiency) [9, 63]; this is especially relevant in our case, as our approach strongly relies on GP. Finally, some C++ features such as overloading and user-defined operators provide useful *syntactic sugar* (syntactic replacement for longer and/or more complex equivalent language constructs). Such shortcuts are helpful to write concise and elegant code.

CONTROL AND EASE OF USE The language lets users control some low-level aspects such as memory management (as in the C language), which is fundamental in the context of efficient scientific software. C++, however, provides more user-friendly means to take advantage of this opportunity, thanks to the programming traits mentioned previously. In the particular case of memory management, objects'

2 We can also mention that Milena makes use of FP.

constructors and destructors help automate the allocation and deallocation of memory (but also the initialization and release of resources such as files or network sockets). This is a good trade-off between handling these elements manually (as in C), which is error-prone and implies a lot of work; and letting the run-time environment take care of it (as in Java, which uses garbage collection) at the expense of some run-time performance—though it is simpler, garbage collection is not yet as efficient as deterministic hand-made memory management.

4.3 ORGANIZATION

Following the motivations for a generic IP software framework exposed in Section 4.1 (p. 113) and the design choices considered relevant for the implementation of such a framework (see Section 4.2), we describe in the present section a proposal for a software infrastructure for generic IP. This proposal tries to handle the three axes of variety encountered in the field of IP: the diversity of data (image) types (see Section 1.1.1, p. 16), the diversity of users (see Section 1.1.2, p. 19) and the diversity of use cases (see Section 1.1.3, p. 20).

4.3.1 *General Architecture*

First of all, handling the many image types required by IP practitioners requires not only requires a framework providing data structures implementing these images types, but also an environment flexible enough to support the addition and the integration of new data types easily. GP is useful here in two ways: firstly, to reduce as much as possible the number of data structures to maintain, by using parameterized types as a factoring means. Immediate illustrations include image types taking their value type as a parameter; for instance, Milena's `image2d<T>`, but also more complex data types, like the ones based on morphers (see Section 3.3.6, p. 95). GP is also of prime importance regarding algorithms: these should be implemented in a generic manner so that the framework scales with the addition of new image types.

Secondly, the diversity of users is visible in the various degrees of knowledge with respect to image processing techniques, the types of images manipulated, software tools, and in many cases, programming; but also in the nature and complexity of the task to perform. Depending on the user and their work, the most fitted operation mode may be an interactive session where IP programs are built with graphical widgets (boxes, buttons, links, etc.) and data are visually represented and updated at each change; or a scripting interface, where the user inputs commands in an interpreter; or an IDE, where the user writes and assembles algorithms to be compiled and run later. To each of these operation modes corresponds a User Interface (UI), which is the set of elements a user manipulates to interact with the framework (a Graphical User Interface (GUI) with visualization capabilities, an interactive-loop interpreter, the Application Program Interface (API) of a library, etc.). We have seen in Section 1.1.4 (p. 22) that most solutions rarely address all the needs of this wide spectrum in a uniform manner. Usually, a tool is designed to properly offer one or two UIs, and it is sometimes extended to cover some other interfaces, sometimes in a degraded form. For instance many image processing libraries initially target a compiled language such as C++ for performance reasons, but also provide in interface for an interpreted language such as Python. While the former requires some non-trivial programming knowledge, the latter can often be used in an interactive mode and can be learned easily and rapidly, so that newcomers may quickly write prototypes and conduct experiments. However, this "second mode" may not offer all the features of the primary (compiled) one (e.g., all data structures or algorithm may not be available) or be as efficient (if the interface introduces a run-time overhead). A reusable framework should therefore provide as many UIs as possible, and be open enough to support future additional UIs.

The third and final axis of diversity is related to the type of work a user wants to perform with their tool. For example the IP task they have to deal with may be purely illustrative, e.g. in pedagogical environments; or to produce figures for a scientific paper; a simple experiment, requiring a few manipulations or a few lines of code); a slightly bigger prototype, needing more lines of code; or a full application, which may require more work to process very large data, be generalized to many data types, or produce the optimized implementations expected in an industrial context. As in the previous item, each of these tasks is achieved efficiently with the right tool. The process of writing, compiling, debugging and running a program is not recommended when one simply want to run a single algorithm to demonstrate it during a lecture. Conversely, an interpreted script is not the most efficient solution to processing a large base of data as fast as possible. Therefore a framework should provide several UIs to also encompass as many *use cases* as possible. Moreover, it might be useful to be able to access a given feature, e.g., a segmentation tool, by different means over time. For instance a segmentation algorithm may be used initially through a GUI so as to test it on a few images, adjust its parameters and evaluate its results. Then, if it is deemed suitable to process a whole data set, it may be later be run from a Command Line Interface (CLI), as a small program run from a terminal. Likewise, more complex processing chains may be developed from a convenient prototyping environment, and later be turned into more efficient, compiled programs.

To address theses three degrees of diversity (data types, users, use cases), we propose a software architecture based on a generic approach, which is well suited to handle the variety of applicable IP operations (see Section 4.1, p. 113). This architecture is based on the following elements:

1. A core made of a generic and efficient C++ library, containing generic data structures and algorithms for image processing, implemented using the SCOOP paradigm (see Chapter 3).

2. "Satellite" components based on this library, such as bindings to dynamic languages such as Python or Ruby, CLIs, GUIs, Web services, etc. These components are to provide various UIs for the various categories of users and the different kinds of tasks they want to perform. They should be kept as light as possible, and let the core library (item 1) do as much work as possible, both to avoid redundancy and concentrate the optimization efforts in this core library.

3. Some "glue code" between items 1 and 2, which consist in automated operations to keep item 2 minimal while leaving item 1 untouched and still benefiting from its qualities (efficiency, reusability).

The benefit of this architecture, first sketched by Duret-Lutz [47], is that most of the difficult work (in particular C++ template programming and metaprogramming) is done in the core library, which is generic, efficient and reusable. The glue code tries to keep as much as possible of the genericity, efficiency and reusability of the core library. Moreover, as time goes, development made with high-level UIs can be refactored and integrated into lower-level tools, so as to improve their genericity, efficiency and more generally their reusability.

This architecture is implemented in the Olena project, a Free Software generic IP platform. The first item is the Milena library mentioned previously, and is detailed in the remaining of this chapter. Items 2 and 3 are sketched in Section 6.3.1 (p. 174).

4.3.2 *Core Library*

According to Stepanov,

> "The fundamental idea of generic programming is to classify abstract software components and their behavior and come up with a standard taxonomy." [9]

Following this principle, we propose to build the core library of our IP platform on a set of orthogonal concepts corresponding to entities of the domain (Image Processing). By orthogonal concepts, we mean concepts with as few dependencies as possible one on another, so as to achieve separation of concerns in the library: the junction between two collaborating components in the library (e.g. an image and a structuring element involved in a morphological dilation) should be minimal, so that one may be changed with no or little impact on the other. For instance, the code of the image type should not be affected when substituting a structuring element based on a set of points for a structuring element describing a (centered) disc solely by its radius.

An essential task of the design of the library is therefore to define concepts (in the sense of GP) representing essential notions such as an image, a point, a neighborhood, a structuring element, etc. In addition to defining the contents of theses concepts (as presented in Section 2.4.1, p. 54), their interactions should be carefully designed, as any change regarding a concept or the connection of two concepts may also have a deep impact on the contents of the library; it may not only affect the concept classes themselves (see Section 3.3.1, p. 80), but also their models (see Section 3.3.2, p. 81) and the algorithms that depend on these concepts (see Section 3.3.3, p. 83). The image-related concepts of the core library are presented in Section 4.4.

Burrus et al. prompt designers of scientific libraries to implement data structures and algorithms as uncorrelated entities [32], a style advocated by the SCOOP paradigm (see Section 3.3.3), and previously the Standard Template Library (STL) [127, 115]. Therefore algorithms are implemented in a procedural fashion, instead of being written as member functions (methods) of classes.

We should also take into account the size of the various populations of users outlined in Section 1.1.2 (p. 19), so as to make the library as usable as possible for everybody. The majority of the public of the framework is expected to be *end users*, that should not be required to have a deep knowledge of OOP, GP, SCOOP or even C++. The task of simply using algorithms from the library and assembling them to create (non-generic) processing chains should remain easy and accessible to a programmer with a basic knowledge of the C programming language. Thanks to the procedural aspect of the library, running algorithm amounts to calling functions, which makes Milena very similar to a classical C library at this level of use, if we except type names carrying parameters (such as `image2d<int>`). Such types can be "hidden" by using `typedef` declarations at the beginning of the code, e.g.

```
typedef image2d<int> I;
```

Other users of the library, who extend it by writing new algorithms or data structures, are expected to be knowledgeable about C++ and the design of the library. The difficulty rises gradually from writing non-generic algorithms, to writing generic algorithms, to writing new (possibly generic) data types and morphers. Writing non-generic algorithms requires some knowledge on the manipulation of data structures (images, site sets, windows, etc.) and iterators, for which an understanding of some OOP principles should suffice. Writing generic algorithms is more difficult, as it involves concepts and other GP- and SCOOP-related notions (associated types, concept checking, conversions to exact type, etc.). Lastly, writing data structures (e.g. a new image type) is an advanced task, as it may require the creation of several collaborating classes (e.g., a new site set, new iterator types, etc.) modeling the corresponding concepts.

4.4 CONCEPTS

This section presents the essential concepts of the core library. Milena contains more than 40 concept, but we only present here the most important ones, starting with the central Image concept [95]. Actual types modeling these concepts are later presented in Section 4.5 (p. 133).

4.4.1 *Images*

We start the presentation of the library's concepts with the Image concept. We have already mentioned this concept in the description of SCOOP's concepts in Section 3.3.1 (p. 80), and its signature has been given as an illustration in Table 2 (p. 82). The Image concept is especially important in Milena, since it appears in all IP-related routines of the library, as a constraint on the input type(s) of an algorithm, as shown in Section 3.3.3 (p. 83).

Let us consider a more detailed explanation of the Image concept in this section, in the context of IP rather than software engineering and GP. The goal of Image is to encapsulate the characteristics of all image types of the library, with as few details as possible, to make it flexible enough to represent all conceivable image types, while keeping enough practical aspects to make it actually useful.

In order to build the Image concept in a top-down fashion, we propose the following definition of an image [94, 117].

Definition. *An image I is a function from a domain $\mathcal{D}$ to a set of values $\mathcal{V}$. The elements of $\mathcal{D}$ are called the* sites *of I, while the elements of $\mathcal{V}$ are its* values.

For example in the case of a binary 2D regular image, the domain $\mathcal{D}$ is a 2D discrete rectangular space aligned with the axes of the image (a 2D "box") and the value $\mathcal{V}$ is set of Boolean values, represented by the `bool` type in C++. Elements of the domain are 2D points within the box, while the value set is $\{$`true`, `false`$\}$.

For the sake of generality, we use the term *site* instead of *point*. Our framework shall be able to manipulate many kinds of images, for which the term "point" would not be appropriate. Consider for instance an initial 2D image on which a segmentation method has been run; from the set of obtain regions, we can create a Region Adjacency Graph (RAG) where each vertex corresponds to a region, and for each pair of neighboring regions (sharing a boundary), an edge is created between the corresponding vertices. If we consider the set of vertices of the graph, and associate to each of them a value of a given set, like the size of the region corresponding to a vertex, we can also consider this "decorated" graph as an image in the sense of the previous definition, associating data to each element of its domain (its set of vertices)[3]. While it is perfectly normal to refer to the elements of the initial image's domain as points (since they represent members of the $\mathbb{Z}^2$ space) it would be incorrect to consider vertices (or edges) of the RAG-based image as points. Therefore, we prefer the term "site" since it covers a more general notion.

Domain and Value Set

In order to materialize the existence of a domain and a value set in each type of image, we build in the Image image two associated types (see Section 3.3.1, p. 80) for $\mathcal{D}$ and $\mathcal{V}$: the associated types `domain_t` and `vset` (see Table 2 (p. 82)). Therefore, for each image type `I` (model of Image), `I::domain_t` and `I::vset` are to represent its domain and value set respectively. However `I::domain_t` and `I::vset` are only *types*, i.e. static descriptions of the domain and value set objects associated with an instance `ima` of `I`. To access these objects, the concept requires accessors in its signature, namely the `domain()` and `values()` methods. The following statement fetches the domain of `ima` and stores it in a value of the appropriate type:

```
const I::domain_t& d = ima.domain();
```

Note that if the previous line were to be used in a template context (i.e., within a generic routine or a parameterized container), the type of `d` should be preceded by the `typename` keyword in order to comply with C++ syntactic rules, leading to long declarations lines:

```
const typename I::domain_t& d = ima.domain();
```

To work around the syntactic verbosity of C++ regarding the use of associated types, we introduce shortcut macros, the same way we did in Section 3.3.6 (p. 98) to simplify the use of iterators. Such macros have the same structure: they are name `mln_`*`type`*,

3 Note that we may as well use the set of edges of the RAG as a domain to build an image in the same way.

where *type* or *type*_t is the associated type corresponding to the type passed as argument to the macro. In the current case, the mln_domain macro would be written as this:

```
#define mln_domain(T) typename T::domain_t
```

For symmetric reasons, we provide versions of such macros without the typename keyword, the name of which contains a trailing underscore character:

```
#define mln_domain_(T) T::domain_t
```

Sites, Point Sites and Values

The Image concept also contains some more associated types regarding sites and values:

- site is the type of an element of the domain. It usually represent an actual location and is meant to convey geometrical information, which is not always the case of the psite associated type (see below). Regarding the original 2D image mentioned earlier, the site type would be a 2D point (implemented by the point2d class in Milena). But in the case of the RAG-based image built from a segmentation of the previous 2D image, site could be a much more complex type. For example, each site object could be a comprehensive geometrical description of the corresponding region, with the list of all its pixels. There is indeed no actual limit to what a site can be (see Section 4.4.2), which makes it a powerful concept of the library as it can be arbitrarily complex and take part in the construction of very sophisticated image types. The drawback of such freedom is that the site associated type cannot be reliably used to access the values of an image efficiently. In the case of the RAG-based image, obtaining the value associated to a region by the means of a site encoding its geometry would necessarily imply some non-trivial computations at each access, making this image type unusable in contexts where efficiency is required. To prevent slow accesses to images, the Image concept introduces another associated type, called psite.

- The psite (short for *point site*) associated type is merely a simple site descriptor (or "site handle"), that shall only contain the minimal information to access the values of an image in an reasonably efficient manner. In the previous case of the RAG-based image, a typical psite object would only contain an identifier of one of the vertices of the graph (e.g., a number and possibly a reference to the graph structure to prevent this point site from being used with a totally different graph-based image). Note that the `psite` and the `site` associated to an image type may refer to the same type if the `psite` suffices to describe the location and geometrical information of a site. This is the case of regular 2D images, where a 2D point (an instance of `point2d`) is enough to represent the location and geometry of an element of a discrete 2D box. Hence `mln_site(image2d<int>)` = `mln_psite(image2d<int>)` = `point2d`. The Image concept contains methods to access values of the image using `psite` values (see below), but there is no such mechanism for `site`s (when `site` $\neq$ `psite`).

- `value` is the type of an element of the value set. However, this type is not used in the signatures of image methods returning values from psite presented hereafter. Instead, the Image concept has two extra associated types corresponding to value accesses performed in read-only and read/write modes, respectively `rvalue` and `lvalue`. In many (but not all) cases, `lvalue` is equal to `value&` and `rvalue` is defined as `const value&` (references to the location of the value in memory). However, some images types provide very different types for these three associated types. In particular, when the actual values are manipulated indirectly, proxy objects (having a type different from `value`) may be built and returned by the image for `lvalue` and `rvalue`.

To access the values of an image, the `Image` concept requires the existence of two methods named `operator()`, for read-only and read/write accesses. Both take `psite`s as argument and return either an `rvalue` or an `lvalue` (see Table 2, p. 82). The definition of a method `operator()` “overloads” the definition of the operator ‘`()`’ for the considered image type, and make it look like a function, or more precisely, a `functor` (see p. 44), as in the following example.

```
// A 3x3 binary, 2D image.
image2d<bool> ima (3, 3);
// A 2d point at row 1 and column 1,
// that can serve as a psite of ‘ima’.
point2d p(1,1);
// Write value ‘3’ into ‘ima’ at psite ‘p’,
// using its ‘operator()’.
ima(p) = 3;
```

All the members of the domain of an image are valid arguments to the image’s `operator()`. But in some cases, one may want to access a location that might not be part of the domain. Accessing the neighbor of a point site of the image is a typical example of non-guaranteed operation. To ensure that such a statement is valid, the Image concept requires its models to provide a `has()`

method checking whether is argument belongs to the image's domain. More generally, the `is_valid()` method checks if the image is globally usable, and answers negatively e.g. if the image has not be initialized.

Iterators

The last part of the Image concept is related to iterators. We only show aspects of iterators related to images in this section; Section 4.4.4 (p. 131) contains more details on the iterator concepts.

As in the STL and many other C++ libraries, the traversal of containers (here, images) is performed with small objects encapsulating the services of moving from one element to another and returning the "pointed" element. Browsing an image requires (point) site iterators or *piters*. Image has three associated iterator types: `fwd_piter` is the type of an iterator traversing the image's data in the forward direction, while `bkd_piter` advances in the backward direction. `piter` is the default piter type, and by default is expected to be an alias of `fwd_piter`. The notions of forward and backward traversals are not strictly formalized; for instance, this order cannot be guaranteed to always follow a lexicographic order on the coordinates of the domain's space—if such order exists. The only constraint required by the Image concept is that a backward iterator should traverse the sites of an image in the reverse order of the corresponding forward iterator.

Piter objects shall be initialized with their iteration target, that is the domain of the image. An iterator browsing the values of an image `ima` of type `I` in no specific order is typically created with the following instruction:

```
mln_piter(I) p(ima.domain());
```

where `mln_piter` is the shortcut macro to access `I`'s `piter` associated type. Milena provides another useful shortcut macro presented in Section 3.3.6 (p. 98) to iterate on the image's domain using this iterator, named `for_all`. As the iterator can be used like a point site, we may use it directly to manipulate the image's values, as in the following example.

```
for_all(p) // Iterate on the domain of ‘ima’.
  ima(p) = ima(p) + 3;
```

A site iterator is bound to the domain of an image, which is a site set (see Section 4.4.3, p. 130), but it is *not* tied to one image in particular. Therefore, a single iterator may be used to iterate on several images sharing the same domain.

```
typedef image2d<int> I;
// Two 10x10 integer images.
I ima1 (10, 10);
// Initialize ima1.
// ...
I ima2 (10, 10);
// Initialize ima2.
// ...

// A third image with the same domain.
I ima3 (10, 10);
// Fill 'ima3' with the point-wise addition
// of 'ima1' and 'ima2' using a single
// piter 'p'.
mln_piter(I) p(ima3.domain());
for_all(p)
  ima3(p) = ima1(p) + ima2(p);
```

We believe the previous code is much more readable for an IP practitioner, as it looks like an algorithm from a paper manipulating points (instead of iterators). The equivalent code in an STL-style would require three iterators (one per image), which makes it both longer and more complex to non-specialists of the C++ language.

We conclude this description of the Image concept with a precision on domains and iterators. The Image concept does not place any constraint on the size of a domain, as long as it countable. A domain may even be infinite. However, the data browsed by the site iterator of an image must be finite, to guarantee that a `for_all` loop will terminate. It is for instance valid to implement a 1D image (a signal) the domain of which is $\mathbb{Z}$, showing non-null values on a finite subset (e.g. the range $[1, 100]$, and having a value of zero on the rest of the domain[4]. While the (theoretic) domain may be is infinite, the set of actual values is finite. To form a valid image, the iterator and the domain objects associated to this image must therefore be implemented to limit the set of sites browsed by iterators to the range of non-null values ($[1, 100]$).

4.4.2 *Sites*

As mentioned in the previous section, the Image concept expects its models to define two site-related associated types: `site` and `psite` (possibly to the same type). We therefore introduce two related concepts, Site and Point_Site. On the one hand, we have seen that there are no actual constraints on the structure and behavior of a site. Milena proposes an empty Site concept to possibly mark some types as explicit sites, but the library does not require sites to inherit from this concept.

4 With usual tools, we can only approximate such an infinite domain (e.g. by using a built-in type like `int` or `long`).

Point_Site		
Associated types		
Type	Model of	Definition
`site`	(Site)	Type of a site
Optional associated types		
Type	Model of	Definition
`point`	Point	Type of point.
`dpoint`	Delta_Point_Site	Type of delta-point.
`coord`		Type of coordinate.
Services		
Method signature		Definition
`operator site`		Conversion to site.
Optional Services		
Method signature		Definition
`const unsigned dim` (attribute)		Dimension
`const point& to_point()`		Conversion to point type
`coord operator[](unsigned i)`		Access to coordinate `i`

Table 5: Signature of the Point_Site concept.

On the other hand, the Point_Site concept has a more precise interface, shown in Table 5. A point site object must able to convert itself into the corresponding site object (but the converse is not true), hence the `site` associated type and the conversion operator '`operator site`'. The C++ language indeed allows the creation of user-defined conversion routines within classes. Such methods can be invoked implicitly (for instance, during an assignment to a value of the target type) or explicitly (by using a cast operator).

The concept also contains optional associated types and services that are only relevant when the site represent a point-like entity in a digital space, like the dimension of the space, the conversion to a an actual point type or the access to a coordinate of the point site. Among these optional elements, a point site may define an associated *delta-point* or *delta-point site* type (dpoint or dpsite for short). A `dpoint` represents the difference between two `point` objects. Delta points are especially useful to implement windows and neighborhoods in regular images (see Section 4.4.5, p. 133).

4.4.3 *Site Sets*

A site set represents a set of point sites[5]. Such as set may contain an actual enumeration of point sites (e.g. as an array of psites) or

5 Despite its name, a site set contains point sites (psites), not sites.

Site_Set		
Associated types		
Type	Model of	Definition
`site`	(Site)	Type of a site
`psite`	Point_Site	Type of a point site
`fwd_piter`	Site_Iterator	Forward iterator type
`bkd_piter`	Site_Iterator	Backward iterator type
Services		
Method signature		Definition
`bool has(const psite& p) const`		Psite membership test

Table 6: Signature of the Site_Set concept.

represent a regular organization of psites described by parameters (e.g. a box aligned on the axes of an orthogonal space defined by two opposite corners).

Table 6 shows the signature of this concept. Like an Image, a Site_Set shall define a `site`, a `psite`, a `fwd_piter` and a `bkd_-piter` types, as well as a `has()` method. All have the same meaning as in Image (see Section 4.4.1, p. 124). In practice many images indeed delegate the definition of these types and services to their underlying `domain_t` type (e.g. for an image type `I`, `I::fwd_-iterator` is often an alias for `I::domain_t::fwd_iterator`).

Site sets serves to define the domains of images (see Section 4.4.1, p. 124). Such site sets often contain structural information of combinatorial, topological or geometrical nature in addition to the task of "containing" psites. For example a 2D digital box (implemented by the `box2d` type in Milena) not only defines a rectangle composed of discrete points and defined by it top right-hand and bottom left-hand corners, but also a regular digital topological space containing no holes, with usual 4- and 8-adjacencies between point sites. Likewise, a site set based on the vertices of a graph exhibit a natural adjacency of sites connected by an edge. Such information are often used by window and neighborhood objects (see Section 4.4.5, p. 133).

It is also possible to define unstructured site sets, acting as bare psite containers (arrays, sets, priority queues, etc.). Such site sets are especially useful in the implementation of generic algorithms, such as the watershed transform presented in Section 5.1 (p. 153).

Section 4.5 (p. 133) presents several image types along with the site sets used as their domains.

4.4.4 *Iterators*

Many C++ libraries have adopted the programming style of the STL, where algorithms take as input a range expressed as a pair

Iterator	
Services	
Method signature	Definition
`bool is_valid() const`	Validity test
`void invalidate()`	Invalidate iterator
`void start()`	Place iterator at the start
`void next()`	Advance to next item

Table 7: Signature of the Iterator concept.

Site_Iterator refines Iterator		
Associated types		
Type	Model of	Definition
`target`		Type of target (iterated set)
`site`	(Site)	Type of a site
Services		
Method signature		Definition
`const target& target();`		Access to target
`operator psite&();`		Conversion to psite
`site& to_site();`		Conversion to site

Table 8: Signature of the Site_Iterator concept.

of two iterators (see p. 43). If we were to traverse an image `ima` of type `I` implemented as an STL container to apply a function `f` to each of its elements, we would write a loop similar to the following one.

```
for (I::iterator i = ima.begin()
       i != ima.end(), ++i)
  f(*i);
```

While this kind of code may look familiar to many C++ programmers, it will probably seem lengthy and complex to IP practitioners having little or no knowledge of C++. For this reason, Milena's iterators have been designed to look like an entity more common in IP, namely points (or more precisely point sites). The previous example translates in Milena as this:

```
mln_piter(I) p(ima.domain();
for_all(p)
  f(p);
```

Iterating on an image is done by iterating on its domain, not the image itself. Domains may be shared among different images, as seen previously. Therefore, iterators on sites (and (p)sites themselves) are not tied to an image.

Table 7 presents the general (not image-related) `Iterator` concept, while Table 8 shows the Site_Iterator concept, describing iterators on domains (site sets), windows and neighborhood.

4.4.5 *Windows and Neighborhoods*

A Window represents a sliding window relative to a point site (reference psite), and mapping this location to other psites of an image's domain. In a sense, a window can be seen as a function from a psite to a set of psites. Windows are for example used to implement structuring elements of mathematical morphology.

Windows on regular domains can be expressed as a constant set of delta-points, that is a set of vector displacements from the reference psite to the psites of the window. On non-regular windows, the psites are computed on the fly: such windows behave more like actual functions.

The Neighborhood concept is similar to Window, but add extra constraints on its models. For instance, a neighborhood must be centered, symmetric and it shall not contain its reference psite. Models of this concept represent neighborhood or adjacency relations, such a the 4- or 8-connectivity in 2D.

Windows and neighborhoods each declare associated (forward and backward) iterator types. These iterators are named “qiters” for windows (by analogy with “piters” on domains), and “niters” for neighborhoods.

The concept of Weighted_Window associates weights to the psites of a window. It is used to implement kernels in convolutions.

4.5 DATA STRUCTURES

In this section, we present some image data structures with their companion types (sites, site sets, etc.).

We have seen in Section 4.4.1 (p. 124) that a generic image type may offer different degrees of parameterization, meaning that the structure of an image can be more or less constrained. The shape of `image2d<T>` is an example of constrained shape: this image has a fixed site set (`box2d`) and its point sites (`point2d`) are also points (in $\mathbb{Z}^2$). On the contrary, a graph-based image can represent a vertex- or edge-valued graph in any “geometrical” space—even one that cannot be represented for technical reasons, such as $\mathbb{R}^{11}$).

We can observe than more general image types (less constrained image types) can often represent more constrained image types. For instance, a vertex-valued graph-based image based on a graph representing a box of $\mathbb{Z}^2$ of type `vertex_image<point2d,`

`bool>` (see Section 4.5.2) can be used to represent the same information as `image2d<bool>`.

However, as `vertex_image<point2d, bool>` may accept *any* graph as domain, it cannot make the same hypotheses as `image2d<bool>` with respect to the shape of data. For instance, accessing the value of a site using its spatial information (e.g., the location $(42, 51)$ in $\mathbb{Z}^2$) is natural with `image2d<bool>`, as its sites are also points (there is here a bijection—the identity—between a psite and a point). The memory location of this value is computed with respect to the address of the image's data and using the row (42) and column (51) indices. However, in the case of `vertex_image<point2d, bool>`, this 2D point is merely some spatial information, not directly linked to the location of the value, as there is only an uni-directional mapping from sites to spatial information in this image type. Getting a psite from a site implies either a cost in time (by browsing the list of psites to find the one associated with the searched point) or a cost in space (by storing the site-to-psite mapping in addition to the existing psite-to-site mapping). Likewise, `vertex_image<point2d, bool>` cannot consider that the 4-connected neighbors of a site are always located at a fixed offset in memory with respect to the location of the site. Contrary to `image2d<bool>` (see Section 4.6.3, p. 139), we cannot use an offset-based fast iteration on neighbors with `vertex_image<point2d, bool>`. There is hence a trade-off between efficiency and *generality* here. Therefore, it is useful to provide various image types, instead of a single one flexible enough to represent any data structure, at the expense of run-time or memory usage penalties.

4.5.1 *Classical Data Structures*

Classical data structures are regular data structure with a compact memory representation. Essentially, they represent boxes on $\mathbb{Z}^n$, with n being most often 2, sometimes 3, and more rarely other values (1, 4, 5 etc.). Images types such as `image2d<T>` or `image3d<T>` are example of regular data structures.

The domain of these data structures is an instance of `box<P>` where `P` is a type of point (e.g. `point2d`, `point3d`).

Iterators associated with this domain contain an instance of this point type. They traverse the hyper-rectangular space defined by this box in the classic forward or backward raster order.

Windows and neighborhoods are composed of delta-points (or dpoints), that is, object representing the difference between two points of the grid. For instance, in Milena the standard `c4` neighborhood object (of type `neighb2d`) contains an array with the following dpoints: $\{(-1, 0), (0, -1), (0, +1), (+1, 0)\}$. Iterators on windows and neighborhoods contains a reference to a center point as well as an index of the previously mentioned array.

4.5.2 *Graphs*

Graph-based images are composed of a triplet:

- A domain object containing an undirected graph structure (with no data attached to vertices nor to edges). This domain can be shared by several images, even if they do not have the same value type. Each point site is assigned a number corresponding to a vertex or edge, depending on the image type.
- A function from the point sites of the image (either the vertices or the edges) to a site type (e.g. `point2d`). This function describes the *geometry* of the graph.
- A function from the point sites of the image to the corresponding values.

The last two items are usually implemented as arrays, as domains shall not change once they are used as domain of an image. In Milena, `vertex_image<P, V>` and `edge_image<P, V>` respectively implement vertex- and edge-valued graph-based images, where vertices are located on points of type `P`, and values of type `V` are attached to vertices and edges respectively.

Iterators on graph-based images contain a reference to their target domain, as well as a number (vertex or edge identifier). Windows and neighborhoods objects act as functions from a point site to a set of point sites. These functions (e.g., mapping a vertex psite to the adjacent edge psites) are actually implemented within neighborhood and window iterators.

4.5.3 *Cell Complexes*

Intuitively, complexes can be seen as a generalization of graphs. An informal definition of a *simplicial complex* (or simplicial d-complex) is “a set of simplices” (plural of simplex), where a simplex or n-simplex is the simplest manifold that can be created using n points (with $0 \leq n \leq d$). A 0-simplex is a point, a 1-simplex a line segment, a 2-simplex a triangle, a 3-simplex a tetrahedron. A graph is indeed a 1-complex. Figure 5 shows an example of simplicial complex.

Likewise, a *cubical complex* or cubical d-complex can be thought as a set of n-faces (with $0 \leq n \leq d$) in $\mathbb{Z}^d$, like points (0-faces), edges (1-faces), squares (2-faces), cubes (3-faces) or hypercubes (4-faces). Figure 6 depicts a cubical complex sample.

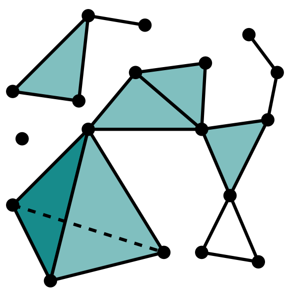


Figure 5: A simplicial 3-complex.

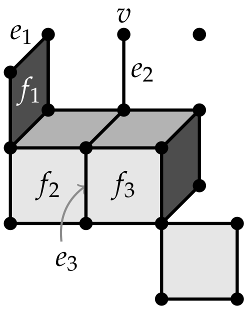


Figure 6: A cubical 2-complex.

Milena provides a general abstract data structure to implement cell complexes, including simplicial and cubical complexes, called `topo::complex<D>`, where `D` is an integer denoting the dimension of the complex. Internally, `topo::complex<D>` contains $\mathtt{D} + 1$ arrays to stores faces, one array per dimension (0 to `D`). What is actually stored for an array cell of `N`-face is the list of adjacent $(\mathtt{N} - 1)$-faces and adjacent $(\mathtt{N} + 1)$-faces (except for 0- and `N`-faces that respectively do not have $(\mathtt{N} - 1)$-faces and adjacent $(\mathtt{N} + 1)$-faces). This structure is not directly related to the image world, much like inner graphs from the previous section.

To represent complexes in the realm of images, the library provides a complex-based site set, `p_complex<D, G>`, wrapping a `topo::complex<D>` object. The `G` parameter is the type of a geometry object, associating location information (sites) to each face of the complex. The image type `complex_image<D, G, V>` uses this site set as a domain, with the same meaning for parameters `D` and `G`, while `V` represents the values associated to each face of the complex image.

Domain iterators, windows, neighborhoods and their iterators works in a similar fashion as in graph-based image, except that there is by default no constraints on the dimension of browsed faces.

Cell complexes are useful to represent *inter-pixel* data structures, allowing users to store data between primary image elements (pixels, voxels, polygons of a mesh, etc.). Examples of use are shown in Chapter 5.

Algorithm 4.1: Non generic implementation of `fill`.

```
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;
}
```

4.6 ALGORITHMS

4.6.1 *Generic Algorithms*

A generic algorithm is an abstract definition of a set of operations to perform some computations. It should contain as little implementation details as possible on the data structures it uses. The exact type of its input should not be fixed in this definition, but instead be represented by parameters, so that its applicability is not restricted to a particular type. Likewise, the types of its output and its intermediate values should not be fixed, but either taken as parameters or deduced from other parameters.

For instance Algorithm 4.1 shows a non generic implementation of the `fill` algorithm, assigning a value to the pixels of an image. An important limitation of this algorithm comes from the use of nested loops to implement the image traversal (one for each dimension). Indeed not only this technique fails to scale to higher dimensions, but it also does not handle special use cases such as the restriction of the image's domain by a mask, nor the non-regular (e.g., graph-based) image types. We have seen that a classical solution to this problem is to resort to iterators, which help to uncouple data structures and algorithms. Another limitation of Algorithm 4.1 is related to value types: this implementation of `fill` can only handle an `image` type the values of which are compatible with the `unsigned int` type.

Algorithm 4.2 shows an implementation that has none of the limitations mentioned previously. Instead of implementing the logic of the traversal itself, it uses a point site iterator (piter) targeting the domain of the input image. Moreover, both the image type and the value type are now free "type variables" (parameters `I` and `V` respectively), making this implementation virtually compatible with any input, as long as the image type models the Image concept.

Chapter 5 presents some examples of more complex IP algorithms implemented in a generic fashion.

Algorithm 4.2: Generic implementation of `fill`.

```
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;
  }
}
```

4.6.2 *Type Deduction*

We have mentioned that generic algorithms should use as type only parameters or types deduced from parameters, so to be as generic as possible. Type deductions are implemented with static programming techniques (see Section 2.6, p. 69), as traits (see Section 2.6.2, p. 71). The most simple case, when the deduced type is the same as one of the template parameter, does not even require a traits class: the parameter can be used directly. Other cases can be more or less complex.

For instance, most IP algorithms producing an output image must be able to create a data structure in memory to receive the values of the output. However, some input types represent images with no actual data. An image type implemented as a uniform function on a domain is an example of such an image: its only data are a single value and a domain: each access to any elements of its domain indeed always present the same value, so there is no need to store it multiple times. Applying an IP operator to such an image, however, may not yield a uniform image. Therefore, the type deduced for the corresponding output data structure cannot be the same as the input type parameter. Milena proposes a mechanism to deduce for each image type a corresponding "concrete" image type, able to store actual values. We do not explain this mechanism here, as it depends on complex metaprogramming techniques that are out of the scope of this thesis. This system is abstracted by the `mln_concrete(I)` macro, returning the concrete data type corresponding to the (image) type `I`. Note that when `I` is already an image type with actual data in memory, `mln_concrete(I)` = `I`.

Likewise, Milena provides the `mln_ch_value_(I, V)` macro, relying on a mechanism similar to `mln_concrete`'s, to deduce the concrete type of `I` while changing its value type to `V`. For instance, `mln_ch_value_(image2d<int>, bool)` returns the concrete type

Algorithm 4.3: Non-generic dilation implementation.

```
image dilation(const image& input)
{
  image output(input.nrows(), input.ncols());
  for (unsigned r = 0; r < input.nrows(); ++r)
    for (unsigned c = 0; c < input.ncols(); ++c)
    {
      unsigned char sup = input(r,c);
      if (r != 0
          && input(r-1,c) > sup)
        sup = input(r-1,c);
      if (r != input.nrows()-1
          && input(r+1,c) > sup)
        sup = input(r+1,c);
      if (c != 0
          && input(r,c-1) > sup)
        sup = input(r,c-1);
      if (c != input.ncols()-1
          && input(r,c+1) > sup)
        sup = input(r,c+1);
      output(r, c) = sup;
    }
  return output;
}
```

of `image2d<int>` with a value type type set to `bool`, that is `image2d<bool>`.

4.6.3 *Efficiency Considerations*

Let us consider two implementations of a classical morphological operator, a dilation with a flat structuring element. Algorithm 4.3 shows a non-generic implementation of this algorithm, while Algorithm 4.4 proposes a generic implementation of this operator [94]. We do not discuss the benefits of the generic approach over the non-generic one, as they have been covered previously. In this section we emphasize efficiency considerations in the context of generic IP software.

The Cost of Abstraction

Table 9 (p. 143) shows execution times of several implementations of the dilation algorithm, including the two previously mentioned. These figures exhibit an important run time overhead in the generic case (Algorithm 4.4), which is about ten times longer to execute than the non-generic one (Algorithm 4.3). This is not a consequence of the GP paradigm *per se*. It is rather due to the highly abstract style of Algorithm 4.4, which in return makes

Algorithm 4.4: Generic dilation implementation.

```
template <typename I, typename W>
I dilation(const Image<I>& input_,
           const Window<W>& win_)
{
  const I& input = exact(input_);
  const W& win = exact(win_);
  I output; initialize(output, input);
  // Iterator on sites of the domain of ‘input’.
  mln_piter(I) p(input.domain());
  // Iterator on the neighbors of ‘p’
  // with respect to ‘win’.
  mln_qiter(W) q(win, p);
  for_all(p)
    {
      // Accumulator computing the supremum
      // with respect to ‘win’.
      accu::supremum<mln_value(I)> sup;
      for_all(q) if (input.has(q))
        sup.take(input(q));
      output(p) = sup.to_result();
    }
  return output;
}
```

the routine very versatile with respect to the context of use. The non-generic version is faster than the generic one because it takes advantage of known features of the input types. For instance the structuring element is “built in the function” (whereas it is an object taken as a generic input in Algorithm 4.4): its size is constant and known at compile-time. Such an implementation trait is useful *static* (compile-time) information that the compiler can use to optimize the code. Hence, what prevents a code from being generic appears to be the condition to generate fast code: implementation details.

Generic Optimizations

The balance between genericity (the ability to handle many different data types) and efficiency (the run-time speed) admittedly depends on the level of details, but these two aspects are not entirely antagonistic: by carefully choosing the amount of specific traits used in an algorithm, one can create intermediate variants showing good run-time performance while keeping many generic traits.

For instance, a means to speed up Algorithm 4.4 is to avoid using site iterators to browse the domain of the input and output images. In Milena, site iterators can be automatically converted into sites (points), that is, locations in the domain of one (or several) image(s). Such location information is not tied to a given image: in the case of a regular 2D image, a site `point2d(42, 51)` is compatible with every regular, 2D, integer coordinate-based domain of the library (including toric spaces, non-rectangular 2D subspaces of $\mathbb{Z}^2$, etc.). This is why iterator `p` is used to refer to the same location in both `input` and `output` in Algorithm 4.4.

The price to pay for such a general expression is usually a run-time overhead: computations have to be performed each time a site iterator is used to access data from an image. However, this flexibility is not always needed when the data to process exhibit noteworthy properties. For instance, an image the values of which are stored in a contiguous, linear memory space, can be browsed using a pointer, directly accessing values in a sequential manner using their memory addresses, instead of computing a location at each access. In Milena, such pointers are encapsulated in small objects called pixel iterators or *pixters* (a pixel refers to a (site, value) pair in an image). Pixters are bound to one image, and cannot be used to iterate on another image. Pixters can also be used to browse spatially invariant structuring elements (windows), as long as the underlying image domain is regular.

Algorithm 4.5 shows a reimplementation of Algorithm 4.4 where site iterators have been replaced by pixel iterators. The code is very similar, except that images `input` and `output` are now browsed with two different iterators (each of them holding a pointer to the data of the corresponding image). Such an implementation of the morphological dilation is less generic than Algorithm 4.4. Even so, it can still be used with a wide variety of image types, as long as their data present a regular organization, which comprises any-dimension classical image using a single linear buffer to store its values. Besides, it is compatible with any spatially invariant structuring element (or in other words, any constant window). Thus it remains more generic than Algorithm 4.3. As for efficiency, Algorithm 4.5 matches almost Algorithm 4.3 in terms of speed (see Table 9), so it is a good alternative to the generic dilation, when the trade-off between genericity and efficiency can be shifted towards the latter.

The approach presented here can be applied to other algorithms of the IP literature for which optimized implementations have been proposed. These optimizations are in practice compatible with a range of input types, so their implementations can be considered as *generic optimizations* since they are not tied to a specific type [97].

Algorithm 4.5: Partially generic optimized dilation.

```
template <typename I, typename W>
I dilation(const Image<I>& input_,
           const Window<W>& win_)
{
  const I& input = exact(input_);
  const W& win = exact(win_);
  I output; initialize(output, input);
  // Iterator on the pixels of ‘input’.
  mln_pixter(const I) pi(input);
  // Iterator on the pixels of ‘output’.
  mln_pixter(I) po(output);
  // Iterator on the neighbor pixels of ‘pi’.
  mln_qixter(const I, W) q(pi, win);
  for_all_2(pi, po)
    {
      accu::supremum<mln_value(I)> sup;
      for_all(q)
        sup.take(q.val());
      po.val() = sup.to_result();
    }
  return output;
}
```

Extra Generic Optimizations

The approach proposed here can be carried further to improve the efficiency of generic optimizations. The idea is to involve data structures in the optimization effort: instead of acting only on algorithms, we can implement new optimized variants by working on their inputs as well.

For instance, in place of a window containing a dynamic array of vectors (e.g., $\{(-1,0), (0,-1), (0,0), (0,+1), (+1,0)\}$ in the case of a 4-connected spatially-invariant structuring element)—the size of which is known at *run time*—we can implement and use a *static* window containing a static array carrying the same data, but the contents and size of which are known at *compile time*. Modern compilers make use of this additional information to perform efficient optimizations (e.g, replace the loop over the elements of the window by an equivalent "unrolled" code). In this particular case, the implementation requires the creation of two new, simple data types (static window and static pixel iterator). No new implementation of the dilation is required: using Algorithm 4.5 with this new window suffices. The resulting code gives run times which are not only faster than the non-generic version of Algorithm 4.3, but which may also be faster than a hand-made, pointer-based optimized (hence non-generic) version of the dilation, as shown hereafter.

Implementation	Time (s) per image (px)		
	512^2	1024^2	2048^2
Non generic (Alg. 4.3)	0.10	0.39	1.53
Non generic, pointer-based (Alg. 4.6)	0.07	0.33	1.27
Generic (Alg. 4.4)	0.99	4.07	16.23
Fast, partly generic (Alg. 4.5)	0.13	0.54	1.95
Alg. 4.5 with a static window	0.06	0.28	1.03

Table 9: Execution times of various dilation implementations.

Results and Evaluation

Table 9 shows execution times of various implementations of the morphological dilation with a 4-connected structuring element (window) applied to images of growing sizes (512×512, 1024×1024 and 2048×2048 pixels). Times shown correspond to 10 iterative invocations. Tests were conducted on a PC running Debian GNU/Linux, featuring an Intel Pentium 4 CPU running at 3.4 GHz with 2 GB RAM at 400 MHz, using the C++ compiler g++ (GCC) version 4.4.5, invoked with optimization option '`-O3`'.

In addition to the dilation implementations presented previously, an additional non-generic version using pointer-based optimizations shown in Algorithm 4.6 has been added to the test suite, so as to further compare non-generic code—mostly optimized by hand—and generic code—mostly optimized by the compiler.

The overhead of the most generic algorithm is important: about ten times longer than Algorithm 4.3. The highly adaptable code of Algorithm 4.4 is free of implementation detail that the compiler could use to generate fast code (image values access with no indirection, statically-known structuring element). Algorithm 4.5 proposes a trade-off between genericity and efficiency: it is about 30% times slower than Algorithm 4.3, but is generic enough to work on many regular image types (as a matter of fact, the most common ones). The case of the dilation with a static window is even more interesting: reusing the same code (Algorithm 4.5) with a less generic input (a static window representing a fixed and spatially invariant structuring-element) makes the code twice faster, to the point that it outperforms the manually optimized pointer-based implementation. Therefore, having several implementations (namely Algorithms 4.4 and 4.5) is useful when flexibility and efficiency are sought.

4.6.4 *Properties of Data Structures and Property-Based Overloading*

In this section, we present a mechanism to automate the selection of the best known variants of an algorithm at compile-time, based

Algorithm 4.6: Non-generic, pointer-based optimized dilation implementation.

```
image dilation(const image& input)
{
  // Offsets corresponding to a 4-connected
  // structuring element moving on ‘input’.
  ptrdiff_t win_offset[4] =
    { &input(-1, 0) - &input(0, 0),
      &input(+1, 0) - &input(0, 0),
      &input(0, -1) - &input(0, 0),
      &input(0, +1) - &input(0, 0) };
  // Initialization of the output image.
  image output(input.nrows(), input.ncols());
  for (unsigned int r = 0; r < input.nrows(); ++r)
  {
    const unsigned* pi = &input(r, 0);
    unsigned* po = &output(r, 0);
    for (; pi < &input(r, 0) + input.ncols();
         ++pi, ++po)
    {
      unsigned char sup = *pi;
      if (r  != 0
          && *(pi + win_offset[0]) > sup)
        sup = *(pi + win_offset[0]);
      if (r  != input.nrows() - 1
          && *(pi + win_offset[1]) > sup)
        sup = *(pi + win_offset[1]);
      if (pi != &input(r, 0)
          && *(pi + win_offset[2]) > sup)
        sup = *(pi + win_offset[2]);
      if (pi  != &input(r, 0) + input.ncols() - 1
          && *(pi + win_offset[3]) > sup)
        sup = *(pi + win_offset[3]);
      *po = sup;
    }
  }
  return output;
}
```

Algorithm 4.7: Partially generic filling implementation.

```
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));
}
```

Algorithm 4.8: Facade of the `fill` dispatch mechanism.

```
// Facade.
template <typename I, typename V>
inline
void fill(Image<I>& ima, const V& v)
{
  // Dispatch following the ‘‘value_access’’
  // property.
  fill_dispatch(mln_trait_image_value_access(I)(),
                ima, v);
}
```

on properties of input types. This technique completes the generic optimization strategy proposed above.

For instance, let us consider the generic implementation of the `fill` algorithm shown in Algorithm 4.2 and a fast variant of this algorithm proposed in Algorithm 4.7. This algorithm is less general than the former. It expects its input image to present its data as a single linear block of values and uses a low-level routine called `data::memset_` generalizing the `memset()` standard C function performing a rapid initialization of a C array.

To automate the selection of the best `fill` variant, we implement a dispatch algorithm composed of three parts.

1. The first element is a *facade* (see Algorithm 4.8) having the same interface as the generic implementation. This routine queries the input type's properties and delegate to the best version based on them. Here, the `mln_trait_image_value_access` macro retrieves the `value_access` (see Table 4, p. 90) property of the image input type (`I`).

2. The second step is to provide a default delegation calling the generic version (see Algorithm 4.9)[6]. This way the dispatch mechanism is guaranteed to find an implementation.

6 The `trait::image::value_access::any` value is the default value of the `value_access` property (see Section 3.3.5, p. 89).

Algorithm 4.9: Default delegation of the `fill` dispatch.

```
// Default dispatch case.
template <typename I, typename V>
void
fill_dispatch(trait::image::value_access::any,
              Image<I>& ima, const V& v)
{
  // Delegate to the generic (non-optimized)
  // version.
  generic::fill(ima, v);
}
```

Algorithm 4.10: Delegation of the `fill` dispatch mechanism for images with direct access to values.

```
// 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& v)
{
  fill_one_block(ima, v);
}
```

3. The last step consists in providing generic optimizations for subsets of image types satisfying a constraint on one or more properties. As shown in Algorithm 4.10, the dispatch delegates the call to the `fill_one_block` variant when the image type `I` provides a direct (i.e. involving no computation) access to its values.

This mechanism, called *property-based overloading*, is much more powerful than simple type-based overloading or than the explicit template specialization mechanism. Each dispatch case of the former covers a whole *subset* of the space of types, while later techniques enable the definition of variants overloaded or (resp. specialized) for a *single* specific type (resp. template). Property-based overloading is a static mechanism resolved by the compiler: the dispatch induces no penalty at run-time.

4.7 IMAGE MORPHERS

Section 3.3.6 (p. 95) presents the idea of morphers as well as some examples. In IP, morphers may be developed for various entities, such a images, site sets or values. We only concentrate on image morphers in this section. Such morphers may be divided in three categories.

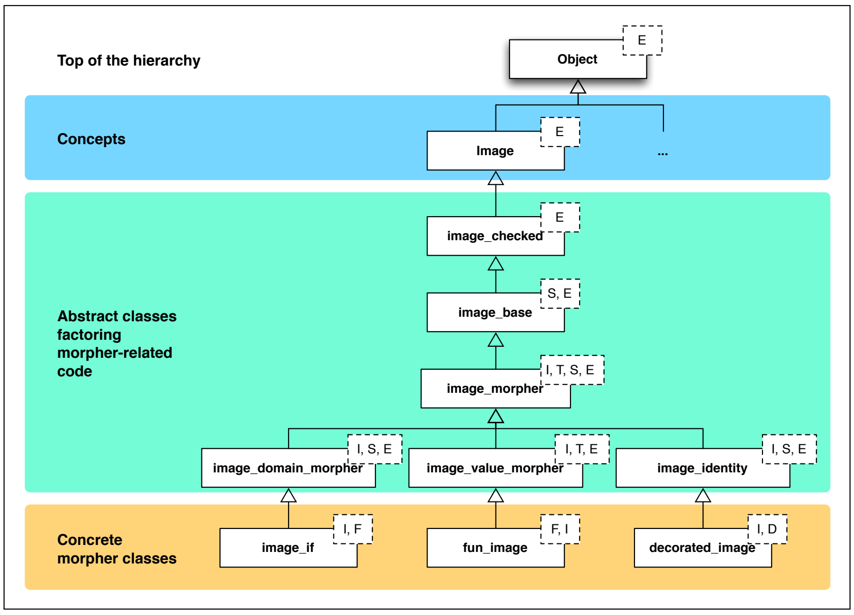


Figure 7: An excerpt of the static hierarchy of image morpher from the Milena library.

DOMAIN MORPHERS These morphers affect the domain of an image. The `image_if` and `slice_image` morphers, changing the domain of its target, are an example of such morphers.

VALUE-RELATED MORPHERS Morphers may also change the behavior of an image's values. The `fun_image`, applying a function on-the-fly is one of them.

IDENTITY MORPHERS The third category of morphers are derived from the identity function, introducing no change to an image. A morpher logging accesses to the value of an image for profiling purpose falls in this category.

Like primary images types, image morphers classes are organized in a hierarchy factoring shared code, as depicted in Figure 7. Morphers of the previous three categories are usually implemented as classes derived from the `image_domain_morpher`, `image_value_morpher` or `image_identity` class.

4.7.1 *Domain Morphers*

A domain image morpher changes the domain of an image, by restricting it, extending it, changing its topology, etc. In Milena, the abstract class template `image_domain_morpher<I, S, E>` factors the implementation of this category of morphers by turning the domain of an image of type `I` into another domain (site set)

Algorithm 4.11: Restriction of an image's domain to a subset using a sub_image morpher.

```
image2d<rgb8> lena = load("lena.png");
rgb8 green(0, 255, 0);
// Region Of Interest.
box2d roi(5,5, 10,10);
// Fill 'lena' restricted to 'roi'.
fill(lena | roi, green);
```

of type S[7]. In particular, this class template redefines image access methods (see Table 2, p. 82) by taking the new domain into account.

The `sub_image<I, S>` morpher class template is used to replace the domain of an image of type `I` with a subset of type `S`. The easiest way to apply this morpher is through operator '`|`', defined as follows:

```
template <typename I, typename S>
sub_image<I, S>
operator | (Image<I>& ima,
            const Site_Set<S>& pset);
```

This operator is a construction shortcut relying on concept-based overloading: the template names `Image` and `Site_Set` in its signature ensure `operator|` returns a `sub_image` only if `I` and `S` are models of Image and Site_Set.

An iteration on a `sub_image<I, S>` is in fact a traversal of the new domain (`pset`, in the above signature) in lieu of the initial image domain (`ima.domain()`). Thus we can change the behavior of an algorithm using this morpher. In the example of Algorithm 4.11, we fill an RGB image with the green value in a Region of Interest (ROI) delimited by a square defined by its corners $(5, 5)$ and $(10, 10)$. The state of the `lena` image object at the beginning and at the end of the code is shown in Figure 8.

Another means to restrict the domain of an image is by using a predicate on the point sites of the original image domain. Milena provides an `image_if<I, F>` class template, mentioned earlier, to create morphed images based on an initial image (of type `I`), and on a predicate functor (of type `F`). As in the case of `sub_image`, it is possible to create an `image_if` object thanks to an operator '`|`'.

```
template <typename I, typename F>
image_if<I,F>
operator| (Image<I>& ima,
           const Function_v2b<F>& f);
```

7 The last parameter of `image_domain_morpher<I, S, E>` (`E`), represents the exact morpher type, as in every non-leaf class of a static hierarchy.

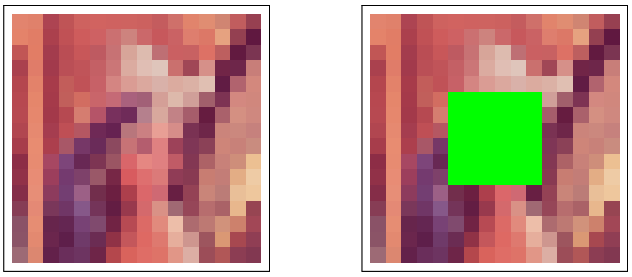

(a) "lena" before. (b) "lena" after.

Figure 8: Input and output of Algorithm 4.11.

Algorithm 4.12: Restriction of an image's domain with a predicate using an `image_if` morpher.

```
image2d<rgb8> lena = load("lena.png");
rgb8 green(0, 255, 0);
// Predicate: $(x,y) \mapsto (x+y) \equiv 0 \pmod 2$.
fun::p2b::chess f;
// Fill 'lena' restricted by 'f'.
fill(lena | f, green);
```

Note that this operator does not introduce any ambiguity with respect to the operator '|' defined previously, as their signature differ by the concepts of their input. In the latter operator, the `Function_v2b` template expresses that `f` shall be a functor returning a `bool` value. During the construction of `image_if<I, F>`, the compatibility of `f`'s inputs and of `ima`'s point sites is checked.

Algorithm 4.12 shows an example of use of `image_if`, where the predicate `f` is an instance of the `fun::p2b::chess` functor returning `true` (resp. `false`) for 2D points corresponding to "white" (resp. "black") squares of a chessboard. The results of this code are shown in Figure 9.

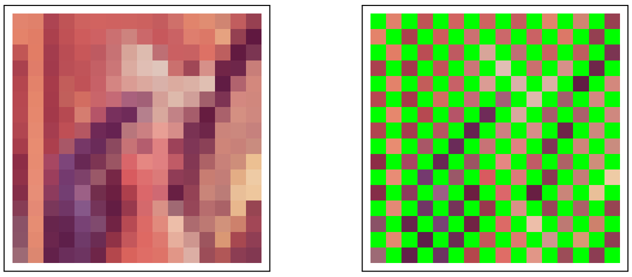

(a) "lena" before. (b) "lena" after.

Figure 9: Input and output of Algorithm 4.12.

Algorithm 4.13: Restriction of an image to a single color channel using a `fun_image` morpher.

```
image2d<rgb8> lena = load("lena.png");
// Filling ‘lena’ restricted to the ‘‘green’’
// channel.
fill(fun::green() << lena, 255);
```

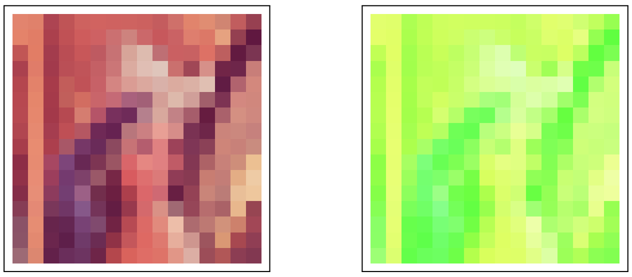

(a) "`lena`" before. (b) "`lena`" after.

Figure 10: Input and output of Algorithm 4.13.

4.7.2 *Value-Related Morphers*

A value image morpher alters the values of an image. It is usually an instance of a class derived from the class template `image_value_morpher<I, T, E>`, where `I` is the type of the underlying image and `T` the new value type (the one of the morphed image).

The `fun_image<F, I>` class template is an example of such a category of morphers. It wraps a read-only image `ima` (of type `I`) in a functor `f` (of type `F`) so that each time a value is read in `fun_image<F, I>(ima, f)` at a location `p`, `f(ima(p))` is returned (instead of `ima(p)`).

The Milena library also provides a syntactic sugar for the construction of a `fun_image` in the form of an operator '«':

```
template <typename F, typename I>
fun_image<F, I>
operator << (const Function_v2v<F>& f,
             const Image<I>& ima);
```

Using this routine, we can create a light-weight image representing the green channel of an RGB image, by using a `fun_image` morpher with function `fun::green()` mapping an RGB value (`rgb8`) to its green component as in Algorithm 4.13 and producing the results shown in Figure 10.

Algorithm 4.14: Counter decoration class template.

```
template <typename I>
struct counter
{
  typedef mln_psite(I) P;
  typedef mln_value(I) V;

  counter() : nreads(0), nwrites(0) {}

  void reading(const I&, const P&) const
  {
    ++nreads;
  }
  void writing(I&, const P&, const V&)
  {
    ++nwrites;
  }

  unsigned nreads;
  unsigned nwrites;
};
```

4.7.3 *Identity Morphers*

Identity image morphers are a third category of morphers not directly related to an image's domain or values, thus introducing no modification of the image itself. These morphers introduce orthogonal changes in an image, like counting the number of read and write operations in an image; displaying an image in a graphical window and updating it at each change; triggering some code when a condition is met in an image; etc. The `image_identity<I, S, E>` class template is responsible for providing a common base class for this category of morphers.

An example of identity morpher is the `decorated_image<I, D>` class template, that attaches a decoration of type `D` to an image of type `I`. A decoration is an object providing two methods, one for read accesses (`reading()`) and one for write accesses (`writing()`). The code in each of these methods is run each time a value is read or written in the initial image.

The counter class template from Algorithm 4.14 is an example of decoration compatible with `decorated_image`. Its purpose is to record the number of read and write accesses in an image. In the following code, we use this combination of `decorated_image` and `counter` with a very simple operation dividing the values of an image by two. At the end of the program, the read and writes counts are displayed.

```
typedef image2d<rgb8> I;
I lena = load("lena.png");
typedef D counter<I>;
D count;
decorated_image<I, D> decorated(lena, count);
mln_piter(I) p(decorated.domain());
for_all(p)
  ima(p) = ima(p) / 2;
std::cout << count.nreads << std::endl;
std::cout << count.nwrites << std::endl;
```

5 APPLICATIONS OF GENERIC IMAGE PROCESSING

This chapter illustrates the possibilities of the framework proposed in this thesis, by presenting examples of actual generic Image Processing algorithms that have been implemented in the Milena library. Their full implementation is given and explained. Applications of theses algorithms is then shown for various data types, to demonstrate the generality of their definition.

The examples shown in this chapter illustrate methods from Mathematical Morphology (MM) and Digital Geometry (DG). These domains have indeed been particularly developed in the Olena project regarding algorithms and data structures. The scope of the platform is however not limited to these domains.

5.1 A GENERIC WATERSHED TRANSFORM IMPLEMENTATION

Our first example presents an implementation of a classic morphological tool used in segmentation, a watershed transform by flooding based on hierarchical queues. Algorithm 5.1 shows an implementation based on the ideas proposed in Chapter 4 and implemented in Milena. This generic routine implements the watershed transform algorithm proposed by Meyer [105].

The idea of the watershed transform, initially proposed by Beucher and Lantuéjoul [22], is to consider a 2D intensity (gray-level or integer value) input image as a landscape, where high values denotes peaks and low values represent valleys. The first step is to identify the regional minima of this image, which are flat zones having no neighbor sites with a lower altitude. The watershed algorithms then simulates the introduction of sources of water in each of these regional minima, progressively flooding the landscape. A barrier is built at locations were two bodies of water (or more) meet. The watershed is made of this set of barriers. Other parts belong to catchment basins attached to an initial minimum.

In many cases, the watershed transform is not performed directly on the initial input image. Instead, a gradient of this image is often computed first, and the watershed transform is run on the intensity image representing the magnitude of this gradient. A high gradient intensity is indeed a clue as to the presence of contours. By applying the watershed transform on this intensity

Algorithm 5.1: Generic implementation of watershed transform.

```
namespace watershed {
  template <typename L, typename I, typename N>
  mln_ch_value(I, L)
  flooding(const Image<I>& input_,
           const Neighborhood<N>& nbh_,
           L& n_basins)
  {
    const I input = exact(input_);
    const N nbh = exact(nbh_);
    typedef L marker;
    // Label of non-minimum (watershed) pixels is 0.
    const marker unmarked = literal::zero;
    typedef mln_value(I) V;
    const V max = mln_max(V);
    // Initialize the output with markers (minima).
    mln_ch_value(I, marker) output =
      labeling::regional_minima (input, nbh, n_basins);
    // Hierarchical queue.
    typedef mln_psite(I) psite;
    typedef p_queue_fast<psite> Q;
    p_priority<V, Q> queue;
    // Image keeping track of processed psites.
    mln_ch_value(I, bool) in_queue;
    initialize(in_queue, input);
    data::fill(in_queue, false);
    // Initialize, then process the hierarchical queue.
    mln_piter(I) p(output.domain());
    mln_niter(N) n(nbh, p);
    for_all(p)
      if (output(p) == unmarked)
        for_all(n)
          if (output.domain().has(n) && output(n) != unmarked)
            {
              queue.push(max - input(p), p);
              in_queue(p) = true;
              break;
            }
    while (! queue.is_empty()) {
      psite p = queue.front(); queue.pop();
      marker adjacent_marker = unmarked;
      bool single_adjacent_marker_p = true;
      mln_niter(N) n(nbh, p);
      for_all(n)
        if (output.domain().has(n) && output(n) != unmarked)
          {
            if (adjacent_marker == unmarked) {
              adjacent_marker = output(n);
              single_adjacent_marker_p = true;
            }
            else if (adjacent_marker != output(n)) {
              single_adjacent_marker_p = false;
              break;
            }
          }
      if (single_adjacent_marker_p) {
        output(p) = adjacent_marker;
        for_all(n)
          if (output.domain().has(n) && output(n) == unmarked
              && ! in_queue(n))
            {
              queue.push(max - input(n), n);
              in_queue(n) = true;
            }
      }
    }
    return output;
  }
}
```

image, we expect the watershed line to follow the contours of the initial image.

The watershed transform produces a segmentation of the initial image. Initially, each of its regional minima is given a unique label. A special label is reserved for pixels[1] which are eventually part of the watershed. Catchment basins made of connected components are supersets of their corresponding regional minima and form the regions of the final segmentation. Watershed pixels form a connected curve separating these regions.

Algorithm 5.1 proposes a generic implementation of this principle based on a hierarchical queue data structure (`queue`, lines 19–21). The special label 0 is reserved for watershed pixels (line 11). The output image is initialized with an image of regional minima computed with the `labeling::regional_minima()` routine. These minima form the initial basins of the landscape that will grow during the algorithm. This steps also determines the number of basins, stored in the `n_basins` variable. At the end of the algorithm, each region (non-watershed) pixel is thus given a label in the range [1,`n_basins`]. To prevent a point site from being processed twice, processed point sites are marked in a binary image (`in_queue`) having the same structure as the input and output image (lines 22–24).

The hierarchical queue is initialized with site sets adjacent to regional minima with a priority inversely proportional to their level (lines 28–36). The queue is then processed until it is empty. Point sites are extracted one by one (line 38) starting with the one having the lowest level (i.e., the highest priority). The neighborhood of this site set is then explored to determine if it belongs to a basin (lines 42–53). If so, the corresponding pixel is given the label of that basin in the output image, and neighboring point sites not yet processed are added to the hierarchical queue (lines 54–62). The image returned by the algorithm is a label map where each pixel of the same region is assigned a common positive label, while watershed pixels are denoted by the label 0.

The implementation proposed in Algorithm 5.1 shows no implementation detail bound to the type of the input image. It can therefore be used with a variety of data structures, as long as their values are scalar and are totally ordered. Likewise, the connectivity of the regions is not fixed, and can be freely chosen thanks to the second argument of the `watershed::flooding` routine. Finally, the last argument is not an input, but an output: `n_basins` is assigned the number of regions of the segmentation

1 For simplification purpose, the name "pixel" is used in this description of the watershed transform. A more appropriate term, not limited to 2D regular images, would be "image element", composed of a point site and the associated value.

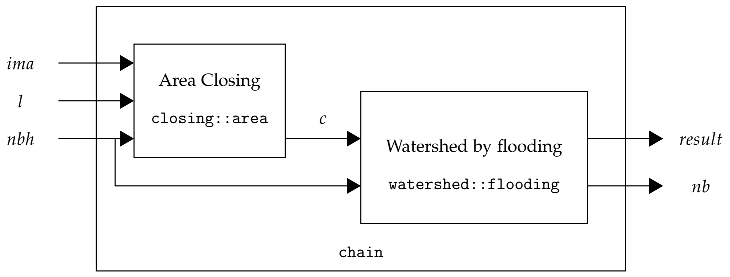


Figure 11: A Generic Morphological Segmentation Chain.

Algorithm 5.2: Implementation of a Generic Morphological Segmentation Chain.

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

during the execution of the algorithm. The parameter `L` is the type of the labels in the output image.

5.1.1 *Generic Morphological Segmentation Chain*

To illustrate the previous algorithm, we present a simple generic IP chain performing a segmentation and composed of two operators, shown on Figure 11 [94]. From an image `ima`, this chain computes an area closing `c` using a criterion value `l`; then, it performs a watershed transform by flooding on `c` to obtain a segmentation `s`. A generic implementation of this chain is given by Algorithm 5.2[2].

The watershed transform often produces "over-segmented" results. As the number of regions is the same as the number of regional (local) minima, it is very sensitive to the local extrema produced by noise. The area closing step is used as an "extrema killer": it flattens small valley components that have an area smaller than the parameter l (in terms of numbers of sites). The resulting segmentation thus contains a smaller number of regions.

2 In Algorithm 5.2, `nb`, despite being technically an argument of `chain`, is considered as an "output value", as it is set by the `watershed::flooding` routine.

5.1.2 *Illustrations*

We have applied the previous segmentation chain on different images `ima`. All of the following illustrations use the exact same Milena code shown in Algorithm 5.2

Regular 2-Dimensional Image

In the example of Figure 12, we have first computed a morphological gradient to create the input image for the processing chain. A 4-connected window is used to compute both this gradient image and the output (Figure 12c), where basins have been labeled with random colors.

Vertex-Valued Graph-Based Image

Figure 13 shows an example of planar graph-based [150] gray-level image, from which a gradient is computed using the vertex adjacency as neighboring relation. The result shows four basins separated by a watershed line on pixels.

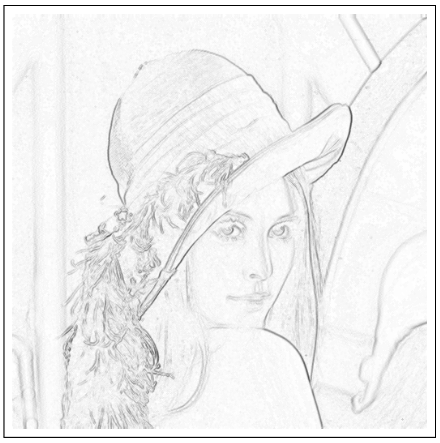
Edge-Valued Graph-Based Image

The example shown in Figure 14 is also based on a graph, but where values are stored on edges instead of vertices. The graph is built from the triangulation of a set of points in a 2D space. The value assigned to each edge is its length. Therefore regional minima on Figure 14b correspond to (locally) shortest edges. The resulting segmentation Figure 14c can be interpreted as a clustering of the initial data set. This approach can be generalized to any n-dimensional digital space, showing that an image segmentation algorithm such as the watershed transform can also be used as a data mining tool.

Simplicial Complex-Based Image

In this last example, a triangular mesh is viewed as a simplicial 2-complex (see Section 4.5.3, p. 135), composed of triangles, edges and vertices (Figure 15a). From this image, we can compute maximum curvature values on each triangle of the complex [119], and compute an average curvature on its edges. Finally, the processing chain is applied and produces a watershed cut [38] on edges; basins are propagated to adjacent triangles and vertices for visualization purpose (Figure 15c).

(a) Input.


(b) Morphological gradient of (a).

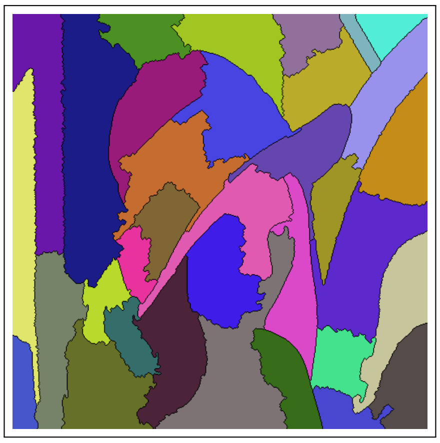
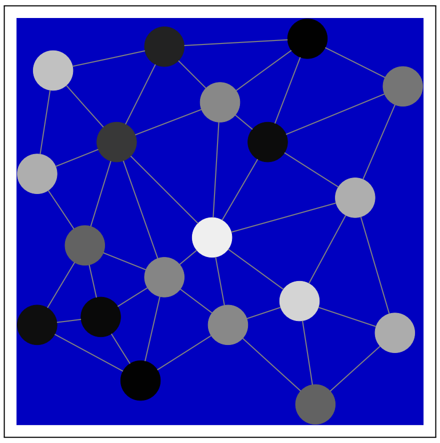
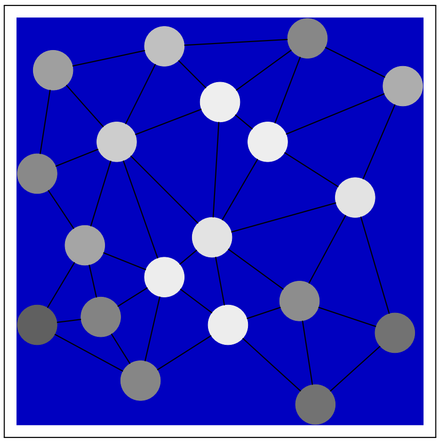

(c) Result of the segmentation on (b).

Figure 12: Results of the segmentation chain from Algorithm 5.2 on a regular 2D image.


(a) Input.


(b) Morphological gradient of (a).

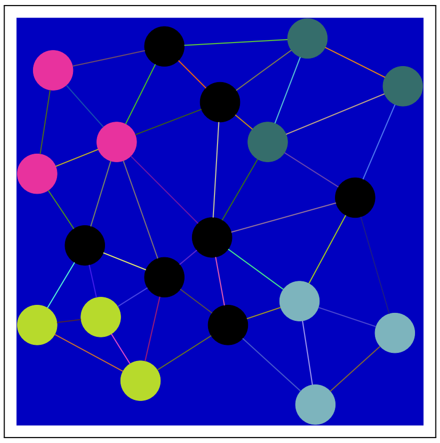
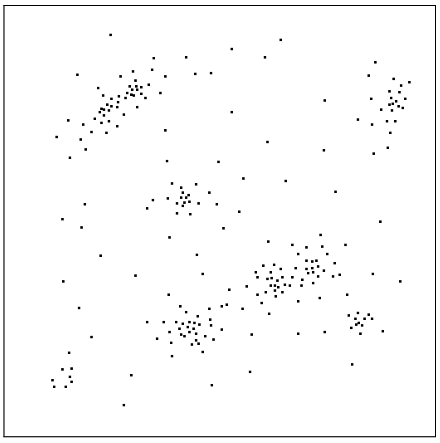
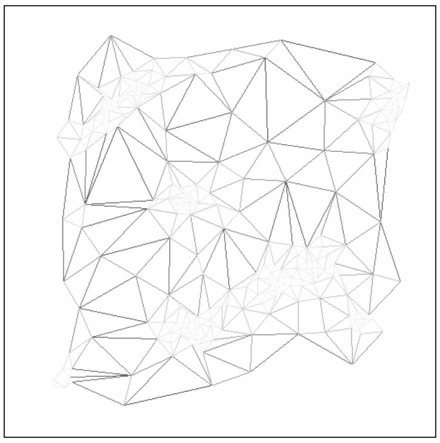

(c) Result of the segmentation on (b).

Figure 13: Results of the segmentation chain from Algorithm 5.2 on a vertex-valued graph-based image.


(a) Vertices of a graph.


(b) Graph obtained by the triangulation of (a), the edges of which are valued with the length.

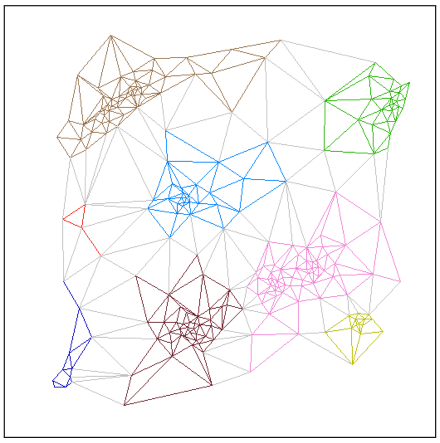
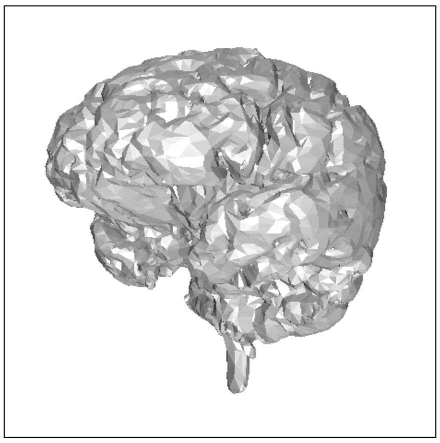
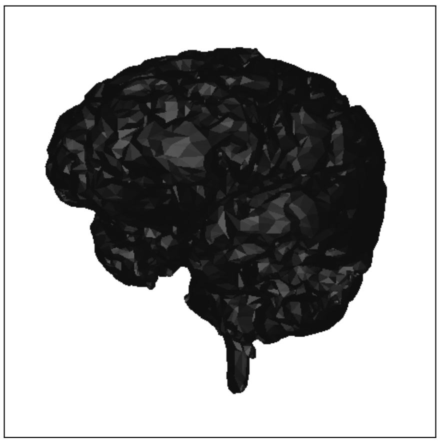

(c) Result of the segmentation on (b).

Figure 14: Results of the segmentation chain from Algorithm 5.2 on an edge-valued graph-based image.


(a) Triangular mesh surface seen as a simplicial complex-based image and used as input.


(b) Maximum surface curvature computed on the edges of (a).

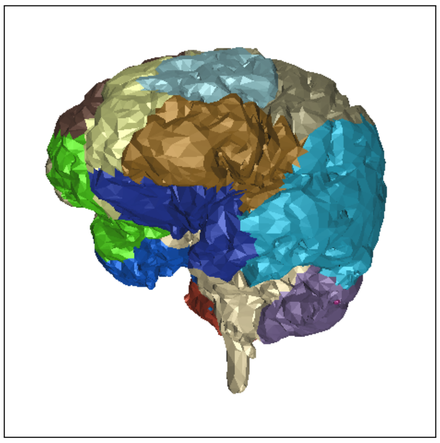

(c) Result of the segmentation on (b) (extended to triangles).

Figure 15: Results of the segmentation chain from Algorithm 5.2 on a simplicial complex-based image.

5.2 A GENERIC HOMOTOPIC THINNING IMPLEMENTATION

In this second example, we present a thinning algorithm used to compute skeletons of binary images. Such an operation can be obtained by the removal of *simple points* (or simple point sites in the Milena parlance) using a breadth-first thinning strategy [20]. A point of an object is said to be simple if its deletion does not change the topology of the object.

Algorithm 5.3 shows a generic implementation of a breadth-first thinning, taking as input an image (`image_`), an adjacency relation (the neighborhood `nbh_`) and three functors: one to characterize a simple point (`is_simple_`), a second to remove such a point (`detach_`) and a last one to express a constraint (`constraint_`), i.e. to prevent the deletion of certain points.

5.2.1 *Simple Point Characterization Implementation*

There are local characterizations of simple points in 2D, 3D and 4D, which can lead to Look-Up Table (LUT) based implementations [36]. However, since the number of configurations of simple and non-simple points in $\mathbb{Z}^d$ is 2^{3^d-1}, this approach can only be used in practice in 2D (256 configurations, requiring a LUT of 32 bytes) and possibly in 3D (67,108,864 configurations, requiring a LUT of 8 megabytes). The 4D case exhibits 2^{80} configurations, which is intractable using a LUT, as it would need 128 zettabytes (128 billions of terabytes) of memory. Couprie and Bertrand have proposed a more general framework for checking for simple points using cell complexes [36] and the collapse operation.

Complexes support a topology-preserving transformation called *collapse*. An *elementary collapse* removes a free pair of faces of a complex, like the square face f_1 and its top edge e_1, or the edge e_2 and its top vertex v, in Figure 6. The pair (f_2, e_3) cannot be removed, since e_3 also belongs to f_3. Successive elementary collapses form a *collapse sequence* that can be used to remove simple points. Collapse-based implementations of simple-point deletion can be always be used in 2D, 3D and 4D, though they are less efficient than their LUT-based counterparts. On the other hand, they provide some genericity as the collapse operation can have a single generic implementation on complexes regardless of their structure.

5.2.2 *Illustrations*

Using this generic approach, Algorithm 5.3 can be used to compute skeletons of various input images.

Algorithm 5.3: Generic implementation of breadth-first thinning.

```
template <typename I, typename N, typename F, typename G,
          typename H>
mln_concrete(I)
breadth_first_thinning(const Image<I>& input_,
                       const Neighborhood<N>& nbh_,
                       Function_v2b<F>& is_simple_,
                       G& detach,
                       const Function_v2b<H>& constraint_)
{
  // Convert arguments to their exact types and
  // initialize ‘output’.
  const I& input = exact(input_);
  const N& nbh = exact(nbh_);
  F& is_simple = exact(is_simple_);
  const H& constraint = exact(constraint_);
  mln_concrete(I) output = duplicate(input);
  // Bind ‘output’ to ‘is_simple’.
  is_simple.set_image(output);
  // Bind ‘output’ to ‘detach’.
  detach.set_image(output);

  // Step #1.  Initialize a FIFO queue with simple points
  // candidates.
  typedef mln_psite(I) psite;
  p_queue_fast<psite> queue;
  // Image ‘in_queue’ records whether a point site is in
  // the queue.
  mln_ch_value(I, bool) in_queue;
  initialize(in_queue, input);
  data::fill(in_queue, false);
  mln_piter(I) p(output.domain());
  for_all(p)
    if (output(p) && constraint(p) && is_simple(p))
    {
      // Push ‘p’.
      queue.push(p);
      in_queue(p) = true;
    }

  // Step #2.  Process the FIFO queue until it is empty.
  while (!queue.is_empty())
  {
    psite p = queue.pop_front();
    in_queue(p) = false;
    if (output(p) && constraint(p) && is_simple(p))
    {
      // ‘p’ is simple and passes the constraint; detach it.
      detach(p);
      // Process the neighbors of ‘p’.
      mln_niter(N) n(nbh, p);
      for_all(n)
        if (output.domain().has(n)
            && output(n) && constraint(n) && is_simple(n)
            && !in_queue(n))
        {
          // Push ‘n’.
          queue.push(n);
          in_queue(n) = true;
        }
    }
  }
  return output;
}
```

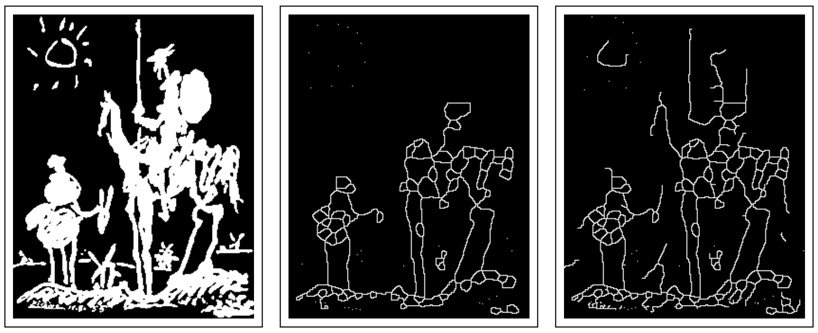

(a) 2D binary image. (b) Skeleton of (a) with no constraint (c) Skeleton of (a) where end points of the initial image have been preserved.

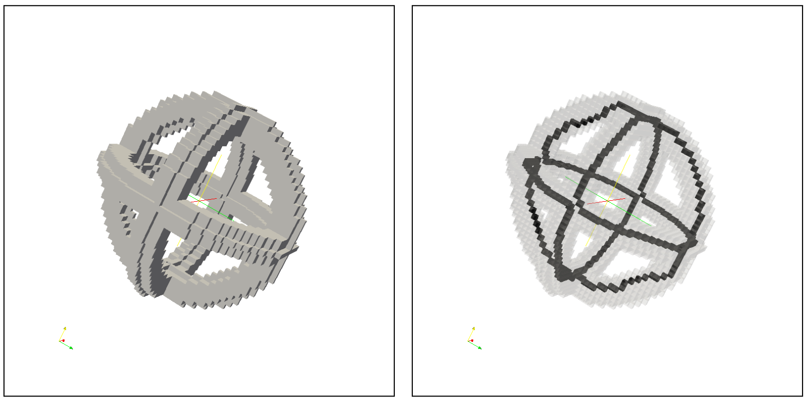
Figure 16: Computation of skeletons from a 2D binary regular image.

Skeleton of a 2D Binary Image

Our first illustration uses a classical 2D binary image built on a square grid (Figure 16a). The following lines produces the result shown on Figure 16b.

```
typedef image2d<bool> I;
typedef neighb2d N;
I output =
  breadth_first_thinning
    (input,
     c4(),
     is_simple_point2d<I, N>(c4(), c8()),
     detach_point<I>(),
     no_constraint());
```

I and N are introduced as aliases of the image and neighborhood types for convenience. The breadth_first_thinning algorithm is called with five arguments, as expected. The first two ones are the input image and the (4-connectivity) neighborhood used in the algorithm. The last three ones are the functors governing the behavior of the thinning operator. The call is_simple_point2d<I, N>(c4(), c8()) creates a simple point predicate based on the computation of the 2D connectivity numbers [20] associated with the 4-connectivity for the foreground and the 8-connectivity for the background. To compute these numbers efficiently, is_simple_point2d uses a LUT containing all the possible configurations in the 8-connectivity neighborhood of a pixel. detach_point<I> is a simple functor removing a pixel by giving it the value "false". Finally, no_constraint is an empty functor representing a lack of constraint.



(a) 3D binary image.

(b) Skeleton of (a) with no constraint (dark voxels) superimposed on the initial image (light voxels).

Figure 17: Computation of the skeleton of a 3D binary regular image.

We also present a variation of the previous example where the fifth argument passed to the function is an actual constraint, preserving all end points of the initial image (see Figure 16c). This result is obtained by invoking `is_not_end_point<I, N>(c4(), input)` in the following lines. This call creates a predicate characterizing end points by their 2D foreground 4-connectivity numbers.

```
I output_with_end_points =
  breadth_first_thinning
    (input,
     c4(),
     is_simple_point2d<I, N>(c4(), c8()),
     detach_point<I>(),
     is_not_end_point<I, N>(c4(), input));
```

Skeleton of a 3D Binary Image

This second example in 3D is similar to the previous one in 2D. The domain of the image is a box on a cubical grid; the 26- and the 6-connectivity are respectively used for the foreground and the background. The output on Figure 17b is obtained from the 3D volume shown in Figure 17a with the following lines.

```
typedef image3d<bool> I;
typedef neighb3d N;
I output =
  breadth_first_thinning
    (input,
     c26(),
     is_simple_point3d<I, N>(c26(), c6()),
```

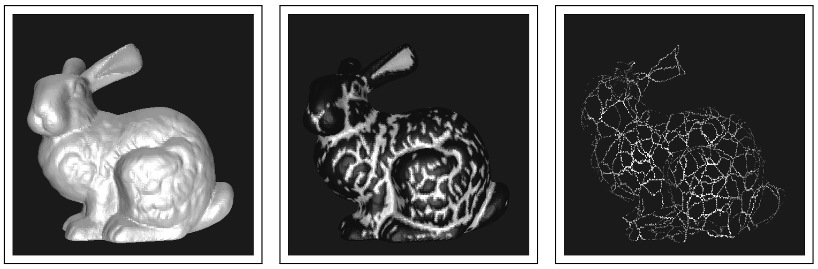

(a) Triangle mesh surface. (b) Surface curvature. (c) Surface skeleton.

Figure 18: Computation of a skeleton using breadth-first thinning. The triangle mesh surface (a) (35,286 vertices and 70,568 triangles) is seen as a simplicial 2-complex. The image of curvature (b) is computed on the edges of the mesh, and simplified using an area opening filter. All curvature regional minima are then removed from the mesh, and the skeleton (c) is obtained with Algorithm 5.3 using the collapse operation.

```
        detach_point<I>(),
        no_constraint());
```

The only real difference with the previous example is the use of the functor `is_simple_point3d`. The default implementation of this predicate uses an on-the-fly computation of 3D connectivity numbers. We have also implemented a version based on a precomputed LUT which showed significant speed-up improvements.

Thick Skeleton of a 3D Mesh Surface

In this third example, we manipulate discrete mesh surfaces composed of triangles. The input of the thinning operator is a surface containing "holes", obtained from the mesh shown in Figure 18a by removing triangles located in regional minima of the surface's curvature [119] (darkest areas of Figure 18b). The result presented in Figure 18c is obtained with the following lines. Types are not shown to make this code more readable.

```
output =
  breadth_first_thinning(input,
                         nbh,
                         is_simple_triangle,
                         detach_triangle,
                         no_constraint());
```

In the previous code, `input` is a triangle-mesh surface represented by an image built on a simplicial 2-complex and `nbh` represents an adjacency relationship between triangles sharing a common edge. Function objects `is_simple_triangle` and `detach_triangle` are

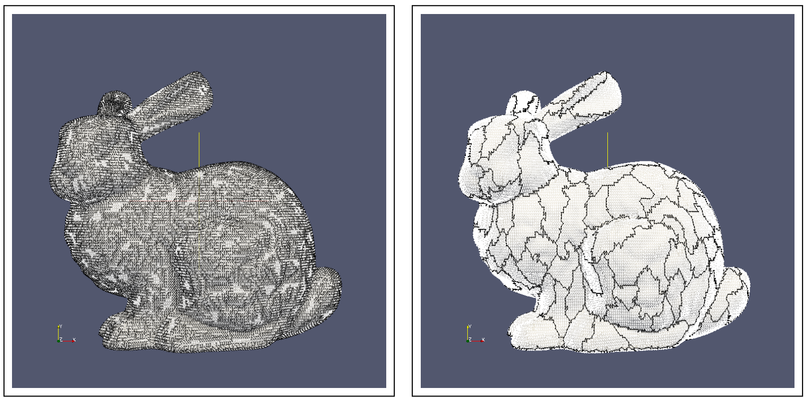

(a) Ultimate 2-collapse of the input image used in Figure 18.
(b) Ultimate 1-collapse of (a)

Figure 19: Thin skeleton obtain by 2- and 1-collapse.

operations compatible with `input`'s type; they are generic routines based on the collapse operation mentioned in Section 5.2.1, working with any complex-based binary image.

The `input` image is constructed so that the sites browsed by the `for_all` loops in Algorithm 5.3 are only 2-faces (triangles), while preserving access to values at 1- and 0-faces. Thus, even though they are passed 2-faces, `is_simple_triangle` and `detach_triangle` are able to inspect the adjacent 1- and 0-faces and determine whether and how a triangle can be completely detached from the surface through a collapse sequence.

The resulting skeleton is said to be *thick*, since it is composed of triangles connected by a common edge. The corresponding complex is said to be pure, as it does not contain isolated 1- or 0-faces (that are not part of a 2-face).

Thin Skeleton of a 3D Mesh Surface

To obtain a *thin* skeleton, we can use a strategy based on successive n-collapse operations, with n decreasing [37]. From the input of the previous example, we can obtain a ultimate 2-collapse by removing all simple pairs composed of a 2- and a 1-face (a triangle and an adjacent edge). The following lines compute such an ultimate 2-collapse. The iteration on `input`'s domain is still limited to triangles (2-faces).

```
collapse2 =
  breadth_first_thinning
    (input,
     nbh,
     is_triangle_in_simple_pair,
     detach_triangle_in_simple_pair,
     no_constraint());
```

Functor is_triangle_in_simple_pair checks whether a given triangle is part of a simple pair, and if so detach_triangle_-in_simple_pair is used to remove the pair. Thinning the initial surface with this simple point (site) definition produces a mesh free of 2-faces (triangles), as shown in Figure 19a.

From this first skeleton, we can compute an ultimate 1-collapse, by removing all simple pairs composed of an edge (1-face) and a vertex (0-face). This skeleton is produced with the following code, where input2 is an image created from collapse2, and for which the domain of has been set to the edges of the complex, (instead of the triangles).

```
collapse1 =
  breadth_first_thinning
    (input2,
     nbh,
     is_edge_in_simple_pair,
     detach_edge_in_simple_pair,
     no_constraint());
```

Here is_edge_in_simple_pair and detach_edge_in_simple_pair respectively test and remove an edge along with a vertex that form a simple pair. The result is a simplified skeleton, with no isolated branches, as the lack of constraint (no_constraint) does not preserve them.

Note that in both cases, the neighborhood object nbh is the same, as it represents the adjacency of two n-faces connected by a common adjacent $(n-1)$-face. In the case of the 2-collapse, the neighborhood of a site (triangle) is the set of adjacent triangles connected by an edge, while in the case of the 1-collapse, the neighborhood of a site (edge) is the set of adjacent edges connected by a vertex.

6 CONCLUSIONS

This last part summarizes the contributions of this thesis. We present some reflexions on the ideas presented in this work. Finally we propose perspectives for future research works.

6.1 CONTRIBUTIONS

This thesis proposes an approach to design and build IP software in a generic manner, in order to create reusable software. We illustrated our point with examples coming from the Olena platform and in particular from the Milena library.

6.1.1 *A Programming Paradigm for Scientific Computing*

The first steps towards a generic architecture are exposed in the first two chapters, which detailed the design and implementation framework of our proposal. Chapter 2 contains a general presentation of the GP paradigm, and why we think it is an ideal context to develop reusable and efficient software. In the language chosen to implement our solution, C++, GP is realized through the `template` keyword. Templates incidentally enable another programming paradigm useful in the context of efficient scientific software, static metaprogramming. This technique consists in diverting templates to use them as a new language within C++ to express compile-time programs (or metaprograms) "executed" by the compiler. Static metaprogramming applications include the computation of values at compile-time, additional user-defined verifications performed by the compiler (static assertions) and functions on types, all of which are later effectively used in the Milena library.

Chapter 3 contains a definition of a new programming paradigm, SCOOP, mixing the benefits of GP and OOP: high-level programming based on abstractions, orthogonal development of data structures and algorithms, and preservation of efficiency. The SCOOP approach also encourages designers to describe characteristics of data structures belonging to a given abstraction as static properties. Thanks to metaprogramming, algorithms can query the properties of their inputs' types at compile-time in order to express static preconditions or to select an alternative implementation that is known to perform better for input types satisfying certain properties.

We also presented a strategy enabled by SCOOP to create lightweight transformations of data such as images or values, called *morphers*. Morphers are especially useful to change the behavior of an algorithm "externally", by applying a transformation to their input, with no modification of the data structure or the algorithm. Morphers are not specific to a data type in particular, and are therefore orthogonal to them, thus increasing the reusability and the expressive power of the framework.

These qualities makes SCOOP a framework really suited to scientific computing and to IP in particular. The paradigm presented in this thesis is a simplified version of a previous proposal [65], which is itself based on a first version [32] used in previous releases of the Olena project [52].

6.1.2 *A Proposal for an Architecture for Generic Image Processing*

Chapter 4 proposes an architecture for generic IP centered on a generic core library. We advocated an orthogonal decomposition of IP software in data (image) structures, values and algorithms, so as to minimize redundancy and maximize reusability. We showed that GP is a good choice to organize this software decomposition. We also justified our choice of SCOOP and C++ to implement our generic core library, because of the available features (compile-time checking, zero run-time overhead, concept- and property-based polymorphisms, morphers) and for practical reasons.

We then introduced a set of IP-related abstractions called concepts, representing essential notions of the domain, starting with a very general definition of an "image". Concrete entities conforming to these concepts, called models, collaborate through the GP notion of associated types, thus exhibiting a first aspect of software decomposition and separation of concerns. More generally, our proposal provides four orthogonal axes regarding the application of generic image processing algorithms:

- the data (image) structure axis,
- the value type axis,
- the algorithm axis,
- and the morpher axis (axes), as one morpher (or more) may be applied to an image before being processed.

Given a data structure, a value type, an algorithm and (possibly) a (sequence of) morpher(s), one may easily create new use cases by changing one of these elements without touching the others.

6.1.3 *Design and Use of Non-Classical Data in Image Processing*

Several actual data structures are also presented in Chapter 4. As long as they respect the requirements of an algorithm, the examples shown are compatible with a single generic definition of it, therefore demonstrating the sustainability of our design. These data structures include less common entities such as vertex- and edge-valued graphs and cell complexes. Chapter 5 showed examples of applications for these various data types.

Our experience has shown that thanks to the generic design of the framework, it is possible to integrate new image-related data structures in a relatively fast and easy way—including collaborating types such as iterators and windows—and such that these new data types are compatible with existing algorithms.

6.1.4 *Addressing Efficiency Issues using Generic Optimizations*

We also proposed a non-specific approach to implement fast variants of existing algorithms for input types having interesting properties. For such an optimized routine, instead of targeting an image type in particular, this strategy circumscribes the set of valid input types by enumerating the required properties, thus allowing a whole subset of the input types compatible with the initial generic implementation. This notion of generic optimization adds to the reusability of the framework.

Moreover, data types' properties enable an automatic selection mechanism of the best estimated variant at compile-time, called property-based overloading. The selection algorithm is written as a static dispatch mechanism.

6.1.5 *Software Contributions*

The ideas presented here have been implemented in the Olena platform, and in particular in Milena, its core library. We can consider Olena as a software contribution demonstrating the practicability of the approach presented in this thesis. Olena version 1.0 has been released during the work of this thesis in 2009. Olena 2.0 has been released at the end of the thesis in 2011, and continues to be developed. This thesis presents some of the important design features of these versions of the project, that was started more than ten years ago.

Milena has served as a basis to achieve several tasks. In particular, the library has been used to implement various segmentation, registration, and evaluation methods in the context of a joint effort with the Institut Gustave-Roussy hospital. We have also experimented the segmentation and skeletonization of digital models of statues, represented as triangular meshes with existing mathematical morphology and discrete geometry tools (watershed transform, connected filters, homotopic thinnings). A module dedicated to Document Image Analysis (DIA) based on Milena has been recently developed within the Olena platform during the SCRIBO project [93], and has been packaged with the 2.0 release of the platform.

Olena is developed following open source principles. The whole source code changes are available on the Web, as well as development tools (problem reports, documentation, mailing lists, etc.). The platform is distributed as Free Software under the GNU General Public License (GNU GPL). We believe Free Software is a good strategy to deliver *reproducible research* results [29, 58]. In addition to providing the source code of the platform, we also gradually include into the code base methods and examples used in publications related to the project, so that readers may themselves reproduce the results of the papers, and compare our approaches with other methods.

6.2 REFLEXIONS

The programming idioms proposed in this thesis and implemented in Milena—such as inheritance-based concept checking; point-like iterators targeting a domain, window or neighborhood; morpher-based control of an algorithm's behavior; and property-based algorithm selection—somehow constitute a new "language" for IP hosted by C++. The eventuality of designing a whole new language for IP has been brought up several times during the design and implementation of the Olena project. The main motivation to create a Domain Specific Language (DSL) to implement the ideas of the generic IP framework proposed in this thesis was to provide a better and simpler tool to users. Although it is a powerful and efficient language, C++ is also complex and not as user-friendly as other popular languages such as Java, C#, Python, Ruby or languages of the ML family (Standard ML, Objective Caml, Haskell, etc.). Its syntax, based on C's, not only inherits the difficult idiosyncrasies of its ancestor but also adds its own constructs often characterized by lengthy statements.

However, we believe like Stroustrup that the cost of designing, implementing and maintaining a new programming language to address a specific need is almost always too high in comparison to the benefits brought by this language [131]. A library written in a widespread general-purpose language is a much more affordable and effective solution to provide domain-specific elements to the programmer. In the same spirit, Veldhuizen and Dennis propose the concept of *active libraries* [148] as an alternative to language extensions, DSLs and object-oriented solution that are hard to optimize. Active libraries provide an abstraction-based design along with the means to optimize them. They are based on techniques such as component generation (e.g. through GP), reflection and meta-level processing, run-time code generation, partial evaluation, multilevel languages or extensible programming tools. Examples of such libraries include the Blitz++ generic array library [145], the POOMA parallel physics library [89], the Matrix Template Library (MTL) [126] and the FFTW (Fastest Fourier Transform in the West) library [60]. By its design, Milena can also be considered as an active library.

Instead of designing and implementing a complete DSL dedicated to Milena, including a whole language tool set (compiler, debugger, profiler, libraries, etc.), several paths have been considered to provide an *alternative syntax* to the C++ library. This approach is also recommended by Stroustrup [131]. Participants to the project proposed several solutions. A first approach was to extend the C++ language with new constructs implementing Milena idioms, such as concept definition, concept checking, GCRTP constructs, property-based dispatch of algorithms and more generally all static metaprogramming constructs which suffers from an unfriendly and verbose syntax. This language extension would thus provide syntactic sugar for the most complex parts of the framework. This proposal was not based on the modification of an existing C++ compiler but on a *program transformation* strategy. An extended C++ program would be processed by a front-end program performing lexical and syntactic analyses (scanning and parsing), building an annotated Abstract Syntax Tree (AST) representing the input program, rewriting this tree by translating elements part of the extension into standard C++ constructs, and finally turning this new AST into a program using the concrete syntax of standard C++. Annotations of the initial AST would serve to recreate a layout of the code as close as possible to the input's. This effort was conducted within the Transformers project [101, 24, 42], based on the Stratego/XT program transformation platform [28]. Alas, because of remaining complex issues regarding C++ syntax disambiguation, the project has not been able to fully handle this C++ extension proposal yet. Other attempts to provide a more user-friendly syntax for Milena included a prototype front-end written in Ruby, and a small language dedicated to SCOOP-based libraries, called SCOOL (Static C++ Object-Oriented Language). This last proposal used ideas similar to the Transformers project: the SCOOL compiler, implemented in the Stratego language, was actually a front end transforming its input into equivalent an C++ program.

The latest ISO C++ standard published in 2011 [87] also contains many interesting additions addressing language issues arising in Milena. To name a few: a new meaning for the `auto` keyword providing simple type inference that will replace long types in templates (already shortened in Milena with the use of macros); r-value references solving the issue of non-const temporary variable passed as arguments, lambda expressions providing anonymous functions in a more concise manner than functors; inherited constructors factoring redundant parts of class hierarchies; and template aliases implementing the service of "template `typedefs`".

We conclude this section by a remark on a current trend of rich run-time environments. Scientific software has been traditionally implemented with compiled language such as Fortran, C or more recently C++ because of the efficiency of the code they produce. However, more and more relatively recent languages, either interpreted (such as Python, Ruby, Tcl, or JavaScript) or running as bytecode in a virtual machine (Java, C#) are used to create scientific applications. They handle the issue of efficiency either by delegating intensive computation tasks to a third party component (e.g. a Fortran or C library) or by using powerful techniques enabled by a rich run-time environment (interpreter, virtual machine): run-time introspection and reflection, Just-In-Time (JIT) compiling and optimization. Even if compiled languages are still considered the best option to produce efficient scientific applications, languages with dynamic features might offer interesting possibilities in the future. The path taken by Milena does not offer so much run-time services, because compile-time efficiency has been considered a priority. However, we try to regain some of the flexibility of dynamic languages through our dynamic-static "bridge" proposal, explained in the following section.

6.3 PERSPECTIVES

This last section develops perspectives for the evolution of the work presented in this thesis and for the Olena project.

6.3.1 *A Dynamic-Static Bridge*

IP projects centered on a C and C++ library sometimes offer an additional layer on top of the library to make this core available outside the originating language. This layer often serves to expose the library to other programming languages, and in particular to dynamic languages such as Python, Ruby, Perl or Tcl; or to encapsulate some of the routines as command line programs. For example VIGRA, ITK, Morph-M, Yayi and Pink [53] expose the contents of their core library through a Python interface. ImageMagick, GraphicsMagick, ImLib3D and CImg offer an interactive use through command-line programs. For CImg, this tool, called G'MIC also serves as a binding for the GIMP (GNU Image Manipulation Program).

These additional User Interfaces (UIs) enlarge the spectrum of users and uses cases targeted by their corresponding project. Command Line Interface (CLI) tools are very convenient to quickly run simple tasks, in comparison to the work required by the library approach, which imposes to write a program (even if it is a small one), compile it, and possibly debug it, before actually running it. CLI tools are also useful to apply the same operation to a list of image files in a batch processing fashion.

Being able to access a C or C++ library from Python, Ruby, Perl or other dynamic languages is useful in many ways. Firstly, these languages often provide an interactive user interface or shell, which is ideal to discover and experiment a new library. Secondly, these languages are simpler and more user-friendly than C and C++. New users can learn their most important features in a couple of hours and be able to write small programs rapidly. Understanding and correcting errors in dynamic programs is also faster and simpler than with compiled languages. Thirdly, these languages offer many ready-to-use libraries. The SciPy initiative [11, 10] shows for example an effort to bring a scientific stack to the Python language to create a scientific environment competing with products such as MATLAB. Finally, dynamic languages are ideal to bring together various software components. In some use cases, languages such a Python or Ruby can be used only to provide a flexible "glue" between efficient components written in C or C++.

Providing such additional UIs requires additional efforts though. Some projects contain a tool to generate command-line programs or bindings for a dynamic languages semi-automatically. This mechanism is usually based on two components: a description of the elements (routines, classes) of the core library to be "wrapped" into an new interface and an interface generator. Developers of the library are expected to provide the former, while the latter is usually a generic component. For example the IP operators of the ImLib3D library are accompanied by XML descriptions that are used to generate the corresponding command line tools [25, Appendix B]. CImg [137] also has its own mechanism, G'MIC, to present algorithms to the CLI. But most projects usually depend on a third party generator. For instance VIGRA, Morph-M, Yayi and Pink use the Boost.Python library [136] to generate Python bindings, while ITK uses the Simplified Wrapper and Interface Generator (SWIG) [19]. The approach chosen by SWIG is interesting, as it proposes to directly use C or C++ header files as a description of the code to be wrapped, while other systems (including Boost.Python) requires this description to be manually written. Another advantage of SWIG is that it can target many languages, including Python, Ruby, Perl, PHP, Java, C#, Tcl, Objective Caml, R and Octave, D and Go.

Nevertheless, in the context of generic code based on C++ templates, the wrapping techniques listed previously are limited. Indeed tools such Boost.Python or SWIG depend on *compiled code*. The glue code they generate is a thin layer delegating the actual work to elements of the core C or C++ library. Therefore, they expect to find this library as a (set of) compiled entitie(s)[1]. However we have seen in Section 2.1.4 (p. 34) that C++ generic libraries are only composed of *non-compiled* code, as they are made of (non instantiated) template classes and functions, thus preventing a classic wrapping process such as the one performed by Boost.Python or SWIG to be used as-is.

A technique that is often used to nevertheless attempt to wrap generic libraries is to manually instantiate the templates with a chosen set of effective parameters and compile them, and then to wrap these explicit instantiations in a traditional way. In other words, the library is specialized once and for all for all the use cases deemed useful. Although this approach does not raise technical difficulties, it suffers from an important limitation: the obtained code is no longer generic. The set of data structures and algorithms available, e.g., in a generated Python interface is indeed limited to the template specializations instantiated explicitly beforehand. Despite this limitation, the Olena project has provided some Python[2] and Ruby[3] bindings using this paradigm of *instantiated genericity*. This approach is depicted on the left part of Figure 20.

To overcome these limitations, we propose another solution based on a small component, not depending on the target language, and providing the service of a *dynamic-static bridge*. This possibility has been mentioned since the early days of the Olena project [47]. The idea of this component is to create a very general interface on top of a generic library, showing only opaque data types such as `var`, `function`, `method`, `ctor` representing respectively a variable, a function, a method and a constructor. These high-level abstract elements are compiled in a library. At run-time, these types can be instantiated and used as a *proxies* representing an actual entity. The power of the dynamic-static bridge is that the underlying actual object can be a pointer to an entity of a type that does not exist *yet*. In this case, the system simply tries to create the type on-the-fly at run time. This operation requires both the name of the entity and the location of its definition so that an interface code may be later created,

1 Depending on the platform, these compiled libraries bear a file name extension such as '`.a`', '`.so`', '`.lib`', '`.dll`', '`.dylib`'.

2 Since Olena 0.7.

3 Since Olena 1.0.

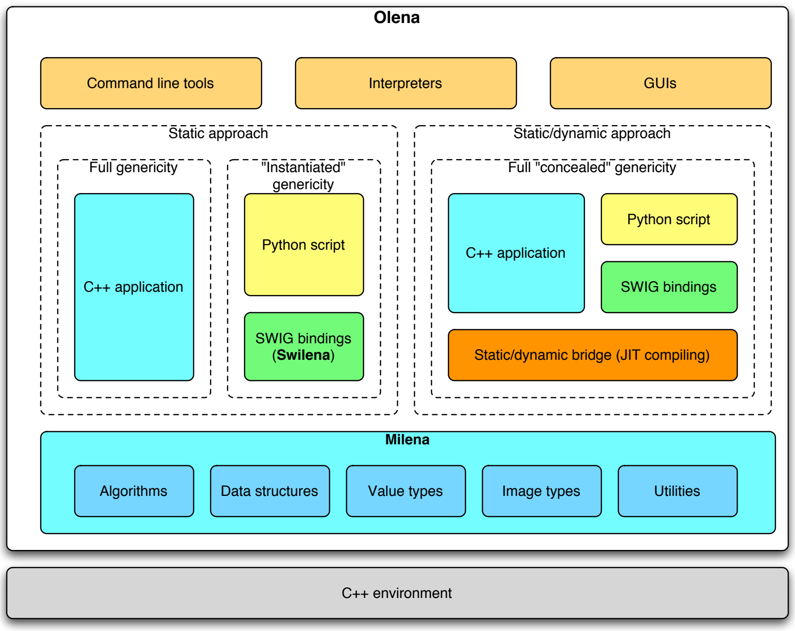


Figure 20: Overview of the Olena Platform

instantiated, compiled and dynamically loaded into the current program. These two pieces of information are passed as character strings.

For instance, let us consider the following template routine from Milena, located in the ‘`mln/data/fill.hh`’ header of the library:

```
template <typename I, typename D>
void fill(Image<I>& ima, const D& data);
```

This template can be represented by an object `fill` of type `fun` in our dynamic-static bridge with the following two instructions.

```
include("mln/data/fill.hh");
fun fill("mln::data::fill");
```

Note that `include` is a routine that has nothing to do with the `#include` C++ preprocessor directive. So far, these statements only *declared* the existence of the template `mln::data::fill`; no code has been instantiated nor compiled, since `mln::data::fill` is not a valid function name, as it is not fully specialized (see Section 2.2.3, p. 38). Constructors of data structures may also be declared to the system similarly, this time as instances of the `ctor` type.

```
include("mln/core/image/image2d.hh");
ctor mk_image2d_int("mln::image2d<int>");
```

The previous lines create an `mk_image2d_int` representing the construction of an instance of the 2D integer image type `mln::image2d<int>`. Here also, no code is instantiated nor compiled.

`mk_image2d_int` is a *functor* (see p. 44), containing a generic `operator()` methods for any number of arguments of any type. Invoking such an operator triggers the wrapping process:

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

The previous lines execute the following actions:

1. A small C++ function containing a call to the constructor with the arguments passed to `mk_image2d_int` (`mln::image2d<int>::image2d(int, int)`) is created.

2. This function is compiled using a Just-In-Time (JIT) compiling mechanism, thus automatically instantiating the `mln::image2d<I>` template class with the effective parameter `I = int`, as well as the constructor mentioned previously.

3. A small dynamic module[4] is created for this compiled function.

4. The above module is loaded dynamically into the program currently running, and the address of the function created in step 1 is stored as a pointer into an attribute of the proxy (`mk_image2d_int` object).

5. The proxy calls the underlying freshly loaded function by using the previous pointer, and passes the arguments it has received through the initial call of its `operator()` (here, the two integers `3, 3`).

6. The value returned by the previous call is then passed to the constructor of the `ima` object of type `var`. `ima` is also a proxy, but this time for a variable. Its constructor stores the values passed as argument in an attribute.

If during this process any error occurs, it is reported as an exception.

The previous example illustrates the philosophy of the dynamic-static bridge. The top-level (dynamic) layer, delimited by the "visible" code shown in the example only manipulates abstract and opaque objects (instances of `var`, `ctor`, `fun`, etc.) and is therefore very simple. For instance, no template code is visible[5]. This interface can be used to simplify the manipulation of a template library, and may be wrapped by a classic tool such as Boost.Python or SWIG. The bottom-level (static) layer comprises the "hidden" code produced as a side effect of the execution of the top-level

4 A dynamic module is similar to a dynamic or shared library on most modern platforms.

5 With the exception of the instantiated type `mln::image2d<int>` mentioned in a character string.

layer. Here, it includes the instantiated and compiled constructor of `mln::image2d<int>` as well as the actual image object built by the call to this constructor and stored in the proxy object `var`.

Despite the actual types of the manipulated data are hidden in the bottom-level layer of the system, the proxy objects can interact at the upper level and delegate the actual actions to the actual functions and objects in the lower level. Consider for instance the following statement:

```
fill(ima, 0);
```

The previous line instantiates the `mln::data::fill<I, D>` routine "contained" in the proxy object `fill` with the effective parameters `I = mln::image2d<int>` and `D = int`, respectively conveyed by the first and the second arguments passed to `fill`'s `operator()`. The routine is then compiled and run, taking as argument the image object stored in the proxy `ima` and the integer value `0`.

It is interesting to note that the dynamic-static bridge presented here shares some similarity with Object-Oriented Programming (OOP)[6]: the code is manipulated through abstractions (expressed as abstract classes in OOP) and the actual execution is delegated to hidden implementations (expressed as concrete subclasses in OOP). Both mechanisms are dynamic: they do not require an "explicit" compiling step to handle new cases, contrary to GP (see Section 2.5.2, p. 65). Thus, the dynamic-static bridge solution is more flexible than solely GP. For instance it can be wrapped like any classic compiled library to produce Python bindings or be linked into a third-party visual programming GUI. From the performance point of view, the dynamic-static bridge induces some run-time penalties caused by the JIT generation, instantiation, compilation and dynamic loading of code, as well as dynamic type conversions. However, most of these costs can be amortized by the use of a cache mechanism. Once a particular instantiation of a generic type or algorithm has been used, this penalty becomes negligible.

We have implemented a prototype based on these ideas in the Olena platform, based on a previous work by Nicolas Pouillard and Damien Thivolle. We have been able to use this component to create truly generic SWIG Python bindings on top of Milena, preserving the genericity of the library while preserving most of its performances. This strategy is represented on the right-hand part of Figure 20.

The two paradigms used in this case, SCOOP and the dynamic-static bridge, show an interesting characteristic of the Olena platform:

6 Besides, let us note that this dynamic-static bridge is implemented using OOP (and GP).

- On the one hand static metaprogramming is used within SCOOP to perform computations at compile time, including manipulations of types, therefore *executing programs at compile-time*.

- On the other hand, the dynamic-static bridge uses JIT compiling techniques to use template code at run-time, thus *compiling programs at run-time*.

This approach demonstrates unusual but effective uses of the C++ language. In a sense, this combination shares some traits with dynamic solutions mentioned at the end of Section 6.2.

Evolutions and improvements of this dynamic-static bridge prototype include the study of compiling toolkits such as LLVM [92] to replace our current solution using an ad hoc JIT compiling component based on the g++ (GCC) compiler.

Furthermore we mentioned in Section 4.3 (p. 120) that developments conducted with high-level tools (e.g. through the Python interface of Milena) that have matured should at some point be reimplemented in lower-level tools (Milena). This effort would improve their genericity, efficiency and reusability. An algorithm written in Python and reimplemented in C++ could thus be made available to other high-level interfaces, such as a Perl or Ruby interpreter. However, this work has to be done by hand currently. Automating or semi-automating this translation would improve the cooperation between users of high-level interfaces and maintainers of the lower-level core library.

6.3.2 *A New Implementation Language?*

We have explained our choice of the C++ programming language in Section 4.2.2 (p. 118). It is however interesting to consider other languages to host Milena and an implementation of the SCOOP paradigm. So far, we have looked at the following languages.

D The D programming language [13] can be described as a cleanly designed successor to C++. It shares many traits with its parent: general-purpose compiled language targeting efficient code generation, multiple paradigm programming (procedural programming, OOP, GP, Functional Programming (FP), metaprogramming), C-style syntax. The syntax and semantics of D are however simpler than those of C++.

D would be a interesting candidate for a possible reimplementation of Olena. SCOOP could probably we adapted for this language, since we have been able to implement the GCRTP idiom (see Section 3.2, p. 79) in D. However, the language does not feature multiple inheritance of (implementation) classes, which is used in several places in Milena.

HASKELL Among the languages supporting the GP paradigm, Haskell stands as one of the best candidates [63]. Haskell is a compiled, general-purpose, purely functional language with strong static typing and using lazy evaluation. Because it is not an imperative nor an object-oriented language, it would probably be difficult to directly adapt the ideas presented in this thesis to Haskell. However, some notions have almost equivalent constructs in that language. For example, concepts can be transposed into *type classes*, used to impose constraints on their instances (models).

None of these languages can compete with C++ regarding general qualities such as its widespread ability, the fact that is taught is many CS curricula, and the large amount of libraries, tools and documentation for the language. Moreover, and despite their advantages, we do not know yet whether the architecture that we propose can be transposed in any of them. Whether these languages actually feature all the properties required by our approach is unknown to this day, and would require a thorough experimentation.

6.3.3 *Parallel Computing*

Another research direction to extend our work is related to the integration of parallel computing into our framework. Parallel programming is becoming more an more important these days, because CPUs have reached physical limits preventing a continuous increase in clock frequencies at a reasonable cost. Progresses in sequential computing offered by a single processing unit are thus being held back. For this reason processor manufacturers have developed during the past years products containing two or more independent processing units. These *multi-core* CPUs make up for the majority of computer processors nowadays.

Harnessing this increased "horizontal" computing power is however not automatic nor easy. Parallel computing is a programming paradigm often requiring a complete redesign of algorithms, programs and libraries to bring substantial gains. Various initiatives such as OpenCL [72] or the Intel Threading Building Blocks [116, 77] have been proposed to simplify the development of parallel programs.

In our case, the main difficulty is to accommodate parallel programming idioms with a generic design. This problem shares some similarities with the generic optimization issue presented in Section 4.6.3 (p. 139), where instead of defining a specific solution to address performance problems, the proposed strategy covers a subset of use cases. We should tackle the issue of parallel programming integration in the same way, so as to preserve the orthogonality of obtained algorithms with data structures. The first step towards this evolution is probably to identify data structure properties related to parallel processing capabilities, such as splittable data containers.

6.3.4 *Impact on Image Processing*

The purpose of this thesis is mainly related to research in software design for IP and scientific applications general, as it can be seen from the conclusions of the previous paragraphs. However, the work presented here also open perspectives in the field of IP as well.

Firstly, the notion of algorithm *canvas* (or algorithm pattern) [46] is a powerful paradigm to create *meta-algorithms*. Generic algorithms ensure a reusability of methods across data types. Canvases however, which are encouraged by GP, are much more powerful. They express a whole *class* of algorithms sharing a common structure. For instance, many mathematical morphology operators share a common pattern based on the traversal of an image combined with the browsing of a sliding window (structuring element) and can be implemented as a canvas. Benefits include easier experimentation and addition of new algorithms featuring a similar structure as well as factored optimization efforts.

Secondly, the IP abstractions proposed in this thesis in order to formalize entities of the domain have been defined in an empirical fashion. We should refine this proposal to define these entities more formally, by identifying their properties carefully. This typology would primarily benefit the framework we propose, but also other IP software projects interested in a formal design of image-related concepts.

We are also interested in the production of a catalog of IP data types and algorithms, in the context of an improved algorithmic study disconnected from any implementation framework. Such a work is often remotely related to implementation concerns, as data structures and algorithms are expressed in different ways: in the former case, a general description is expected, while in the later situation, software and hardware constraints usually dictate the form of the expression. In GP however, the representation of algorithms, and to lesser extent, of data types, is close to abstract definitions found in textbooks or catalogs.

Our final point of view pertains to transverse explorations made possible by a generic design. We believe that our approach simplifies the transposition of a technique initially proposed in a given context to another situation, such as another application domain or other data types. Highlighting relevant examples of such transpositions from and for real applications would add value to the reusable software strategy.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications.

- Guillaume Lazzara, Roland Levillain, Thierry Géraud, Yann Jacquelet, Julien Marquegnies, and Arthur Crépin-Leblond. The SCRIBO module of the Olena platform: a free software framework for document image analysis. In *Proceedings of the 11th International Conference on Document Analysis and Recognition (ICDAR)*, Beijing, China, September 2011. International Association for Pattern Recognition (IAPR)

- Roland Levillain, Thierry Géraud, and Laurent Najman. Une approche générique du logiciel pour le traitement d'images préservant les performances. In *Proceedings of the 23rd Symposium on Signal and Image Processing (GRETSI)*, Bordeaux, France, September 2011. In French

- Roland Levillain, Thierry Géraud, and Laurent Najman. Why and how to design a generic and efficient image processing framework: The case of the Milena library. In *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, pages 1941–1944, Hong Kong, September 2010

- Roland Levillain, Thierry Géraud, and Laurent Najman. Writing reusable digital geometry algorithms in a generic image processing framework. In *Proceedings of the Workshop on Applications of Digital Geometry and Mathematical Morphology (WADGMM)*, pages 96–100, Istanbul, Turkey, August 2010. URL `http://mdigest.jrc.ec.europa.eu/wadgmm2010/`

- Roland Levillain, Thierry Géraud, and Laurent Najman. Milena: Write generic morphological algorithms once, run on many kinds of images. In Michael H. F. Wilkinson and Jos B. T. M. Roerdink, editors, *Mathematical Morphology and Its Application to Signal and Image Processing – Proceedings of the Ninth International Symposium on Mathematical Morphology (ISMM)*, volume 5720 of *Lecture Notes in Computer Science*, pages 295–306, Groningen, The Netherlands, August 2009. Springer Berlin / Heidelberg

- Thierry Géraud and Roland Levillain. 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), Paphos, Cyprus, July 2008

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

BIBLIOGRAPHY

[1] DGtal – digital geometry tools and algorithms. http://liris.cnrs.fr/dgtal/.

[2] Gandalf. http://gandalf-library.sourceforge.net/.

[3] ImageJ. http://rsbweb.nih.gov/ij/.

[4] The computer language benchmarks game. http://shootout.alioth.debian.org/.

[5] Python imaging library (PIL). http://www.pythonware.com/products/pil/.

[6] Sage. http://www.sagemath.org/.

[7] Scilab – free open source software for numerical computation. http://www.scilab.org/.

[8] eXtensible Imaging Platform™ (XIP™). http://www.openxip.org/.

[9] Al Stevens interviews Alex Stepanov. *Dr. Dobb's Journal*, March 1995.

[10] Numpy 1.6 reference guide. http://docs.scipy.org/doc/numpy-1.6.0/reference/, May 2011.

[11] Scipy 0.10.0 reference guide. http://docs.scipy.org/doc/scipy-0.10.0/reference/, December 2011.

[12] Adobe. Generic Image Library (GIL). http://opensource.adobe.com/gil.

[13] Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley Professional, 2010. ISBN 978-0321635365.

[14] Jesús Angulo and Jocelyn Chanussot. Color and multivariate images. In Laurent Najman and Hugues Talbot, editors, *Mathematical Morphology—From Theory to Applications*, chapter 11. Wiley-ISTE, July 2010. ISBN 978-1-84821-215-2.

[15] Advanced Computing Laboratory at Los Alamos National Laboratory. POOMA (parallel object-oriented methods and applications). http://acts.nersc.gov/pooma/.

[16] Russell R. Atkinson, Barbara H. Liskov, and Robert W. Scheifler. Aspects of implementing CLU. In *Proceedings of the 1978 annual conference*, ACM '78, pages 123–129, New

York, NY, USA, 1978. ACM. ISBN 0-89791-000-1. doi: http://doi.acm.org/10.1145/800127.804079.

[17] Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley professional computing series. Addison-Wesley, 1999.

[18] John Barton and Lee Nackman. *Scientific and engineering C++*. Addison-Wesley, 1994.

[19] David M. Beazley. SWIG: an easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th conference on USENIX Tcl/Tk Workshop*, volume 4, Berkeley, CA, USA, 1996. USENIX Association.

[20] Gilles Bertrand and Michel Couprie. Transformations topologiques discrètes. In David Coeurjolly, Annick Montanvert, and Jean-Marc Chassery, editors, *Géométrie discrète et images numériques*, chapter 8, pages 187–209. Hermes Sciences Publications, 2007.

[21] Nicolas Beucher and Centre de Morphologie Mathématique. Mamba. http://www.mamba-image.org/.

[22] Serge Beucher and Christian Lantuéjoul. Use of watersheds in contour detection. In *International Workshop on Image Processing: Real-time Edge and Motion Detection/Estimation*, Rennes, France, September 1979.

[23] Dan Bloomberg. Leptonica. http://www.leptonica.com/.

[24] Alexandre Borghi, Valentin David, and Akim Demaille. C-Transformers — A framework to write C program transformations. *ACM Crossroads*, 12(3), Spring 2006. http://www.acm.org/crossroads/xrds12-3/contractc.html.

[25] Marcel Bosc. *Contribution à la détection de changements dans des séquences IRM 3D multimodales*. PhD thesis, Université Louis Pasteur, Strasbourg, France, December 2003.

[26] Marcel Bosc and Torbjørn Vik. ImLib3D. http://imlib3d.sourceforge.net/, 2005.

[27] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly, 2008. ISBN 978-0596516130.

[28] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008. ISSN 0167-6423. doi: DOI:10.

1016/j.scico.2007.11.003. URL http://www.sciencedirect.com/science/article/pii/S0167642308000452. Special Issue on Second issue of experimental software and toolkits (EST).

[29] Jonathan B. Buckheit and David L. Donoho. WaveLab and reproducible research. Technical Report 474, Stanford University, Stanford CA 94305, USA, 1995.

[30] Timothy Budd. *Multiparadigm programming in Leda*. Addison Wesley, Reading, MA, USA, 1995. ISBN 0-201-82080-3.

[31] Wilhelm Burger and Mark J. Burg. *Digital Image Processing: An Algorithmic Introduction using Java*. Springer, 2008. ISBN 978-1-84628-379-6.

[32] Nicolas Burrus, Alexandre Duret-Lutz, Thierry Géraud, David Lesage, and Raphaël Poss. 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, October 2003.

[33] Centre de Morphologie Mathématique. MICROMORPH. http://cmm.ensmp.fr/Micromorph/mmorph.html.

[34] Steve Cleary and John Maddock. Boost.StaticAssert. http://www.boost.org/doc/libs/release/doc/html/boost_staticassert.html, 2005.

[35] James O. Coplien. Curiously recurring template patterns. In Stanley B. Lippman, editor, *C++ Gems*. Cambridge Press University & Sigs Books, 1996.

[36] Michel Couprie and Gilles Bertrand. New characterizations of simple points in 2D, 3D, and 4D discrete spaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31 (4):637–648, April 2009. ISSN 0162-8828. doi: http://doi.ieeecomputersociety.org/10.1109/TPAMI.2008.117.

[37] Jean Cousty, Gilles Bertrand, Michel Couprie, and Laurent Najman. Collapses and watersheds in pseudomanifolds. In *Proceedings of the 13th International Workshop on Combinatorial Image Analysis (IWCIA)*, IWCIA '09, pages 397–410. Springer-Verlag, 2009. ISBN 978-3-642-10208-0.

[38] Jean Cousty, Gilles Bertrand, Laurent Najman, and Michel Couprie. Watershed cuts: minimum spanning forests and the drop of water principle. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(8):1362–1374, August 2009.

[39] Jean Cousty, Laurent Najman, and Jean Serra. Some morphological operators in graph spaces. In Springer-Verlag, editor, *Proceedings of the Ninth International Symposium on Mathematical Morphology (ISMM)*, Lecture Notes in Computer Science Series, Groningen, The Netherlands, August 2009.

[40] Krzysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technical University of Ilmenau, October 1998.

[41] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. ISBN 0-201-30977-7.

[42] Valentin David, Akim Demaille, and Olivier Gournet. Attribute grammars for modular disambiguation. In *Proceedings of the IEEE 2nd International Conference on Intelligent Computer Communication and Processing (ICCP'06)*, Technical University of Cluj-Napoca, Romania, September 2006.

[43] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Meyers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, SIGPLAN Notices volume 30 number 10, pages 156–168, 1995.

[44] James C. Dehnert and Alexander A. Stepanov. Fundamentals of generic programming. In Mehdi Jazayeri, Rüdiger G. K. Loos, and David R. Musser, editors, *Selected Papers from the International Seminar on Generic Programming*, pages 1–11. Springer-Verlag, 1998. ISBN 3-540-41090-2.

[45] Fábio Dias, Jean Cousty, and Laurent Najman. Some morphological operators on simplicial complex spaces. In *Proceedings of the 16th IAPR international conference on Discrete Geometry for Computer Imagery (DGCI)*, pages 441–452, Nancy, France, 2011. Springer-Verlag. ISBN 978-3-642-19866-3.

[46] Marcos Cordeiro d'Ornellas. *Algorithmic Patterns for Morphological Image Processing*. PhD thesis, Universiteit van Amsterdam, 2001.

[47] Alexandre Duret-Lutz. 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, October 2000. ISBN 3-89683-932-2.

[48] Alexandre Duret-Lutz, Thierry Géraud, and Akim Demaille. Generic design patterns in C++. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 189–202, San Antonio, TX, USA, January-February 2001. USENIX Association.

[49] John W. Eaton. *GNU Octave Manual*. Network Theory, 2002. ISBN 978-0954161729.

[50] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN 0-201-51459-1.

[51] Raffi Enficiaud. Yayi. `http://raffi.enficiaud.free.fr/`.

[52] EPITA Research and Developpement Laboratory (LRDE). The Olena image processing platform. `http://olena.lrde.epita.fr`.

[53] ESIEE Engineering. Pink image processing library. `http://pinkhq.com/`.

[54] Jacques-Olivier Lachaud et al. ImaGene, generic digital image library. `http://gforge.liris.cnrs.fr/projects/imagene`, .

[55] John W. Eaton et al. GNU Octave. `http://www.gnu.org/software/octave/`, .

[56] Ullrich Köthe et al. VIGRA – vision with generic algorithms. `http://hci.iwr.uni-heidelberg.de/vigra/`, .

[57] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL a computational geometry algorithms library. *Software - Practice and Experience*, 30(11):1167–1202, 2000.

[58] Sergey Fomel and Jon F. Claerbout. Guest editors' introduction: Reproducible research. *Computing in Science and Engineering*, 11(1):5–7, 2009. ISSN 1521-9615. doi: 10.1109/MCSE.2009.14.

[59] Centre for Mathematical Morphology. Morph-M: Image processing software specialized in mathematical morphology. `http://cmm.ensmp.fr/Morph-M/`.

[60] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[61] Jacques Froment. MegaWave2 user's guide. `http://megawave.cmla.ens-cachan.fr/`, May 2004.

[62] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[63] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17:145–205, March 2007. ISSN 0956-7968. doi: 10.1017/S0956796806006198.

[64] Thierry Géraud and Alexandre Duret-Lutz. Generic programming redesign of patterns. In *Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 283–294, Irsee, Germany, July 2000. UVK, Univ. Verlag, Konstanz.

[65] Thierry Géraud and Roland Levillain. 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)*, Paphos, Cyprus, July 2008.

[66] Thierry Géraud, Yoann Fabre, Alexandre Duret-Lutz, Dimitri Papadopoulos-Orfanos, and Jean-François Mangin. Obtaining genericity for image processing and pattern recognition algorithms. In *Proceedings of the 15th International Conference on Pattern Recognition (ICPR)*, volume 4, pages 816–819, Barcelona, Spain, September 2000. IEEE Computer Society.

[67] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison-Wesley, third edition, 2005. ISBN 0321246780.

[68] Mentor Graphics. VSIPL++ specification 1.1. `http://s3.mentor.com/embedded/vsipl-specification.pdf`, October 2011.

[69] Douglas Gregor, Jaakko Järvi, Mayuresh Kulkarni, Andrew Lumsdaine, David Musser, and Sibylle Schupp. Generic programming and high-performance libraries. *International*

Journal of Parallel Programming, 33:145–164, June 2005. ISSN 0885-7458. doi: 10.1007/s10766-005-3580-8.

[70] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek. Proposed wording for concepts (revision 9). Technical Report N2773=08-0283, JTC1/SC22/WG21 – The C++ Standards Committee, September 2008.

[71] GraphicsMagick Group. GraphicsMagick image processing system. `http://www.graphicsmagick.org/`.

[72] Khronos OpenCL Working Group. The OpenCL specification, version 1.2. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`, November 2011.

[73] Brian Guenter and Diego Nehab. Neon: A domain-specific programming language for image processing. Microsoft TechReport MSR-TR-2010-175, Microsoft Research, 2010.

[74] Leonard G. C. Hamey. Efficient image processing with the Apply language. In *Proceedings of the International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, pages 533–540, 2007.

[75] Luis Ibáñez, Will Schroeder, Lydia Ng, Josh Cates, and the Insight Software Consortium. *The ITK Software Guide*. Kitware, Inc., second edition, November 2005.

[76] Indiana University. ConceptGCC. `http://www.generic-programming.org/software/ConceptGCC/`.

[77] Intel. Intel Threading Building Blocks reference manual. `http://threadingbuildingblocks.org/uploads/81/91/LatestOpenSourceDocumentation/Reference.pdf`, November 2011.

[78] ECMA International. Standard ECMA-334 — C# language specification, June 2006.

[79] ISO/IEC. ISO/IEC 14882:1998(E). Programming languages — C++, 1998.

[80] ISO/IEC. ISO/IEC 9899:1999(E). Programming languages — C, 1999.

[81] ISO/IEC. ISO/IEC 14882:2003(E). Programming languages — C++, 2003.

[82] ISO/IEC. Ada reference manual, ISO/IEC 8652:1995(E), with technical corrigendum 1 and amendment 1 – language and standard libraries, 2005.

[83] ISO/IEC. ISO/IEC 23270:2006(E). Information technology — Programming languages — C#, 2006.

[84] ISO/IEC. ISO/IEC 25436:2006(E). Information technology — Eiffel: Analysis, design and programming language, 2006.

[85] ISO/IEC. ISO/IEC 1539-1:2010(E). Information technology — Programming languages – Fortran – part 1: Base language, 2010.

[86] ISO/IEC. ISO/IEC 9899:2011(E). Programming languages — C, December 2011.

[87] ISO/IEC. ISO/IEC 14882:2011(E). Programming languages — C++, September 2011.

[88] Mehdi Jazayeri, Rüdiger Loos, David Musser, and Alexander Stepanov. Report of the Dagstuhl seminar (98061) on generic programming. `http://www.dagstuhl.de/98171`, April 1998.

[89] Steve Karmesin, James Crotinger, Julian Cummings, Scott Haney, William J. Humphrey, John Reynders, Stephen Smith, and Timothy Williams. Array design and expression evaluation in POOMA II. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE)*, number 1505 in Lecture Notes in Computer Science, pages 231–238. Springer-Verlag, 1998. ISBN 3-540-65387-2.

[90] Robert Klarer, John Maddock, Beman Dawes, and Howard Hinnant. Proposal to add static assertions to the core language (revision 3). Technical report, JTC1/SC22/WG21 – The C++ Standards Committee, October 2004.

[91] Ullrich Köthe. Reusable software in computer vision. In Bernd Jähne, Horst Haussecker, and Peter Geißler, editors, *Handbook of Computer Vision and Applications*, volume 3: Systems and Applications, pages 103–132. Academic Press, San Diego, CA, USA, 1999.

[92] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, Palo Alto, California, Mar 2004.

[93] Guillaume Lazzara, Roland Levillain, Thierry Géraud, Yann Jacquelet, Julien Marquegnies, and Arthur Crépin-Leblond. The SCRIBO module of the Olena platform: a free software framework for document image analysis. In

Proceedings of the 11th International Conference on Document Analysis and Recognition (ICDAR), Beijing, China, September 2011. International Association for Pattern Recognition (IAPR).

[94] Roland Levillain, Thierry Géraud, and Laurent Najman. Milena: Write generic morphological algorithms once, run on many kinds of images. In Michael H. F. Wilkinson and Jos B. T. M. Roerdink, editors, *Mathematical Morphology and Its Application to Signal and Image Processing – Proceedings of the Ninth International Symposium on Mathematical Morphology (ISMM)*, volume 5720 of *Lecture Notes in Computer Science*, pages 295–306, Groningen, The Netherlands, August 2009. Springer Berlin / Heidelberg.

[95] Roland Levillain, Thierry Géraud, and Laurent Najman. Why and how to design a generic and efficient image processing framework: The case of the Milena library. In *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, pages 1941–1944, Hong Kong, September 2010.

[96] Roland Levillain, Thierry Géraud, and Laurent Najman. Writing reusable digital geometry algorithms in a generic image processing framework. In *Proceedings of the Workshop on Applications of Digital Geometry and Mathematical Morphology (WADGMM)*, pages 96–100, Istanbul, Turkey, August 2010. URL http://mdigest.jrc.ec.europa.eu/wadgmm2010/.

[97] Roland Levillain, Thierry Géraud, and Laurent Najman. Une approche générique du logiciel pour le traitement d'images préservant les performances. In *Proceedings of the 23rd Symposium on Signal and Image Processing (GRETSI)*, Bordeaux, France, September 2011. In French.

[98] Barbara Liskov. A history of CLU. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, pages 133–147, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4. doi: http://doi.acm.org/10.1145/154766.155367.

[99] Barbara Liskov, Alan Snyder, Russell Atkinson, and J. Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[100] LORIA. Qgar®. http://www.qgar.org/.

[101] LRDE — EPITA Research and Developpement Laboratory. Transformers home page, 2005. http://transformers.lrde.epita.fr.

[102] MathWorks. MATLAB®. http://www.mathworks.com/products/matlab/.

[103] Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[104] Bertrand Meyer. Genericity versus inheritance. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Aplications (OOPSLA)*, pages 391–405, Portland, OR, USA, 1986.

[105] Fernand Meyer. Un algorithme optimal de ligne de partage des eaux. In *Actes du 8e Congrès AFCET*, pages 847–857, Lyon-Villeurbanne, France, 1991. AFCET.

[106] Scott Meyers. *More Effective C++*. Addison-Wesley Professional, 1996. ISBN 020163371X.

[107] Scott Meyers. How non-member functions improve encapsulation. *C/C++ Users Journal*, 18(2):44–??, February 2000. ISSN 1075-2838.

[108] David R. Musser and Alexander A. Stepanov. Generic programming. In *Symbolic and Algebraic Computation (Proceedings of ISSAC'88)*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 1989. ISBN 978-3-540-51084-0.

[109] David R. Musser and Alexander A. Stepanov. Algorithm-oriented generic libraries. *Software - Practice and Experience*, 24:623–642, July 1994. ISSN 0038-0644. doi: 10.1002/spe.4380240703.

[110] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley professional computing series. Addison-Wesley, 2001.

[111] Nathan Myers. A new and useful template technique: "traits". In Stanley B. Lippman, editor, *C++ Gems*, pages 451–457. Cambridge Press University & Sigs Books, 1996.

[112] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, 7(5):32–35, June 1995.

[113] National Library of Medicine. Insight segmentation and registration toolkit (ITK). http://www.itk.org/.

[114] John K. Ousterhout and Ken Jones. *Tcl and the Tk Toolkit*. Addison-Wesley Professional Computing Series. Addison-Wesley, second edition, 2009. ISBN 978-0321336330.

[115] P.J. Plauger, Meng Lee, David Musser, and Alexander A. Stepanov. *The C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000. ISBN 0134376331.

[116] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, first edition, July 2007. ISBN 0596514808.

[117] Gerhard X. Ritter, Joseph N. Wilson, and Jennifer L. Davidson. Image algebra: an overview. *Computer Vision, Graphics, and Image Processing*, 49(3):297–331, 1990. ISSN 0734-189X. doi: http://dx.doi.org/10.1016/0734-189X(90)90106-6.

[118] Guido Van Rossum. *The Python Language Reference Manual*. Network Theory, 2003. ISBN 978-0954161781.

[119] Szymon Rusinkiewicz. Estimating curvatures and their derivatives on triangle meshes. In *Proceedings of the 2nd International Symposium on 3D Data Processing, Visualization, and Transmission*, pages 486–493. IEEE Computer Society, 2004. ISBN 0-7695-2223-8. doi: 10.1109/3DPVT.2004.54.

[120] Graziano Lo Russo. An interview with A. Stepanov. http://www.stlport.org/resources/StepanovUSA.html.

[121] Antonio Scuri. ImLab 2.3 – a free experimental system for image processing. http://imlab.sourceforge.net/.

[122] SGI. Standard template library programmer's guide. http://www.sgi.com/tech/stl/.

[123] Jeremy Siek and Andrew Lumsdaine. The Boost Concept Check Library (BCCL). http://www.boost.org/libs/concept_check/concept_check.htm, 2000.

[124] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, October 2000.

[125] Jeremy G. Siek. A modern framework for portable high performance numerical linear algebra. Master's thesis, University of Notre Dame, Indiana, 1999.

[126] Jeremy G. Siek and Andrew Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE)*, number 1505 in Lecture Notes in Computer Science, pages 59–70. Springer-Verlag, 1998.

[127] Alexander Stepanov and Meng Lee. The standard template library. Technical report, HP Labs, 1995.

[128] Michael Still. *The Definitive Guide to ImageMagick*. Definitive Guides. Apress, December 2005. ISBN 1590595904.

[129] Bjarne Stroustrup. Parameterized types for C++. In *Proceedings of the USENIX C++ Conference*, Denver, USA, October 1988.

[130] Bjarne Stroustrup. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., 1994. ISBN 0-201-54330-3.

[131] Bjarne Stroustrup. A rationale for semantically enhanced library languages. In *Proceedings of the Workshop on Library-Centric Software Design (LCSD)*, San Diego, California, USA, October 2005.

[132] Bjarne Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of Programming Languages*, pages 4–1–4–59, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-X. doi: http://doi.acm.org/10.1145/1238844.1238848.

[133] Bjarne Stroustrup. Simplifying the use of concepts. Technical Report N2906=09-0096, JTC1/SC22/WG21 – The C++ Standards Committee, June 2009.

[134] Herb Sutter. Sutter's mill: Virtuality. *C/C++ Users Journal*, 19(9), September 2001.

[135] Hugues Talbot. Imview. `http://hugues.zahlt.info/software_imview.html`.

[136] The Boost Project. Boost C++ libraries. `http://www.boost.org/`.

[137] David Tschumperlé. The CImg library. `http://cimg.sourceforge.net/`.

[138] Erwin Unruh. Prime number computation. Technical report, ISO SC22 WG21, 1994.

[139] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2003.

[140] Todd Veldhuizen. Using C++ template metaprograms. In Stanley B. Lippman, editor, *C++ Gems*, pages 459–473. Cambridge Press University & Sigs Books, 1996.

[141] Todd Veldhuizen. Expression templates. In Stanley B. Lippman, editor, *C++ Gems*, pages 475–487. Cambridge Press University & Sigs Books, 1996.

[142] Todd L. Veldhuizen. Who invented...[template metaprogramming, expression templates]? http://osl.iu.edu/~tveldhui/papers/priority.html.

[143] Todd L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[144] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7 (5):26–31, June 1995. ISSN 1040-6042. Reprinted in C++ Gems, ed. Stanley Lippman.

[145] Todd L. Veldhuizen. Arrays in Blitz++. In Denis Caromel, R. R. Oldehoeft, and Marydell Tholburn, editors, *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, pages 223–230. Springer-Verlag, 1998. ISBN 3-540-65387-2.

[146] Todd L. Veldhuizen. Techniques for scientific C++. Technical Report 542, Indiana University Department of Computer Science, August 1999.

[147] Todd L. Veldhuizen. C++ templates are Turing complete. Technical report, Indiana University, 2003.

[148] Todd L. Veldhuizen and Gannon Dennis. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998.

[149] Todd L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Conference on Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, volume 1343 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, New York, Tokyo, 1997. Springer-Verlag.

[150] Luc Vincent. Graphs and mathematical morphology. *Signal Processing*, 16(4):365–388, April 1989.

[151] Wolfram. Mathematica®. http://www.wolfram.com/mathematica/.

[152] Stephen Wolfram. *The Mathematica® Book*. Wolfram Media, fifth edition, 2003. ISBN 1-57955-022-3.

INDEX