# Data Exchange with the Columbus Schema for C++

Rudolf Ferenc, Árpád Beszédes
University of Szeged, Hungary
Research Group on Artificial Intelligence
{ferenc,beszedes}@cc.u-szeged.hu

## Abstract

*To successfully carry out a software maintenance or reengineering task, a suitably assembled set of tools is required, which interoperate seaminglessly. To achieve this goal, an exchange format is needed that can be used to represent the facts extracted from a software system in a standardized way; serving as an output of one tool and as an input for other tools. In this paper we propose a modular schema for C++, called the Columbus Schema. The schema has been implemented in the Columbus/CAN front end framework tool and is already utilized in several usages, one of which is its representation in the GXL form.*

### Keywords

Tool interoperability, standard exchange format, C++ schema, front end, AST, Columbus/CAN, GXL

## 1 Introduction

In recent years it became increasingly apparent that in order to successfully perform a real-life software maintenance or reengineering task, a suitably assembled *set of tools* is required. This is in contrast with a popular belief from prior times that one integrated tool must be found which suits all of the needs of a reverse engineering project. However, this has as a consequence that in order to retain the flexibility of these tools' interchange, one must have also an *interchange format* for these tools.

Recently, many researchers indeed realized the importance of such interoperability of tools and a common exchange format; and this became an active research topic as well (e.g. [1, 2], [3], [6] and [12]). However, for the C++ language no such schema has been proposed until now that is accepted as a standard (or at least reference) description of C++ for interchange purposes among reverse engineering tools.

In this article we propose a new C++ schema—called Columbus—as a candidate for such exchange. The proposed schema satisfies some important requirements of an exchange format. It reflects the low-level (AST) structure of the code, as well as higher level semantic information (e.g. semantics of types). Furthermore, the structure of the schema and the used standard notation (UML Class Diagrams [14]) make its implementation straightforward, and what is even more important, an API is very simple to issue as well. Therefore, the Columbus schema is a good candidate for exchanging information among tools of various nature, e.g. a C++ parser front end and code-rewriters, metrics tools, documentation tools, and even compilers. This is already supported by several use cases, where the Columbus schema has been successfully used for data exchange. The most important of these applications is probably GXL, an already accepted medium for information interchange in reengineering [9, 10].

In the following, we present our schema in detail, followed by an example. In Section 3 we give details about the implementation and report our experience in recent use cases of the schema's application and finally we conclude our paper.

## 2 Columbus Schema for C++

The project team at the University of Szeged (in cooperation with Nokia Research Center and FrontEndART Ltd. [7]) created a C++ schema for various reengineering and reverse engineering tasks such as creating UML class diagrams and calculating metrics [17]. The ISO/IEC C++ standard of 1998 [11] served as the basis for all design decisions. More precisely, the schema models the "clean" C++ language syntax (preprocessed source code), it does not deal with macros and other preprocessor issues.

The Columbus schema is also used as the internal representation in the C/C++ extractor module of the *Columbus* reverse engineering tool called *CAN* [5] (see Section 3). The schema evolved into its current state in parallel with its implementation and is used also for analysis (e.g. resolving

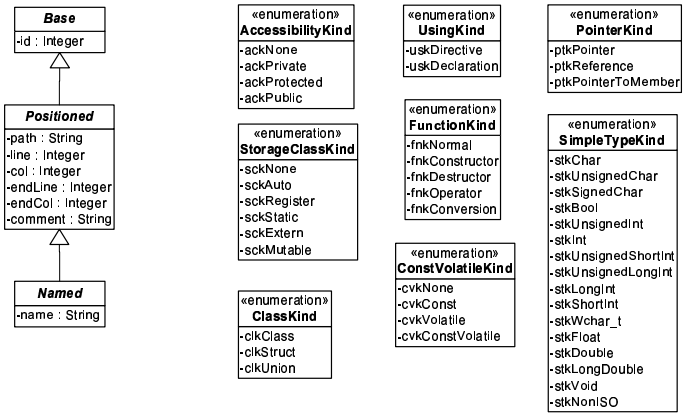


**Figure 1. Class diagram of the *base* package**

type names and scopes), as well as data exchange.

We present the Columbus C++ schema using UML Class Diagram notations [14], which has several advantages. Firstly, it uses a standard notation; secondly, at the same time it is close to implementation level and sufficiently abstracts the language. Thirdly, the schema can be used as a basis for an API to such a tool that uses the schema. Additionally, an implementation of the schema might define further semantic constraints in the class diagrams that make the schema more precise. This could include, for example, certain cases when an attribute of some base class is not used by one of its derived classes.

Externally, the model of a software system using this schema can be stored in various physical formats, including GXL [9, 10].

### 2.1 The structure of the Schema

Because of the high complexity of the C++ language, we have modularized our schema similarly to the proposal in the discussion part of our previous work [6]. This opens up also the possibility for its extension/modification. We divided the schema into seven packages:

- *base:* the base package containing the base classes and data types for the remaining parts of the schema.
- *struc:* this package models the main program elements according to their scoping structure, such as objects, functions and classes.
- *type:* the classes in this package are used to represent the types of the elements in the *struc*, *templ* and *expr* packages.
- *templ:* the package covers the representation of template parameter and argument lists, and is used by the *struc* and *type* packages.
- *statm:* the package contains classes modelling the statements.
- *expr:* the classes in this package represent all kinds of expressions.

Packages *statm* and *expr* are not presented in this paper because of space constraints. However, the schema without these packages is still a rounded whole, especially for certain applications where higher-level models are sufficient (e.g. extracting UML Class Diagrams).

### 2.2 The *base* package

The base package (Figure 1) contains the abstract class *Base*, which is the base class of all classes in our schema. A singly rooted hierarchy has many advantages (e.g. all classes have a common interface). It has one attribute: the node identifier *id*. The second class in this package is called *Positioned*. This abstract class extends *Base* with *location* (path, line, column, ...) and *comment* information. This class is the base class of all classes that represent source code elements, which have a position in the code. The third class *Named* extends the previous one with *name* information.

Apart from these classes the package contains different enumerations (*AccessibilityKind*, *StorageClassKind*, *ClassKind*, etc.) used by the schema.

### 2.3 *struc* – the structure package

This package contains classes that model the main program elements and their scoping structure, which are organized around *Member*, the most important child class of *base::Named* (see Figure 2). This abstract class is the parent of all kinds of elements, which appear in a scope (we use the term “member” in a more general way than usual). It has

the properties *accessibility* (`private`, `protected`, ...), *storageClass* (`static`, `extern`, ...) and *nonISOSpec* for capturing any non-ISO specifiers from different language dialects. In addition, member declarations can refer to their definition.

The first child of the *Member* class is the abstract class *Scope*, which is a member as well, because it can be contained by another scope. The composition from *Scope* to *Member* enables the class to store other members in an ordered way, which is a natural representation of the scope nesting in the C++ language (i.e. it is a *composite*). This recursive containment along with the other compositions named *"contains"* builds the basic *skeleton structure* tree of the modelled system.

The *Namespace* class is used to represent C++ namespaces. In a schema instance there must always exist at least one namespace object, which is called *"global namespace."* Our schema is basically designed in a project-oriented view that gives namespace scopes priority over file scopes (both cannot be represented in an AST at the same time, because namespaces can be defined across files/compilation units). However, the original path information is stored in the schema and the files can be restored from there.

The class named *Class* is similar to *Namespace*, because both represent scopes. It has the fields *kind* (`class`, `struct` or `union`) and if it is abstract and defined or not. Additionally, it is composed with two classes that represent the base classes and friends. The first one is the class *BaseSpecifier*, which models the inheritance relationship between two classes or between a class and a template instance. The additional information of accessibility and virtuality is stored as attributes. The second one is the class *FriendSpecifier*, which models the friend relationship between two classes, between a class and a function, or between a class and a template instance. In the case of modelling these two relationships among classes and template classes/functions, these classes are also composed with template argument lists that represent the actual template arguments used for the instantiation of the referred template. E.g. in the case of deriving class `A` from the template class `C`, which is nested in template class `B`, the base specifier will contain the argument lists `<int>` and `<D,char>`:
`class A : public B<int>::C<D,char>`.

The classes *Function*, *Object*, *Typedef* and *Parameter* are very similar, so they will be described together. Basically, these are the language elements that have a type. Our schema represents this with aggregations with the *TypeRep* class from the *type* package (see Section 2.4 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 if the object is a *bitfield* or not. In the case of functions there are additional associations: *throws* models the eventual exception specifications defined by the function, while *hasBody* and *hasLabel* are used to model the body (block scope with statements) and the jump labels of the function. The schema makes no difference in representing class attributes and local variables (these are all *Object*-s). 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 objects. Note, that the class *Parameter* is not a member (it is derived from the class *base::Named*). Instead of being contained by a scope, it can be a child of a function, object (if it is a pointer to a function), typedef (type alias of a pointer to a function) or another parameter (if the parameter is a pointer to a function).

The *Using* and *NamespaceAlias* classes are special as well, because they have a position in the code, but neither is a real member. The *Using* class refers to a namespace whose symbols can be used from the point of definition, or a single namespace member or a base class member that can be used with e.g. a modified accessibility. The *NamespaceAlias* class refers to a namespace and defines a new name (alias) for it.

Similarly to scopes, the *Enumeration* class is a composite, which stores an arbitrary number of *Enumerator*-s.

Our schema makes no difference in representing the two kinds of templates of C++ (class- and function-templates) by separating the template representation (the template parameters) from the actual template object, similarly to the type representation. This template representation is called *ParameterList* and is located in the *templ* package (see Section 2.5). The two template classes are *ClassTempl* and *FunctionTempl*, which represent class templates and function templates, respectively.

Template specializations (classes *ClassTemplSpec* and *FunctionTemplSpec*) are handled in a similar way as templates: they are composed with the *ArgumentList* class from the *templ* package, which is representing the template specialization arguments. In addition, the template specialization is referring to the template, which is being specialized.

Template instances can also be considered as members, although they do not have an exact position (they are not present in the source code directly but are generated by the front end). Similarly to the template specializations, the *TemplInstance* class is composed with the class *templ::ArgumentList*, and it is referring to the template it is being instantiated from.

## 2.4 The *type* package

The type representations (class *TypeRep*) are stored completely separated from the language elements, which are using them (e.g. functions or objects, see Section 2.3). This opens the possibility for storing each type only once, and referring to them from multiple nodes (see Figure 3).

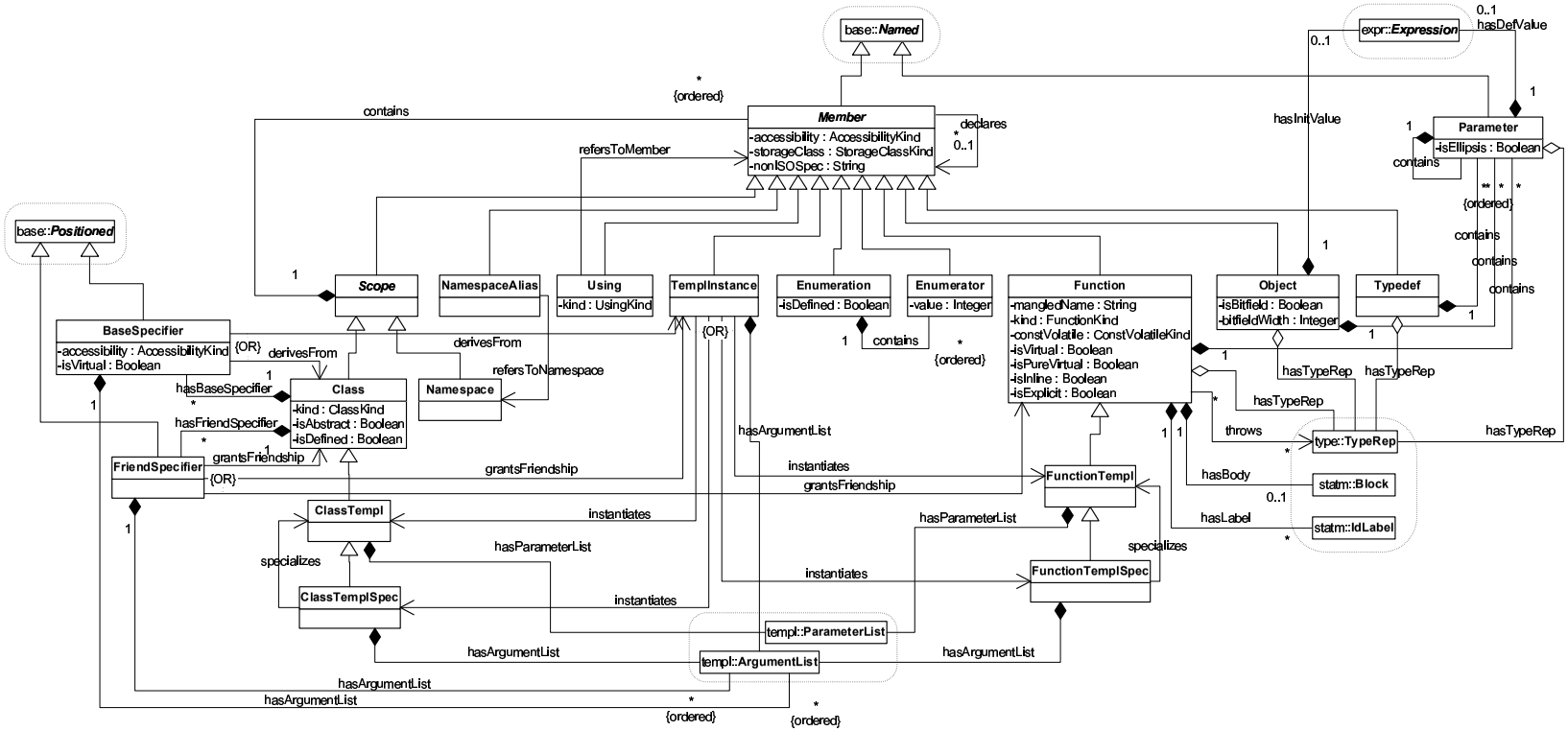


**Figure 2. Class diagram of the *struc* package**

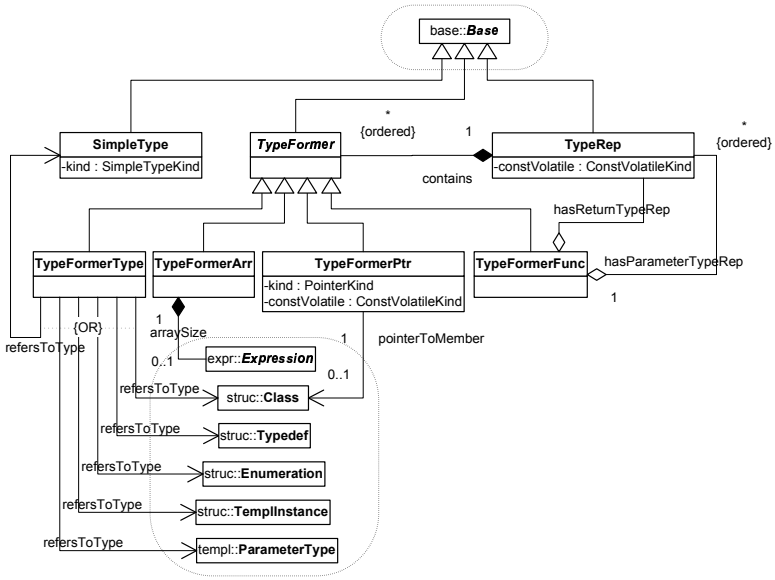


**Figure 3. Class diagram of the *type* package**

The type representation class *TypeRep* is composed of small parts that we call *TypeFormer*-s. These can be simple type formers like arrays (*TypeFormerArr*) and pointers/references (*TypeFormerPtr*); or type references (*TypeFormerType*), which can refer to simple (built-in) types (class *SimpleType*) or some other elements that represent types, like classes or typedefs. The *TypeFormerFunc* is a special type former, which is present if we are representing the types of functions or parts of a more complex type representation containing pointers to functions. It refers to the type representations of the function's parameters and its return type. Eventual const-volatile qualifiers that belong to a type are represented by an attribute in the *TypeRep* class.

The order of the type formers captures how the type is built up semantically. For example, the declaration `int *array[SIZE]` is represented with this *TypeRep*: (1) array of (2) pointers to (3) `int`-s.

Note, that this type representation allows an arbitrary number of recursions in representing the types of e.g. parameters that are pointers to functions that have parameters that are in turn pointers to functions.

## 2.5 *templ* – the template package

There are two special language elements that class and function templates and template specializations have: the template parameter list (represented with the class *ParameterList*) and the template argument list (represented with the class *ArgumentList*); see Figure 4. Both are composites, which store *Parameter*-s and *Argument*-s, respectively.

In C++ there are three kinds of template parameters: the "usual" type name, the non-type (e.g. value) and another template. The type name parameter is represented with the class *ParameterType*, which can possibly refer to a type representation as its default value. The non-type parameters are modelled with the class *ParameterNonType*, which represents its type the same way as, e.g., the function parameters. It can have a default value, so it may refer to the corresponding expression. Finally, the parameter that is another template is represented with the class *ParameterTempl*, which simply stores its template parameter list with a recursion: it is composed with the *ParameterList* class. Additionally, it can have a default value, so it may refer to the appropriate *ClassTempl* or *FunctionTempl* class.

Every template parameter has its corresponding template argument when the template is instantiated: the class *ArgumentType* represents a reference to a concrete type representation for the type parameter. Similarly, the *ArgumentNonType* class refers to a concrete expression and the *ArgumentTempl* class refers to the concrete class or function template.

## 2.6 Example schema instance

We illustrate the use of our schema through an example instance of it. We use a slightly simplified version of the example in [6] (see Figure 5). The AST for the example is given in Figure 6. We use an Object Diagram-like notation, where the object instances of the schema's classes are represented and the links that connect them clearly show the instances of various association and aggregation relations. The ordered associations are represented by numbering the links. We have simplified the diagram for clarity by omit-

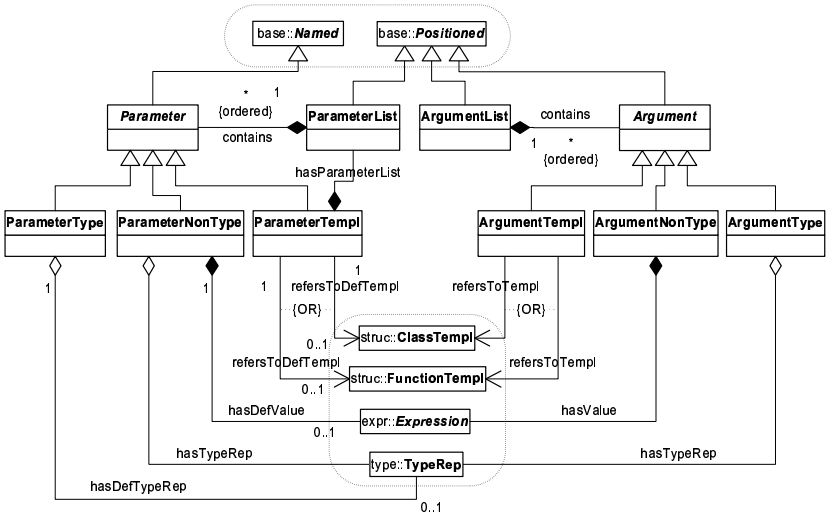


**Figure 4. Class diagram of the *templ* package**

ting attributes, such as line numbers, which are not necessary for the description.

The schema uses integers as unique identifiers for nodes, so the key of the topmost node is "1." The class of each node is given to the right of the key number; for example, the class of node 1 is *struc::Namespace*. The schema uses a *name* attribute in some nodes to give the name of the source item being represented; for example, the name attribute of node 1 is "*global namespace.*"

We will use the example to explain how the template class, the types, the object and function in it are represented in the AST according to the Columbus schema.

The template class *Array* is represented by node 11. It contains a template parameter list (node 12), which has two children (the order is shown on the connecting edges) that represent the two template parameters. The first one is a type name *T* that it is referred from two other nodes. The second one is the non-type parameter (value) *Size*, which has a type. This type is represented by *TypeRep* node 18.

The template class *Array* has two children (ordered): object *arr* (node 15) and function *get* (node 16). The type of object *arr* is represented by *TypeRep* node 20.

The facts that function *get* is virtual and const are stored as attribute fields in node 16. The function has one parameter with the name *idx* (node 17). The type of this node is the same as the type of template parameter *Size*, so it refers to the same *TypeRep* 18. We will present the type representation of function *get* in detail (node 24).

The *TypeRep* 24 contains a *TypeFormerFunc* node (25), which refers to the *TypeRep* of the function's return type (node 26) and the *TypeRep* of the parameter *idx* (node 18). The return type representation (node 26) stores the constness of the return type and contains two type formers for the type itself. The first one is a pointer (reference) former (*TypeFormerPtr* 27), which means that the return type is a reference to a node that represents a type. The second type former is a *TypeFormerType* (node 28), which refers to template parameter 13. So the overall meaning of the type representation is: a reference to template parameter *T*.

## 3 Implementation experiences

We will provide a two-fold application of the Columbus schema in our implementation. Firstly—as already mentioned in the Introduction—in parallel to the schema a reverse engineering tool also called Columbus [5] has been developed. Its C/C++ extractor module uses the schema as its internal representation. Because the schema description is in form of UML Class Diagrams the implementation of the extractor is rather close to the schema. This opens also a possibility to issue a schema-conforming API for accessing the internal representation. The second application is that the final result of the extraction can be presented according to the schema in various formats, such as GXL.

The Columbus tool implements a *general framework* for combining a number of reverse engineering tasks and provides a common interface for them. It supports project handling, data extraction, data representation, data storage, filtering and visualization. Extractors for different programming languages can be integrated into the Columbus framework using a *plug-in API*. The current version has a C/C++ extractor and a Java extractor is under development. The Columbus tool is available on the web, and it is free for scientific and educational purposes [7].

During extraction Columbus produces separate internal representations for each precompiled compilation unit and

```
1  template <class T, int Size>
2  class Array {
3    T arr[Size];
4  public:
5    virtual const T& get(int idx) const;
6  };
```

*The example implements a generic array, which expects two parameters (the type of the stored elements and the size of the array) and has a public function* get.

**Figure 5. Example**

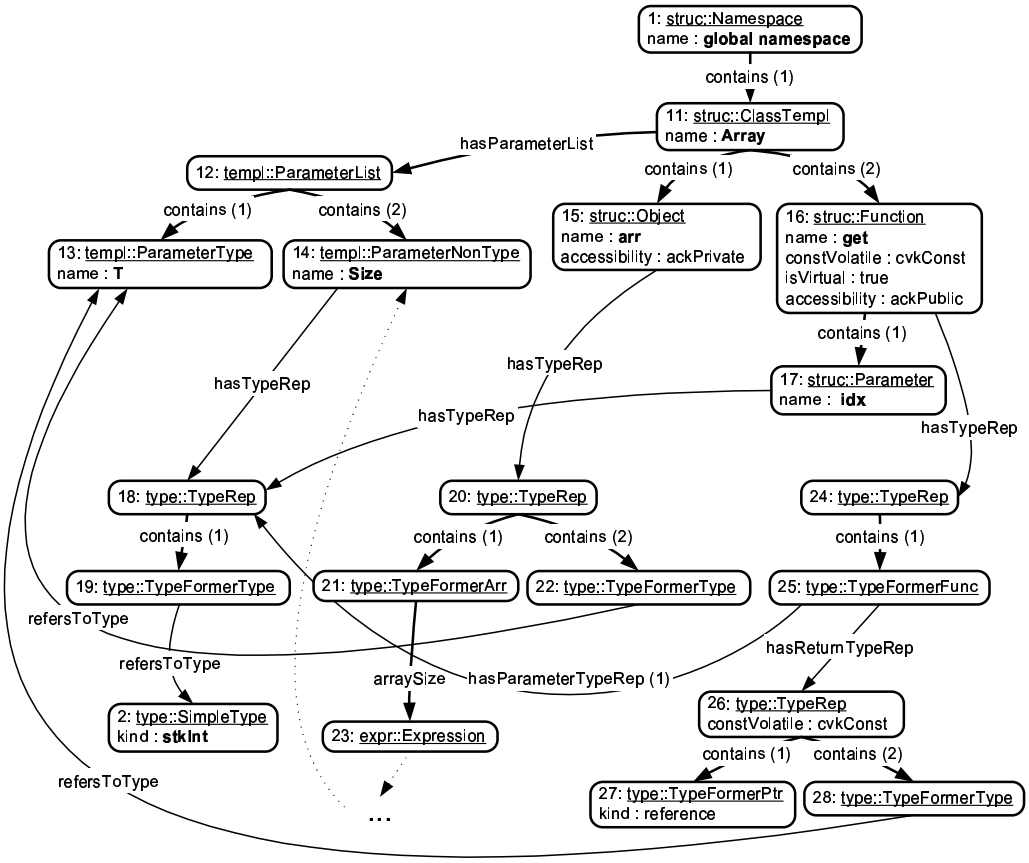


**Figure 6. AST for the example**

the *linker plug-in* merges these units into a unified AST. During linking duplicated elements, such as declarations in header files, are removed. After linking the extracted data can be exported into various formats. Some of these formats are simple reports (e.g. HTML), while others conform to the Columbus schema. One format is CPPML, our XML format specially designed for the schema. The most important one is however, the GXL output [9, 10], which realizes the Columbus schema as an XML-based graph description of the presented UML Class Diagrams.

Successful interchange of the data created by the Columbus tool according to the Columbus schema has been achieved in several usages. The first application was by Nokia Research Center in Nokia's proprietary UML design environment *TDE* [18]. The *Maisa* project of the Helsinki University for recognizing simple standard Design Patterns [8] in C++ programs also successfully utilized the output created by Columbus [4]. Another example of the schema's use is in a FAMOOS project with the Crocodile metrics tool [17]. One of the most important applications is probably the currently ongoing work on exchange between Columbus and the GUPRO tool, which uses GXL as its input format [3]. Another example of a successful interchange is with the *rigi* graph visualizer tool [13, 16].

## 4 Conclusion

This work was motivated by the observation that successful data exchange is crucial among reverse engineering tools. This requires a common *format*, which is applicable in various reverse engineering tools, such as front ends and metrics tools. A standard (or at least reference) schema must be found. In this paper we propose an exchange schema for the C++ language, called the Columbus Schema.

The Columbus schema is modular, thus providing additional flexibility for its extension/modification. It captures the (preprocessed) C++ language at low details (AST) and also contains higher-level elements. The description of the schema is given using UML Class Diagrams, which enables its simple implementation and easy physical representation (e.g. using GXL).

The schema is implemented in a reverse engineering framework tool Columbus and we also provide our experiences with this application of the schema, as well as several exchange examples with other tools.

In the near future we plan to validate the Columbus schema further by applying it in new tool interoperability usages, such as the interchange with *Rational Rose* [15]. We also plan to make available an Application Programming Interface according to the Columbus schema, which could be used to access the internal representation of the extracted C++ system by the Columbus tool. Apart from this, we wish to publish the details of the *statm* and *expr* packages.

## References


[1] Bell Canada Inc., Montréal, Canada. *DATRIX – Abstract Semantic Graph reference manual*, Version 1.2, Jan. 2000.

[2] The DATRIX homepage. `http://www.iro.umontreal.ca/labs/gelo/datrix`

[3] J. Ebert, R. Gimnich, H. H. Stasch, and A. Winter. GUPRO – Generische Umgebung zum Programmverstehen, 1998.

[4] R. Ferenc, J. Gustafsson, L. Müller, and J. Paakki. Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa. In *Proceedings of SPLST 2001*, pages 58–70. University of Szeged, June 2001.

[5] R. Ferenc, F. Magyar, Á. Beszédes, Á. Kiss, and M. Tarkiainen. Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems. In *Proceedings of SPLST 2001*, pages 16–27. University of Szeged, June 2001.

[6] R. Ferenc, S. E. Sim, R. C. Holt, R. Koschke, and T. Gyimóthy. Towards a Standard Schema for C/C++. In *Proceedings of WCRE 2001*, pages 49–58. IEEE Computer Society, Oct. 2001.

[7] Homepage of FrontEndART Ltd. `http://www.frontendart.com`

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.

[9] The GXL homepage. `http://www.gupro.de/GXL`

[10] R. Holt, A. Winter, and A. Schürr. GXL: Towards a Standard Exchange Format. In *WCRE 2000*, pages 162–171, Nov. 2000.

[11] International Standards Organization. *Programming languages — C++*, ISO/IEC 14882:1998(E), 1998.

[12] E. Mamas and K. Kontogiannis. Towards Portable Source Code Representations using XML. In *Proceedings of WCRE 2000*, pages 172–182, Nov. 2000.

[13] H. A. Müller, K. Wong, and S. R. Tilley. Understanding software systems using reverse engineering technology. In *Proceedings of the 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences (ACFAS)*, 1994.

[14] Object Management Group Inc. *OMG Unified Modeling Language Specification*, Version 1.3, 1999.

[15] The Rational Rose web site. `http://www.rational.com/products/rose`

[16] The Rigi web site. `http://www.rigi.csc.uvic.ca`

[17] C. Riva, M. Przybilski, and K. Koskimies. Environment for Software Assessment. In *Workshop on Object-Oriented Architectural Evolution, ECOOP'99*, 1999.

[18] A. Taivalsaari and S. Vaaraniemi. TDE: Supporting Geografically Distributed Software Design with Shared, Collaborative Workspaces. In *Proceedings of CAiSE'97*, LNCS 1250, pages 389–408. Springer Verlag, 1997.