

A short introduction to the Columbus Proposal for a standard C/C++ Schema

Rudolf Ferenc

ferenc@cc.u-szeged.hu

RGAI – University of Szeged

H-6720 Szeged, Aradi Vértanúk tere 1, Hungary

This paper is a short description of the Columbus Proposal for a standard C/C++ schema.

1. Introduction

Parsing C++ source code and extracting information from it is inherently a very difficult task (and for most researchers it is out of their scope of work). Finding a standard way of interchanging this information among tools – e.g. a C++ parser front-end and a metrics tool – is of crucial importance.

As a basically already accepted medium for information interchange in re-engineering, GXL was chosen as the underlying representation. One of the most powerful features of GXL is the handling of schemas that can be represented as UML class diagrams. Therefore the standard C/C++ schema proposal is also presented as a UML class diagram. The model is supplemented with key examples (C/C++ snippets).

2. The model

This version of the model covers all C/C++ constructs *except the function bodies* (statements, expressions). The only deficiency is regarding the template specializations: the specialization arguments are not represented yet.

Please note, that our model covers the “clean” C/C++ language syntax (preprocessed source code), it does not deal with macros and other preprocessor issues.

The model itself consists of two parts: the first one is language independent and the second one is the C/C++ specific.

2.1 The language independent base-model

The first part of the model is language independent. This means that it can be used for modeling other programming languages as well. This could be useful because if all these other models would use this “base-model”, then all models would be in some manner similar to each other.

The base-model (the upper part of the class diagram above the dashed line in Figure 1) consists of only seven abstract classes: At the top of the hierarchy is the class called “*object*.” It simply stores an *identifier* that is inherited by all other classes. A singly rooted hierarchy has many advantages (e.g. all classes have an interface in common).

2.1.1 Members

A class called “*node*” is derived from the base by extending it with a *name* field. All classes in the C/C++ model that are “leaves” (that do not contain further objects, e.g. scope members) are derived from this class.

The “*composite*” class is derived from the node, which follows the standard composite design pattern. This means that it is a container that can store (or refer to) other objects (the aggregation between the *object* and the *composite*).

2.1.2 Type representation

The other child class of the object is called “*typeform*.” This is an abstract class for forming different kinds of type formers, like pointer-to or array-of, etc. These typeforms are gathered together by the “*typerep*” class that contains two composites: one for the type prefix and one for the type suffix, respectively.

2.1.3 Relations

It is common in programming languages that some objects are referring to other objects. The easiest way to model such relationships is using relations. The “*relation*” class is a composite that stores only edges of one certain type (e.g. friend edges or inheritance edges in the case of C++). This way multiple relations can be put over the objects of the model in a sense of layers (the edges are not hard-coded into the objects). The “*edge*” class refers to two objects that are somehow connected with each other (from, to).

2.2 Extending the base-model to C/C++

This part of the model (the lower part of the class diagram below the dashed line) is modeling the actual

C/C++ constructs. As in the case of the base-model an abstract “*Base*” class is introduced for building a singly rooted hierarchy of classes. It is important to emphasize that every class in the model is inherited from one (or both) of these abstract base classes and that the multiple inheritance is acyclic.

2.2.1 Members

The most important child class of the base is called “*Member*.” This abstract class is the parent of all kinds of members a composite can store (we use the term members in a more general way than the usual). It has properties like visibility (private, protected, public) and location information (path, line, end line).

The first child of the member class is the abstract “*Scope*” class, which is a member as well, because it can be contained by another scope. It is also the child of the *composite* class from the base-model that enables the class to store other objects. It represents all kinds of scopes, like block scopes, classes, etc.

The “*BlockScope*” will contain the statements and expressions that will be presented in the next version of the model and this paper.

The “*Namespace*” class is the first class presented that can be instantiated to an object. Actually there must always exist at least one namespace object that is called the *global namespace*. Our model is basically designed in a project-oriented view, which means that namespace scopes have priority over file scopes (both cannot be represented at the same time, because namespaces can be defined across files/compilation units). However, the original path information is stored in the member objects and the files can be restored from there.

The “*Class*” class can be considered as an extension of the namespace, it has only some additional fields like class kind (class, struct or union) and if it is abstract and defined or not.

All the following classes in this chapter (except the Enum class) are derived also from the *node* class from the base-model (beside the *Member* class) that provides the class an identifier and a name.

The classes “*Variable*,” “*Function*,” “*Parameter*” and “*Typedef*” are very similar, so they will be described together. Basically, these are the members that have a type. Our model represents this by composing them with the “*TypeRep*” class (see section 2.2.2 for more on this). All these classes have some custom attributes that are needed for storing special information like function kind (constructor, destructor, etc.) or the initial value of the variable. The model makes no difference in representing class attributes and local variables. Typedefs are special because they are

not “real members” in the usual sense (just type aliases), but syntactically they look almost exactly the same as variables.

The “*Using*” and “*NamespaceAlias*” classes are special as well, because they have a position in the code, but are not “real members.” The *Using* class refers a namespace whose symbols can be used from the point of definition, a single namespace member or base class members that can be used with e.g. a modified visibility. The *NamespaceAlias* class refers to a namespace and defines a new name/alias for it.

Similarly to scopes, the “*Enum*” class is a composite that stores an arbitrary number of “*Enumerator*”-s.

2.2.2 Type representation

C/C++ types consist basically of two parts: the type prefix and the type suffix. These are already represented in the *typerep* class of the base-model, the C/C++ part subclasses it only with the class “*TypeRep*.” The actual composite, which represents the type prefix and -suffix, is called “*TypeForms*.”

Both the type prefix and -suffix consist of small parts that we call “*TypeForm*”-s. These can be simple type formers like pointers, arrays and parentheses or composites like declaration specifier lists and function parameter lists.

The simple type formers include the classes “*TypeFormPtr*” for representing pointers and references, “*TypeFormArr*” for modeling arrays, and “*TypeFormPrth*” for parentheses.

The first composite is called “*TypeFormSpecs*” and is stored by the *TypeForm-s* child class “*TypeFormSpec*.” It stores primitive specifiers modeled as “*PrimSpec*” objects (e.g. int, void, extern, inline, etc.) or references to objects that are representing types like *Classes*, *Typedefs* and *Enums*. The *TypeFormSpec* class is always the first one in the type prefix.

The philosophy of the second composite is quite similar to the first one: It is called “*TypeFormParams*” and is stored by the class “*TypeFormParam*.” It stores function parameters (*Parameter*) of the actual function. The *TypeFormParam* class is always part of the type suffix.

Note, that this flexible type representation allows an arbitrary number of recursions in representing parameters that are pointers to functions that have parameters that are in turn pointers to functions!

2.2.3 Templates

There are two kinds of templates in C++: class- and function templates. Our model makes no difference in representing them by separating the template

representation from the actual template object (similarly to the type representation). This template representation is a composite called "*TemplRep*." The two template classes are "*ClassTempl*" and "*FuncTempl*" that represent the class template and the function template, respectively.

Templates can have three kinds of template parameters: the "usual" type name, the non-type (e.g. value) and another template. All these parameter kinds have an abstract base class called "*TemplParam*." The type name parameter is represented by the class "*TemplParamTypeName*," which stores the type in the same way as the typed scope members do: it is composed with a *TypeRep* object. The non-type parameter is modeled with the class "*TemplParamNonType*" that is composed with a *Parameter* object. A non-type template parameter is syntactically the same as a function parameter. Finally, the parameter that is another template is represented by the class "*TemplParamTempl*", which simply stores the parameter template with a recursion: it is composed with a *ClassTempl* or *FuncTempl* class. Additionally, it can have a default value, so it may refer to the appropriate *ClassTempl* or *FuncTempl* object.

2.2.4 Relations

In C++ there are situations when objects are referring to other objects. In our model two of these relationships are modeled using relations.

The first one is the "*FriendRelation*," which stores friend edges. The "*Friend*" edge refers a class and a class or a function that is a friend of the first class.

The second one is the "*InheritanceRelation*," which stores inheritance edges. The "*Inheritance*" edge refers to two classes that are in direct inheritance relationship with each other. Some additional information is stored as well, like visibility and virtuality.

3. An alternative approach

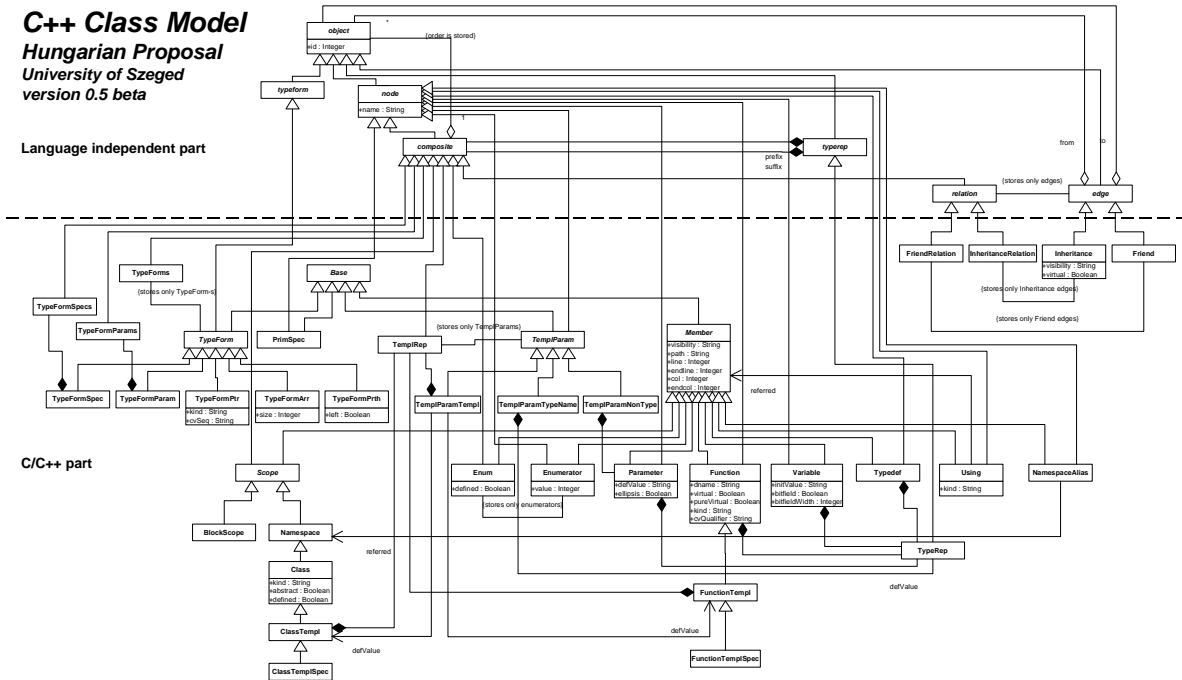
An alternative approach would be a model proposal without the language independent part. This may result in a more clear-cut model, but all advantages gained from the two-part architecture would be lost. In addition, all services of the language independent part would be needed to be integrated into the current C/C++ part. However, many of the aggregation and association relations would be more specialized to the concrete classes, and therefore many current constraints would disappear.

4. Conclusion

As it can be seen from the supplemented examples, the model described in this paper is suitable for representing any kind of C/C++ constructs, thus it is a good candidate for the standard exchange format for C/C++. For confirming this statement we have already implemented this model and we use it in our Columbus/CAN C/C++ front-end.

C++ Class Model
Hungarian Proposal
 University of Szeged
 version 0.5 beta

Language independent part



C/C++ part

Constraints:

- **typeprop**: stores only TypeForms-es
- **TypeFormSpecs**: stores only PrimSpec-s or type references (Class, Typedef)
- **Namespace**: location information is not used
- **Parameter**: If ellipsis is true => the other attributes are empty
- **Variable**: if bitfield is false => bitfieldWidth is empty

Deficiencies:

- **Template Specializations**: specialization arguments are not represented

Figure 1

Example 1: Namespaces

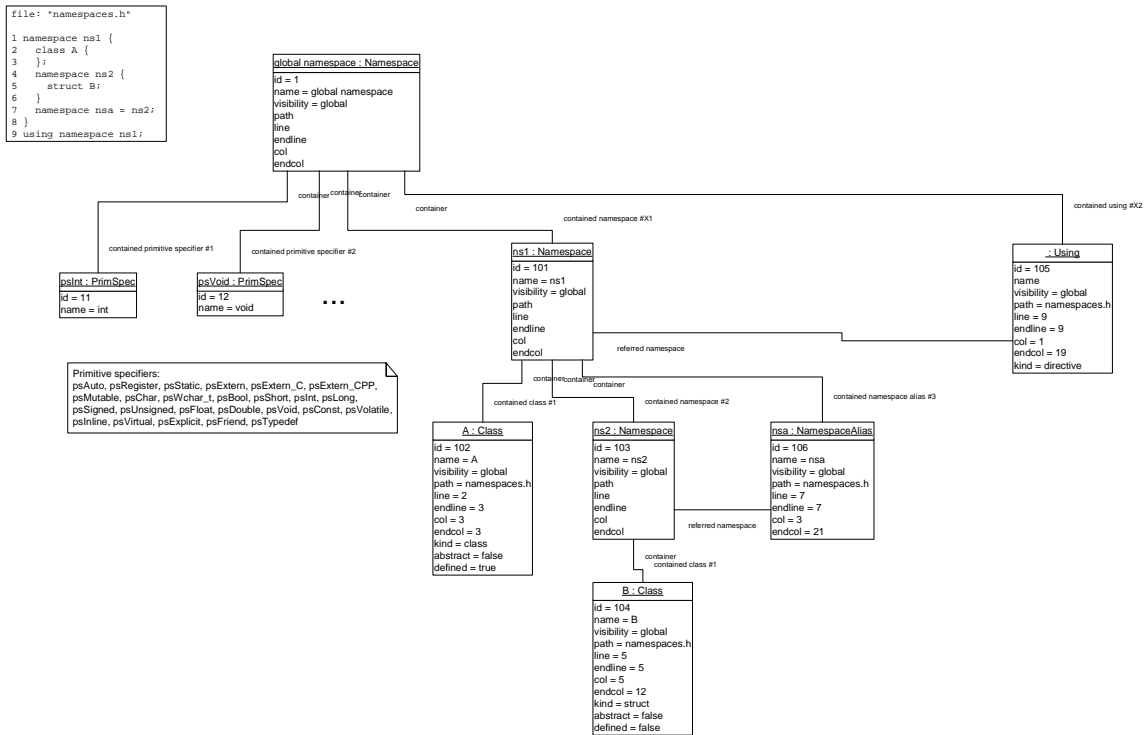


Figure 2

Example 2: Classes & Simple Types

```
file: "classes.h"
1 class A {
2   int *i;
3 public:
4   virtual void foo() const = 0;
5 };
6
7 struct B;
8
9 class C : public A {
10  friend struct B;
11  friend class E;
12  union D { /*...*/ };
13 };
14
15 class E;
```

C++ Class Model
Hungarian Proposal
version 0.5 beta

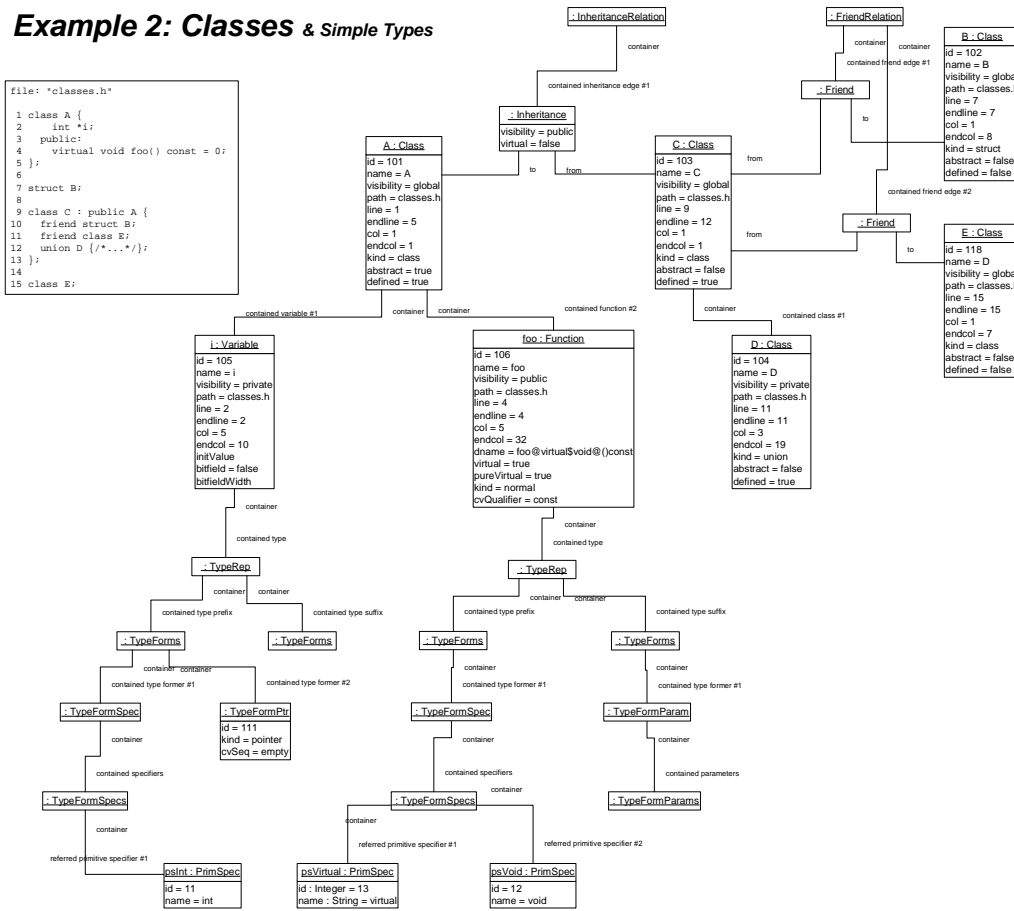


Figure 3

Example 3: Enums

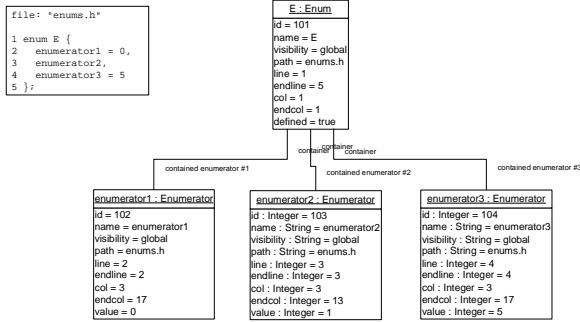


Figure 4

Example 4: Types

```
file: "types.h"
1 unsigned long int& (*pfoo)(A& a, int i = 0);
2
3 // the type representation is the same in
4 // the case of Functions, Variables,
5 // Parameters, Typedefs and TemplParamTypeNames
```

C++ Class Model
Hungarian Proposal
version 0.5 beta

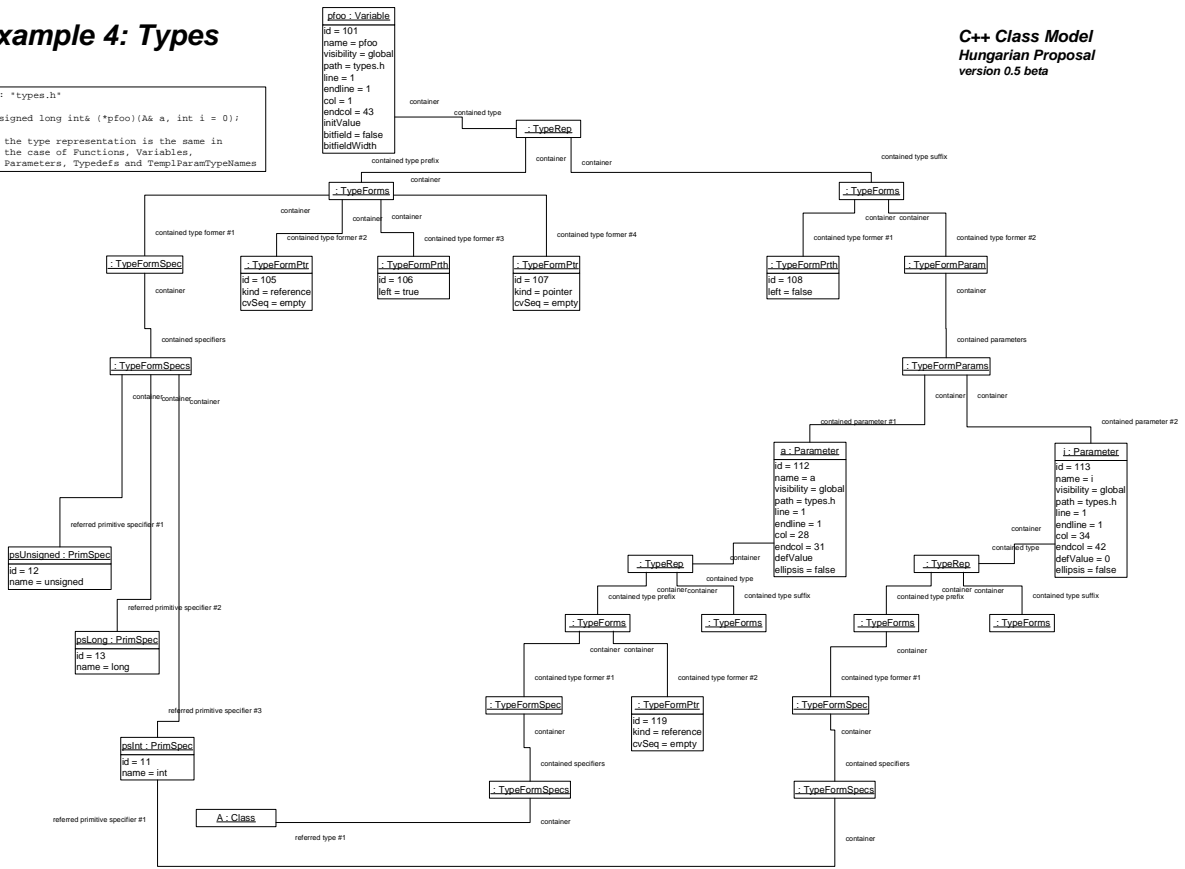


Figure 5

Example 5: Templates

```
file: "templates.h"
1 template <typename T2> class TC1, typename T1=int, int par1 = 3>
2 class TempClass { /*...*/ };
3
4 // function template is similar
```

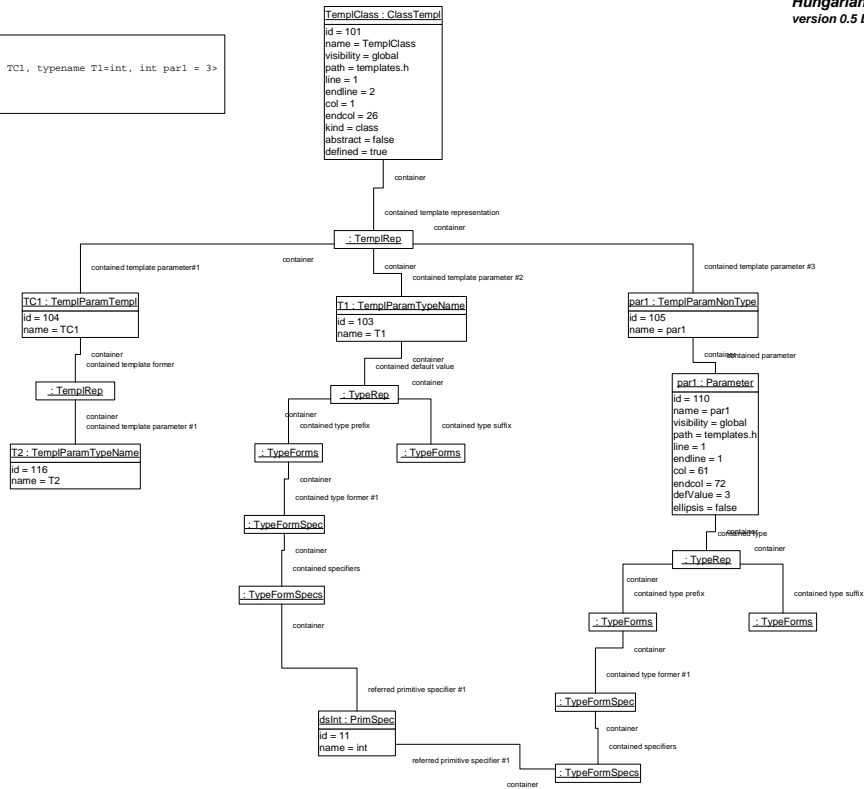


Figure 6