A Static C++ Object-Oriented Programming (SCOOP) Paradigm Mixing Benefits of Traditional OOP and Generic Programming

Nicolas Burrus, Alexandre Duret-Lutz, Thierry Géraud, David Lesage, and Raphaël Poss

EPITA Research and Development Laboratory
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre, France
firstname.lastname@lrde.epita.fr

Abstract. Object-oriented and generic programming are both supported in C++. OOP provides high expressiveness whereas GP leads to more efficient programs by avoiding dynamic typing. This paper presents SCOOP, a new paradigm which enables both classical OO design and high performance in C++ by mixing OOP and GP. We show how classical and advanced OO features such as virtual methods, multiple inheritance, argument covariance, virtual types and multimethods can be implemented in a fully statically typed model, hence without run-time overhead.

1 Introduction

In the context of writing libraries dedicated to scientific numerical computing, expressiveness, reusability and efficiency are highly valuable. Algorithms are turned into software components that handle mathematical abstractions while these abstractions are mapped into types within programs.

The object-oriented programming (OOP) paradigm offers a solution to express reusable algorithms and abstractions through abstract data types and inheritance. However, as studied by Driesen and Hölzle [18], manipulating abstractions usually results in a run-time overhead. We cannot afford this loss of performance since efficiency is a crucial issue in scientific computing.

To both reach a high level of expressiveness and reusability in the design of object-oriented scientific libraries and keep an effective run-time efficiency for their routines, we have to overcome the problem of "abstractions being inefficient". To cope with that, one can imagine different strategies.

A first idea is to find an existing language that meets our requirements, i.e., a language able to handle abstractions within programs without any penalty at execution time. This language has to be either well-known or simple enough to ensure that a scientist will not be reluctant to use our library. Unfortunately we do not feel satisfied with existing languages; for instance LOOM and PolyTOIL by Bruce et al. [11, 9] have the precise flavor that we expect but, as prototypes, they do not feature all what a complete language can offer.

A second approach, chosen by Baumgartner and Russo [6] and Bracha et al. [8] respectively for C++ and Java, is to extend an existing expressive language by adding ad hoc features making programs more efficient at run-time. Yet, this approach requires a too great amount of work without any guarantee that extensions will be adopted by the language community and by compiler vendors. To overcome this problem, an alternate approach is to propose a front-end to translate an extended language, more expressive, into its corresponding primary language, efficient, such as Stroustrup [49] did with his erstwhile version of the C++ language. This approach has been made easier than in the past thanks to recently available tools dedicated to program translation, for instance Xt [57]. However, we have not chosen this way since we are not experimented enough with this field.

Another strategy is to provide a compiler that produces efficient source codes or binaries from programs written in an expressive language. For that, several solutions have been developed that belong to the fields of static analysis and partial evaluation, as described by Chambers et al. [14], Schultz [42], Veldhuizen and Lumsdaine [56]. In particular, how to avoid the overhead of polymorphic method calls is studied by Aigner and Hölzle [2], Bacon and Sweeney [4] for C++ and by Zendra et al. [58] for Eiffel. However, most of these solutions remain prototypes and are not implemented in well-spread compilers.

Last, we can take an existing object-oriented language and try to bend it to make some constructs more efficient. That was for instance the case of the expression templates construct defined by Veldhuizen [54] in C++, later brought to Ada by Duret-Lutz [19], and of mixin-based programming by Smaragdakis and Batory [44] in C++. These solutions belong to the field of the *generic programming* (GP) paradigm, as described by Jazayeri et al. [26]. This programming style aims at implementing algorithms as general so reusable as possible without sacrificing efficiency obtained by parameterization—related to the `template` keyword in C++ and to the `generic` keyword in Ada and Eiffel. However, from our experience in developing a scientific library, we notice several major drawbacks of GP that seriously reduce expressiveness and affect user-friendliness, whereas these drawbacks do not exist with "classical" OOP. A key point of this paper is that we do *not* subscribe to "traditional" GP because of these drawbacks. Said shortly, they have their origin in the unbounded structural typing of parameterization in C++ which prevents from having strongly typed signatures for functions or methods. Consequently, type checking at compile-time is awkward and overloading is extremely restricted. Justifications of our position and details about GP limitations are given later on in this paper.

Actually, we want to keep the best of both OOP and GP paradigms—inheritance, overloading, overriding, and efficiency—without resorting to a new language or new tools—translators, compilers, or optimizers. The advent of the C++ Standard Template Library, mostly inspired by the work of Stepanov et al. [47], is one the first serious well-known artifact of GP. Following that example a lot of scientific computing C++ libraries arose during the past few years(they are referenced by oonumerics [39]), one of the most predominant being Boost

[7]. Meanwhile, due to the numerous features of C++, many related GP techniques appeared and are described in the books by Czarnecki and Eisenecker [17], Alexandrescu [3], Vandevoorde and Josuttis [53]. Moreover, Striegnitz and Smith [48], Järvi and Powell [25], Smaragdakis and McNamara [45] have shown that some features offered by a non-object-oriented paradigm, namely the functional one, can be supported by the native C++ language. Knowing these C++ programming techniques, we then thought that this language was able to support an OOP-like paradigm without compromising efficiency. The present paper describes this paradigm, namely a proposal for "Static C++ Object-Oriented Programming": SCOOP.

This paper is composed of three parts. Section 2 discusses the OOP and GP paradigms, their limitations, existing solutions to overcome some of these limitations, and finally what we expect from SCOOP. Section 3 shows how SCOOP is implemented. Finally some technical details and extra features have been moved into appendices.

2 OOP, GP, and SCOOP

A scientific library offers data structures *and* algorithms. This procedural point of view is now consensual [35] although it seems to go against OOP. Actually, an algorithm is intrinsically a general entity since it deals with abstractions. To get the highest decoupling as possible between data and algorithms, a solution adopted by the C++ Standard Library and many others is to map algorithms into functions. At the same time, data structures are mapped into classes where most of the methods are nothing but the means to access data. Last, providing reusable algorithms is an important objective of libraries so we have to focus on algorithms. It is then easier to consider that algorithms and all other entities are functions (such as in functional languages) to discuss typing issues. For all these reasons, we therefore adopt in this section a function-oriented approach of algorithms.

2.1 About Polymorphisms

A function is polymorphic when its operands can have more than one type, either because there are several definitions of the function, or because its definition allows some freedom in the input types. The right function to call has to be chosen depending on the context. Cardelli and Wegner [13] outline four different kinds of polymorphism.

In **inclusion polymorphism**, a function can work on any type in a *type class*. Type classes are named sets of types that follow a uniform interface. Functional languages like Haskell allow programmers to define type classes explicitly, but this polymorphism is also at the heart of OO languages. In C++, inclusion polymorphism is achieved via two mechanisms: subclassing and overriding of virtual functions.

Subclassing is used to define sets of types. The `class` (or `struct`) keyword is used to define types that can be partially ordered through a hierarchy: i.e., an inclusion relation[1]. A function which expects a pointer or reference to a class `A` will accept an instance of `A` or any subclass of `A`. It can be noted that C++'s typing rules make no difference between a pointer to an object whose type is exactly `A` and a pointer to an object whose type belongs to the type class of `A`[2].

Overriding of virtual functions allows types whose operations have different implementations to share the same interface. This way, an operation can be implemented differently in a subclass of `A` than it is in `A`. Inclusion polymorphism is sometime called *operation polymorphism* for this reason.

These two aspects of inclusion polymorphism are hardly dissociable: it would make no sense to support overriding of virtual functions without subclassing, and subclassing would be nearly useless if all subclasses had to share the same implementation.

In **parametric polymorphism**, the type of the function is represented using at least one generic type variable. Parametric polymorphism really corresponds to ML generic functions, which are compiled only once, even if they are used with different types. Cardelli and Wegner states that Ada's generic functions are not to be considered as parametric polymorphism because they have to be *instantiated explicitly* each time they are used with a different type. They see Ada's generic functions as a way to produce several monomorphic functions by macro expansion. It would therefore be legitimate to wonder whether C++'s function templates achieve parametric polymorphism. We claim it does, because unlike Ada's generics, C++'s templates are instantiated *implicitly*. In effect, it does not matter that C++ instantiates a function for each type while ML compiles only one function, because this is transparent to the user and can be regarded as an implementation detail[3].

These two kinds of polymorphism are called *universal*. A nice property is that they are open-ended: it is always possible to introduce new types and to use them with existing functions. Two other kinds of polymorphism do not share this property. Cardelli and Wegner call them *ad-hoc*.

Overloading corresponds to the case where several functions with different types have the same name.

Coercion polymorphism comes from implicit conversions of arguments. These conversions allow a monomorphic function to appear to be polymorphic.

All these polymorphisms coexist in C++, although we will discuss some notable incompatibilities in section 2.3. Furthermore, apart from virtual functions,

---

[1] Inclusion polymorphism is usually based on a subtyping relation, but we do not enter the debate about "subclassing v. subtyping" [15].

[2] In Ada, one can write `access A` or `access A'Class` to distinguish a pointer to an instance of `A` from a pointer to an instance of any subclass of `A`.

[3] This implementation detail has an advantage, though: it allows specialized instantiations (i.e., template specializations). To establish a rough parallel with *inclusion polymorphism*, template specializations are to templates what method overriding is to subclassing. They allow to change the implementation for some types.

the resolution of a polymorphic function call (i.e., choosing the right definition to use) is performed at compile-time.

2.2 About the Duality of OOP and GP

Duality of OOP and GP has been widely discussed since Meyer [33]. So we just recall here the aspects of this duality that are related to our problem.

Let us consider a simple function `foo` that has to run on different image types. In traditional OOP, the image abstraction is represented by an abstract class, `Image`, while a concrete image type (for instance `Image2D`) for a particular kind of 2D images, is a concrete subclass of the former. The same goes for the notion of "point" that gives rise to a similar family of classes: `Point`, which is abstract, and `Point2D`, a concrete subclass of `Point`. That leads to the following code[4]:

```
struct Image {
  virtual void set(const Point& p, int val) = 0;
};

struct Image2D : public Image {
  virtual void set(const Point& p, int val) { /* impl */ }
};

void foo(Image& input, const Point& p) {
  // does something like:
  input.set(p, 51);
}

int main() {
  Image2D ima; Point2D p;
  foo(ima, p);
}
```

`foo` is a polymorphic function thanks to *inclusion through class inheritance*. The call `input.set(p, 51)` results in a run-time dispatch mechanism which binds this call to the proper implementation, namely `Image2D::set`. In the equivalent GP code, there is no need for inheritance.

```
struct Image2D {
  void set(const Point2D& p, int val) { /* impl */ }
};

template <class IMAGE, class POINT>
void foo(IMAGE& input, const POINT& p) {
  // does something like:
  input.set(p, 51);
}
```

[4] Please note that, for simplification purpose, we use `struct` instead of `class` and that we do not show the source code corresponding to the `Point` hierarchy.

```
int main() {
  Image2D ima; Point2D p;
  foo(ima, p);
}
```

`foo` is now polymorphic through *parameterization*. At compile-time, a particular version of `foo` is instantiated, `foo<Image2D, Point2d>`, dedicated to the particular call to `foo` in `main`. The basic idea of GP is that all exact types are known at compile-time. Consequently, functions are specialized by the compiler; moreover, every function call can be inlined. This kind of programming thus leads to efficient executable codes.

The table below briefly compares different aspects of OOP and GP.

notion	OOP	GP
typing	named typing through class names so explicit in class definitions	structural so only described in documentation
abstraction (e.g., image)	abstract class (e.g., `Image`)	formal parameter (e.g., `IMAGE`)
inheritance	is the way to handle abstractions	is only a way to factorize code
method (set)	no-variant (`Image::set(Point, int)` `Image2D::set(Point, int)`)	— — —
algorithm (foo)	a single code at program-time (`foo`) and a unique version at compile-time (`foo`)	a single meta-code at program-time (`template<..> foo`) and several versions at compile-time (`foo<Image2D,Point2D>`, etc.)
efficiency	poor	high

From the C++ compiler typing point of view, our OOP code can be translated into:

$$\text{type Image} = \{\ \text{set} : \text{Point} \to \text{Int} \to \text{Void}\ \}$$
$$\text{foo} : \text{Image} \to \text{Point} \to \text{Void}$$

`foo` is restricted to objects whose types are respectively subclasses of `Image` and `Point`. For our GP code, things are very different. First, the image abstraction is not explicitly defined in code; it is thus unknown by the compiler. Second, both formal parameters of `foo` are anonymous. We then rename them respectively "I" and "P" in the lines below and we get:

$$\forall\ \text{I},\quad \forall\ \text{P},\quad \text{foo} : \text{I} \to \text{P} \to \text{Void}$$

Finally, if these two pieces of code seem at a first sight equivalent, they do not correspond to the same typing behavior of the C++ language. Thus, they are treated differently by the compiler and have different advantages and drawbacks. The programmer then faces the duality of OOP and GP and has to determinate which paradigm is best suited to her requirements.

During the last few years, the duality between OOP and GP has given rise to several studies.

Different authors have worked on the translation of some design patterns [22] into GP; let us mention Géraud et al. [23], Langer [27], Duret-Lutz et al. [20], Alexandrescu [3], Régis-Gianas and Poss [40].

Another example concerns the *virtual types* construct, which belongs to the OOP world even if very few OO languages feature it. This construct has been

proposed as an extension of the Java language by Thorup [51] and a debate about the translation and equivalence of this construct in the GP world has followed [10, 52, 41].

Since the notion of virtual type is of high importance in the following of this paper, let us give a more elaborate version of our previous example. In an augmented C++ language, we would like to express that both families of image and point classes are related. To that aim, we could write:

```
struct Image {
  virtual typedef Point point_type = 0;
  virtual void set(const point_type& p, int val) = 0;
};

struct Image2D : public Image {
  virtual typedef Point2D point_type;
  virtual void set(const point_type& p, int val) { /* impl */ }
};
```

`point_type` is declared in the `Image` class to be an "abstract type alias" (`virtual typedef .. point_type = 0;`) with a constraint: in subclasses of `Image`, this type should be a subclass of `Point`. In the concrete class `Image2D`, the alias `point_type` is defined to be `Point2D`. Actually, the behavior of such a construct is similar to the one of virtual member functions: using `point_type` on an image object depends on the exact type of the object. A sample use is depicted hereafter:

```
Image* ima = new Image2D();
// ...
Point* p = new (ima->point_type)();
```

At run-time, the particular exact type of `p` is `Point2D` since the exact type of `ima` is `Image2D`.

An about equivalent GP code in also an augmented C++ is as follows:

```
struct Image2D {
  typedef Point2D point_type;
  void set(const point_type& p, int val) { /* impl */ }
};

template <class I>
where I {
  typedef point_type;
  void set(const point_type&, int);
}
void foo(I& input, const typename I::point_type& p) {
  // does something like:
  input.set(p, 51);
}

int main() {
```

```
  Image2D ima; Point2D p;
  foo(ima, p);
}
```

Such as in the original GP code, inheritance is not used and typing is fully structural. On the other hand, a *where clause* has been inserted in `foo`'s signature to precise the nature of acceptable type values for `I`. This construct, which has its origin in CLU [30], can be found in Theta [29], and has also been proposed as an extension of the Java language [36]. From the compiler point of view, `foo`'s type is much more precise than in the traditional GP code. Finally, in both C++ OOP augmented with virtual types and C++ GP augmented with where clauses, we get stronger expressiveness.

2.3 OOP and GP Limitations in C++

Object-Oriented Programming relies principally on the inclusion polymorphism. Its main drawback lies in the indirections necessary to run-time resolution of virtual methods. This run-time penalty is undesirable in highly computational code; we measured that getting rid of virtual methods could speed up an algorithm by a factor of 3 [24].

This paradigm implies a loss of typing: as soon as an object is seen as one of its base classes, the compiler looses some information. This limits optimization opportunities for the compiler, but also type expressiveness for the developer. For instance, once the exact type of the object has been lost, type deduction (`T::deducted_type`) is not possible. This last point can be alleviated by the use of virtual types [52], which are not supported by C++.

The example of the previous section also expresses the need for covariance: `foo` calls the method `set` whose expected behavior is covariant. `foo` precisely calls `Image2D::set(Point2D,int)` in the GP version, whereas the call in the OOP version corresponds to `Image::set(Point,int)`.

Generic Programing on the other hand relies on parametric polymorphism and proscribes virtual functions, hence inclusion polymorphism. The key rule is that the exact type of each object has to be known at compile-time. This allows the compiler to perform many optimizations. We can distinguish three kinds of issues in this paradigm:

- the rejection of operations that cannot be typed statically,
- the closed world assumption,
- the lack of template constraints.

The first issues stem from the will to remain statically typed. Virtual functions are banished, and this is akin to rejecting inclusion polymorphism. Furthermore there is no way to declare an heterogeneous list and to update it at run-time, or, more precisely to dynamically replace an attribute by an object of a compatible subtype. These operations cannot be statically typed, there can be no way around this.

The closed world assumption refers to the fact that C++'s templates do not support separate compilation. Indeed, in a project that uses parametric polymorphism exclusively it prevents separate compilation, because the compiler must always know all type definitions. Such monolithic compilation leads to longer build times but gives the compiler more optimization opportunities. The C++ standard [1] supports separate compilation of templates via the `export` keyword, but this feature has not been implemented in mainstream C++ compilers yet.

```
void foo(A1& arg)
{
  arg.m1()
}

void foo(A2& arg)
{
  arg.m2()
}
```

```
template<class A1>
void foo(A1& arg)
{
  arg.m1()
}

template<class A2>
void foo(A2& arg) // illegal
{
  arg.m2()
}
```

```
template<class A1>
void foo(A1& arg)
{
  arg.m1()
}

template<>
void foo<A2>(A2& arg)
{
  arg.m2()
}
```

Fig. 1. Overloading can be mixed with inclusion polymorphism (left), but will not work with unconstrained parametric polymorphism (middle and right).

The remaining issues come from bad interactions between parametric polymorphism and other polymorphisms in C++. For instance, because template arguments are unconstrained, one cannot easily overload function templates. Figure 1 illustrates this problem. When using inclusion polymorphism (left), the compiler knows how to resolve the overloading: if `arg` is an instance of a subclass of `A1`, resp. `A2`, it should be used with the first resp. second definition of `foo()`. We therefore have two implementations of `foo()` handling two different sets of types. These two sets are not closed (it is always possible to add new subclasses), but they are constrained. Arbitrary types cannot be added unless they are subtypes of `A1` or `A2`. This constraint, which distinguishes the two sets of types, allows the compiler to resolve the overloading.

In generic programming, such an overloading could not be achieved, because of the lack of constraints on template parameters. The middle column on Figure 1 shows a straightforward translation of the previous example into parametric polymorphism. Because template parameters cannot be constrained, the function's arguments have to be generalized *for any type* `A`, and *for any type* `B`. Of course, the resulting piece of code is not legal in C++ because both functions have the same type. A valid possibility (on the right of Figure 1), is to write a definition of `foo` for any type `A1`, and then *specialize* this definition for type `A2`. However, this specialization will only work for one type (`A2`), and would have to be repeated for each other type that must be handled this way.

Solving overloading is not the only reason to constrain template arguments, it can also help catching errors. Libraries like STL, which rely on generic programming, document the requirements that type arguments must satisfy. These constraints are gathered into *concepts* such as *forward iterator* or *associative container* [47]. However, these concepts appear only in the documentation, not in typing. Although some techniques have been devised and implemented in SGI's STL to check concepts at compile-time, the typing of the library still allows a function expecting a *forward iterator* to be instantiated with an *associative container*. Even if the compilation will fail, this technique will not prevent the compiler from instantiating the function, leading to cryptic error messages, because some function part of the *forward iterator* requirements will not be found in the passed associative container. Could the *forward iterator* have been expressed as a constraint on the argument type, the error would have been caught at the right time i.e. during the attempt to instantiate the function template, not after the instantiation.

2.4 Existing Clues

As just mentioned, some people have already devised ways to check constraints. Siek and Lumsdaine [43] and McNamara and Smaragdakis [32] present a technique to check template arguments. This technique relies on a short checking code inserted at the top of a function template. This code fails to compile if an argument does not satisfy its requirements and is turned into a no-op otherwise. This technique is an effective means of performing structural checks on template arguments to catch errors earlier. However, constraints are just *checked*, they are not *expressed* as part of function types. In particular, overloading issues discussed in the previous section are not solved. Overloading has to be solved by the compiler *before* template instantiation, so any technique that works after template instantiation does not help.

Ways to *express* constraints by subtyping exist in Eiffel [34] and has been proposed as a Java extension by Bracha et al. [8]. Figure 2 shows how a similar C++ extension could be applied to the example from Section 2.2.

We have introduced an explicit construct through the keyword `concept` to express the definition of `image`, the structural type of images. This construct is also similar to the notion of signatures proposed by Baumgartner and Russo [6] as a C++ extension. Having explicitly a definition of `image` constraints the formal parameter `I` in `foo`'s type.

Some interesting constructions used to constrain parametric polymorphism or to emulate dynamic dispatch statically rely on a idiom known as the *Barton and Nackman trick* [5] also known as the *Curiously Recurring Template Pattern* [16]. The idea is that a super class is parameterized by its immediate subclass (Figure 3), so that it can define methods for this subclass.

For instance the Barton and Nackman trick has been used by Furnish [21] to constrain parametric polymorphism and simplify the Expression Template technique of Veldhuizen [54].

```
concept image {
  typedef point_type;
  void set(const point_type& p, int val);
};

struct Image2D models image {
  typedef Point2D point_type;
  void set(const point_type& p, int val) { /* impl */ }
};

template <class I models image>
void foo(I& input, const typename I::point_type& p) {
  // does something with:
  input.set(p, 51);
}

int main() {
  Image2D ima; Point2D p;
  foo(ima, p);
}
```

Fig. 2. Extending C++ to support concept constraints

```
template <class T>
struct super
{
  void foo(const T& arg)
  {
    // ...
  }
};
```

```
struct infer : public super<infer>
{
  // ...
};
```

Fig. 3. The Barton and Nackman trick

2.5 Objectives of SCOOP

Our objective in this paper is to show how inclusion polymorphism can be almost completely emulated using parametric polymorphism in C++ while preserving most OOP features. Let us define our requirements.

Class Hierarchies. Developers should express (static) class hierarchies just like in the traditional (dynamic) C++ OOP paradigm. They can draw UML static diagrams to depict inheritance relationships between classes of their programs. When they have a class in OO, say `Bar`, its translation in SCOOP is a single class template: `Bar`[5].

Named Typing. When a scientific practitioner designs a software library, it is convenient to reproduce in programs the names of the different abstractions of the application domain. Following this idea, there is an effective benefit to make explicit the relationships between concrete classes and their corresponding abstractions to get a more readable class taxonomy. We thus prefer named typing over structural typing for SCOOP.

Multiple Inheritance. In the object model of C++, a class can inherits of several classes at the same time. There is no reason to give up this feature in SCOOP.

Overriding. With C++ inheritance come the notions of pure virtual functions, of virtual functions, and of overriding functions in subclasses. We want to reproduce their behavior in SCOOP but without their associated overhead.

Virtual Types. This convenient tool (see sections 2.2 and 2.3) allows to express that a class encloses polymorphic `typedef`s. Furthermore, it allows to get covariance for member functions. Even if virtual types does not exist in primary C++, we want to express them in SCOOP.

Method Covariance. It seems reasonable to support method covariance in SCOOP, and particularly binary methods. Since our context is static typing with parametric polymorphism, the C++ compiler may ensure that we do not get typing problems eventually.

Overloading. In the context of scientific computing, having overloading is crucial. For instance, we expect from the operator "`+`" to be an over-overloaded function in an algebraic library. Moreover, overloading helps to handle a situation that often arises in scientific libraries: some algorithms have a general implementation but also have different more efficient implementation for particular families of objects. We want to ensure in SCOOP that overloading is as simply manageable as in OOP.

[5] We are aware of a solution to encode static class hierarchies that is different to the one presented later on in this paper. However, one drawback of this alternate solution is to duplicate every class: having a class `Bar` in OOP gives rise to a couple of classes in the static hierarchy. To our opinion, this is both counter-intuitive and tedious.

Multimethods. Algorithms are often functions with several input or arguments. Since the source code of an algorithm can also vary with the nature and number of its input, we need multimethods.

Parameter Bounds. Routines of scientific libraries have to be mapped into strongly typed functions. First, this requirement results in a comfort for the users since it prevents them from writing error-prone programs. Second, this requirement is helpful to disambiguate both overloading and multimethod dispatch.

3 Description of SCOOP

3.1 Static Hierarchies

Static hierarchies are meta-hierarchies that result in real hierarchies after various static computations like parameter valuations. With them, we are able to know all types statically hence avoiding the overhead of virtual method resolution. Basically, the core of our static hierarchy system is a generalization of the Barton & Nackman trick [5]. Veldhuizen [55] had already discussed some extensions of this technique and assumed the possibility to apply it to hierarchies with several levels. We effectively managed to generalize these techniques to entire, multiple-level hierarchies.

Our hierarchy system is illustrated in Figure 4. This figure gives an example of a meta-hierarchy, as designed by the developer, and describes the different final hierarchies obtained, according to the instantiated class. The corresponding C++ code is given in Figure 5. This kind of hierarchy gives us the possibility to define abstract classes (class `A`), concrete extensible classes (class `B`), and final classes (class `C`). Non final classes[6] are parameterized by `EXACT` that basically represents the type of the object effectively instantiated. Additionally, any class hierarchy must inherit from a special base class called `Any`. This class factorizes some general mechanisms whose role are detailed later.

Instantiations of abstract classes are prevented by protecting their constructors. The interfaces and the dispatch mechanisms they provide are detailed in Section 3.2.

Extensible concrete classes can be instantiated and extended by subclassing. Since the type of the object effectively instantiated must be propagated through the hierarchy, this kind of class has a double behavior. When such a class `B` is extended and is not the instantiated class, it must propagate its `EXACT` type parameter to its base classes. When it is effectively instantiated, further subclassing is prevented by using the `Itself` terminator as `EXACT` parameter. Then, `B` cannot propagate its `EXACT` parameter directly and should propagate its own type, `B<Itself>`. To determine the effective `EXACT` parameter to propagate, we

[6] Non final classes are abstract classes or concrete classes that can be extended. Non parameterized classes are necessarily final in our paradigm.

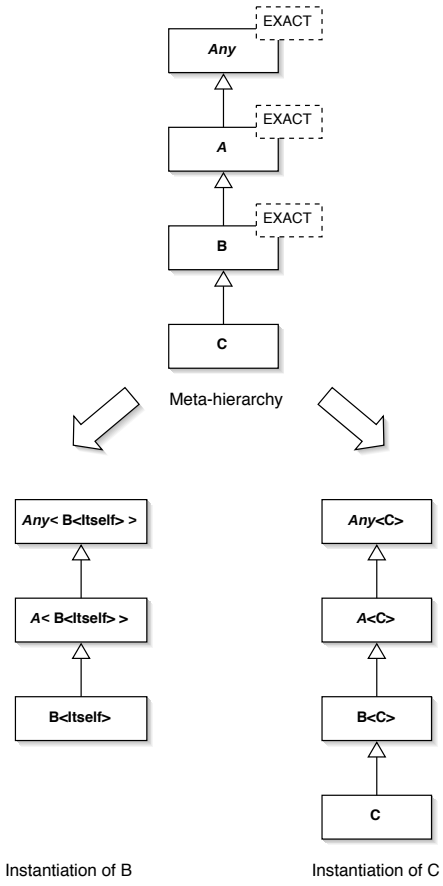

Fig. 4. Static hierarchy unfolding sample
A single meta-hierarchy generates one class hierarchy per instantiable class. Our model can instantiate both leaf classes and intermediate ones. In this example, only `B` and `C` are instantiable, so only the above two hierarchies can be instantiated. Non final classes are parameterized by `EXACT` which represents the type of the object effectively instantiated. The type `Itself` is used as a terminator when instantiating extensible concrete classes.

```
// Hierarchy apparel

struct Itself
{ };

// find_exact utility macro
#define find_exact(Type) // ...

template <class EXACT>
class Any
{
  // ...
};
```

```
// Hierarchy

// purely abstract class
template <class EXACT>
class A: public Any<EXACT>
{
  // ...
};

// extensible concrete class
template <class EXACT = Itself>
class B: public A<find_exact(B)>
{
  // ...
};

// final class
class C: public B<C>
{
  // ...
};
```

Fig. 5. Static hierarchy sample: C++ code
`find_exact(Type)` mechanism is detailed in Appendix A.1.

use a meta-program called `find_exact(Type)` whose principle and C++ implementation are detailed in Appendix A.1. One should also notice that `Itself` is the default value for the `EXACT` parameter of extensible concrete classes. Thus, `B` sample class can be instantiated using the `B<>` syntax.

Itself classes cannot be extended by subclassing. Consequently, they do not need any `EXACT` parameterization since they are inevitably the instantiated type when they are part of the effective hierarchy. Then, they only have to propagate their own types to their parents.

Within our system, any static hierarchy involving n concrete classes can be unfolded into n distinct hierarchies, with n distinct base classes. Effectively, concrete classes instantiated from the same meta-hierarchy will have different base classes, so that some dynamic mechanisms are made impossible (see Section 2.3).

3.2 Abstract Classes and Interfaces

In OOP, abstraction comes from the ability to express class interfaces without implementation. Our model keeps the idea that C++ interfaces are represented by abstract classes. Abstract classes declare all the services their subclasses should provide. The compliance to a particular interface is then naturally ensured by the inheritance from the corresponding abstract class.

Instead of declaring pure virtual member functions, abstract classes define abstract member functions as dispatches to their actual implementation. This manual dispatch is made possible by the `exact()` accessor provided by the `Any` class. Basically, `exact()` downcasts the object to its `EXACT` type made available by the static hierarchy system presented in Section 3.1. In practice, `exact()` can be implemented with a simple `static_cast` construct, but this basic mechanism forbids virtual inheritance[7]. Within our paradigm, an indirect consequence is that multiple inheritance implies inevitably virtual inheritance since `Any` is a utility base class common to all classes. Advanced techniques, making virtual and thus multiple inheritance possible, are detailed in Appendix A.2.

An example of an abstract class with a dispatched method is given in Figure 6. The corresponding C++ code can be deduced naturally from this UML diagram. In the abstract class `A`, the method `m(...)` calls its implementation `m_impl(...)`. Method's interface and implementation are explicitly distinguished by using different names. This prevents recursive calls of the interface if the implementation is not defined. Of course, overriding the implementation is permitted. Thanks to the `exact()` downcast, `m_impl(...)` is called on the type of the object effectively instantiated, which is necessarily a subclass of `A`. Thus, overriding rules are respected. Since the `EXACT` type is known statically, this kind of dispatch is entirely performed at compile-time and does not require the use of virtual symbol tables. Method dispatches can be inlined so that they finally come with no run-time overhead.

[7] Virtual inheritance occurs in diamond-shape hierarchies.

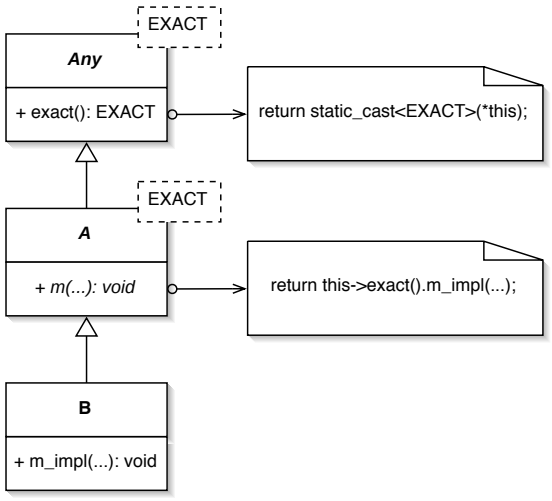

Fig. 6. Abstract class and dispatched abstract method sample

3.3 Constraints on Parameters

Using SCOOP, it becomes possible to express constraints on types. Since we have inheritance between classes, we can specify that we only want a subclass of a particular type, thereby constraining the input type. Thus, OOP's ability to handle two different sets of types has been kept in SCOOP, as demonstrated in Figure 7.

Actually, two kinds of constraints are made possible: accept a type and all its subclasses or accept only this type. Both kinds of constraints are illustrated in Figure 8. We have the choice between letting the `EXACT` parameter free to accept all its subclasses, or freezing it (generally to `Itself`) to accept only this exact type.

3.4 Associations

In SCOOP, the implementation of object composition or aggregation is very close to its equivalent in C++ OOP. Figure 9 illustrates the way an aggregation relation is implemented in our paradigm, in comparison with classical OOP. We want a class `B` to aggregate an object of type `C`, which is an abstract class. The natural way to implement this in classical OOP is to maintain a pointer on an object of type `C` as a member of class `B`. In SCOOP, the corresponding meta-class `B` is parameterized by `EXACT`, as explained in Section 3.1. Since all types have to be known statically, `B` must know the effective type of the object it aggregates. A second parameter, `EXACT_C`, is necessary to carry this type. Then, `B` only has to keep a pointer on an object of type `C<EXACT_C>`. As explained in Section 3.3, this syntax ensures that the aggregated object type is a subclass of `C`. This provides

```
void foo(A1& arg)
{
   // ...
}

void foo(A2& arg)
{
   // ...
}
```

```
template <class EXACT>
void foo(A1<EXACT>& arg)
{
   // ...
}

template <class EXACT>
void foo(A2<EXACT>& arg)
{
   // ...
}
```

Fig. 7. Constraints on arguments and overloading
Left (classical OOP) and right (SCOOP) codes have the same behavior. Classical overloading rules are applied in both cases. Subclasses of `A1` and `A2` are accepted in SCOOP too; the limitation of GP has been overcome.

```
template <class EXACT>
void foo(A<EXACT>& a)
{
  // ...
}
```

```
void foo(A<Itself>& a)
{
  // ...
}
```

Fig. 8. Kinds of constraints
On the left, `A` and all its subclasses are accepted. On the right, only exact `A` arguments are accepted. As mentioned in section 2.1, contrary to other languages like Ada, C++ cannot make this distinction; this is therefore another restriction overcome by SCOOP.

stronger typing than the generic programming idioms for aggregation proposed in [20].

As for hierarchy unfolding (Section 3.1), this aggregation pattern generates as many versions of `B` as there are distinct parameters `EXACT_C`. Each effective version of `B` is dedicated to a particular subclass of `C`. Thus, it is impossible to change dynamically the aggregated object for an object of another concrete type. This limitation is directly related to the rejection of dynamic operations, as mentioned in Section 2.3.

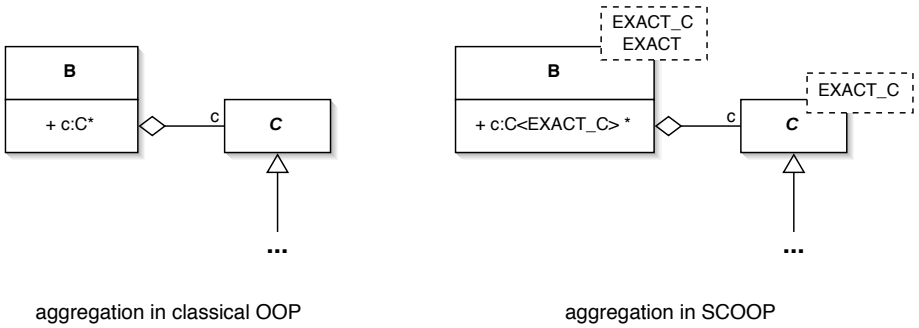

Fig. 9. Comparison of aggregation in OOP and SCOOP

3.5 Covariant Arguments

Covariant parameters may be simulated in C++ in several ways. It can be done by using a `dynamic_cast` to check and convert at run-time the type of the argument. This method leads to unsafe and slower programs. Statically checked covariance has already been studied using templates in Surazhsky and Gil [50]. Their approach was rather complex though, since their typing system was weaker.

Using SCOOP, it is almost straightforward to get statically checked covariant parameters. We consider an example with images and points in 2 and 3 dimensions to illustrates argument covariance. Figure 10 depicts a UML diagram of our example. Since an `Image2d` can be seen as an `Image`, it is possible to give a `Point3d` (seen as a `Point`) to an `Image2d`. This is why classical OO languages either forbid it or perform dynamic type checking when argument covariance is involved.

Figure 11 details how this design would be implemented in SCOOP. This code works in three steps:

– Take a `Point<P>` argument in `Image::set` and downcast it into its exact type `P`. Taking a `Point<P>` argument ensures that `P` is a subclass of `Point` at this particular level of the hierarchy.

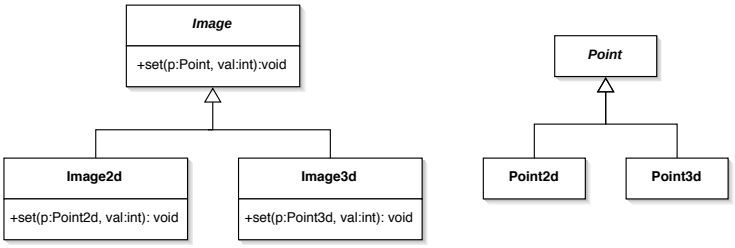

Fig. 10. Argument covariance example in UML

- Lookup `set_impl` in the exact image type. Since the point argument has been downcasted towards `P`, methods accepting `P` (and not just `Point<P>`) are candidate.
- In SCOOP, since method dispatch is performed at compile-time, argument covariance will be checked statically. The compilation fails if no method accepting the given exact point type is available.

Finally, we have effectively expressed argument covariance. Points have to conform to `Point` at the level of `Image`, and to `Point2d` at the level of `Image2d`.

3.6 Polymorphic `typedefs`

In this section we show how we can write virtual `typedefs` (we also call them polymorphic `typedefs`) in C++. From a base class we want to access `typedefs` defined in its subclasses. Within our paradigm, although base classes hold the type of their most derived subclass, it is not possible to access fields of an incomplete type. When a base class is instantiated, its `EXACT` parameter is not completely constructed yet because base classes have to be instantiated before subclasses. A good solution to cope with this issue is to use traits [37, 55]. Traits can be defined on incomplete types, thereby avoiding the infinite recursion.

The overall mechanism is described in Figure 12. To allow the base class to access `typedefs` in the exact class, traits have been defined for the exact type (`image_traits`). To ensure correct typedef inheritance, we create a hierarchy of traits which reproduces the class hierarchy. Thus, `typedefs` are inherited as if they were actually defined in the class hierarchy. As for argument covariance, virtual `typedefs` are checked statically since method dispatch is performed at compile-time. The compilation fails if a wrong point type is given to an `Image2d`.

There is an important difference between classical virtual types and our virtual `typedefs`. First, the virtual `typedefs` we have described are not constrained. The `point_type` virtual `typedef` does not have to be a subclass of `Point`. It can be any type. It is possible to express a subclassing constraint though, by checking it explicitly using a meta-programming technique detailed in Appendix A.3.

One should note that in our paradigm, when using `typedefs`, the resulting type is a single type, not a class of types (with the meaning of Section 3.3).

```
template <class EXACT>
struct Point : public Any<EXACT> {};

template <class EXACT = Itself>
struct Point2d : public Point<find_exact(Point2d)>
{
  // ...
};

template <class EXACT = Itself>
struct Point3d : public Point<find_exact(Point3d)>
{
  // ...
};

template <class EXACT>
struct Image : Any<EXACT>
{
  template <class P>
  void set(const Point<P>& p, int val) {
    // static dispatch
    // p is downcasted to its exact type
    return this->exact().set_impl(p.exact(), val);
  }
};

template <class EXACT = Itself>
struct Image2d : public Image<find_exact(Image2d)>
{
  template <class P>
  void set_impl(const Point2d<P>& p, int val) {
    // ...
  }
};

int main() {
  Image2d<> ima;
  ima.set(Point2d<>(), 42); // ok
  ima.set(Point3d<>(), 51); // fails at compile-time
}
```

Fig. 11. Argument covariance using SCOOP
Compilation fails if the compiler cannot find an implementation of `set_impl` for the exact type of the given point in `Image2d`.

```
// Point, Point2d and Point3d

// A forward declaration is enough to define image_traits
template <class EXACT> struct Image;

template <class EXACT> struct image_traits;

template <class EXACT>
struct image_traits< Image<EXACT> >
{
  // default typedefs for Image
};

template <class EXACT>
struct Image : Any<EXACT>
{
  typedef typename image_traits<EXACT>::point_type point_type;

  void set(const point_type& p, int val) {
    this->exact().set_impl(p, val);
  }
};

// Forward declaration
template <class EXACT> struct Image2d;

// image_traits for Image2d inherits from image_traits for Image
template <class EXACT>
struct image_traits< Image2d<EXACT> >
  : public image_traits<Image <find_exact(Image2d)> >
{
  // We have to specify a concrete type, we cannot write:
  // typedef template Point2d point_type;

  typedef Point2d<Itself> point_type;
  // ... other default typedefs for Image2d
};

template <class EXACT = Itself>
struct Image2d : public Image<find_exact(Image2d)>
{
  // ...
};

int main() {
  Image2d<> ima;
  ima.set(Point2d<>(), 42); // ok
  ima.set(Point3d<>(), 51); // fails at compile-time
}
```

Fig. 12. Mechanisms of virtual `typedef`s with SCOOP

A procedure taking this type as argument does not accept its subclasses. For instance, a subclass `SpecialPoint2d` of `Point2d` is not accepted by the `set` method. This problem is due to the impossibility in C++ to make `template typedef`s, thus we have to bound the exact type of the class when making a `typedef` on it. It is actually possible to overcome this problem by encapsulating open types in boxes. This is not detailed in this paper though.

3.7 Multimethods

Several approaches have been studied to provide multimethods in C++, for instance Smith [46], which relies on preprocessing.

In SCOOP, a multimethod is written as a set of functions sharing the same name. The dispatching is then naturally performed by the overloading resolution, as depicted by Figure 13.

```
template <class I1, class I2>
void algo2(Image<I1>& i1, Image<I2>& i2);

template <class I1, class I2>
void algo2(Image2d<I1>& i1, Image3d<I2>& i2);

template <class I1, class I2>
void algo2(Image2d<I1>& i1, Image2d<I2>& i2);

// ... other versions of algo2

template <class I1, class I2>
void algo1(Image<I1>& i1, Image<I2>& i2)
{
  // dispatch will be performed on the exact image types
  algo2(i1.exact(), i2.exact());
}
```

Fig. 13. Static dispatch for multi-methods
`algo1` downcasts `i1` and `i2` into their exact types when calling `algo2`. Thus, usual overloading rules will simulate multimethod dispatch.

4 Conclusion and Perspectives

In this paper, we described a proposal for a Static C++ Object-Oriented Programming (SCOOP) paradigm. This model combines the expressiveness of traditional OOP and the performance gain of static binding resolution thanks to

generic programming mechanisms. SCOOP allows developers to design OO-like hierarchies and to handle abstractions without run-time overhead. SCOOP also features constrained parametric polymorphism, argument covariance, polymorphic `typedef`s, and multimethods for free.

Yet, we have not proved that resulting programs are type safe. The type properties of SCOOP have to be studied from a more theoretical point of view. Since SCOOP is static-oriented, object types appear with great precision. We expect from the C++ compiler to diagnose most programming errors. Actually, we have the intuition that this kind of programming is closely related to the *matching* type system of Bruce et al. [11]. In addition, functions in SCOOP seem to be f-bounded [12].

The main limitations of our paradigm are common drawbacks of the intensive use of templates:

- closed world assumption;
- heavy compilation time;
- code bloat (but we trade disk space for run-time efficiency);
- cryptic error messages;
- unusual code, unreadable by the casual reader.

The first limitation prevents the usage of separated compilation and dynamic libraries. The second one is unavoidable since SCOOP run-time efficiency relies on the C++ capability of letting the compiler perform some computations. The remaining drawbacks are related to the convenience of the *core developer*, i.e. the programmer who designs the hierarchies and should take care about core mechanisms. Cryptyc error messages can be helped by the use of structural checks mentionned in Section 2.4, which are not incompatible with SCOOP.

This paradigm has been implemented and successfully deployed in a large scale project: Olena, a free software library dedicated to image processing [38]. This library mixes different complex hierarchies (images, points, neighborhoods) with highly generic algorithms.

Although repelling at first glance, SCOOP can be assimilated relatively quickly since its principles remain very close to OOP. We believe that SCOOP and its collection of constructs are suitable for most scientific numerical computing projects.

Bibliography

[1] International standard: Programming language – C++. ISO/IEC 14882:1998(E), 1998.

[2] G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In *In the Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP)*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–167. Springer-Verlag, 1996.

[3] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[4] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *In the Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 324–341, 1996.

[5] J. Barton and L. Nackman. *Scientific and engineering C++*. Addison-Wesley, 1994.

[6] G. Baumgartner and V. F. Russo. Implementing signatures for C++. *ACM Transactions on Programming Languages and Systems*, 19(1):153–187, January 1997.

[7] Boost. Boost libraries, 2003. URL `http://www.boost.org`.

[8] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *In the Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[9] K. B. Bruce, A. Fiech, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems (ToPLAS)*, 25(2):225–290, March 2003.

[10] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *In the Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549, Brussels, Belgium, July 1998. Springer-Verlag.

[11] K. B. Bruce, L. Petersen, and A. Fiech. Subtyping is not a good "match" for object-oriented languages. In *In the Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127, Jyväskylä, Finland, 1997. Springer-Verlag.

[12] P. S. Canning, W. R. Cook, W. L. Hill, J. C. Mitchell, and W. G. Olthoff. F-bounded polymorphism for object-oriented programming. In *In the Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 73–280, London, UK, September 1989. ACM.

[13] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[14] C. Chambers, J. Dean, and D. Grove. Wholeprogram optimization of object-oriented languages. Technical Report UW-CSE-96-06-02, University of Washington, Department of Computer Science and Engineering, June 1996.

[15] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 125–135, San Francisco, California, USA, January 1990. on l'a pas.

[16] J. Coplien. *Curiously Recurring Template Pattern*. In [28].

[17] K. Czarnecki and U. Eisenecker. *Generative programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[18] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In *In the Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, SIGPLAN Notices 31(10), pages 306–323, 1996.

[19] A. Duret-Lutz. Expression templates in Ada. In *In the Proceedings of the 6th International Conference on Reliable Software Technologies, Leuven, Belgium, May 2001 (Ada-Europe)*, volume 2043 of *Lecture Notes in Computer Science*, pages 191–202. Springer-Verlag, 2001.

[20] A. Duret-Lutz, T. Géraud, and A. Demaille. Design patterns for generic programming in C++. In *In the Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 189–202, San Antonio, Texas, USA, January-February 2001. USENIX Association.

[21] G. Furnish. Disambiguated glommable expression templates. *Computers in Physics*, 11(3):263–269, 1997.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns – Elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, 1995.

[23] T. Géraud, A. Duret-Lutz, and A. Adjaoute. Design patterns for generic programming. In M. Devos and A. Rüping, editors, *In the Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP'2000)*. UVK, Univ. Verlag, Konstanz, July 2000.

[24] T. Géraud, Y. Fabre, and A. Duret-Lutz. Applying generic programming to image processing. In M. Hamsa, editor, *In the Proceedings of the IASTED International Conference on Applied Informatics – Symposium Advances in Computer Applications*, pages 577–581, Innsbruck, Austria, February 2001. ACTA Press.

[25] J. Järvi and G. Powell. The lambda library: Lambda abstraction in C++. In *In the Proceedings of the 2nd Workshop on Template Programming (in conjunction with OOPSLA)*, Tampa Bay, Florida, USA, October 2001.

[26] M. Jazayeri, R. Loos, and D. Musser, editors. *Generic Programming: International Seminar, Dagstuhl Castle, Germany, 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*, 2000. Springer-Verlag.

[27] A. Langer. Implementing design patterns using C++ templates. Tutorial at the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 2000.

[28] S. B. Lippman, editor. *C++ Gems*. Cambridge Press University & Sigs Books, 1998.

[29] B. Liskov, D. Curtis, M. Day, S. Ghemawhat, R. Gruber, P. Johnson, and A. C. Myers. Theta reference manual. Technical Report 88, Programming Methodology Group, MIT Laboratory for Computer Science, Cambridge, MA, USA, February 1995.

[30] B. Liskov, A. Snyder, R. Atkinson, and J. C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[31] F. Maes. Program templates: Expression templates applied to program evaluation. In J. Striegnitz and K. Davis, editors, *In the Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL; in conjunction with PLI)*, number FZJ-ZAM-IB-2003-10 in John von Neumann Institute for Computing (NIC), Uppsala, Sweden, August 2003.

[32] B. McNamara and Y. Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[33] B. 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.

[34] B. Meyer. *Eiffel: the Language*. Prentice Hall, 1992.

[35] S. Meyers. How non-member functions improve encapsulation. 18(2):44–??, Feb. 2000. ISSN 1075-2838.

[36] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for java. In *In the Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 132–145, Paris, France, January 1997.

[37] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, 7 (5):32–35, June 1995.

[38] Olena. Olena image processing library, 2003. URL `http://olena.lrde.epita.fr`.

[39] oonumerics. Scientific computing in object-oriented languages, 2003. URL `http://www.oonumerics.org`.

[40] Y. Régis-Gianas and R. Poss. On orthogonal specialization in C++: Dealing with efficiency and algebraic abstraction in Vaucanson. In J. Striegnitz and K. Davis, editors, *In the Proceedings of the Parallel/High-performance Object-Oriented Scientific Computing (POOSC; in conjunction with ECOOP)*, number FZJ-ZAM-IB-2003-09 in John von Neumann Institute for Computing (NIC), Darmstadt, Germany, July 2003.

[41] X. Rémy and J. Vouillon. On the (un)reality of virtual types. URL `http://pauillac.inria.fr/~remy/work/virtual/`. March 2000.

[42] U. P. Schultz. Partial evaluation for class-based object-oriented languages. In *Program as Data Objects: International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 2001,*

Proceedings, volume 2053 of *Lecture Notes in Computer Science*, pages 173–198. Springer-Verlag, 2001.

[43] J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, October 2000.

[44] Y. Smaragdakis and D. Batory. Mixin-based programming in C++. In *In the Proceedings of the 2nd International Conference on Generative and Component-based Software Engineering (GCSE)*, pages 464–478. tranSIT Verlag, Germany, October 2000.

[45] Y. Smaragdakis and B. McNamara. FC++: Functional tools for object-oriented tasks. *Software - Practice and Experience*, 32(10):1015–1033, August 2002.

[46] J. Smith. C++ & multi-methods. *ACCU spring 2003 conference*, 2003.

[47] A. Stepanov, M. Lee, and D. Musser. *The C++ Standard Template Library*. Prentice-Hall, 2000.

[48] J. Striegnitz and S. A. Smith. An expression template aware lambda function. In *In the Proceedings of the 1st Workshop on Template Programming*, Erfurt, Germany, October 2000.

[49] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[50] V. Surazhsky and J. Y. Gil. Type-safe covariance in C++, 2002. URL `http://www.cs.technion.ac.il/~yogi/Courses/CS-Scientific-Writing/exampl%es/paper/main.pdf`. Unpublished.

[51] K. K. Thorup. Genericity in Java with virtual types. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471, Jyväskylä, Finland, June 1997. Springer-Verlag.

[52] K. K. Thorup and M. Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In R. Guerraoui, editor, *In the Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 186–204, Lisbon, Portugal, June 1999. Springer-Verlag.

[53] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2003.

[54] T. Veldhuizen. *Expression Templates*, pages 475–487. In [28].

[55] T. L. Veldhuizen. Techniques for scientific C++, August 1999. URL `http://extreme.indiana.edu/~tveldhui/papers/techniques/`.

[56] T. L. Veldhuizen and A. Lumsdaine. Guaranteed optimization: Proving nullspace properties of compilers. In *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, volume 2477 of *Lecture Notes in Computer Science*, pages 263–277. Springer-Verlag, 2002.

[57] Xt. A bundle of program transformation tools. Available on the Internet, 2003. URL `http://www.program-transformation.org/xt`.

[58] O. Zendra, D. Colnet, and S. Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *In the Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems,*

Languages and Applications (OOPSLA), volume 32 of *Issue 10*, pages 125–141, Athlanta, GA, USA, October 1997.

A Technical Details

A.1 Implementation of `find_exact`

The `find_exact` mechanism, introduced in Section 3.1, is used to enable classes that are both concrete and extensible within our static hierarchy system. This kind of class is parameterized by `EXACT`: the type of the object effectively instantiated. Contrary to abstract classes, concrete extensible classes cannot propagate directly their `EXACT` parameter to their parents, as explained in Section 3.1. A simple utility macro called `find_exact` is necessary to determine the `EXACT` type to propagate. This macro relies on a meta-program, `FindExact`, whose principle is described in Figure 14. and the corresponding C++ code is given in Figure 15.

FindExact($Type$, $EXACT$)
{
  if $EXACT \neq$ "$Itself$"
    return $EXACT$;
  else
    return $Type < Itself >$;
}

Fig. 14. `FindExact` mechanism: algorithmic description

```
// default version
template <class T, class EXACT>
struct FindExact
{
  typedef EXACT ret;
};

// version specialized for EXACT=Itself
template <class T>
struct FindExact<T, Itself>
{
  typedef T ret;
};

// find_exact utility macro
#define find_exact(Type) typename FindExact<Type<Exact>, Exact>::ret
```

Fig. 15. `FindExact` mechanism: C++ implementation

A.2 Static Dispatch with Virtual Inheritance

Using a `static_cast` to downcast a type does not work when virtual inheritance is involved. Let us consider an instance of EXACT. It is possible to create an `Any<EXACT>` pointer on this instance. In the following, the address pointed to by the `Any<EXACT>` pointer is called "the `Any` address" and the address of the `EXACT` instance is called the "exact address".

The problem with virtual inheritance is that the `Any` address is not necessarily the same as the exact address. Thus, even `reinterpret_cast` or `void*` casts will not help. We actually found three solutions to cope with this issue. Each solution has its own drawbacks and benefits, but only one is detailed in this paper.

The main idea is that the offset between the `Any` address and the exact address will remain the same for all the instances of a particular class (we assume that C++ compilers will not generate several memory model for one given class). The simplest way to calculate this offset is to compute the difference between an object address and the address of an `Any<EXACT>` reference to it. This has to be done only once per exact class. The principle is exposed in Figure 16.

This method has two drawbacks. First, it requires a generic way to instantiate the `EXACT` classes, for instance a default constructor. Second, one object per class (not per instance!) is kept in memory. If an object cannot be empty (for example storing directly an array), this can be problematic. However, this method allows the compiler to perform good optimizations. In addition, only a modification of `Any` is necessary, a property which is not verified with other solutions we found.

A.3 Checking Subclassing Relation

Checking if a subclass of another is possible in C++ using templates. The `is_base_and_derived<T,U>` tool from the Boost [7] `type_traits` library performs such a check. Thus, it becomes possible to prevent a class from being instantiated if the virtual types does not satisfy the required subclassing constraints.

B Conditional Inheritance

Static hierarchies presented in Section 3.1 come with simple mechanisms. These parameterized hierarchies can be considered as meta-hierarchies simply waiting for the exact object type to generate real hierarchies. It is generally sufficient for the performance level they were designed for. In order to gain modeling flexibility and genericity, one can imagine some refinements in the way of designing such hierarchies. The idea of the conditional inheritance technique is to adapt automatically the hierarchy according to statically known factors. This is made possible by the C++ two-layer evaluation model (evaluation at compile-time and evaluation at run-time) [31]. In practice, this implies that the meta-hierarchy comes with static mechanisms to discriminate on these factors and to determine the inheritance relations. Thus, the meta-hierarchy can generate different final hierarchies through these variable inheritance links.

```
template <class EXACT>
struct Any
{
  // exact_offset has been computed statically
  // A good compiler can optimize this code and avoid any run-time overhead
  EXACT& exact() {
    return *(EXACT*)((char*)this - exact_offset);
  }

private:
  static const int exact_offset;
  static const EXACT exact_obj;
  static const Any<EXACT>& ref_exact_obj;
};

// Initialize an empty object
// Require a default constructor in EXACT
template <class EXACT>
const EXACT Any<EXACT>::exact_obj = EXACT();

// Upcast EXACT into Any<EXACT>
template <class EXACT>
const Any<EXACT>& Any<EXACT>::ref_exact_obj = Any<EXACT>::exact_obj;

// Compute the offset
template <class EXACT>
const int Any<EXACT>::exact_offset =
  (char*)(&Any<EXACT>::ref_exact_obj)
   - (char*)(&Any<EXACT>::exact_obj);
```

Fig. 16. One method to handle virtual inheritance
The offset between the `Any` address and the address of the `EXACT` class is computed once by using a static object. Since everything is static and const, the compiler can optimize and remove the cost of the subtraction.

```
// ...

template <bool b>
struct type_assert
{};

template <>
struct type_assert<true>
{
  typedef void ret;
};

#define ensure_inheritance(Type, Base)      \
typedef typename                            \
  type_assert<                              \
    is_base_and_derived<Base, Type>::value \
  >::ret  ensure_##Type

template <class EXACT>
struct Image : Any<EXACT>
{
  typedef typename image_traits<EXACT>::point_type point_type;

  // Will not compile if point_type is not a Point since ret
  // is not defined if the assertion fails.
  ensure_inheritance(point_type, Point<point_type>);
};

// ...
```

Fig. 17. Specifying constraints on virtual types

To illustrate the conditional inheritance mechanism, we introduced a UML-like symbol that we called an *inheritance switch*. Figure 18 gives a simple use case. This example introduces an image hierarchy with a concrete class whose inheritance is conditional: `SpecialImage`. `SpecialImage` is parameterized by an unsigned value `Dim`. We want this class to inherit from `Image2d` or `Image3d` depending on the value of `Dim`. `SpecialImage`'s inheritance is thus represented by an inheritance switch. Figure 19 presents the corresponding C++ code. The inheritance switch is implemented by the `ISwitch` trait parameterized by the dimension value. Its specialization on 2 (resp. 3) defines `Image2d` (resp. `Image3d`) as result type. Finally, `SpecialImage<Dim>` only has to inherit from `ISwitch<Dim>`'s result type.

The factors on which inheritance choices are made are necessarily static values. This includes types, provided by *typedef*s or parameterization, as constant integer values. The effective factors are not necessarily directly available data but can be deduced from static pieces of information. Trait structures can then be used to perform more or less complex information deductions. One should also note that the discriminative factors must be accessible while defining the hierarchy. This implies that these factors must be independent from the hierarchy or externally defined. In practice, class-related factors can be made available outside the hierarchy thanks to trait structures and polymorphic *typedef*s (see Section 3.6).

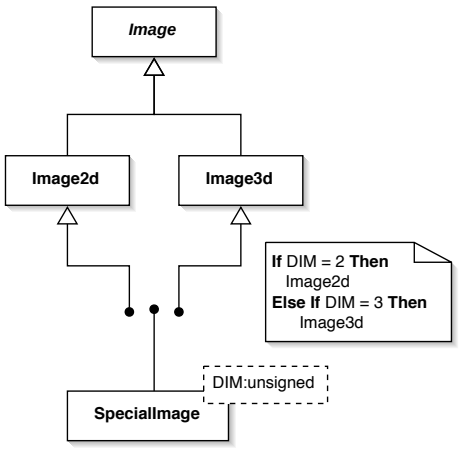

Fig. 18. Simple conditional inheritance sample: UML-like description.

Conditional inheritance mechanisms become particularly interesting when objects are defined by several orthogonal properties. A natural way to handle such a modeling problem is to design a simple sub-hierarchy per property. Unfortunately, when defining the final object, the combinatorial explosion of cases

```
class Image
{
  // ...
};

class Image2d: public Image
{
  // ...
};

class Image3d: public Image
{
  // ...
};
```

```
template <unsigned Dim>
struct ISwitch;

template <>
struct ISwitch<2>
{
  typedef Image2d ret;
};

template <>
struct ISwitch<3>
{
  typedef Image3d ret;
};

template <unsigned Dim>
class SpecialImage
  : public ISwitch<Dim>::ret
{
  // ...
};
```

Fig. 19. Simple conditional inheritance sample: C++ code

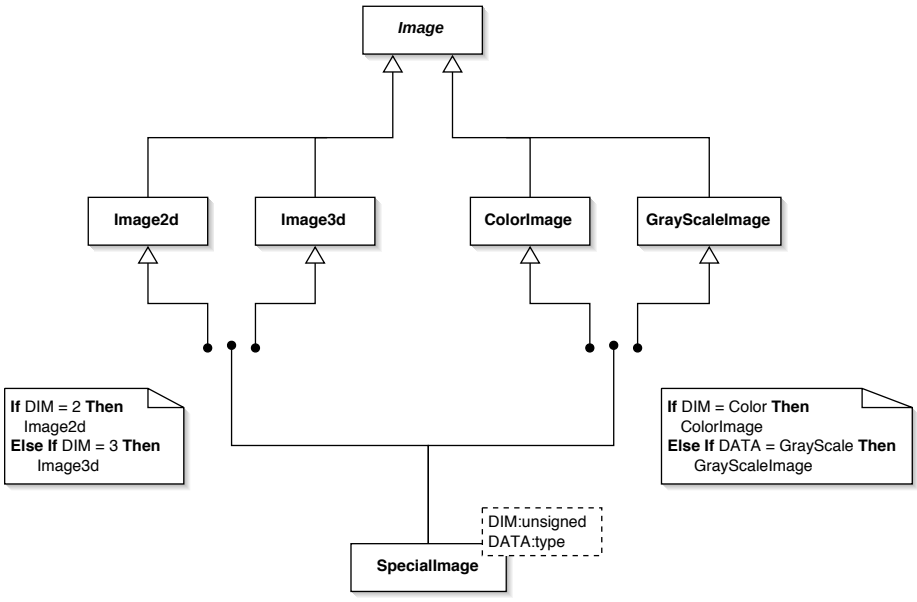

Fig. 20. Conditional inheritance: multiple orthogonal factors.

usually implies a multiplication of the number of concrete classes. Figure 20 illustrates an extension of the previous image hierarchy, with more advanced conditional inheritance mechanisms. We extended the image hierarchy with two classes gathering data-related properties, `ColorImage` and `GrayScaleImage`. The hierarchy is now split into two parallel sub-hierarchies. The first one focuses on the dimension property while the second one focuses on the image data type. The problem is then to define images that gather dimension- and data-related properties without multiplying concrete classes. The idea is just to implement a class template `SpecialImage` parameterized by the dimension value and the data type. Combining conditional and multiple inheritance, `SpecialImage` inherits automatically from the relevant classes. This example introduces the idea of a programming style based on object properties. A `SpecialImage` instance is only defined by its properties and the relevant inheritance relations are deduced automatically.

Finally, mixing conditional inheritance mechanism with other classical and static programming techniques results in powerful adaptive solutions. This work in progress has not been published yet.