Generic Programming: Controlling Static Specialization with Concepts in C++

Bruno Bachelet, Antoine Mahul, Loïc Yon

Generic Programming: Controlling Static Specialization with Concepts in C++

Bruno Bachelet [1,4], Antoine Mahul [2,5]
and Loïc Yon [3,4]

Research Report LIMOS/RR-10-18

December 7, 2010

1. bruno.bachelet@isima.fr - http://frog.isima.fr/bruno
2. antoine.mahul@clermont-universite.fr
3. loic.yon@isima.fr - http://www.isima.fr/~loic
4. LIMOS, UMR 6158-CNRS, Université Blaise Pascal, BP 10125, 63173 Aubière, France.
5. CRRI, Clermont Université, BP 80026, 63177 Aubière, France.

Abstract

In generic programming, software components are parameterized on types (and sometimes static values, as in C++) rather than dynamic values. When available, a static specialization mechanism allows building, 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 (or sometimes the static value) of parameters, which may not be accurate enough and may lead to ambiguities or false specializations. This is mainly due to the fact that some relationships between types, which can be considered as similar in some ways, are missing. Thus, it is not always possible to determine an ordering of the specializations of a generic component.

Concepts can be used to introduce relationships between "similar" types: a concept represents a set of requirements for a component that among others refer to its interface and its behavior. This report describes generic programming techniques in C++ for declaring concepts, "modeling" relationships (between a type and a concept) and "refinement" relationships (between two concepts), and for controlling template specializations based on a taxonomy of concepts. This control relies on a metaprogram that finds the most specialized concept for a type involved in the specialization process of a given generic component.

The solution presented here is open for retroactive extension: at any time, a new concept or a new modeling/refinement relationship can be declared, or a new template specialization can be defined; and this new statement will be picked up by the specialization mechanism. The control is also improved by avoiding false specializations and many ambiguities during the specialization process.

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

Résumé

En programmation générique, les composants logiciels sont paramétrés sur des types (et parfois des valeurs statiques, comme en C++) plutôt que sur des valeurs dynamiques. Quand il est disponible, un mécanisme de spécialisation statique permet de construire, pour un jeu de paramètres donné, une version plus adaptée d'un composant générique que sa version initiale. Le mécanisme normal de spécialisation d'un *template* en C++ repose sur le patron de type (ou parfois la valeur statique) de paramètres, ce qui n'est parfois pas assez précis et peut conduire à des ambiguïtés ou à de fausses spécialisations. Ceci est principalement dû au fait que certaines relations entre types, qui peuvent être considérés comme similaires d'une certaine manière, sont manquantes. Ainsi, il n'est pas toujours possible d'établir un ordre des spécialisations d'un composant générique.

Les concepts peuvent être utilisés pour introduire des relations entre types "similaires" : un concept représente un ensemble de spécifications pour un composant qui peuvent, entre autres, faire référence à son interface et à son comportement. Ce rapport décrit des techniques de programmation générique en C++ pour déclarer des concepts, des relations de "modélisation" (entre un type et un concept) et des relations de "raffinement" (entre deux concepts), et pour contrôler les spécialisations de *templates* à partir d'une taxonomie de concepts. Ce contrôle repose sur un métaprogramme qui trouve le concept le plus spécialisé pour un type impliqué dans le processus de spécialisation d'un composant générique donné.

La solution présentée ici est ouverte à une extension rétroactive : à tout moment, un nouveau concept ou une nouvelle relation de modélisation/raffinement peut être déclarée, ou une nouvelle spécialisation de *template* peut être définie ; et cette nouvelle déclaration sera prise en compte par le mécanisme de spécialisation. Le contrôle est également amélioré en évitant les fausses spécialisations et de nombreuses ambiguïtés au cours du processus de spécialisation.

Mots clés : programmation générique, spécialisation de *template*, surcharge/spécialisation basée concept, métaprogrammation.

Abstract

In generic programming, software components are parameterized on types (and sometimes static values, as in C++) rather than dynamic values. When available, a static specialization mechanism allows building, 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 (or sometimes the static value) of parameters, which may not be accurate enough and may lead to ambiguities or false specializations. This is mainly due to the fact that some relationships between types, which can be considered as similar in some ways, are missing. Thus, it is not always possible to determine an ordering of the specializations of a generic component.

Concepts can be used to introduce relationships between "similar" types: a concept represents a set of requirements for a component that among others refer to its interface and its behavior. This report describes generic programming techniques in C++ for declaring concepts, "modeling" relationships (between a type and a concept) and "refinement" relationships (between two concepts), and for controlling template specializations based on a taxonomy of concepts. This control relies on a metaprogram that finds the most specialized concept for a type involved in the specialization process of a given generic component.

The solution presented here is open for retroactive extension: at any time, a new concept or a new modeling/refinement relationship can be declared, or a new template specialization can be defined; and this new statement will be picked up by the specialization mechanism. The control is also improved by avoiding false specializations and many ambiguities during the specialization process.

1 Introduction

Through the notion of generic component (i.e. a component with parameters that are types or static values, and not only dynamic values as in functions or methods), generic programming aims at providing components (mainly algorithms and data structures) as general as possible and broadly adaptable and interoperable, without loosing efficiency (when the language supports generics without run-time overhead) [8].

1.1 Static Specialization

Similar to inheritance in object-oriented programming, which allows the specialization of classes, some languages provide a mechanism to specialize generic components. The decision to select a specialized version of a generic component is made by the compiler based on the type signature (or static value) of parameters during instantiation. Here is a C++ example of a generic class, `ArrayComparator`, that allows to compare two arrays of length `N` and containing elements of type `T`. A specialization for `T` = `char`, presumably more efficient than the general version, is proposed.

```
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));
 }
};

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

With generic programming, the assembling of components goes through the instantiation of the parameter types of a generic component, which arises two concerns: *(i)* to ensure that an instantiated type fulfills the requirements to be used by the generic component (e.g. `T` must provide the `<` and `==` operators in the `ArrayComparator` class); *(ii)* to find the best static specialization of the generic component for an instantiated type (e.g. for `T = char`, specialization `ArrayComparator<char,N>` must be selected).

To gain expressiveness and earliest checking in the process of assembling components, the notion of "concept" has been introduced [3]. When specifying a type as parameter of a generic component, it must follow a set of requirements called a "concept" that can be related to its interface (e.g. existence of a method), its behavior (e.g. complexity of a method) or anything else relevant to its use by the generic component. When a type fulfills the requirements of a concept, it is said that the type "models" the concept. The notion of specialization between concepts is possible, and is referred to as "refinement": a concept "refines" a more generic concept. For instance, the type `int` models the concept `Integral`, and the concept `Integral` refines the concept `Numerical`.

1.3 Challenges

The concerns expressed in Section 1.2 about generic programming can be addressed using concepts.

Concern (i) is referred to as "concept checking" [14], and its goal is to detect, as soon as possible during the compilation process, the types that does not model the concepts required by a generic component. A concept acts like a contract between the users and the author of a generic component: the author specifies requirements on the parameter types with concepts, and the users must provide types that fulfill these requirements, i.e. types that model the specified concepts, to be allowed to instantiate the generic component.

With C++, concepts can not be defined explicitly, and they are usually only indicated in the documentation (e.g. *Standard Template Library* [5]), which leads to late error detections, and thus to cryptic error messages [14]. With Java, constraints can be expressed on the parameter types of a generic component, in the following form: parameter type `T` must be a "subtype" of `U` (`T` is a subtype of `U` whether `T` inherits class `U`, or `T` implements interface `U`, or `T` is type `U`) [4]. Concepts are thus reduced to interfaces in Java.

Concern (ii) is referred to as "concept-based overloading" [9], but in this article, we propose to use the term "concept-based specialization" to emphasize that we consider class as well as function specialization. Its goal is to control the specialization of generic components with concepts instead of type signatures (by signature, we mean a type or a form of type like `T*`, or a "template template" parameter [15]), which can lead to ambiguities (the compiler can not decide between two possible specializations) or false specializations (the compiler selects the wrong specialization), as it will be presented in Section 2.

Several attempts have been made to represent concepts with the C++ language. [12, 14] propose C++ implementations for concept checking, mainly to ensure interface conformance of the instantiated types. On the other hand, [11] proposes a C++ implementation for concept-based specialization. In this article, the specialization is based on both the SFINAE (*substitution failure is not an error*) principle [2] and a mechanism to answer to the question *"does type T model concept C ?"* (through the `enable_if` template), but this solution may lead to ambiguities [6].

6. See the documentation of `enable_if` at: http://www.boost.org/doc/libs/release/libs/utility/enable_if.html

More recently, an attempt has been initiated to define an extension of the C++ language itself to support concepts [6, 13], which may be added to the C++ Standard [7], and is yet available within the experimental compiler ConceptGCC [6, 10]. In the meantime, there seems to be no satisfying solution for concept-based specialization that avoids ambiguities and false specializations.

1.4 Proposal

In this article, we propose a C++ implementation that focuses on the concept-based specialization aspect. In this solution, concepts can be defined, and possibly refined, and types can be declared as modeling one or more concepts. Then these concepts can be used to specify specializations of generic components.

Even if this proposal does not detect directly concept mismatches to provide more understandable errors, it needs to perform concept checking to control the specialization process. The checking is only based on "named conformance" [12] (i.e. check on whether a candidate has been declared as modeling a given concept), and eludes "structural conformance" (i.e. check on whether a candidate has the right interface). However, it could be possible to add structural conformance, as proposed in [12]. One can notice that named conformance can be related to the "concept map" notion of [6], in the sense that the user is free to declare that a type models a concept (without a direct compiler checking), in order to increase the possibilities of using any generic component.

In order to follow the key ideas of generic programming [8] (mainly to express components with minimal assumptions, and to allow that the most specialized, and thus presumably the most efficient, form of a generic component is chosen), our proposal allows (i) any new concept to be modeled by a type (which can not be done with Java for instance: a class must list all its interfaces when defined, and afterward, a new interface, i.e. a new concept, can not be added); (ii) a new specialization of a generic component to be added at any time (if a new concept is defined, a specialization of any generic component can be added for this concept).

Section 2 explains several issues encountered with static specialization, and justifies the use of concepts to bypass them. Section 3 proposes a C++ implementation for concept-based specialization, and presents examples using this proposal. Section 4 shows compilation performances of the solution depending on the number of concepts and relationships among them in a whole code. Full source code of all the examples and our proposal is available for download [7].

2 Issues with Static Specialization

We present several issues that may occur during the process of static specialization: (i) several types may have the same interface, and should have the same specialization; (ii) specialization based on type signature may lead to false specialization; (iii) a type may match several specializations, hence to avoid any ambiguity, a relationship between the specializations is necessary. First, these issues are illustrated with an example that attempts to provide serialization of objects. Then, existing solutions for concept-based specialization are briefly recalled and discussed, using a simpler and academic example.

7. Full source code available at: http://forge.clermont-universite.fr/projects/show/cpp-concepts.

2.1 Specialization Based on Type Signatures

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). A general version of the class, that makes a raw copy of the bytes of an object in memory, can be proposed.

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

This version may not be adapted for complex objects, like containers, where the internal state may have pointers that should not be stored (because they will lead to memory inconsistency after restoring). If we consider "sequence containers" of the STL (*Standard Template Library* [5]) like vectors, lists..., we can provide a specialized version of `Serializer`.

```
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> & cont);
 public: static int inflate(CONTAINER<T,ALLOC> & cont, const char * copy);
};
```

This specialization is based on the type signature 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. Let us consider now "sorted containers" of the STL, like sets and maps. They have a type signature different from the one of sequence containers (e.g. sets have one more parameter to compare elements), whereas they have a common interface for elementary operations, which should ideally allow to define the same specialization of `Serializer` for both sequence and sorted containers. However, as specialization is based here on type signature, another specialization of `Serializer` is necessary.

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

Notice that with some compilers (e.g. Embarcadero – formerly Borland – C++ 6.20 and GNU GCC 3.4.5), this specialization is ambiguous with the one for sequence containers, presumably due to default types for some parameters. Moreover, any type with this signature matches the specialization. For instance, the `std::string` class of the C++ standard library is an alias for a type that matches the signature of sets.

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

Finally, if we consider STL maps, the signature is also different, and a specialization must be provided for `Serializer< CONTAINER<K,T,COMP,ALLOC> >`, whereas maps and sets have similar interfaces (the main difference is that sets store single elements and maps pairs of elements), which should ideally allow to define the same specialization.

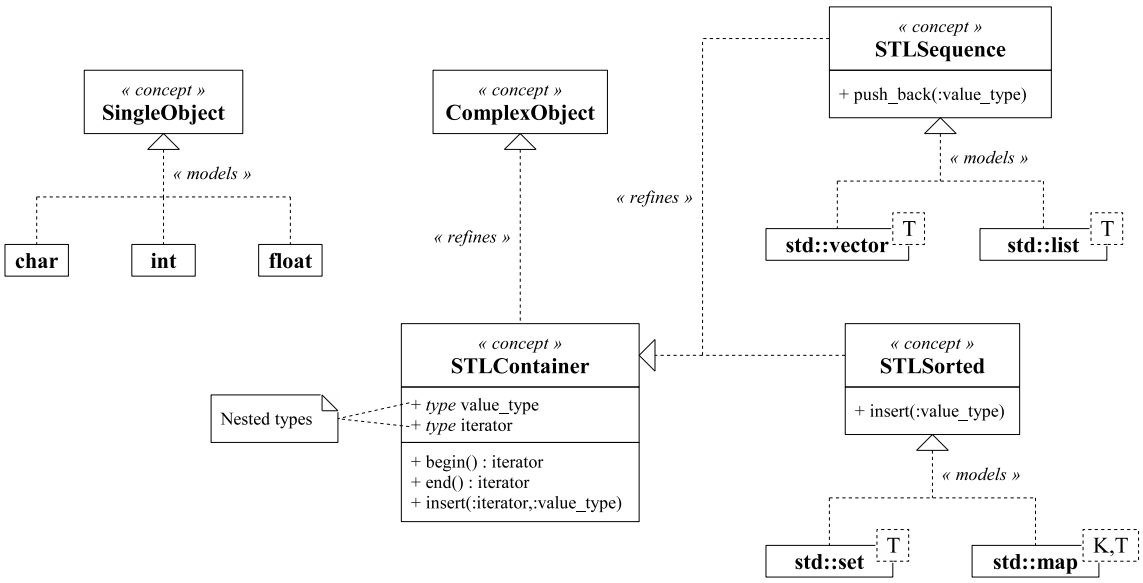

Figure 1: Example of concepts for serialization.

At the beginning of the section we expressed three issues that could be addressed with concepts: (i) several types with the same interface can model a common concept, and a specialization for this concept should be provided; (ii) concepts avoid false specialization: if a type models a concept, that means it has the associated requirements; (iii) if a type matches several specializations, that means it models the concepts for these specializations, either there is no relationship between the concepts and any specialization can be chosen, or one concept is more specialized (through refinement) than the others, and its specialization must be selected.

Figure 1 proposes concepts and their relationships for the example discussed in this section. The `SingleObject` and `STLContainer` concepts will allow to provide the two specializations for `Serializer` discussed previously: one based on raw copy, and another one based on the common interface of all STL containers. More specialized versions of `Serializer` will be possible for the `STLSequence` and `STLSorted` concepts, using more specific interfaces.

2.2 Existing Solutions Based on Concepts

Several solutions have already been proposed to control static specialization with concepts. To briefly illustrate their mechanism, we chose an example based on three concepts: `Numerical`, `Integral` and `Floating`. Each one respectively represents requirements for numerical values, integral values, and floating-point values. Hence, `int` models both `Numerical` and `Integral`, and `double` models both `Numerical` and `Floating`. Moreover, `Integral` and `Floating` refine `Numerical`, and thus should be chosen above `Numerical` during specialization.

2.2.1 Concept-Based Dispatch

[12] implements concepts with "static interfaces" in C++, and proposes a dispatch mechanism to control static specialization with concepts. The solution is based on the `StaticIsA` template that provides concept checking: `StaticIsA<T,C>::valid` is true if `T` models concept `C`. Here is an overview of how concepts can be used to dispatch specializations.

```
enum { IS_NUMERICAL, IS_INTEGRAL, IS_FLOATING, UNSPECIFIED };

template <class T> struct Dispatcher {
 static const int which = StaticIsA<T,Floating>::valid ? IS_FLOATING
                        : StaticIsA<T,Integral>::valid ? IS_INTEGRAL
                        : StaticIsA<T,Numerical>::valid ? IS_NUMERICAL
                        : UNSPECIFIED;
};

template <class T> struct ErrorSpecializationNotFound;

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

template <class T> struct Example<T,IS_NUMERICAL> { ... };
template <class T> struct Example<T,IS_FLOATING> { ... };
template <class T> struct Example<T,IS_INTEGRAL> { ... };
```

A dispatcher must be defined for each generic component that is specialized, which can rapidly be tedious. A more automatic solution should be provided, especially for the selection of the best concept in the dispatch. Notice that the general version of the generic inherits a class that is not defined, the aim being to detect, as soon as possible, that a type can not be used for instantiation: usually, the error appears using the instantiated component, and the messages of the compiler could be cryptic for the user [14], whereas with this solution, the error appears at instantiation and the messages should be clearer for the user.

2.2.2 Concept-Based Overloading

[11] provides the `enable_if` template to control static specialization with concepts. Its definition, provided in the Boost Library [1], is as follows.

```
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> {};
```

The idea is that if `B` is true, there is a nested type inside `enable_if_c` and thus inside `enable_if`. Combined with the SFINAE principle [2], `enable_if` allows some control on static specialization. Here is an example that tries to perform the same specialization presented previously with the dispatch technique.

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

template <class T>
struct Example<T, typename enable_if< is_numerical<T> >::type> { ... };

template <class T>
struct Example<T, typename enable_if< is_integral<T> >::type> { ... };

template <class T>
struct Example<T, typename enable_if< is_floating<T> >::type> { ... };
```

In this example, `is_C<T>` is a generic class that is specialized so `is_C<T>::value` is true if `T` models concept `C`. The SFINAE principle is: if an invalid parameter is formed during an

instantiation, this instantiation is ignored. For instance, the instantiation `Example<int>` induces an attempt to instantiate `Example<int,typename enable_if< is_floating<int> >::type>`[8], and because `enable_if< is_floating<int> >` has no member `type`, this specialization is not considered. If there is only one valid instantiation, it is selected for specialization. However, there remains two possible instantiations for `Example<int>`, which leads to an ambiguity in the specialization process: `int` models both `Integral` and `Numerical`. This ambiguity should ideally be avoided: `Integral` is more specific than `Numerical`, so the specialization for `Integral` should be chosen. A refinement relationship between `Integral` and `Numerical` should be expressed and used to avoid this ambiguity.

3 C++ Proposal for Concept-Based Specialization

It appears that concepts are better suited to control static specialization than type signatures, but to our knowledge, there is no solution that addresses all the issues brought up in the previous section. We propose now a C++ implementation that attempts to tackle these issues, while waiting for changes in the C++ language specification. As concepts are not part of the C++ language yet, a solution is first introduced to declare concepts and relationships (modeling and refinement) between concepts. Then, a mechanism based on concepts is presented to control static specialization.

3.1 Metafunctions

To implement concept-based specialization, some fundamental "metafunctions" are necessary. These generic classes are usual in metaprogramming libraries (e.g. in the Boost Library). A metafunction acts similarly to an ordinary function, but instead of manipulating dynamic values, it deals with "metadata", i.e. entities that can be handled at compile time in C++: mainly types and static integer values [1]. In order to manipulate indifferently types and static values in metafunctions, metadata are embedded inside classes, as follows[9].

```
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;
```

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

The parameters of a metafunction (which are the parameters of the generic class representing the metafunction) are assumed to represent metadata (i.e. they are classes with an embedded member `type` or `value`). The "return" of a metafunction is implemented with inheritance: the metafunction inherits a class representing a metadata. This way the metafunction itself can become a parameter of another metafunction. Here are metafunctions necessary in this section.

8. `typename` is necessary in C++ to confirm that the member `type` is actually a type and not a value.
9. We chose to add the prefix "`gnx_`" to all the metafunctions and macrocommands of our solution.

```
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 static specialization to fully implement their behavior. Metafunction `gnx_same` returns whether two types are identical: `gnx_same<T1,T2>` inherits `gnx_true` if `T1` and `T2` are identical, or `gnx_false` otherwise. Thus, `gnx_same<T1,T2>::value` contains the return of the function. Metafunction `gnx_if` acts similarly to the common `if` instruction: `gnx_if<T,A,B>` inherits `A` if `T::value` is true, or `B` otherwise. If `A` and `B` represent metadata, then `gnx_if<T,A,B>` inherits the member nested in `A` or `B`.

3.2 Declaring Concepts

Concepts need to be identified so they can be used for static specialization. We propose to represent them as types. For instance, concept `STLContainer` of the example of Section 2 is simply declared as follows: `struct STLContainer {};`

3.2.1 Typelists

To be manipulated during specialization, concepts need to be stored in a list. [2] provides the "typelist" technique that allows to store types into a linked list.

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

struct gnx_nil {};
```

Type `gnx_nil` models "no type" (we chose to avoid using `void`, 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 `int` and `double` in a list can be declared as follows.

```
typedef gnx_list< int,gnx_list<double,gnx_nil> > mylist1;
```

Operations can be defined on typelists [2], like adding a type into a list.

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

However, this solution is too static: a typelist can not be modified, instead a new list is created. We need a solution where the list can be modified at compile time to add a new concept, without changing the identifier of the list. Typelists will nevertheless be useful during the specialization process, and operations to merge and to search lists will be necessary.

3.2.2 Numbering Concepts

To provide a list where concepts can be added at any time, we propose to design an indexed list with a metafunction: adding a concept into the list is performed by specializing 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> {};
```

To number the concepts by hand is not acceptable, so a solution to get the number of concepts already indexed is necessary.

```
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>
        > {};

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 is not working as is, because each time `gnx_nb_concept<>` is used, all the instantiations of `gnx_concept`, from `gnx_concept<0>` to `gnx_concept<N+1>` where `N` is the number of concepts, are performed. Hence the specializations for `STLContainer` and `STLSequence` can not be done, because instantiation `gnx_concept<N+1>` has already been performed (with the general version of `gnx_concept`). To cope this phenomena, the counting is parameterized with an "observer": each time the concepts need to be counted, a new observer must be provided. Thus, to get the next available number for a new concept, this concept is used as observer for counting.

```
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>
        > {};

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> {};
```

The `gnx_nb_concept` metafunction requires $O(n)$ operations, where n is the number of concepts already in the code. Hence, numbering n concepts requires $O(\sum_{i=1}^{n} i) = O(n^2)$ operations.

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

The macrocommand `gnx_declare_concept` is defined to declare concepts with a single instruction. For the example of Section 2 (cf. Figure 1), concepts are declared as follows.

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

3.3 Modeling Concepts and Refinement

The "models" relationship between types and concepts is represented with a metafunction that answers whether a type `T` models a concept `C`.

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

By default, the metafunction returns false, and specializations allow to declare new relationships by returning true. The refinement relationship between concepts will also be defined through this metafunction. For the example of Section 2 (cf. Figure 1), the following relationships are declared.

```
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>,STLSorted> : gnx_true {};

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

Notice that `gnx_models_concept` provides an answer for direct relationship only. If a type `T` models a concept `C1` that refines a concept `C2`, this metafunction will return false for a relationship between `T` and `C2`. Additional metafunctions, necessary to find any relationship between a type and a concept, are briefly introduced below (their complete code is available for download [7]).

– `gnx_direct_concepts<T>` provides a list (using the typelist technique [2]) of all the concepts directly related to type (or concept) `T`. It goes through all the concepts using their number, and checks whether `T` models each concept using `gnx_models_concept`. Assuming that retrieving a concept from its number (i.e. a call to `gnx_concept`) is a constant time operation, metafunction `gnx_direct_concepts` requires $O(n)$ operations, where n is the number of concepts in the whole code.

– `gnx_matching_concepts<T>` provides a list of all the concepts that a type (or a concept) `T` models, even the concepts associated by refinement. It uses `gnx_direct_concepts` to list the concepts directly related to `T`, 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 relationships between concepts in the whole code: at worst, all the n concepts are asked for their direct concepts (i.e. a call to `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).
– `gnx_matches_concept<T,C>` returns whether a type (or a concept) `T` models a concept `C`, directly or indirectly. This metafunction searches for `C` in the list of concepts provided by `gnx_matching_concepts` and requires $O(n^2 + rn)$ operations: $O(n^2 + rn)$ operations to build the list, and $O(n)$ for the search.

From now on, we will make a difference between the terms "models" and "matches": "`T` models `C`" means `T` is directly related to `C`, whereas "`T` matches `C`" means `T` is directly or indirectly related to `C`.

3.4 Specialization Based on Concepts

3.4.1 Declaring Specializations

Our goal is to provide specializations for a generic component based on concepts. For the example of Section 2 (cf. Figure 1), we would like to provide specializations of `Serializer` for `SingleObject` and `STLContainer` concepts. We also propose to define a specialization for `STLSorted` that should be selected when a type matches both `STLContainer` and `STLSorted`, the latter being more specific. Here is the code of the `Serializer` generic class with our proposal.

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

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

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

template <class T> class Serializer<T,STLSorted> { ... };
```

An additional parameter is provided to the generic component that is the best concept of type `T` for the specialization of `Serializer`. This best concept is automatically provided by `gnx_best_concept`, which is a macrocommand to lighten the syntax for calling metafunction `gnx_contextual_concept`.

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

The metafunction requires a "context", which is a type that represents the context of a specialization: each generic component that uses static specialization based on concepts requires a different context. There are two reasons for defining this notion of context: (i) as seen previously, metafunction `gnx_nb_concept`, called by many other metafunctions, requires an observer to perform correctly and to allow defining new concepts at any time, and this observer will be the context of the specialization; (ii) our proposal does not modify the C++ language itself, and we need a mechanism to know which concepts have to be considered for a specific specialization.

Back to our example, we need to explicitly indicate which concepts must be considered to specialize `Serializer`. For that purpose, context `SerializerContext` is defined and the concepts it needs are associated by the metafunction `gnx_uses_concept`, whose use is similar to `gnx_models_concept`: it has to be specialized to indicate that a context `X` provides a specialization for a concept `C`.

```
struct SerializerContext {};

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

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

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

3.4.2 Selecting the Best Specialization

Based on the list of concepts to be considered in a specialization context, and the relationships between concepts and types, metafunction `gnx_contextual_concept` provides the best concept for specialization of a given type.

If we consider the fictitious example of relationships between types and concepts of Figure 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 class `A`: concept `C2`, candidates are `C1` and `C2`, but `C2` is more specific.
- For class `B`: no concept, there is no candidate in the context's list, `gnx_nil` is returned.
- For class `C`: concept `C5`, it is the only choice.
- For class `D`: concepts `C1` or `C5`, both concepts are valid (`D` matches both), and there is no relationship between them to make a deterministic choice. The one selected for specialization depends on the implementation (with ours, the concept with the highest number is selected). To force the selection, one can specialize `gnx_contextual_concept`.

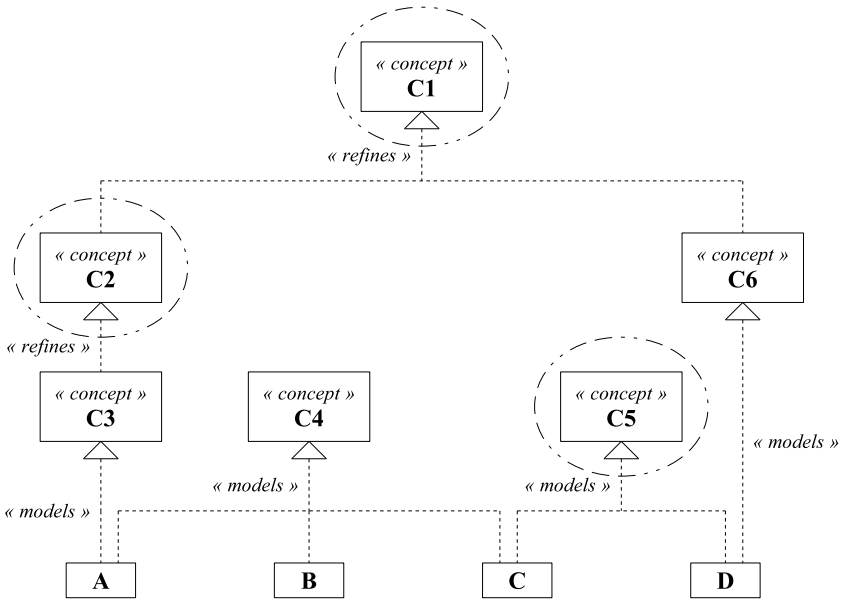

Figure 2: Example of best concept selection.

Metafunction `gnx_contextual_concept<X,T>` goes through the list of all the concepts matched by type `T` (provided by `gnx_matching_concepts<T>`), and selects the one that does not match any other concept in the list (using `gnx_matches_concept`) and is associated with 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_matching_concepts` has already performed all the necessary `gnx_matches_concept` instantiations).

3.5 Full Example

Two examples have been presented all along this section: one concerning serialization of objects, and the other, more fictitious and with more complex relationships between concepts. To conclude, we propose to illustrate our solution on the example about numbers introduced in Section 2.2.

```
gnx_declare_concept(Numerical);
gnx_declare_concept(Integral);
gnx_declare_concept(Floating};

template <> struct gnx_models_concept<Integral,Numerical> : gnx_true {};
template <> struct gnx_models_concept<Floating,Numerical> : gnx_true {};

template <> struct gnx_models_concept<int,Integral>       : gnx_true {};
template <> struct gnx_models_concept<float,Floating>     : gnx_true {};
template <> struct gnx_models_concept<MyNumber,Numerical> : gnx_true {};

struct ExampleContext;

template <class T,class = gnx_best_concept(ExampleContext,T)>
struct Example : ErrorSpecializationNotFound<T> {};

template <> struct gnx_uses_concept<ExampleContext,Numerical> : gnx_true{};

template <class T> struct Example<T,Numerical> { ... };

template <> struct gnx_uses_concept<ExampleContext,Integral>  : gnx_true{};

template <class T> struct Example<T,Integral> { ... };

template <> struct gnx_uses_concept<ExampleContext,Floating>  : gnx_true{};

template <class T> struct Example<T,Floating> { ... };
```

3.6 Conclusion

Several steps are necessary for concept-based specialization with our proposal: (i) declaring concepts and defining modeling and refinement relationships; (ii) declaring a specialization context by enumerating the concepts that will be used. These steps are not monolithic, and new concepts and specializations can be defined at any time (before the first instantiation of the affected 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, can be automated using for instance the `StaticIsA` template proposed in [12].

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

However, a few issues arise from our solution. First, it modifies the signature of the generic components: for each classic parameter, an additional parameter providing its best concept may be necessary. For instance, the `Serializer` generic class is seen by users with a single parameter, whereas it actually has two. Secondly, the instantiation problem of `gnx_nb_concept` (cf. Section 3.2.2) has been almost bypassed with observers, but there are very specific situations where the issue remains. For instance, the following specialization may be troublesome.

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

It implies a 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. One should avoid to instantiate `gnx_contextual_concept` before final use of the affected generic component. In this example, the solution is to avoid full instantiation:

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

4 Compilation Performances

Theoretical performances have been established for the metafunctions of our solution. We assumed some operations of the compiler to be constant time, so it seems important to compare these performances with practical ones. The metafunctions, as presented in this paper, are not optimized for compilation, and some template structures may lead to unacceptable compilation times. Thus, we propose an optimized version of our implementation [7], whose code is much less readable, but gives better compilation times.

The tests presented here have been performed 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). Concepts and relationships between them have been randomly generated to provide different instances to compile. Each compilation time presented here is expressed in seconds and is the mean of 10 compilations of different instances.

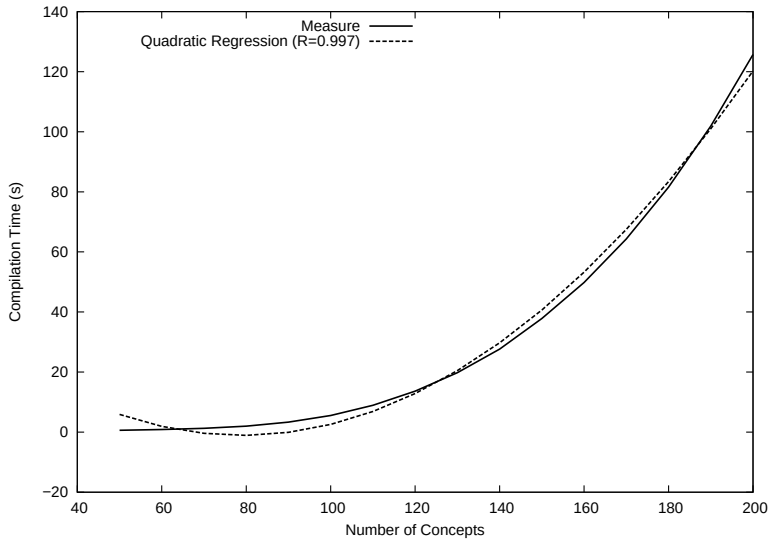

Figure 3: Time for numbering concepts, depending on n ($r = 100$).

Figure 3 shows the compilation time for numbering the concepts, depending on the number n of concepts in the whole code. As predicted, there is a quadratic dependence on n (confirmed by a quadratic regression with a correlation coefficient [10] $R = 0.997$).

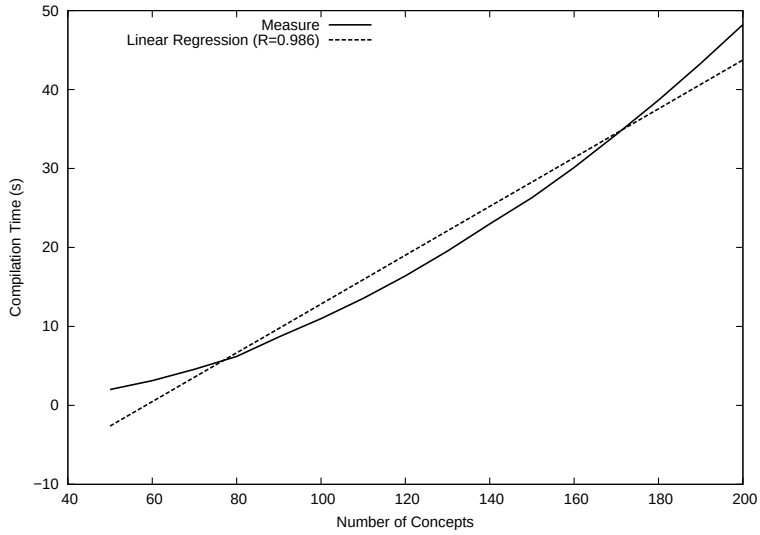

Figure 4: Time for 50 instantiations of `gnx_direct_concepts`, depending on n ($r = 100$).

Figure 4 shows the compilation time, depending on n, for 50 instantiations of metafunction `gnx_direct_concepts` (which finds the direct concepts of a given type or concept). It has been predicted to be linearly dependent on n, but practical performances seem to show otherwise, which we think is related to our assumption that retrieving a concept from its number (i.e. a call to `gnx_concept`) was a constant time operation. It seems that to access a specialization of a generic, the compiler may require a number of operations dependent on the total number of specializations for the generic. 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 experiment, and the instantiations of `gnx_direct_concepts` are only a single step in the whole compilation process.

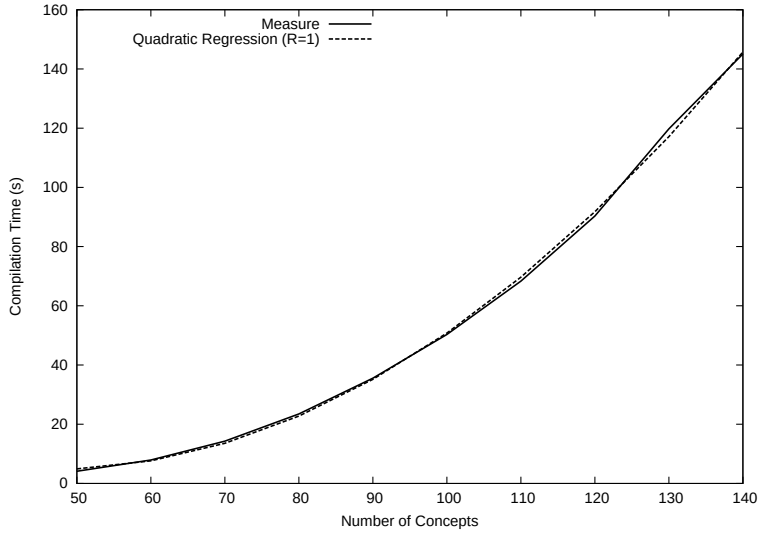

Figure 5: Time for 50 instantiations of `gnx_contextual_concept`, depending on n ($r = 300$).

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

Figures 5 and 6 show the compilation time, depending respectively on n and r, for 50 instantiations of `gnx_contextual_concept` (which gets the best concept for specialization of a given type). The performances of the intermediate metafunctions are not shown, as they are similar. As predicted, there is a quadratic dependence on n (confirmed with $R = 1$), and a linear dependence on r (confirmed with $R = 0.989$).

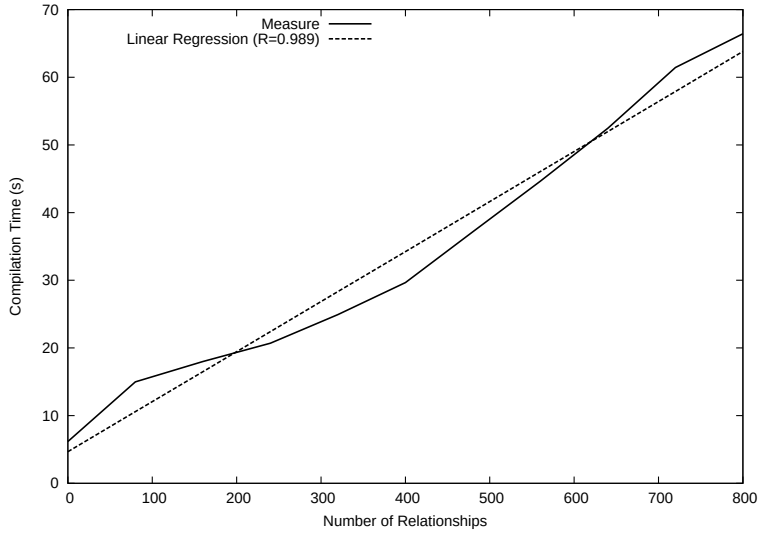

Figure 6: Time for 50 instantiations of `gnx_contextual_concept`, depending on r ($n = 80$).

Our proposal for concept-based specialization has been tested successfully on several recent compilers: GNU GCC from 3.4.5 to 4.4.3, Embarcadero C++ 6.20, Microsoft Visual C++ 10. Figures 7 and 8 show the time of the whole compilation process for those compilers, from numbering the concepts to finding the best concept for specialization, depending on n and r. We were not able to test all the instances with Embarcadero's compiler, due to a hard limitation of 256 levels of template recursion.

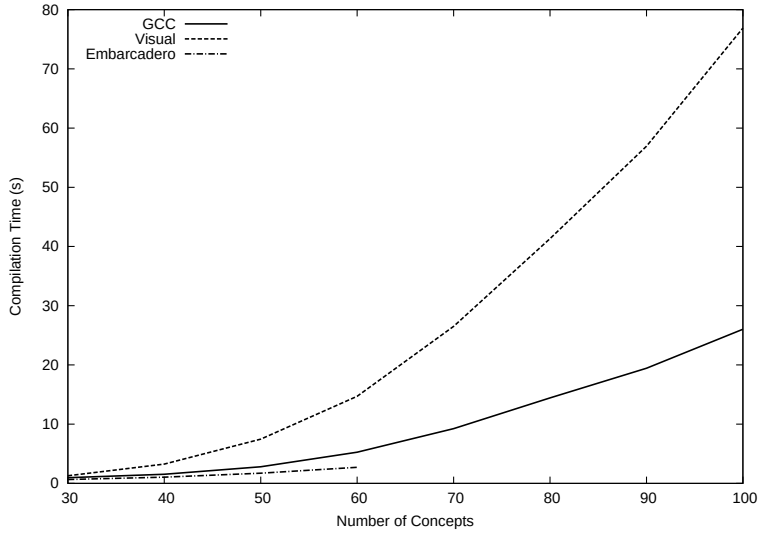

Figure 7: Whole compilation time, 30 instantiations of `gnx_contextual_concept`, depending on n ($r = 100$).

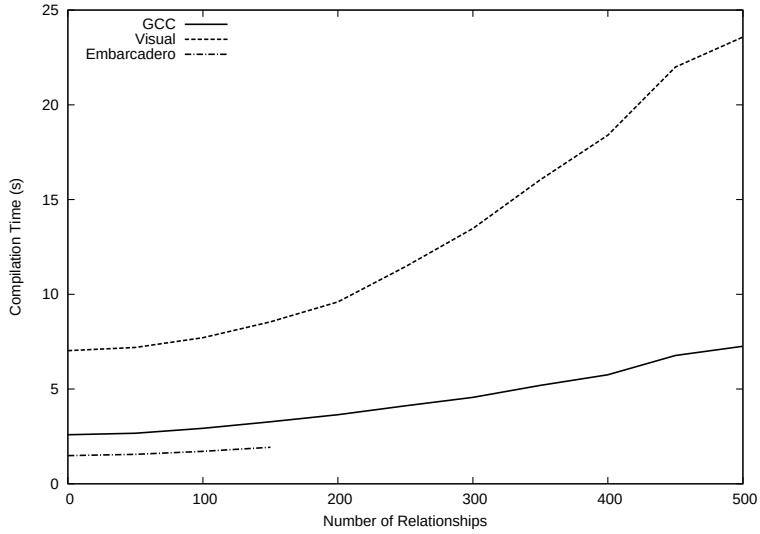

Figure 8: Whole compilation time, 50 instantiations of `gnx_contextual_concept`, depending on r ($n = 50$).

5 Conclusion

An implementation for the C++ language has been proposed to control the specialization of generic components with concepts. As concepts are not part of the C++ language yet, a solution has first been proposed to declare concepts and relationships (modeling and refinement) between concepts. It is based on an automatic numbering of the concepts, which allows a flexible way to declare new concepts and relationships: at any time, new concepts can be declared, and types can be specified to model a concept. This ensures that if a new concept is needed to control the specialization of a generic component, any already defined type can be related to this new concept.

Then, our solution proposes a mechanism based on concepts to control static specialization. Specializations of a generic component are defined for specific concepts, and during instantiation, the best concept for a type in the context of a given specialization is automatically found. Our proposal is a bit invasive because a default parameter must be added to the generic to specialize. However, after the first definition of a generic, specializations can be added non intrusively: at any time, a specialization can be defined for any concept. This ensures that if a specialization is necessary for new types, it can be defined.

To perform correctly, our solution needs to identify the concepts used to define the specializations of a given generic component. As we limited our approach to existing C++ functionalities, we were not able to automate this aspect, and the notion of "context" has to be added, and requires to explicitly identify the concepts involved in the specialization of a generic component.

To conclude, theoretical and practical performances have been presented to show the compilation time overhead our solution may induce. Even if a quadratic dependence on the number of concepts has been identified, the compilation time seems reasonable for many applications: compiling 50 specializations with 50 concepts and 250 relationships on an average computer requires less than 5 seconds.

References

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

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

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

[4] Gilad Bracha. Generics in the Java Programming Language. Technical report, Sun Microsystems, 2004.

[5] Silicon Graphics. Standard Template Library Programmer's Guide. Web site. http://www.sgi.com/tech/stl/.

[6] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic Support for Generic Programming in C++. In *Proceedings of OOPSLA'06*, pages 291–310, 2006.

[7] Douglas Gregor, Bjarne Stroustrup, Jeremy Siek, and James Widman. Proposed Wording for Concepts (Revision 3). Technical report, N2421=07-0281, ISO/IEC JTC 1, 2007.

[8] Mehdi Jazayeri, Rüdiger Loos, David Musser, and Alexander Stepanov. Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, 1998.

[9] Jaakko Järvi, Douglas Gregor, Jeremiah Willcock, Andrew Lumsdaine, and Jeremy G. Siek. Algorithm Specialization in Generic Programming: Challenges of Constrained Generics in C++. In *Proceedings of PLDI'06*. ACM Press, 2006.

[10] Jaakko Järvi, Mat Marcus, and Jacob N. Smith. Programming with C++ Concepts. In *Science of Computer Programming*, volume 75, pages 596–614. Elsevier, 2010.

[11] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Concept-Controlled Polymorphism. In *Lecture Notes in Computer Science*, volume 2830, pages 228–244. Springer-Verlag, 2003.

[12] Brian McNamara and Yannis Smaragdakis. Static Interfaces in C++. In *First Workshop on C++ Template Programming*, 2000.

[13] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ Concepts. In *Proceedings of POPL'06*, pages 295–308. ACM Press, 2006.

[14] Jeremy G. Siek and Andrew Lumsdaine. Concept Checking: Binding Parametric Polymorphism in C++. In *First Workshop on C++ Template Programming*, 2000.

[15] Roland Weiss and Volker Simonis. Exploring Template Template Parameters. In *Lecture Notes in Computer Science*, volume 2244, pages 500–510. Springer-Verlag, 2001.