Scientific Programming 21 (2013) 43–61
DOI 10.3233/SPR-130362
IOS Press

Template metaprogramming techniques for concept-based specialization

Bruno Bachelet [a,b,*], Antoine Mahul [c] and Loïc Yon [a,b]

[a] *Clermont Université, Université Blaise Pascal, LIMOS, BP 10448, F-63000 Clermont-Ferrand, France*
[b] *CNRS, UMR 6158, LIMOS, F-63171 Aubière, France*
[c] *Clermont Université, Université Blaise Pascal, CRRI, F-63000 Clermont-Ferrand, France*

Abstract. In generic programming, software components are parameterized on types. When available, a static specialization mechanism allows selecting, for a given set of parameters, a more suitable version of a generic component than its primary version. The normal C++ template specialization mechanism is based on the type pattern of the parameters, which is not always the best way to guide the specialization process: type patterns are missing some information on types that could be relevant to define specializations.

The notion of a concept, which represents a set of requirements (including syntactic and semantic aspects) for a type, is known to be an interesting approach to control template specialization. For many reasons, concepts were dropped from C++11 standard, this article therefore describes template metaprogramming techniques for declaring concepts, modeling relationships (meaning that a type fulfills the requirements of a concept), and refinement relationships (meaning that a concept refines the requirements of another concept).

From a taxonomy of concepts and template specializations based on concepts, an automatic mechanism selects the most appropriate version of a generic component for a given instantiation. Our purely library-based solution is also open for retroactive extension: new concepts, relationships, and template specializations can be defined at any time; such additions will then be picked up by the specialization mechanism.

Keywords: Generic programming, template specialization, concept-based overloading/specialization, template metaprogramming

1. Introduction

Generic programming focuses on providing parameterized software components, notably algorithms and data structures, as general as possible and broadly adaptable and interoperable [14], and as efficient as non-parameterized components. Generic programming relies on the notion of a generic component that is a class, a function, or a method with parameters that are types or static values, instead of dynamic values as the usual arguments of functions and methods.

With modern compilers, no loss of efficiency occurs when the parameters of a generic component are bound at compile time, which makes generic programming particularly adapted for scientific programming (e.g., [4,5,15,17,21]). When designing scientific libraries with generic programming, template specialization is a major concern as it allows assembling components together at compile-time in an optimal manner, for instance, selecting the most appropriate code for an algorithm based on the types bound to its template parameters.

The normal C++ template specialization mechanism is based on the type pattern of the template parameters, which is known to have many drawbacks. In this article, we propose a solution based on metaprogramming techniques to control template specialization with concepts. It enables declaring a taxonomy of concepts that can be used to control template specialization: template parameters are constrained by concepts (instead of type patterns) to define a specialization. At instantiation time, an automatic mechanism selects the most appropriate version of a generic component based on the concepts of the types bound to the template parameters.

1.1. Template specialization

Similar to inheritance in object-oriented programming, which allows the specialization of classes, C++

*Corresponding author. Tel.: +33473405044; bruno.bachelet@univ-bpclermont.fr.

provides a mechanism to specialize generic components (called *templates*). At instantiation time, the compiler selects a version, the primary or a specialized one, of a template based on the type pattern of the types (or static values) bound to the parameters. Here is a C++ example of a generic class, `ArrayComparator`, that allows comparing two arrays that contain `N` elements of type `T`.

```
template <class T, int N>
class ArrayComparator {
 public:
  static int
  run(const T * a, const T * b) {
   int i = 0;
   while (i<N && a[i]==b[i]) ++i;
   return
   (i==N ? 0 : (a[i]<b[i] ? -1 : 1));
  }
};
```

The comparison of arrays of characters is presumably more efficient using a built-in function. Therefore, a specialization of the template with `T` = `char` can be provided.

```
template <int N>
class ArrayComparator<char,N> {
 public:
  static int
  run(const char * a, const char * b)
  { return memcmp(a,b,N); }
};
```

1.2. Concepts

In generic programming, instantiating a generic component raises two concerns: (i) how to ensure that a type bound to a parameter fulfills the requirements of the generic component (e.g., any type bound to `T` must provide operators < and == in the `ArrayComparator` class); (ii) how to select the most appropriate specialization of the generic component for a given binding of the parameters (e.g., if type `char` is bound to parameter `T`, then specialization `ArrayComparator<char,N>` is selected; but how to make another type benefit from the same specialization).

To address these issues, the notion of a concept has been introduced [3]. When a type is bound to a parameter of a generic component, it must satisfy a set of requirements represented by a concept. These requirements define syntactic constraints (i.e., on the interface of the type) and semantic constraints (i.e., on the behavior of the type). When a type fulfills the requirements of a concept, it is said that the type "models" the concept. The notion of a specialization between concepts is called "refinement": a concept that includes the requirements of another concept is said to refine this concept.

For instance, let us define the concept `Integral` that captures the requirements of an integral number, and the concept `Numerical` that captures the requirements of any kind of number. One can state that type `int` models concept `Integral`, and concept `Integral` refines concept `Numerical`.

1.3. Challenges with concepts

Concern (i) of the previous section is called "concept checking" [18], and its goal is to detect the types bound to the parameters of a generic component that do not model the required concepts. A concept acts like a contract between the users and the author of a generic component: the author specifies requirements on the parameters using concepts, and the users must bind the parameters to types that fulfill these requirements (i.e., to types that model the specified concepts).

In C++, concepts can not be defined explicitly, and for now, they are only documentation (e.g., *Standard Template Library*). This leads to late error detections, and thus to cryptic error messages [18]: for instance, let us declare the instantiation `ArrayComparator<X,10>`; if type `X` has no operator <, the error will be detected in method `run`, and not at the instantiation point. In some languages, specific features are used to support concepts for generic programming (e.g., type classes in Haskell, deferred classes in Eiffel ... [8]).

In Java and C#, concepts are represented with interfaces, but this approach restricts concepts to syntactic requirements. Moreover, concepts bring more flexibility, because a type is not predestined to model any given concept. A type models a concept either implicitly (it fulfills automatically all the requirements of a concept, cf. "auto concepts" [9]), or explicitly (one has to declare the modeling relationship and to make explicit how the type fulfills the requirements, cf. "concept maps" [9]).

Concern (ii) of the previous section usually deals with "concept-based overloading" [11], as generic programming in C++ has a central notion of generic algorithms where function template specialization is essential. In this article, we propose a solution for the specialization of both function and class templates, so we choose to use the term "concept-based specialization". This approach uses the partial specialization capability of templates that is only available for classes in C++. Therefore, this solution is basically designed

for class template specialization, but is fully usable for function template specialization (as explained in Section 3.2, the specialization process of a function template can be easily delegated to a class template).

The goal of concept-based specialization is to control the specialization of generic components with concepts rather than type patterns. By type pattern, we mean a type or a parameterized type (e.g., `T*` or `vector<T>`), or a template template parameter [23] (e.g., `template <class> class U`). Specialization based on type patterns can lead to ambiguities (the compiler cannot decide between two possible specializations) or false specializations (the compiler selects an unintended specialization), as explained in Section 2. Furthermore, the extensibility of specialization based on type patterns is limited: to control the specialization of a template for a new type (i.e., a type that was not considered in the specialization process before), a new specialization often needs to be defined (unless the type already matches the type pattern of a suitable specialization).

Several attempts have been made to represent concepts in C++. On one hand, implementations for concept checking have been proposed, mainly to ensure interface conformance of types bound to template parameters [16,18]. On the other hand, an implementation for concept-based specialization has been proposed [13]. In this solution, the specialization is based on both the SFINAE (*substitution failure is not an error*) principle [2] and a mechanism to answer the question *"does type T model concept C?"* (through the `enable_if` template). However this approach may still lead to ambiguities.[1]

More recently, an extension of the C++ language to support concepts [7,9] has been proposed to be included into the C++ standard [10]. This extension is available within the experimental compiler ConceptGCC [9,12], and is also implemented as ConceptClang in Clang, a C language family front-end for the LLVM compiler [22]. The inclusion of concepts has been deferred from C++11 standard, and a new extension, Concepts Lite [20], has been designed and implemented as a branch of GCC 4.8. This extension introduces "template constraints", a.k.a. "concepts lite", which is a subset of concepts that allows the use of predicates to constrain template parameters. It is undeniably an improvement on existing solutions to control template specialization, but concepts lite can not be considered to be full concepts, as some features are missing. The long-term goal of this extension is to propose a complete definition of concepts.

A library-based emulation of C++0x concepts, called the Origin Concept library, has also been developed based on new features of C++11 [19]. The primary goal of this solution is to provide a uniform interface to defining and using concepts, but no new technique is proposed for concept-based specialization as it favors the Boost's `enable_if` approach [13].

1.4. Proposal

In this article, a solution focused on the concept-based specialization aspect only is proposed. Due to portability concerns, our goal is to provide a purely library-based solution that could be used with any standard C++ compiler, and no need of an additional tool. The proposed technique enables declaring concepts, modeling relationships, and refinement relationships. Once a taxonomy of concepts has been declared, it can be used to control the specialization of templates: to define a specialization, concepts (instead of type patterns) are used to constrain parameters. At instantiation time, the most appropriate version of a template is selected based on the concepts modeled by the types bound to the parameters: a metaprogram determines, for each one of these types, the most specialized concept to consider for this instantiation.

Even if the proposed technique does not detect directly concept mismatches to provide more understandable error messages, it needs to perform some checking on concepts to lead the specialization process. The checking is only based on "named conformance" [16] (i.e., checking on whether a type has been declared to model a given concept), and does not consider "structural conformance" (i.e., checking on whether a type implements a given interface).

One key idea of generic programming is to express components with minimal assumptions [14], therefore our solution is open for retroactive extension:

- A new concept or a new relationship (modeling or refinement) can be declared at any time. The declaration of such relationships is distinct from the definition of types and concepts, contrary to class inheritance and interface implementation that have to be declared with the definition of classes.
- A new specialization based on concepts can be defined at any time, but only for templates that have been prepared for concept-based specialization.

[1] As explained in Boost documentation: http://www.boost.org/doc/libs/release/libs/utility/enable_if.html.

Section 2 discusses several issues encountered with template specialization, and shows how concepts can be used to bypass most of them. Section 3 presents template metaprogramming techniques for concept-based specialization, and an example using our library-based solution. Section 4 reports the compile-time performance of the library depending on the number of concepts and the number of relationships (modeling and refinement) declared in a program. The full source code of the library and of the examples is available for download.[2]

2. Issues with template specialization

This section presents several issues that may occur with template specialization based on type patterns, and how they can be addressed with concepts:

(i) Some types that can be considered somehow similar (e.g., with a common subset of operations in their interface) could be bound to the same specialization of a template, but if they have no type pattern in common, several specializations must be defined.

(ii) A specialization based on type patterns may lead to false specialization (i.e., an unintended specialization), because a type pattern can be insufficient to capture the requirements that a template needs for a parameter.

Existing solutions, proposed by McNamara and Smaragdakis [16] and Järvi et al. [13], that use concepts to control template specialization in C++ are discussed in this section. Refinement relationships appear to be necessary to address another issue:

(iii) A type can possibly be bound to different specializations of a template, when it models concepts that constrain different specializations. If there is no clear ordering between these concepts, to choose one specialization is not possible.

The solution presented in this paper allows concept-based specialization for both function and class templates, but it is basically designed for class template specialization, because, as detailed in Section 3, it uses the partial specialization capability of templates that is not available for functions. However, concept-based overloading can easily be enabled: the function template to be specialized calls a static method of a class template where the whole specialization process is defined with our approach (cf. Section 3.2 for details). Therefore, the discussion in this paper is illustrated with an example of class template specialization.

[2]Source code is available at: http://forge.clermont-universite.fr/projects/show/cpp-concepts.

2.1. Specialization based on type patterns

As an example, we propose to develop a generic class, `Serializer`, to store the state of an object into an array of bytes (the "deflate" action), or to restore the state of an object from an array of bytes (the "inflate" action). The primary version of the template, which makes a bitwise copy of an object in memory, is defined as follows.

```
template <class T> class Serializer {
 public:
  static int
  deflate(char * copy, const T & object);
  static int
  inflate(T & object, const char * copy);
};
```

This version should not be used for complex objects, such as containers, where the internal state may have pointers that should not be stored (because these versions of the deflate and inflate actions would lead to memory inconsistency after restoring). Let us define a specialized version of `Serializer` for the sequence containers of the STL (*Standard Template Library*), such as vectors and lists.

```
template
<class T, class ALLOC,
 template <class,class> class CONTAINER>
class Serializer< CONTAINER<T,ALLOC> > {
 public:
  static int
  deflate(char * copy, const
          CONTAINER<T,ALLOC> & container);
  static int
  inflate(CONTAINER<T,ALLOC> & container,
          const char * copy);
};
```

For this specialization, parameter `CONTAINER` is constrained with the type pattern of the STL sequence containers: they are generic classes with two parameters, the type `T` of the elements to be stored, and the type `ALLOC` of the object used to allocate elements.

Now, let us consider associative containers of the STL, such as sets and maps. Their type pattern is different from the one of sequence containers (they have at least one more parameter `COMP` to compare elements), whereas sequence and associative containers have a common subset of operations in their interface that should allow defining a common specialization of `Serializer`. However, as specialization is based on type pattern for now, another specialization of `Serializer` is necessary.

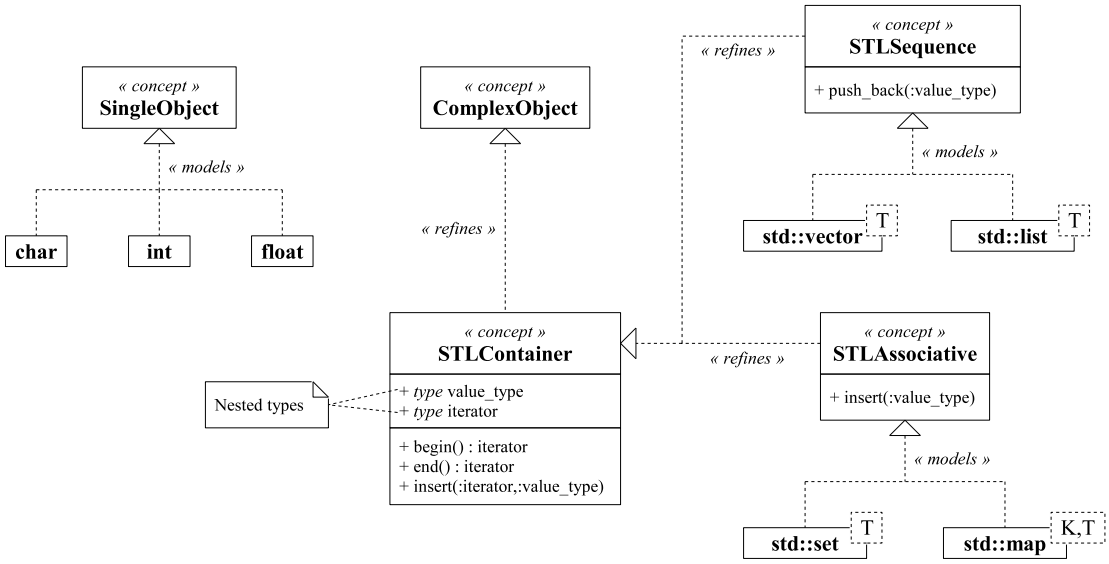

Fig. 1. Taxonomy of concepts for the serialization example.

```
template <class T, class COMP,
          class ALLOC,
          template <class,class,class>
          class CONTAINER>
class
Serializer< CONTAINER<T,COMP,ALLOC> >
{ [...] };
```

Notice that this specialization of `Serializer` is only suitable for sets, and not for maps, because their type pattern is different: maps have an additional parameter `K` for the type of the keys associated with the elements of the container. The specialization `Serializer< CONTAINER<K, T, COMP, ALLOC> >` is necessary for maps, whereas maps and sets have a common subset of operations in their interface and should share the same specialization.

The specialization for sets has been written having only STL associative containers in mind, but any type matching the same type pattern can be bound to the specialization. Thus, there could be an unintended match. For instance, the `std::string` class of the C++ standard library is an alias for a type that matches the type pattern of sets:

```
std::basic_string< char,
                   std::char_traits<char>,
                   std::allocator<char> >
```

The first two issues presented in the introduction of the section have been illustrated here. They could be addressed with concepts:

(i) "Similar" types (i.e., sharing a common subset of features) could model the same concept, and a specialization for this concept could be defined. Therefore, "similar" types with different type patterns could be bound to the same specialization.

(ii) Concepts could avoid false specialization: with template specialization based on concepts, any template parameter could be constrained by a concept, and only types that model this concept could be bound to the parameter. This way, only the types that satisfy the requirements of a specialization could be considered.

For the example of serialization discussed here, Fig. 1 proposes concepts and their relationships. The `SingleObject` and `STLContainer` concepts are defined to provide two specializations for `Serializer`: one based on bitwise copy, and another one based on the common subset of operations shared by all STL containers, respectively. As sequence and associative containers are of different natures, one can think of different ways of optimizing the serialization operations. For this reason, the `STLContainer` concept is refined into the `STLSequence` and `STLAssociative` concepts to provide specializations of `Serializer` for sequence containers and associative containers respectively.

2.2. *Specialization based on concepts*

Existing solutions for concept-based specialization in C++ [13,16] are discussed here. They use concepts

to guide the specialization of templates, and enable addressing the two first issues presented in the introduction of the section. However, about the third issue, that is to find the most appropriate specialization when a type can possibly be bound to several specializations, the solutions presented here are not fully satisfactory.

2.2.1. Concept-based dispatch

The solution of McNamara and Smaragdakis [16] implements concepts with "static interfaces" in C++, and proposes a "dispatch" mechanism to control template specialization with concepts. The solution is based on the `StaticIsA` template that provides some concept checking: `StaticIsA<T,C>::valid` is true if `T` models concept `C`. Let us assume that `StaticIsA` answers accordingly to the taxonomy of concepts of Fig. 1 (see the source code for details). Here is an example of the dispatch mechanism for the specialization of the `Serializer` generic class.

```
enum { IS_SINGLE_OBJECT, IS_STL_CONTAINER,
       IS_STL_SEQUENCE,
       IS_STL_ASSOCIATIVE, UNSPECIFIED };

template <class T> struct Dispatcher {
 static const int which
  = StaticIsA<T,STLAssociative>::valid ?
    IS_STL_ASSOCIATIVE
    : StaticIsA<T,STLSequence>::valid  ?
    IS_STL_SEQUENCE
    : StaticIsA<T,STLContainer>::valid ?
    IS_STL_CONTAINER
    : StaticIsA<T,SingleObject>::valid ?
    IS_SINGLE_OBJECT
    : UNSPECIFIED;
};

template <class T>
struct ErrorSpecializationNotFound;

template
<class T, int = Dispatcher<T>::which>
class Serializer
: ErrorSpecializationNotFound<T> {};

template <class T>
class Serializer<T,IS_SINGLE_OBJECT>
{ [...] };
template <class T>
class Serializer<T,STL_CONTAINER>
{ [...] };
template <class T>
class Serializer<T,STL_SEQUENCE>
{ [...] };
template <class T>
class Serializer<T,STL_ASSOCIATIVE>
{ [...] };
```

The `Dispatcher` template goes through all the concepts (in a well-defined order) until its parameter `T` models a concept. The symbolic constant associated with the found concept is stored in the `which` attribute of `Dispatcher`. For instance, `Dispatcher< vector<int> >::which` is equal to `IS_STL_SEQUENCE`.

Compared to the version of the `Serializer` template based on type patterns, there is an additional parameter with a default value that is the answer of the dispatcher for parameter `T`. This value is used rather than the type pattern of `T` to define the specializations of `Serializer`. This way, it is possible to provide a specialization for any concept. For instance, `Serializer< vector<int> >` instantiates in fact `Serializer< vector<int>, IS_STL_SEQUENCE >` and matches the specialization for the `STLSequence` concept.

Notice that the primary version of the template inherits from a class that is only declared, the aim being that this version could not be instantiated. This way, compilation errors related to the fact that `T` has been instantiated with a wrong type occurs at the instantiation of `Serializer`, rather than inside the code of `Serializer` where it tries to call invalid operations on `T`. This solution avoids usual error messages that could be cryptic for the user [18].

In this solution, a dispatcher (and dispatch rules) is defined for nearly each context of specialization (i.e., for each template that is specialized), which can quickly become tedious. A dispatcher can be reused, but only between specialization contexts that are identical (i.e., providing specializations based on the same concepts). Moreover, to define a new specialization for a template implies to change its dispatch rules. A solution where the dispatch rules, for each context of specialization, are automatically deduced from the modeling and refinement relationships of the taxonomy of concepts should be provided.

2.2.2. Concept-based overloading

The solution proposed by Järvi et al. [13] relies on the `enable_if` template, which can be found in the Boost Library [1], and the SFINAE (*substitution failure is not an error*) principle [2], to provide some control on template specialization with concepts. The definition of `enable_if` is recalled here.

```
template <bool B, class T = void>
struct enable_if_c { typedef T type; };
```

```
template <class T>
struct enable_if_c<false,T> {};

template <class COND, class T = void>
struct enable_if
: enable_if_c<COND::value,T> {};
```

At instantiation time, if `B` is true, there is a nested type `type` inside `enable_if_c`, and thus inside `enable_if`, if its parameter `COND` has an attribute `value` set to true. Let us assume that, for each concept `C` of the taxonomy of Fig. 1, a template `is_C<T>` is defined so `is_C<T>::value` is true if `T` models concept `C` (see the source code for details). Here is an example of the use of `enable_if` for the specialization of the `Serializer` generic class.

```
template <class T, class = void>
class Serializer
: ErrorSpecializationNotFound<T> {};

template <class T> class
Serializer<T, typename
           enable_if< is_SingleObject<T> >
           ::type>
{ [...] };

template <class T> class
Serializer<T, typename
           enable_if< is_STLContainer<T> >
           ::type>
{ [...] };

template <class T> class
Serializer<T, typename
           enable_if< is_STLSequence<T> >
           ::type>
{ [...] };

template <class T> class
Serializer<T, typename
           enable_if<
            is_STLAssociative<T> >
           ::type>
{ [...] };
```

The SFINAE principle is: if there is an error when binding types to the parameters of a template specialization, this specialization is discarded. For instance, the instantiation `Serializer< vector<int> >` implies an attempt to instantiate `Serializer< vector<int>, typename enable_if< is_SingleObject< vector<int> > >::type>`,[3] and because `enable_if` has no member `type` in this case, the specialization for concept `SingleObject` is ignored.

This solution keeps only the specializations constrained with a concept modeled by the type bound to `T`. If more than one specialization remain, the compiler has to deal with an ambiguity: for instance, `vector<int>` models both `STLContainer` and `STLSequence` concepts. This ambiguity could be avoided: concept `STLSequence` is more specialized than concept `STLContainer`, so the specialization for `STLSequence` should be selected.

2.2.3. Conclusion

In this section, solutions have been presented to control template specialization with concepts. Concept-based dispatch allows considering refinement relationships, but the selection of the specialization is not automatic and requires some specific code for each context of specialization. At the opposite, concept-based overloading allows an automatic selection of the specialization, but is not able to deal with ambiguities that could be avoided considering refinement relationships.

3. A solution for concept-based specialization

Concepts appear to be better suited than type patterns to control template specialization, but to our knowledge, there is no solution that addresses all the issues brought up in the previous section. We propose here template metaprogramming techniques that enable defining a taxonomy of concepts, and using this taxonomy to automatically select the most appropriate specialization of a template.

Two main goals have guided our choices toward this library-based solution: to provide a fully portable C++ code (meaning that we do not want to modify the C++ language itself, and to provide an extra tool to preprocess the code), and to be open for retroactive extension (new concepts, relationships, and template specializations can be defined at any time).

3.1. Example

Let us consider the example of the `Serializer` generic class with our solution. In a first step, the taxonomy of concepts of Fig. 1 is defined: concepts and relationships (modeling and refinement) are declared. Then, the `Serializer` template is defined: first its primary version, and then its specializations for each concept. Details on the implementation of the library are presented afterward.

[3] `typename` is necessary in C++ to declare that the member `type` is actually a type and not a value.

Concepts declaration

```
gnx_declare_concept(SingleObject);
gnx_declare_concept(ComplexObject);
gnx_declare_concept(STLContainer);
gnx_declare_concept(STLSequence);
gnx_declare_concept(STLAssociative);
```

Modeling and refinement relationships

```
template <> struct gnx_models_concept<char,SingleObject>  : gnx_true {};
template <> struct gnx_models_concept<int,SingleObject>   : gnx_true {};
template <> struct gnx_models_concept<float,SingleObject> : gnx_true {};

template <class T>
struct gnx_models_concept<std::vector<T>,STLSequence>   : gnx_true {};

template <class T>
struct gnx_models_concept<std::list<T>,STLSequence>     : gnx_true {};

template <class T>
struct gnx_models_concept<std::set<T>,STLAssociative>   : gnx_true {};

template <class K, class T>
struct gnx_models_concept<std::map<K,T>,STLAssociative> : gnx_true {};

template <>
struct gnx_models_concept<STLContainer,ComplexObject>   : gnx_true {};

template <>
struct gnx_models_concept<STLSequence,STLContainer>     : gnx_true {};

template <>
struct gnx_models_concept<STLAssociative,STLContainer>  : gnx_true {};
```

Template primary version

```
struct SerializerContext;

template <class T,class = gnx_best_concept(SerializerContext,T)>
class Serializer : ErrorSpecializationNotFound<T> {};
```

Template specialized versions

```
template <>
struct gnx_uses_concept<SerializerContext,SingleObject> : gnx_true {};

template <class T> class Serializer<T,SingleObject> { [...] };

template <>
struct gnx_uses_concept<SerializerContext,STLContainer> : gnx_true {};

template <class T> class Serializer<T,STLContainer> { [...] };

template <>
struct gnx_uses_concept<SerializerContext,STLSequence> : gnx_true {};

template <class T> class Serializer<T,STLSequence> { [...] };

template <>
struct gnx_uses_concept<SerializerContext,STLAssociative> : gnx_true {};

template <class T> class Serializer<T,STLAssociative> { [...] };
```

Concepts are declared using macro gnx_declare_concept. The modeling and refinement relationships are equally declared using metafunction gnx_models_concept. To control the specialization, a "specialization context" must be declared (SerializerContext in our example). Each specialization of Serializer based on a concept must be declared and associated with the specialization context SerializerContext, using metafunction gnx_uses_concept. The most appropriate concept for a type bound to parameter T is automatically determined by the gnx_best_concept macro and stored in an additional parameter of the Serializer template, enabling template specialization based on this parameter.

3.2. Concept-based overloading

Our solution uses the partial specialization capability of templates that is not available for functions in C++. However, concept-based overloading of functions is possible with very few extra code. For instance, let us define function inflate that calls the inflate static method of Serializer; this function benefits indirectly of the concept-based specialization of the class:

```
template <class T>
inline int
inflate(T & object, const char * copy)
{ return
  Serializer<T>::inflate(object,copy); }
```

3.3. Metafunctions

Some fundamental metafunctions are necessary to implement our library. These generic classes are common in metaprogramming libraries (e.g., in the Boost MPL Library). A metafunction acts similarly to an ordinary function, but instead of manipulating dynamic values, it deals with metadata, meaning entities that can be handled at compile time in C++: mainly types and static integer values [1]. In order to manipulate equally types and static values in metafunctions, metadata are embedded inside classes, as follows.[4]

[4]We chose to prefix all the metafunctions and macros of our library with "gnx_". We also chose to use our own metafunctions instead of the ones of MPL for two reasons: we only need few of them and we want to be able to easily change their implementation to optimize the compile time.

```
template <class TYPE>
struct gnx_type { typedef TYPE type; };

template <class TYPE, TYPE VALUE>
struct gnx_value
{ static const TYPE value = VALUE; };

typedef gnx_value<bool,true>  gnx_true;
typedef gnx_value<bool,false> gnx_false;
```

Template gnx_type<T> represents a type and provides a type member type that is T itself. The same way, template gnx_value<T,V> represents a static value and provides an attribute value that is the value V of type T. Based on template gnx_value, types gnx_true and gnx_false are defined to represent the boolean values.

The parameters of a metafunction, which are the parameters of the template representing the metafunction, are assumed to be metadata (i.e., to be classes with a member type or value). The "return value" of a metafunction is implemented with inheritance: the metafunction inherits from a class representing a metadata. This way the metafunction itself has a member type or value, and can be a parameter of another metafunction. Here are metafunctions necessary for the discussion of this section.

```
template <class TYPE1, class TYPE2>
struct gnx_same : gnx_false {};

template <class TYPE>
struct gnx_same<TYPE,TYPE> : gnx_true {};

template
<class TEST, class IF, class ELSE,
 bool = TEST::value>
struct gnx_if : ELSE {};

template
<class TEST, class IF, class ELSE>
struct gnx_if<TEST,IF,ELSE,true> : IF {};
```

Metafunctions usually need template specialization to fully implement their behavior. Metafunction gnx_same determines whether two types are identical: gnx_same<T1,T2> inherits from gnx_true if T1 and T2 are the same type, or from gnx_false otherwise. Thus, the value returned by metafunction gnx_same<T1,T2> is stored in its value attribute. Metafunction gnx_if acts similarly to the common if instruction: gnx_if<T,A,B> inherits from A if T::value is true, or from B otherwise. If A and B represent metadata, then gnx_if<T,A,B> inherits the member nested in A or B.

3.4. Declaring concepts

This section describes how concepts are represented and automatically indexed in order to be manipulated afterward by metafunctions that control concept-based specialization. In our solution, an empty structure, defined using macro `gnx_declare_concept`, represents a concept. For instance, `struct STLContainer {};` declares concept `STLContainer`.

3.4.1. Typelists

Concepts also need to be stored in a container, in order to be manipulated by metafunctions, for instance, to determine the most appropriate concept for a template specialization. Notably, the "typelist" technique [2,6], based on metaprogramming, allows building a static linked list to store types, and can be defined as follows.

```
template <class CONTENT, class NEXT>
struct gnx_list {
 typedef CONTENT content;
 typedef NEXT    next;
};

struct gnx_nil {};
```

Type `gnx_nil` represents "no type" (`void` is not used, as it could be a valid type to be stored in a list), and is used to indicate the end of a list. For instance, to store the `STLSequence` and `STLAssociative` concepts in a list:

```
typedef
gnx_list< STLSequence,
          gnx_list<STLAssociative,
                   gnx_nil> > mylist1;
```

Common operations on linked lists can be defined on typelists [2]. For instance, to add concept `STLContainer` in the previous list:

```
typedef gnx_list<STLContainer,mylist1>
mylist2;
```

However, typelists are too static for our needs: in the previous example, list `mylist1` cannot be modified to add a type, so a new list `mylist2` has to be created instead. In the following section, a solution is proposed to build a list of concepts that can be modified at compile time to add new concepts, without changing the identifier of the list. Typelists will nevertheless be useful in our solution for several metafunctions where operations for merging and searching lists of concepts are necessary.

3.4.2. Indexing concepts

To design a list where concepts can be added at any time, a mechanism for indexing the concepts is proposed. The metafunction `gnx_concept` is defined: it has one parameter that is an integer value, and it returns the concept associated with this number. Adding a concept to the list is performed by the specialization of the metafunction.

```
template <int ID> struct gnx_concept
: gnx_type<gnx_nil> {};

template <> struct gnx_concept<1>
: gnx_type<STLContainer> {};
template <> struct gnx_concept<2>
: gnx_type<STLSequence> {};
[...]
```

Indexing the concepts by hand is not acceptable, so a solution to get the number of concepts already in the list is needed. For this purpose, a preliminary version of the `gnx_nb_concept` metafunction is proposed. It goes through all the concepts in the list by increasing an index until finding `gnx_nil`.

```
template <int N = 0> struct gnx_nb_concept
: gnx_if< gnx_same<typename
                   gnx_concept<N+1>::type,
                   gnx_nil>,
          gnx_value<int,N>,
          gnx_nb_concept<N+1>
        > {};
```

For an automatic indexing of the concepts, one would use the return value of metafunction `gnx_nb_concept` to determine the next index to assign to a new concept.

```
template <> struct
gnx_concept<gnx_nb_concept<>::value+1>
: gnx_type<STLContainer> {};

template <> struct
gnx_concept<gnx_nb_concept<>::value+1>
: gnx_type<STLSequence> {};

[...]
```

However, this solution does not work, because using `gnx_nb_concept<>` infers that `gnx_concept` is instantiated from `gnx_concept<0>` to `gnx_concept<N+1>`, where `N` is the number of indexed concepts. Due to this fact, specializing `gnx_concept` for `STLContainer` and `STLSequence` in the previous example is not possible, because `gnx_concept<N+1>` has already been instantiated based on the primary version of `gnx_concept`. To

eliminate this flaw, an additional parameter, called here "observer", is added to both metafunctions `gnx_concept` and `gnx_nb_concept`.

```
template <int ID, class OBS = gnx_nil>
struct gnx_concept : gnx_type<gnx_nil> {};

template <class OBS, int N = 0>
struct gnx_nb_concept
: gnx_if< gnx_same<typename
                   gnx_concept<N+1,OBS>
                   ::type, gnx_nil>,
          gnx_value<int,N>,
          gnx_nb_concept<OBS,N+1>
        > {};
```

The idea is to provide a different observer each time the concepts need to be counted to determine the next index to assign to a new concept: the new concept itself will be the observer. With this solution, counting the concepts with observer `OBS` induces the instantiation of `gnx_concept<N+1,OBS>`, so any specialization for index `N+1` with an observer other than `OBS` is still possible. Finally, concepts are indexed as follows.

```
template <class OBS> struct
gnx_concept<gnx_nb_concept<STLContainer>
            ::value+1, OBS>
: gnx_type<STLContainer> {};

template <class OBS> struct
gnx_concept<gnx_nb_concept<STLSequence>
            ::value+1, OBS>
: gnx_type<STLSequence> {};

[...]
```

To declare a concept in a single and easy instruction, as presented in the example at the start of the section, the `gnx_declare_concept` macro is defined.

```
#define gnx_declare_concept(CONCEPT)    \
 struct CONCEPT {};                     \
                                        \
 template <class OBS> struct            \
 gnx_concept<gnx_nb_concept< CONCEPT > \
             ::value+1, OBS>            \
 : gnx_type< CONCEPT > {}
```

To conclude, the `gnx_nb_concept` metafunction requires O(n) operations, where n is the number of concepts already declared in the program. Hence, at compile time, indexing n concepts requires O($\sum_{i=1}^{n} i$) = O(n^2) operations.

3.5. Modeling and refinement relationships

This section describes how modeling and refinement relationships are represented to build a taxonomy of concepts, and presents several metafunctions to get useful information from a taxonomy. Modeling relationships, between a type and a concept, and refinement relationships, between two concepts, are declared equally in our solution with the `gnx_models_concept` metafunction.

```
template <class TYPE_OR_CONCEPT,
          class CONCEPT>
struct gnx_models_concept : gnx_false {};
```

The primary version of the template returns false, and the relationships are declared through specializations of the template: if type `X` models concept `C` (or concept `X` refines concept `C`), then specialization `gnx_models_concept<X,C>` must return true.

```
template <> struct gnx_models_concept<X,C>
: gnx_true {};
```

Notice that `gnx_models_concept` provides an answer for a direct relationship only. If a type `T` models a concept `C1` that refines a concept `C2`, this metafunction returns false for a relationship between `T` and `C2`. Additional metafunctions, necessary in our solution to find any relationship between a type and a concept (or between two concepts), are briefly presented below (see the source code for details).

- Metafunction `gnx_direct_concepts<X>` provides a list (using the typelist technique) of all the concepts directly modeled by a type (or refined by a concept) `X`. It goes through all the concepts using their index, and checks whether `X` models (or refines) each concept using metafunction `gnx_models_concept`. Assuming that to retrieve a concept from its index (i.e., to call metafunction `gnx_concept`) is a constant time operation, metafunction `gnx_direct_concepts` requires O(n) operations, where n is the number of concepts declared in the program.
- Metafunction `gnx_all_concepts<X>` provides a list of all the concepts directly or indirectly modeled by a type (or refined by a concept) `X`. It calls `gnx_direct_concepts` to list the concepts directly related to `X`, and recursively gets all the concepts related to each one of the direct concepts. This metafunction requires O($n^2 + rn$) operations, where r is the number of modeling and refinement relationships declared

in the program: at worst, all the n concepts are asked for their direct concepts (i.e., a call to metafunction `gnx_direct_concepts`), which requires O(n^2) operations; to build the final list, at worst all the r relationships are considered, and each time the list of the currently found concepts is merged with the list of the newly found concepts, which requires O(rn) operations (at worst $2n$ operations are necessary for the merging, as it avoids duplicates).
- Metafunction `gnx_matches_concept<X,C>` returns whether a type (or a concept) `X` models (or refines) a concept `C`, directly or indirectly. This metafunction searches for `C` in the list of concepts provided by metafunction `gnx_all_concepts` and requires O($n^2 + rn$) operations: O($n^2 + rn$) operations to build the list, and O(n) for the search.

3.6. Specialization based on concepts

This section explains how to declare template specializations based on concepts and how the "best" specialization is automatically selected at instantiation time.

3.6.1. Declaring specializations

With our solution, controlling the specialization of a template with concepts that constrain one of its parameters implies an additional parameter. In the example, class `Serializer` has initially one parameter `T`, and based on different concepts that types bound to `T` might model, several specializations of `Serializer` must be provided. For this purpose, an extra parameter is added to `Serializer`.

```
template <class T, class
 = gnx_best_concept(SerializerContext,T)>
class Serializer
: ErrorSpecializationNotFound<T> {};
```

This additional parameter is the most specialized concept that a type bound to `T` models and that is of interest for the specialization of `Serializer`. This "best" concept is obtained using the `gnx_best_concept` macro, which eases the call to metafunction `gnx_contextual_concept`.

```
#define gnx_best_concept(CONTEXT,TYPE) \
 typename                              \
 gnx_contextual_concept<CONTEXT,TYPE>  \
 ::type
```

Notice that metafunction `gnx_contextual_concept` requires a "specialization context", which is a type that represents the context of a given template specialization. Each template that uses specialization based on concepts requires its own context.

There are two main reasons for this notion of a specialization context: (i) as seen previously, metafunction `gnx_nb_concept`, called by many metafunctions, requires an observer to perform correctly and to allow defining new concepts at any time, and this observer will be the specialization context; (ii) as explained in the next section, it is necessary to known which concepts are of interest for a given specialization context (i.e., which concepts are involved in the specialization control), and we want a library-based solution compliant with the C++ standard, so each one of these concepts must be associated with the context using the `gnx_uses_concept` metafunction.

```
template <> struct
gnx_uses_concept<SerializerContext,
                 STLContainer>
: gnx_true {};
```

In our example, the `SerializerContext` context has been declared for the specialization of `Serializer`. Among others, concept `STLContainer` is used to define a specialization of `Serializer`, so `gnx_uses_concept` is specialized (the same way as `gnx_models_concept`) to specify that concept `STLContainer` is used in the `SerializerContext` context.

3.6.2. Selecting the best specialization

Based on the list of concepts declared in a specialization context, and a taxonomy of concepts, metafunction `gnx_contextual_concept` determines the "best" concept for a type `T`, meaning the most specialized concept that `T` models and that is of interest for the context of specialization.

If we consider the taxonomy of concepts of Fig. 2, and a context `X` that provides specializations for concepts `C1`, `C2` and `C5` in this example, the following best concepts should be selected.

- For type `A`: concept `C2`, candidates are `C1` and `C2`, but `C2` is more specialized.
- For type `B`: no concept, there is no candidate in the context's list, `gnx_nil` is returned.
- For type `C`: concept `C5`, it is the only choice.
- For type `D`: concepts `C1` or `C5`, both concepts are valid (because `D` models both), and there is no relationship between them to determine that one is more specialized than the other. The selected one depends on the implementation of `gnx_contextual_concept`. In our solution, the concept with the highest index is selected. But

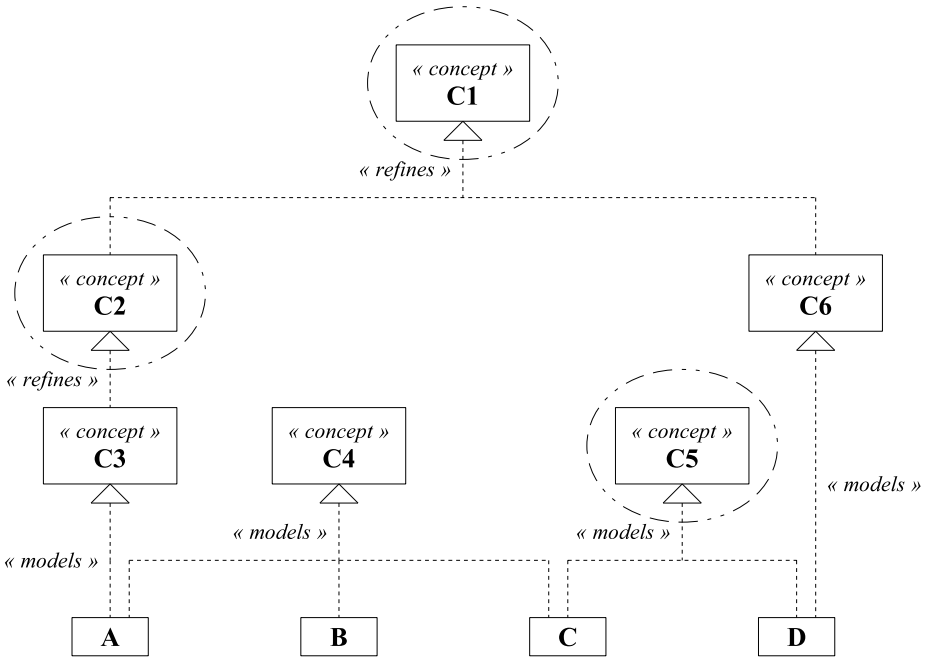

Fig. 2. Example of best concept selection.

to avoid this arbitrary selection, one can add relationships to the taxonomy of concepts, or can specialize metafunction `gnx_contextual_concept` for type `D` in context `X`.

Metafunction `gnx_contextual_concept<X, T>` goes through the list of all the concepts modeled directly or indirectly by type `T` (provided by `gnx_all_concepts<T>`), and selects the one that does not refine directly or indirectly any other concept in the list (using metafunction `gnx_matches_concept`) and that is declared in context `X`. This metafunction requires O($n^2 + rn$) operations: O($n^2 + rn$) operations to build the list, and O(n) to select the best candidate (because `gnx_all_concepts` has already achieved all the necessary `gnx_matches_concept` instantiations).

3.7. Conclusion

Two steps are necessary for concept-based specialization with our solution: (i) to declare concepts and modeling/refinement relationships in order to define a taxonomy of concepts; (ii) for each context of specialization, to declare the concepts that are used to control the specialization. These steps are not monolithic, and new concepts, relationships, and specializations can be defined at any time (but before the first instantiation of the targeted generic component), which provides high flexibility with minimal assumptions about components.

The selection of the best specialization is fully automatic and safe as long as the modeling and refinement relationships are correct. Notice that those relationships, declared manually with our solution, could be automated using a mechanism to check structural conformance, such as the "concept classes" introduced in [19] (i.e., type traits that indicate whether a type models a concept), or the `StaticIsA` template of Section 2.2.1:

```
template <class TYPE, class CONCEPT>
struct gnx_models_concept
: gnx_value<bool, StaticIsA<TYPE,CONCEPT>
            ::valid> {};
```

3.7.1. Pitfalls

However, a few issues occur with our solution. First, the type pattern of any template specialized based on concepts is altered: for each primary template parameter, an extra "hidden" parameter may be added to get its best concept. For instance, users of the `Serializer` generic class could think that this template has only one parameter, whereas it actually has two.

Secondly, the notion of an observer, which is totally hidden from the users of a template specialized based on concepts, has been introduced to bypass an instanti-

ation problem with metafunction gnx_nb_concept (cf. Section 3.4.2). However there are very specific situations where the issue remains. For instance, the following specialization may be troublesome.

```
template <> class Serializer<int>
{ [...] };
```

It induces the full instantiation of Serializer that forces the default value of the hidden parameter to be instantiated (i.e., gnx_contextual_concept<SerializerContext,int>), which itself forces gnx_nb_concept to be instantiated for observer SerializerContext. If concepts are added after this code, another call to metafunction gnx_contextual_concept with context SerializerContext will ignore the new concepts. Hence, one should avoid to instantiate gnx_contextual_concept before the final use of the targeted template. In our example, the full instantiation can be avoided as follows.

```
template <class CONCEPT>
class Serializer<int,CONCEPT>
{ [...] };
```

3.7.2. Usage

For the designers of generic libraries, using our solution for concept-based specialization helps improving extensibility and maintainability, as specialization is controlled by concepts that are more relevant and robust to guide the process than type patterns, and because retroactive extension is possible (i.e., a specialization, a concept, or a relationship, can be added at any time, with no impact on existing code).

Designing a generic library with concept-based specialization also offers new possibilities to end-users. They have the opportunity to easily extend the taxonomy of concepts. For instance, if std::deque were not yet in the taxonomy, one can associate it with concept STLSequence, and automatically, std::deque benefits from the specialization of Serializer for STLSequence. Nonintrusively, the users can also define new specializations for concepts that were not foreseen by the designers of the generic library, but it requires some documentation on this library.

4. Compile-time performance

The theoretical performance of the metafunctions of our solution has been studied in this paper. We assumed some operations of the compiler to be constant time, so it is important to confirm the theoretical performance with practical experiments. The initial implementation of the library, that is presented in this paper, is meant for understanding. Thus, a second version of the library has been designed to optimize the compile time. Nevertheless, the metaprogramming techniques and how to use the library remain unchanged with this new version. To understand what kind of optimization has been performed, let us discuss on the following example of metafunction.

```
template <class A> struct plain_meta
: gnx_if < test<A>, branch1<A>,
           branch2<A> > {};
```

At instantiation time, both branch1<A> and branch2<A> are instantiated. But depending on the value of test<A>, only one of the two templates actually needs to be instantiated. In our library, such cases occur many times and lead to a lot of unnecessary instantiations. Metafunctions can be rewritten using an intermediate template that hides the two possible branches of the conditional statement in separate specializations of the template. Here is an optimized version of the example that shows the technique that has been applied on all the metafunctions of the library.

```
template <class A, bool TEST>
struct _optimized_meta_;

template <class A> struct optimized_meta
: _optimized_meta_<A, test<A>::value> {};

template <class A>
struct _optimized_meta_<A,true>
: branch1<A> {};
template <class A>
struct _optimized_meta_<A,false>
: branch2<A> {};
```

The tests presented here have been performed with the optimized version[5] of the library on an Intel Core 2 Duo T8100 2.1 GHz with 3 GB of memory, and using GNU G++ 4.3.4 (its template recursion limit set to 1024). Instances with different numbers n of concepts and r of modeling/refinement relationships declared in the whole program have been randomly generated (see the source code for details). Each compile time presented here is expressed in seconds and is the mean of compilations of 10 different instances.

Figure 3 reports the compile time, depending on n, for indexing concepts. As predicted by the theoretical performance analysis, there is a quadratic dependence on n (confirmed by a quadratic regression with a correlation coefficient[6] $R = 0.997$).

[5] Optimized version 2011-08-25 was used for the experiments.

[6] R = Pearson's correlation coefficient; the closer to 1, the more the regression fits the curve.

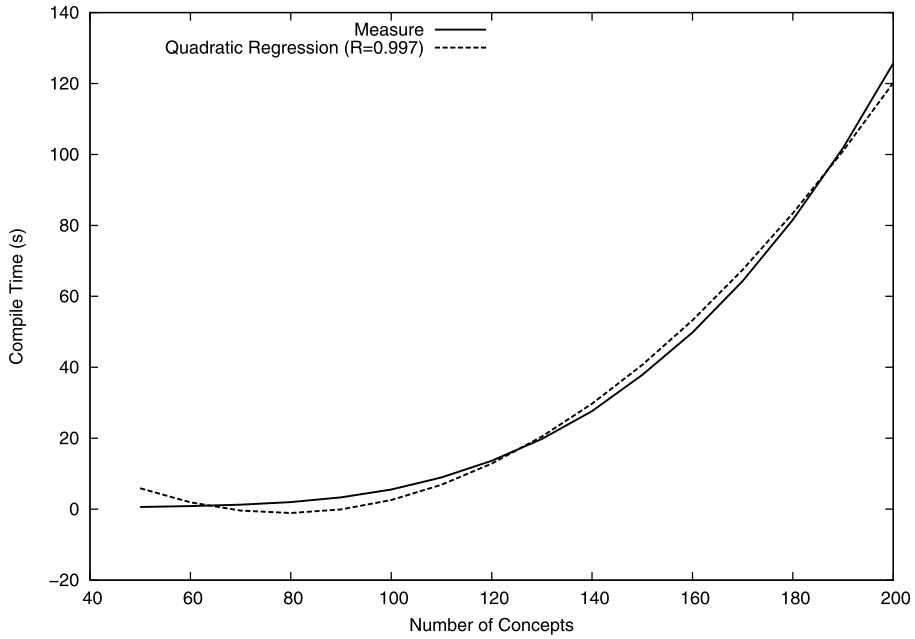

Fig. 3. Compile time for indexing concepts ($r = 100$).

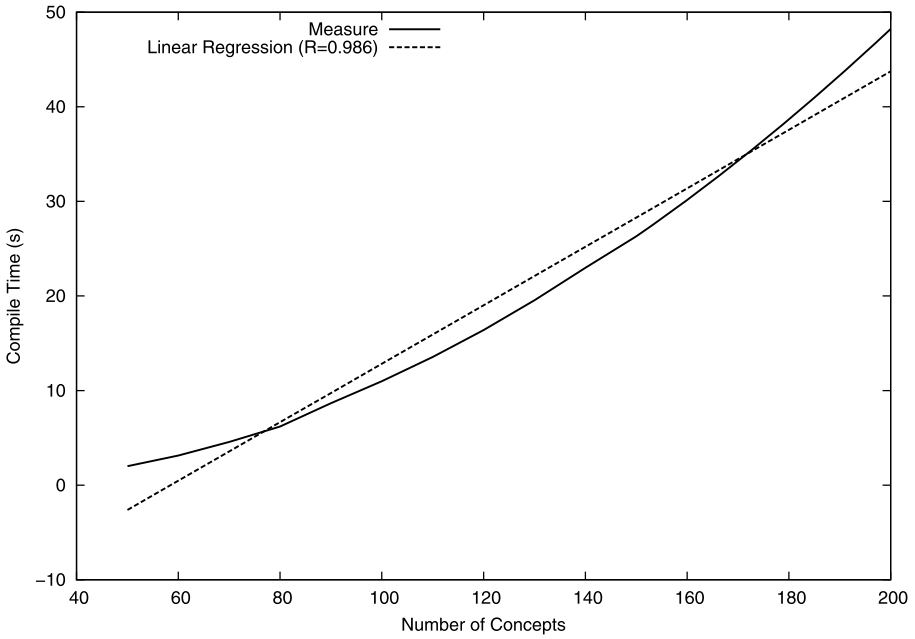

Fig. 4. Compile time for `gnx_direct_concepts` (50 instantiations, $r = 100$).

Figure 4 reports the compile time, depending on n, of 50 instantiations of metafunction `gnx_direct_concepts` (which lists the concepts that a given type or concept directly models or refines respectively). The theoretical performance analysis predicted a linear dependence on n, but the practical results show otherwise, which we think is related to our assumption that accessing a concept through its index (i.e., a call to `gnx_concept`) was constant time. It seems that to find a specialization of a template, the compiler may

require a number of operations dependent on the total number of specializations for this template. However, this non-linear dependence is not so significant, as the linear regression shows a correlation coefficient $R = 0.986$ in the range of our experiments, and the instantiations of `gnx_direct_concepts` represent only one step of the whole compilation process.

Figures 5 and 6 report the compile time, depending respectively on n and r, of 50 instantiations of `gnx_contextual_concept` (which determines

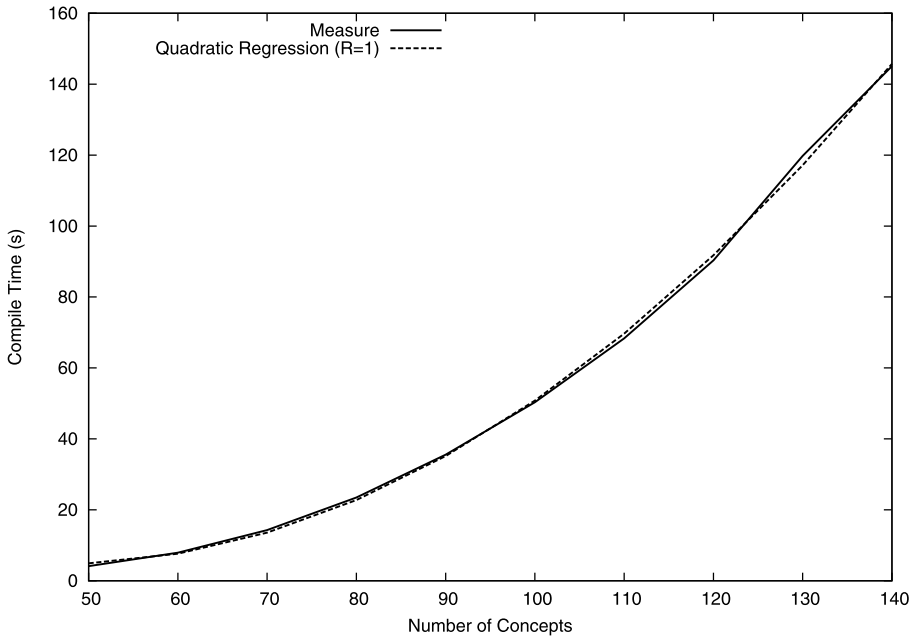

Fig. 5. Compile time for `gnx_contextual_concept` (50 instantiations, $r = 300$).

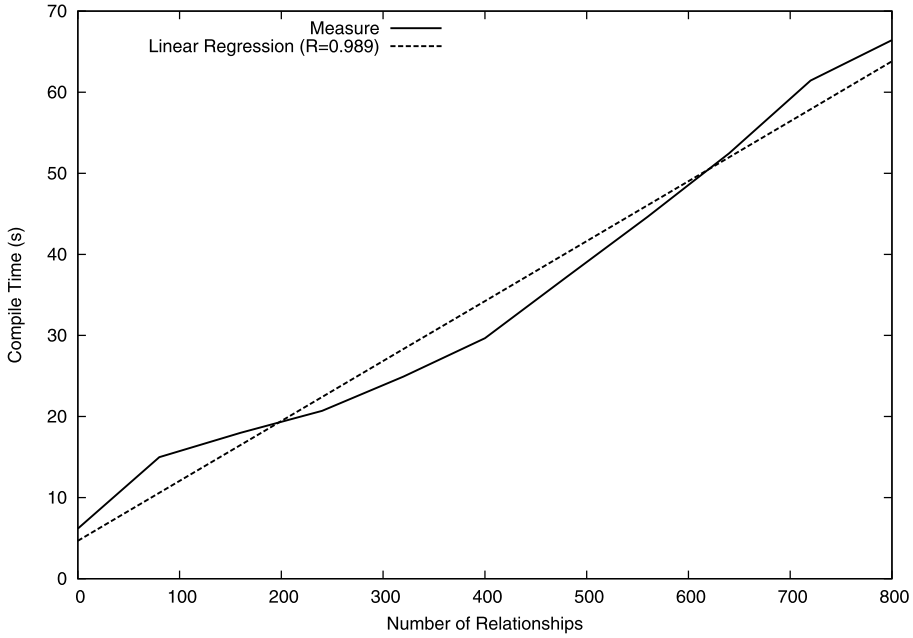

Fig. 6. Compile time for `gnx_contextual_concept` (50 instantiations, $n = 80$).

the best concept for a type bound to a template parameter). The performance of each intermediate metafunction is not shown, as it is similar. As predicted by the theoretical performance analysis, there is a quadratic dependence on n (confirmed with $R = 1$), and a linear dependence on r (confirmed with $R = 0.989$).

Our library has been tested successfully on several compilers: GNU GCC from 3.4.5 to 4.4.3, Microsoft Visual C++ 10, and Embarcadero C++ 6.20. Figures 7 and 8 report the time of the whole compilation pro-

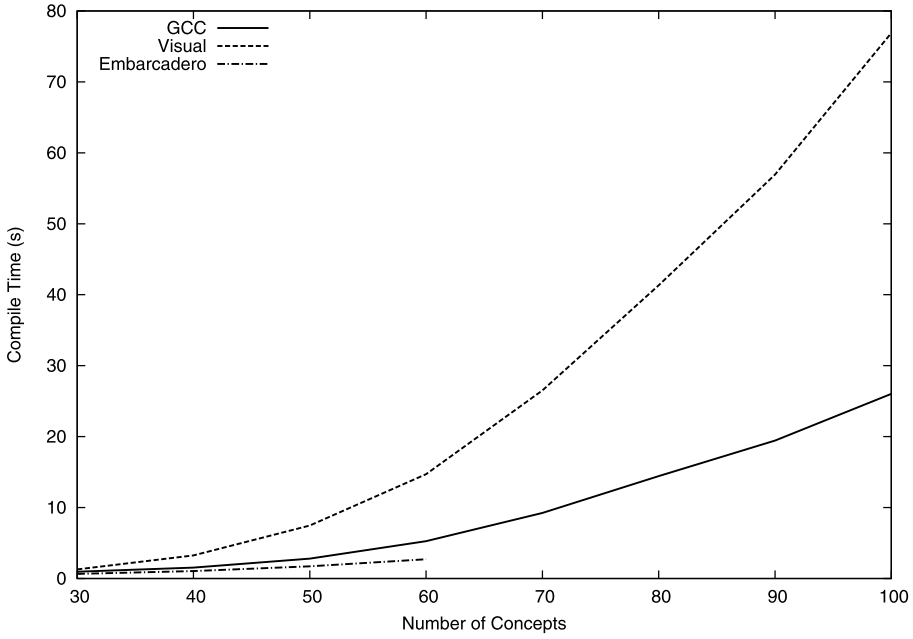

Fig. 7. Whole compile time (with 30 instantiations of `gnx_contextual_concept`, $r = 100$).

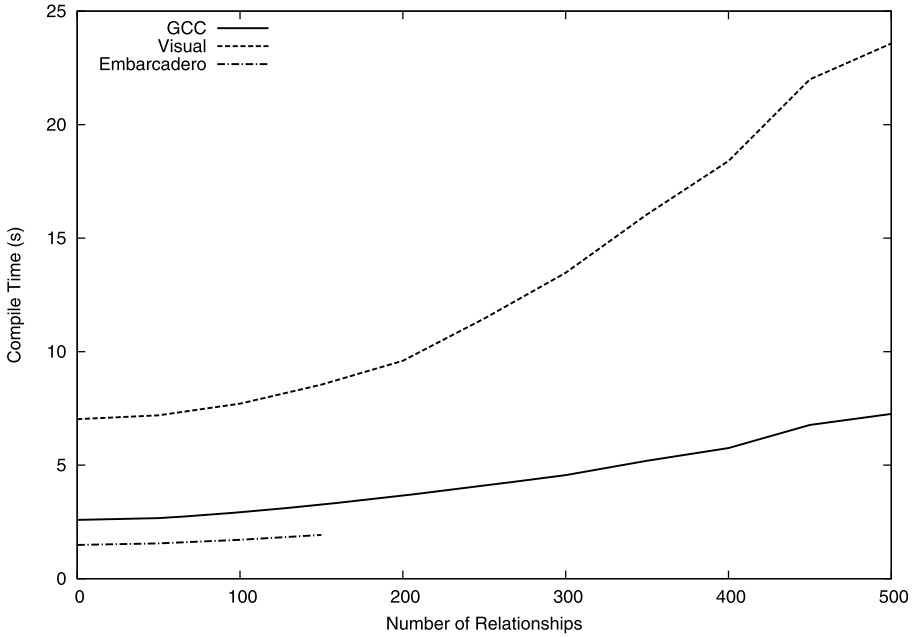

Fig. 8. Whole compile time (with 50 instantiations of `gnx_contextual_concept`, $n = 50$).

cess for those compilers, from indexing the concepts to finding the best concepts for types bound to template parameters, depending on n and r. Notice that we were not able to test all the instances with Embarcadero's compiler, due to a hard limitation of 256 levels in the template recursion.

5. Conclusion

This paper describes template metaprogramming techniques to control the specialization of generic components with concepts. As concepts are not part of the C++ language yet, a library-based solution is provided to declare concepts and modeling/refinement relationships in order to define a taxonomy of concepts. It relies on an automatic indexing of the concepts that allows a retroactive extension: at any time, new concepts and modeling/refinement relationships can be declared.

The library also provides a mechanism to automatically select the most appropriate specialization of a template based on concepts. Specializations of generic components can be defined by constraining template parameters with concepts rather than type patterns. At instantiation time, a metafunction determines the most specialized concept, for a given specialization context, of any type bound to a template parameter, and thus guides the selection of the most appropriate specialization. Our solution is invasive because an extra parameter (invisible to the user) must be added to any template that is intended to be specialized based on concepts; but after the definition of the primary version of the template, specializations based on concepts can be added non intrusively and retroactively.

The retroactive extension enabled by the proposed technique provides high flexibility with minimal assumptions about the components: the coupling between a template and the types bound to its parameters only occurs at instantiation time, while the most appropriate specialization is selected. However, because our goal was to provide a fully portable C++ code with no extra tool, we were not able to automate the identification of the concepts that control the specialization of a given template. Therefore, the notion of a specialization concept is necessary and requires to explicitly declare each concept that is involved in the control of a specialization.

To conclude, a theoretical performance analysis and the performance of practical experiments have been presented to show the compile time overhead of our solution. Even if a quadratic dependence on the number of concepts has been identified, the compile time is reasonable for many applications: compiling 50 specializations with 50 concepts and 250 modeling/refinement relationships on an average computer requires less than 5 seconds.

Acknowledgements

The authors would like to thank the editor and the anonymous reviewers for their constructive comments that helped to improve the manuscript.

References

[1] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, 2004.

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

[3] M.H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, Addison-Wesley, 1999.

[4] B. Bachelet, A. Mahul and L. Yon, Designing generic algorithms for operations research, *Software: Practice and Experience* **36**(1) (2006), 73–93.

[5] C.G. Baker and M.A. Heroux, Tpetra, and the use of generic programming in scientific computing, *Scientific Programming* **20**(2) (2012), 115–128.

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

[7] G. Dos Reis and B. Stroustrup, Specifying C++ concepts, in: *Proceedings of POPL'06*, ACM Press, 2006, pp. 295–308.

[8] R. Garcia, J. Järvi, A. Lumsdaine, J.G. Siek and J. Willcock, A comparative study of language support for generic programming, in: *Proceedings of OOSPLA'03*, 2003, pp. 115–134.

[9] D. Gregor, J. Järvi, J.G. Siek, B. Stroustrup, G. Dos Reis and A. Lumsdaine, Concepts: Linguistic support for generic programming in C++, in: *Proceedings of OOPSLA'06*, 2006, pp. 291–310.

[10] D. Gregor, B. Stroustrup, J.G. Siek and J. Widman, Proposed wording for concepts (Revision 3), Technical Report N2421=07-0281, ISO/IEC JTC 1, 2007.

[11] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine and J.G. Siek, Algorithm specialization in generic programming: Challenges of constrained generics in C++, in: *Proceedings of PLDI'06*, ACM Press, 2006.

[12] J. Järvi, M. Marcus and J.N. Smith, Programming with C++ concepts, *Science of Computer Programming* **75** (2010), 596–614.

[13] J. Järvi, J. Willcock and A. Lumsdaine, Concept-controlled polymorphism, in: *Lecture Notes in Computer Science*, Vol. 2830, Springer-Verlag, 2003, pp. 228–244.

[14] M. Jazayeri, R. Loos, D. Musser and A. Stepanov, Generic programming, in: *Report of the Dagstuhl Seminar on Generic Programming*, 1998.

[15] U. Köthe, STL-style generic programming with images, *C++ Report Magazine* **12**(1) (2000), 24–30.

[16] B. McNamara and Y. Smaragdakis, Static interfaces in C++, in: *First Workshop on C++ Template Programming*, 2000.

[17] J.G. Siek, L.-Q. Lee and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*, Addison-Wesley, 2002.

[18] J.G. Siek and A. Lumsdaine, Concept checking: Binding parametric polymorphism in C++, in: *First Workshop on C++ Template Programming*, 2000.

[19] A. Sutton and J.I. Maletic, Emulating C++0x concepts, *Science of Computer Programming* **78** (2013), 1449–1469.

[20] A. Sutton, B. Stroustrup and G. Dos Reis, Concepts lite: Constraining templates with predicates, Technical Report, N3580, ISO/IEC JTC 1, 2013.

[21] T.L. Veldhuizen, Arrays in Blitz++, in: *Lecture Notes in Computer Science*, Vol. 1505, Springer-Verlag, 1998, pp. 223–230.

[22] L. Voufo, M. Zalewski and A. Lumsdaine, ConceptClang: an implementation of C++ concepts in clang, in: *7th ACM SIGPLAN Workshop on Generic Programming*, 2011.

[23] R. Weiss and V. Simonis, Exploring template template parameters, in: *Lecture Notes in Computer Science*, Vol. 2244, Springer-Verlag, 2001, pp. 500–510.