# Towards a Standard Schema for C/C++

Rudolf Ferenc
University of Szeged
Research Group on
Artificial Intelligence
ferenc@cc.u-szeged.hu

Susan Elliott Sim
University of Toronto
Department of
Computer Science
simsuz@cs.utoronto.ca

Richard C. Holt
University of Waterloo
Department of
Computer Science
holt@uwaterloo.ca

Rainer Koschke
Universität Stuttgart
Institut für Informatik
koschke@informatik.
uni-stuttgart.de

Tibor Gyimóthy
University of Szeged
Research Group on
Artificial Intelligence
gyimi@cc.u-szeged.hu

## Abstract

*Developing a standard schema at the abstract syntax tree level for C/C++ to be used by reverse engineering and reengineering tools is a complex and difficult problem. In this paper, we present a catalogue of issues that need to be considered in order to design a solution. Three categories of issues are discussed. Lexical structure is the first category and pertains to characteristics of the source code, such as spaces and comments. The second category, syntax, includes both the mundane and hard problems in the C++ programming language. The final category is semantics and covers aspects such as naming and reference resolution. Example solutions to these challenges are provided from the Datrix schema from Bell Canada and the Columbus schema from University of Szeged. The paper concludes with a discussion of lessons learnt and plans for future work on a C/C++ AST standard schema.*

**Keywords**
Standard exchange format, C/C++ schema, GXL, parser, extractor, front end, AST, reverse engineering, Datrix, Columbus

## 1 Introduction

Recently, GXL (Graph eXchange Language) has emerged within the reverse engineering community as a standard exchange format (SEF) for data [3, 13]. In order to keep the format flexible, GXL does not prescribe a schema for software data. Instead, features have been provided for users to specify their own schema, separate from instance data. This concept is analogous to databases, which have a schema that is distinct from instance data. By **schema**, we mean a description of the form of the data, in terms of a set of entities with attributes and relationships that prescribe the form of the instance data. A schema is derived from a **model**, a description that relates entities in the schema to their real-world counterparts, thus providing a basis for meaningful interpretation of the data.

With a common syntax for exchange established, work now proceeds to defining standard schemas for various levels of analysis. A **standard schema** is a schema that tool builders have agreed upon to facilitate data exchange between tools. In this paper, we present results from work towards a standard schema for C/C++ at the abstract syntax tree level. These results are a catalogue of issues to be considered and some sample solutions. By presenting intermediate results we seek to advance the shared understanding within the field and to provide a basis for discussion. One of the hallmarks in the development of GXL was ongoing consultation and consensus building with the end user community, and we plan to follow in this tradition. Moreover, the creation of a standard schema for C/C++ is a complex problem and it would be beneficial to have an open process.

In this introduction, we lay the groundwork for discussion. We begin by defining the terms we will be using. We then provide a motivation for our work and describe the scope of the problem we are attempting to solve. Finally, we give a short overview of the remaining sections in this paper.

### 1.1 Terminology

Grammars are used to define the syntactic structure of a programming language. For a given sentence that is in the language, the grammar induces a certain derivation (or more than one if the grammar is ambiguous), characterized by a **parse tree**. This parse tree is determined by the underlying grammar and may contain unnecessary artifacts like chain rule derivations. The purpose of an **abstract syntax tree (AST)** is to describe the syntactic decomposition of the represented program, which is basically a tree-nesting structure, but without any unnecessary detail. What constitutes an unnecessary detail depends upon the intended use of the abstract syntax tree. Compilers, for instance, are generally not interested in brackets used to specify associativity within expressions, as this information is represented by the nesting structure of the represented expressions. On the other hand, source-to-source transformation tools have to reproduce the code as true to the original as possible and therefore need to retain information about brackets.

During the semantic analysis phase of front ends the ASTs are usually decorated with some additional information, such as resolution of names. These decorated ASTs are sometimes referred to as abstract semantic graphs (ASG). However, in this paper we use AST for denoting a decorated abstract syntax tree.

### 1.2 Motivation

Building tools is expensive, in terms of both time and resources. An infrastructure for tool interoperability allows tool designers and users to re-purpose tools, which helps to amortize this cost. Consequently, it is important to design an SEF that uses a representation that is compatible with the largest number of tools possible.

Flexibility is particularly important with a C++ front end because its output is the basis for many other downstream analyses, such as design recovery, clone detection, calculating metrics, architecture analysis, and code transformation. Writing a robust and reliable C++ front end is expensive and is a project with little research value. Nevertheless, such a tool is needed to handle a growing number of industrial case studies.

These two factors taken together illustrate the importance of designing a robust schema for C++ ASTs. Consequently, we need to work carefully on this standard schema and to solicit feedback through activities such as the discussion and publication of intermediate results, as we do in this paper.

### 1.3 Fundamental Issues

When undertaking a large and complex project, an important first step is to define its scope by articulating both what we are doing and what we are not doing.

- *We are creating a standard schema for C/C++ at the AST level.*
- *We are primarily working on defining a schema rather than specifying a front end.* However, we will also make prescriptions for front ends that implement the schema. It is difficult to completely separate the two; for example reference resolution is not a design decision that affects the schema, but it does need to be specified in order to create a standard format for exchange.
- *The schema should be independent of any parsing technology.* This independence would permit changes to the schema and grammar without affecting the abstract syntax, e.g., the grammar from which the schema is derived may need to be restructured into an LALR(1) grammar if yacc is used to generate the parser. This change should not affect the schema.
- *We are initially focusing on ANSI C++.* We intend to work on this language first and broaden later. We are aware of the need to be compatible with other languages (e.g. Ada, COBOL) and in particular, dialects of C and C++ (K&R, ANSI, Borland, Microsoft, etc.).

Finally, there is one fundamental issue that we have not yet resolved: who are the end users of this schema and what are their requirements? Clearly, the usefulness of this schema will depend on where it will be used. Transformation tools require a "source code complete" representation, i.e. one with enough detail to allow regeneration of the original source. If this schema could support this level of detail, would this format even be useful for such an application?

### 1.4 Overview

The remainder of this paper will be a catalogue of issues in creating a standard schema for C++ ASTs and examples of solutions for these issues. The schemas that provide those examples are introduced in Section 2.

Section 3 discusses three levels of Representation Issues. This list of issues extends previous work on developing an exchange format by Bowman, Godfrey, and Holt [8]. Issues on the lowest level, Lexical Structure, include the Line/Column Number Problem, and the Project Handling Problem. The second level, Syntax, encompasses the basic mechanisms of how the format will represent C++ AST. Finally, the highest level, Semantics, deals with issues such as the Resolution Problem and the Naming Problem. A Discussion of lessons learnt and alternative solutions is presented in Section 4, and followed by a Conclusion in Section 5.

## 2 Existing Schemas

Because the main purpose for ASTs is to represent the syntactic structure of the program, the abstract syntax is derived from the grammar. We can divide the methods of deriving the AST into two cases, automatically derived schemas and manually derived schemas.

*Automatically derived schemas.* The translation from the parse tree to the AST can be formally specified by a set of transformations, which can be automatically interpreted to carry out the translation from parse tree to AST. Some transformations may be applied to reduce the size of the tree. A typical transformation is to eliminate tree nodes for concrete terminals. Changes to the language's grammar influence the abstract syntax. A key advantage of automatically derived schemas is that creation of the abstract syntax is completely automatic, so schemas for new languages can be derived quickly. Also, every syntactic detail can be captured in the ASTs with automatically derived schemas. For example, Semantic Designs [5] and the Dutch research group at CWI [23] derive their schemas automatically from the underlying grammar and use other transformations to trim the resulting ASTs.

*Manually derived schemas.* The AST can be designed manually. In this case, the translation is carried out by software engineers. Manually derived schemas are generally more abstract, because human judgment is superior in deciding what is essential in a given language. The Datrix and Columbus schemas described in this paper are both manually derived.

Schemas can be *language-specific*, that is, applicable to only one language or *language-independent*, applicable to a set of languages. The Bauhaus schema, for instance, models C and a subset of Ada in one joint schema [18]. The Datrix and Columbus schemas both support a form of generalization across languages that are similar to C++. Multi-language and multi-platform compiler suites generally use a common intermediate format to minimize the number of interfaces between front- and back ends. For example, the Stanford University Intermediate Format (SUIF) works with different languages, currently C and Fortran [4].

Another way to attain independence from the language's grammar is through an application programming interface (API) that provides traversal and query operations on the AST. One successful example is the Ada Semantic Interface Specification [16].

In the remainder of this section, we briefly introduce the Datrix schema and the Columbus schema for C++ ASTs. These two schemas will provide sample solutions to issues discussed in the next section. Unfortunately, lack of space prevents us from including other noteworthy schemas, such as ones from CIA, gen++, Semantic Designs [5], Visual Age C++ [17], and cppML (Waterloo version) [19].

## 2.1 Datrix Schema

The Datrix Group at Bell Canada developed a schema for representing an AST for C/C++ and Java programs. Their goal was to create a common front end for an extensive set of software analysis and assessment tools.

**Schema documentation.** The Datrix schema is specified in detail in a report available via the web from Bell Canada [1].

**Source completeness.** The Datrix schema represents essentially all syntactical and some semantic information about the source program. Source completeness is violated in two aspects. First, the schema represents only preprocessed code, therefore all macro and other preprocessor information is lost. Second, semantically meaningless details, such as redundant semicolons and brackets are not retained. The first problem is hard to solve (see Section 3.1.1), while it is relatively easy to overcome the second by extending the schema.

**Language independence.** The Datrix schema was designed initially to handle C++, C and Java. While this original schema handles a limited set of languages, the intention was to extend and specialize the schema to handle other languages.

**Bell Canada Implementation of Datrix Schema**

The Datrix team at Bell Canada implemented a front end that parses C/C++/Java and emits information in VCG (for visualization) [22], and TA [14]. It has been used in production to assess software from vendors and serves as a proof of concept that the Datrix schema is workable. The implementation is proprietary to Bell Canada, but a binary executable is available to research institutions free of charge [2].

**CPPX Implementation of Datrix Schema**

In the first half of 2001, the team of Thomas Dean, Andrew Malton and Ric Holt built CPPX (C++ Extractor) [9]. This Open Source tool is based on GNU's GCC front end and produces information according to the Datrix schema for C++ in VCG, TA, and GXL formats.

## 2.2 Columbus Schema

The project team at the University of Szeged (in a cooperation with the Nokia Research Center) created a C/C++ schema for various reengineering and reverse engineering tasks such as creating UML Class Diagrams [20] and calculating metrics [21]. The ISO/IEC C++ standard of 1998 served as the basis for all design decisions [15].

**Schema documentation.** A description of the Columbus schema with examples is available on the web [10].

**Source completeness.** In this aspect, the Columbus schema is similar to the Datrix schema, except that it captures redundant parentheses.

**Language independence.** The schema has a *language-independent* part and a *language-specific* part. The former acts as a *base-schema* to provide a common root for modeling other programming languages. It consists of abstract classes that offer language independent representations of language elements (*nodes*), scopes (*composites* [12]), and types. The language-specific schema is built on top of the base-schema and extends it with C/C++ elements.

**Implementation of Columbus Schema**

The Columbus schema is used as the internal representation in the C/C++ extractor module of the *Columbus* reverse engineering tool. The schema evolved into its current state in parallel with its implementation and is used also for analysis (e.g. resolving type names and scopes) as well as data exchange.

The Columbus system has been developed in cooperation between the Nokia Research Center and the University of Szeged, Hungary by the team of Árpád Beszédes, Rudolf Ferenc, Ferenc Magyar, Tibor Gyimóthy, Mikko Tarkiainen and Gábor Márton. This 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 [6, 11]. 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.

During extraction the system produces separate internal representations for each precompiled compilation unit and the *linker plug-in* merges these units into a unified AST. During linking, duplicated elements, such as declarations in header files, are removed.

# 3 Representation Issues

In designing an AST-level schema for a language, we need to decide what elements from each of three levels of abstraction we want to represent.

- *Lexical Structure:* tokens, separators, layout, and compilation units.
- *Syntax:* features of the language specified by the grammar.
- *Semantics:* interpretations placed on the syntax, such as name resolution, control and data flow information.

Some decisions can be made easily. For example, we definitely want the syntax. We also need some semantic aspects, like name resolution (including resolution of overloading). When multiple compilation units are linked, we also need some kind of global name resolution. Other semantic information such as control and data flow are not intrinsically part of the AST, but other tools should be able to compute these flows using the AST. It is these kinds of semantic analysis, and not parsing, that makes writing a C++ front end difficult.

## 3.1 Lexical Structure

In C/C++, it is straightforward to identify the lexical tokens; it is less straightforward to decide which ones should be stored in the schema. Some tokens are not syntactically significant, such as spaces and line breaks, so some front ends elect not to store them. These and other lexical problems are discussed in this section and we show how they are solved in the two schemas.

The Datrix and Columbus schemas handle lexical structure similarly; both deal with preprocessed code and maintain all (or almost all) lexical information in the AST.

### 3.1.1 Preprocessing

Both the Datrix and Columbus schemas deal only with preprocessed code. Macros in C/C++ are a complex and expressive programming language on their own. They can be used to set up conditional compilation and even to tamper with the syntax of C/C++. Due to this flexibility, it is difficult to represent (and subsequently reconstruct) arbitrary preprocessor artifacts.

### 3.1.2 Line/Column Number Problem

This problem concerns the recording of location information for entities in the schema and has been discussed by Bowman, Godfrey, and Holt in Section 3.3 of their paper [8]. They noted some complications within this problem: 1) the path in a mangled file name needs to be invariant over different installations; 2) location ranges for a single entity are not always contiguous (examples are C++ classes and conditional compilation units); and 3) the level of detail required varies significantly between tools (some need line numbers, others require column numbers as well, and yet others use file offset).

**Datrix Schema**

The Datrix schema records the line and column position of most items in the AST. Each node uses the *beg* (begin) and *end* attributes to record line and column numbers. For example, the attributes *beg: 24.8* and *end: 25.13* indicate that an element begins at line 24, column 8 and ends on line 25, column 13.

File names are stored in special nodes within the AST, at the point where the file is included. This approach assumes that included files contain sub-trees of the AST, which is sometimes not true. Due to this difficulty, the CPPX implementation has adopted an alternate approach, much like the one used in the Columbus schema.

**Columbus Schema**

Every scope element (e.g. class, variable, function) has an attribute called *path* for storing the path and file name of its declaration/definition and *line, endline, col* and *endcol* attributes for saving the exact location in the file. Language elements are organized according to scopes (namespaces, classes) and file information is stored in the members. The schema does not have an entity for representing files/compilation units.

### 3.1.3 Project Handling Problem

A software project, or system, consists of a set of source code files (compilation units). The language elements in a project can be organized hierarchically by compilation units or by namespaces. These two hierarchical structures offer two different ways of constructing an AST for the entire project; thus we have the Project Handling Problem. Sometimes known as the Linking Problem, this issue is also related to the Naming Problem in Section 3.3.1.

**Datrix Schema**

The Datrix schema uses mangled names for externally visible symbols in the same way that standard compilers and linkers use mangled names to allow linkage between separately compiled parts of a whole program. The graphs for each separate compilation are linked to form a single large graph, in which common parts, e.g., header files, are combined. At the time of writing, neither the Bell Canada nor the CPPX implementations perform this linkage.

**Columbus Schema**

The Columbus schema gives namespace scopes precedence over file scopes. There must always be at least one namespace object, the *global namespace*, which represents the top-level namespace of the project files. Path information is stored in the member objects and can be used to reconstruct files. Similarly to Datrix, the Columbus schema uses mangled names for externally visible symbols to help link separate compilation units.

## 3.2 Syntax

Now we turn our attention to Syntax, the raison d'être of the AST. We will use a common example to explain how the Datrix and Columbus schemas represent C++ templates, types, functions, and statements. The code for the example is in Figure 1, while the ASTs for the two schemas are given in Figures 2 and 3, respectively. We have simplified both diagrams for clarity by omitting nodes and attributes, such as line numbers, that are not necessary for our discussion.

The two schemas and their diagrams use some common conventions. Both schemas use integers as unique identifiers for nodes, so the key of the topmost node in both figures is "1". The class of each node is given to the right of the key number; for example, in Figure 2, the class of node 1 is *Generic*. Both schemas use a *name* attribute in nodes to give the name of the source item being represented; for example, in both figures, the name attribute of node 1 is *Array*.

Edges in the basic AST are drawn using thick lines, while other relationships, such as references to declarations, are drawn using thin or dotted lines. Sometimes edges are ordered and this is shown in the diagrams using parenthesized order numbers, for example, in Figure 2, *ArcSon(2)* from node 1 to node 3 indicates that this is the second *ArcSon* edge leaving node 1.

We will now use the common example to explain how the template, types, function and statements are represented in the two schemas.

### 3.2.1 Templates

C++ templates are well known for being complex and difficult to handle. In the example, the template has two kinds of parameters: a type (*class T*) and a value (*int Size*).

**Datrix Schema**

The Datrix schema represents templates by a straightforward encoding of their syntactic structure. For example, in Figure 2, the *Array* template is represented by node 1, together with its four descendent nodes, numbered 2 to 5. These four represent template parameter *T*, template parameter *Size*, private member *arr*, and public member *get*.

```
template <class T, int Size>
class Array {
  T arr[Size];
public:
  virtual const T& get(int idx) {
    T& t = arr[idx];
    return t;
  }
  /* set()...*/
};
```

**Figure 1. The common example**

*The source code excerpt implements a generic (template) array, which expects two parameters (the type of the stored elements and the size of the array) and has a public function* get *that returns the stored element. The language features in this example were chosen to illustrate key decisions in designing the two schemas.*

The edges from node 1 to its descendents are *ArcSon* edges (they are arcs to sons). Node 2 is a *TemplParamType* object, which indicates that *T* is a *type* template parameter. Analogously, node 3 is a *FormalFctParam* object, which indicates *Size* is a *value* parameter. The contents of the template are located by the third and fourth *ArcSon* edges descending from *Array*'s node 1, which connect the nodes for *arr* and *get*.

**Columbus Schema**

There are two kinds of templates in C++: class- and function templates. The schema makes no difference in representing them by separating the template representation from the actual template object. This template representation is a composite node called *TemplRep* (node 2 in Figure 3). The two template classes are *ClassTempl* (node 1) and *FuncTempl* that represent the class template and the function template, respectively.

Templates can have three kinds of template parameters: the "usual" type name called *TemplParamTypeName* (node 3), the non-type or value (node 4), and another template. The non-type parameter is modeled with the class *TemplParamNonType* that contains a *Parameter* (node 5) object. A non-type template parameter is syntactically the same as a function parameter. Finally, the parameter that is itself a template is represented by the class *TemplParamTempl*, which simply stores the parameter template recursively composed with a *ClassTempl* or *FuncTempl* class.

The schema handles both class and function templates in the same way by separating the template representation from the actual template object. Consequently, the template representation can be treated as a module separate from the rest of the schema.

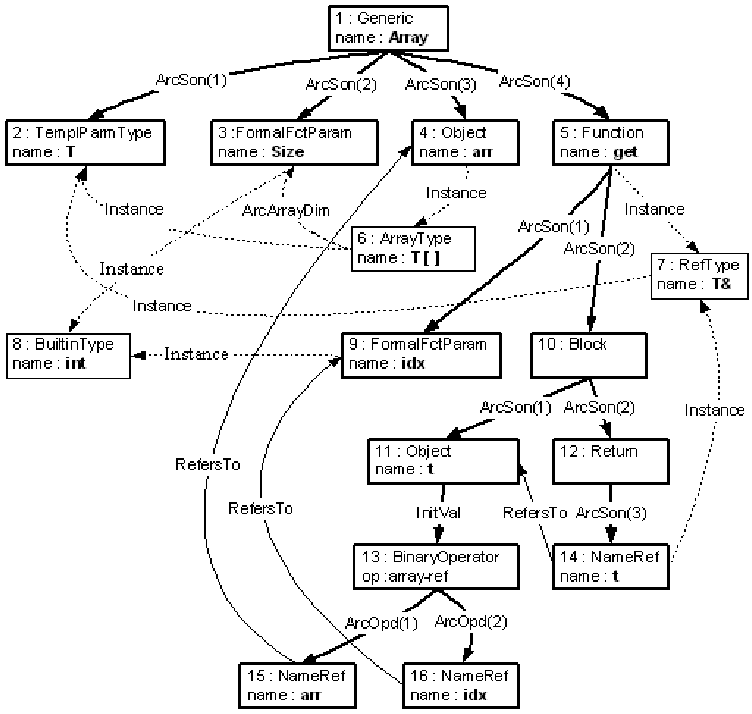


**Figure 2. Datrix AST for the common example**

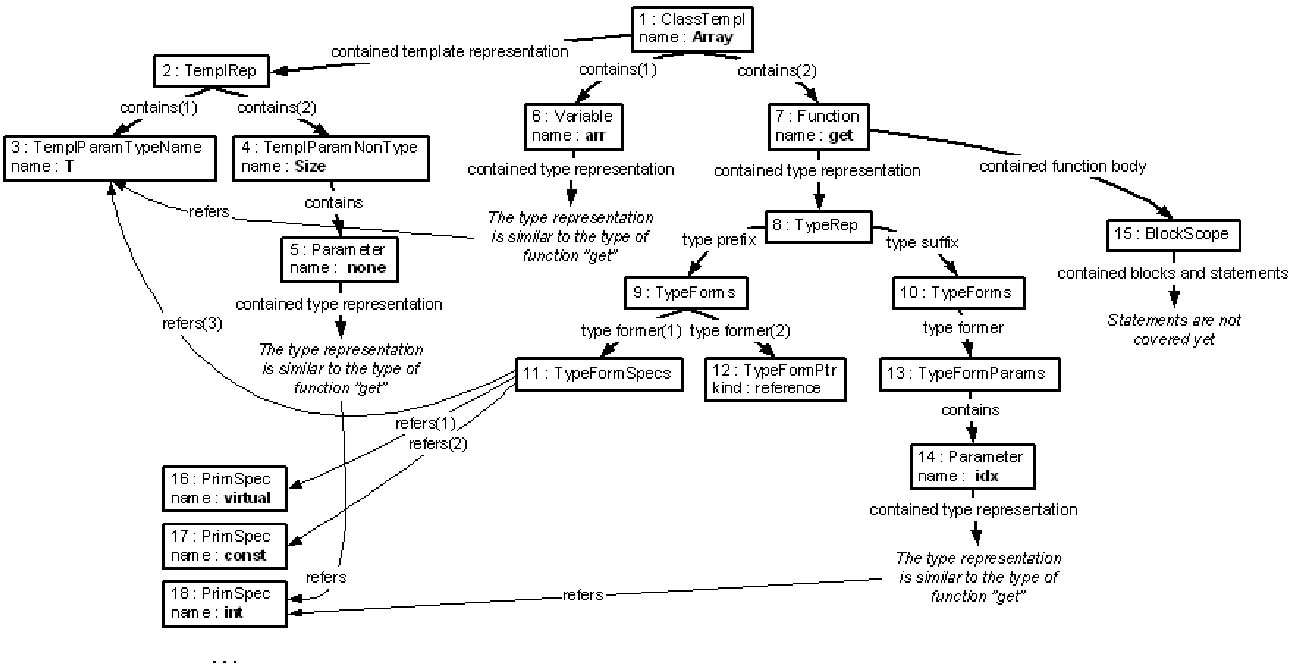


**Figure 3. Columbus AST for the common example**

#### 3.2.2 Types

Like templates, the representation of types is also a complex and difficult task for the C/C++ language. In the common example there are three elements that have types: an array (*T arr[Size]*), a function (*virtual const T& get(int)*) and a function parameter (*int idx*).

**Datrix Schema**

The general pattern for representing types in Datrix is to use a node for the element and to add nodes and edges to show its type and value. In the example, *T arr[Size]* is represented starting at node 4 in Figure 2. The array, *arr* has an *Instance* edge connecting it to *ArrayType* (node 6). From there, an *Instance* edge indicates that the array element is of type *T* (node 2), and an *ArcArrayDim* edge points to the array size, *Size* (node 3). The reference type *T&* is represented with the class *RefType* (node 7) and the function parameter *int idx* is represented with a *FormalFctParam* (node 9).

**Columbus Schema**

The type representation is modeled with the class *TypeRep* (node 8 in Figure 3) that contains two composite nodes called *TypeForms* that stand for the type prefix and suffix (nodes 9 and 10, respectively). The type prefix represents the part of the type before the name of the typed object (*virtual const T&*), while the type suffix represents the part after it; typically arrays (*[Size]*) and function parameters (*int idx*).

The type prefix and suffix consist of small building blocks called *TypeForms* (nodes 11-13). These can be simple ones like pointers (*TypeFormPtr*), arrays (*TypeFormArr*) and parentheses (*TypeFormPrth*); or composites like declaration specifier lists (*TypeFormSpecs*) and function parameter lists (*TypeFormParams*). The first composite (node 11) stores references to primitive specifiers modeled as *PrimSpec* objects (nodes 16-18) or to objects that are representing types such as classes, typedefs and enums. The second composite (node 13) stores the function parameters. Note, that this flexible type representation can encode an arbitrarily deep nesting of parameters that are pointers to functions.

Similar to the template representation, the type representation is also completely separate from the typed object (e.g. a variable). This way a unified type representation is achieved and enforced in the schema, which contributes to the modularization of the schema as well.

#### 3.2.3 Functions

The AST for a function needs to record the function's name, its parameters, its return type and body. In the common example (Figure 1), the function *get* has one parameter, *idx*, a return type of *T&*, and a body containing two statements.

**Datrix Schema**

Each function is represented by a sub-tree rooted at a node of type *Function* with the function name stored in the *name* attribute. There are edges from this node to function parameter(s), the body of the function, and return type. In Figure 2, the sub-tree for the *get* function is rooted at node 5. From there, the first son arc, *ArcSon(1)*, locates its parameter *idx* (node 9) and the second son arc points to a *Block* object starting the function body sub-tree (node 10). An *Instance* edge connects its return type *T&* in node 7.

**Columbus Schema**

In the Columbus schema all member objects (variables, functions, typedefs, etc.) are represented exactly in the same way: they are derived from the abstract base class called *Member*, which has common properties for all members like visibility and location information. Individual member classes are: *Variable* (node 6), *Typedef*, *Function* (node 7), *Parameter* (nodes 5, 14), *Enum*, *Enumerator*, *Using* and *NamespaceAlias*. These classes also contain attributes (virtual, default value, etc.) specific to the elements they are representing.

Function parameters are treated as part of its type representation (see Section 3.2.2), so parameters of function pointers are handled the same way.

#### 3.2.4 Statements

The AST needs to represent statements and also expressions. These are illustrated in the common example (Figure 1), by the two statements on lines 6 and 7 in the body of the *get* function.

**Datrix Schema**

In the Datrix schema, structured statements, such as for-loops and if-statements, are represented using specialized node classes. Expressions are represented by an operator node with an attribute that records the particular operator and edges to its operands. Simple statements, notably assignment statements, are considered to be expressions.

Statement 6 in the example declares local variable *t* and initializes it. This is represented in Figure 2 starting at node 11. An *Instance* edge indicates *t*'s type, *T&* (node 7). An *ArcInitVal* edge points to an expression rooted at node 13 that gives *t*'s initial value. In this *BinaryOperator* node, the *op* (operator) attribute is *array-ref* to represent the subscripting expression, *arr[idx]*. Its operands are found by following the *ArcOpd* edges to its children (nodes 15 and 16).

**Columbus Schema**

The Columbus schema does not cover statements yet, but this is ongoing work. Design decisions and experiences from the Datrix schema will be used to make the schema as useful as possible.

The elements discussed in this subsection give the flavor of how the two schemas deal with syntax. For more information about how they handle other language features, consult their respective documentation schemas [1, 10].

### 3.3 Semantics

In this section we discuss some of the semantic information to be included in the schema, in particular naming and resolution. While this information is not strictly part of the abstract syntax, it is needed by downstream analysis tools.

### 3.3.1 Naming Problem

This problem was discussed by Bowman, Godfrey, and Holt in their paper [8]. When storing and exchanging data with GXL, each entity must have a unique identifier. While automatically generated identifiers suffice, they make it difficult to combine data from different compilation units (and different tools), and to compare different versions of the same software system. Some approaches use the name of the entity itself plus some path information to generate the unique identifier. Unfortunately C/C++ allows different elements to use the same name. For example, labels and variables can share names because they are stored in different symbol tables.

**Datrix Schema**

Entities in the Datrix schema have unique identifiers that are positive integers assigned arbitrarily by the front end. The approach does not use mangled names nor the full path name; rather, it simply uses the names as they appear in the source program. Nodes with the same name are distinguished by their unique identifiers. Since neither implementation of the Datrix schema currently performs global linking, mangled names are not (yet) needed.

**Columbus Schema**

The Columbus schema uses the same approach as Datrix to identify the nodes. Besides the unique identifier, a so-called *mangled/decorated* name is created for each function in a manner very similar to compiler systems. Since C++ allows function overloading, multiple functions with the same name can exist within the same scope. This mangled name allows us to distinguish them by including the name and return type of the function together with the names and types of its parameters. For example, the function *get* in the common example would have the mangled name: *get@virtual$const$T$&@(int)*).

Different identifiers assigned to the same entity in different compilation units (e.g. included from the same header file) are reconciled during the linking phase. The front end re-assigns these identifiers and all references to the original identifiers are checked and corrected. Like Datrix, scoping information is not part of the name, because it can be unambiguously determined from the hierarchical structure of the AST.

### 3.3.2 Resolution Problem

Bowman, Godfrey, and Holt also discussed this problem [8]. They wrote, "Fact extractors detect when one source code entity refers to another, and record this as a relation. The algorithm the extractor uses to determine which entity is referred to depends on the source language and the implementation of the extractor" (Page 96). They gave four categories of resolution:

1. *Not resolved.*
2. *Resolved to declaration.* Typically, compilers only resolve to this level. An identifier may have multiple declarations, but only one definition.
3. *Resolved to static definition.* Typically, a linker is used to resolve all references to global variables and functions to the appropriate definition.
4. *Resolved to dynamic definition.* This level of resolution requires dynamic analysis tools and is only needed for languages that use dynamic binding, which includes C/C++.

**Datrix Schema**

The Datrix schema resolves the use of an identifier to its corresponding declaration, as per category two in the above list.

**Columbus Schema**

Not all references are resolved in the Columbus schema, because statements are not yet represented. However, the linker in the front end resolves references to forward declarations (e.g. global variables and functions) to their static definition, as per category three.

## 4 Discussion

In this section, we will consider some of the lessons learnt from attempting to design a standard schema for C++ ASTs. We begin with a comparison of the two schemas and then critically examine our approach to solving this data exchange problem. This discussion also considers alternative approaches, such as source code tagging, standard APIs, and modular schemas.

### 4.1 Comparison of the Two Schemas

Not surprisingly, the independently developed Columbus and Datrix schema have a lot in common since they are both derived from the C++ grammar. However, they differ in their terminology and details. Since the Columbus schema is not yet complete, we cannot compare their handling of statements and expressions, but we can compare them in other respects.

One significant difference between the two schemas is the representation of types. The type representation of the Columbus schema directly reflects the syntactic decomposition of the type declaration whereas the Datrix schema has a more straightforward semantic representation. The Columbus schema separates prefix and suffix parts for declarations and captures type qualifiers and specifiers as nodes within the prefix and suffix parts. Though this modeling eases regenerating the original code, it complicates type comparisons because prefix and suffix parts need to be combined to get the full type information.

The Columbus approach appears to be more detailed, more symmetric and uses additional nodes and entity types to encode structure than the Datrix approach. Consequently, the Datrix schema requires the reader to have greater knowledge of the source language information. For instance, the Columbus schema clearly distinguishes the template parameters from the member declarations of the template by a distinguishable edge and a node *TemplRep* that contains the template parameters (node 2 vs. nodes 6 and 7 in Figure 3). In contrast, the Datrix representation has only *ArcSon* edges leading to both template parameters and members (see edges coming from node 1 in Figure 2). Consequently, edges become more or less untyped since they refer to incomparable nodes: to parameters (nodes 2 and 3), and to members (nodes 4 and 5). As a negative consequence, one has to traverse all outgoing *ArcSon* edges of a template (including those that lead to parameters) in order to find the members of a template. Also, the Datrix schema "overloads" edge types, e.g., the *Instance* edge does not mean just a direct instance.

Both schemas have room for improvement. Both use flags in some nodes to encode subtyping and these should be changed to use real subtyping. For instance, Datrix has just an abstract class for operators and the exact operator subtype is encoded as an additional attribute *op*. Similarly, in the Columbus schema, references and pointers are both of the same class *TypeFormPtr* and are distinguished by a flag *kind*.

Finally, neither schema includes typing information for subexpressions. Type inference in the presence of overloaded operators and functions is a difficult task. If we want to take advantage of a front end that gives typing information, the schema needs to have placeholders to capture this information. It would be good if implicit type conversions were represented explicitly. Otherwise the resulting AST would contain type errors.

### 4.2 Other Approaches for Exchange

Given all of the difficulties and problems that we have identified in this paper, one question that arises is: Are we using the right approach? Are the complications a consequence of the solution we have chosen or the basic problem itself? We can attempt to answer this question by examining other approaches to sharing data about source code.

Rather than using XML to encode extracted data as a graph (as GXL does), another approach is to encode the AST itself using the structure of XML documents. The Harmonia framework for building CASE tools [7] and cppML developed at University of Waterloo [19] both use this approach. While Harmonia adds tags to source code as metadata to achieve this structure, cppML only uses tags and records the additional information as attributes on the tags. Although this approach is more consistent with how XML tools handle documents, i.e. translating nested tags into a hierarchy, the problem they are solving is not inherently simpler.

Another approach is to provide an API (application programming interface) to an in-memory representation of the parsed source code. IBM's VisualAge C++ compiler provides an API to its internal data structures [17]. As mentioned earlier, the Ada community has a standard API, called ASIS, to allow other tools to use the intermediate representations inside compilers [16]. It should be noted that ASIS follows a previous unsuccessful effort (DIANA) to promote tool interoperability by standardizing ASTs for internal representation.

A major difficulty of attempting to standardize a schema for C++ ASTs is to bring tools into compliance. This significant effort of modifying tools to produce and use the new information format may not be acceptable for tool builders and vendors. One possible solution to solve this problem would be to develop a framework for modular schemas with well-defined interfaces between modules. Each module would contain the subset of the schema relating to a particular problem, for example, templates, projects, or type handling. The framework would allow people to pick and choose among different predefined sub-schema "plug-ins" and should be extensible so that the users may define their own sub-schema solutions. The framework may include predefined transformation algorithms that map between different schemas, similar to ones discussed by Bowman, Godfrey, and Holt in Sections 4.2-3 of their paper [8].

Consider, for example, the type representation portions of the Datrix and Columbus schemas. These are quite separate from the other parts of the schema and could easily be two alternative type representation plug-ins. It would also be possible to define a general sub-schema, which covers several possible solutions. However, such generalization may not always be possible when solutions are quite distinct, as is the case with the template handling in Datrix and Columbus schemas.

## 5 Conclusion

We plan to continue our work on creating a standard schema for C++ at the AST level. In doing so, we hope to advance the state of tool interoperability in the reverse engineering and reengineering community through the use of an SEF, GXL. Whether or not a standard schema proves to be the final answer for exchanging low-level software data, we believe the lessons learnt and the schema itself will be valuable in future efforts. Indeed, we have learned a lot about our own schemas and ways they can be improved.

We invite others to participate in this effort to define a standard schema. In particular, we are looking for designers of analysis tools to test and critique our schemas as we refine them. Anyone interested in this problem should contact one of the authors directly or join the mailing list (instructions are available at:

*http://rgai.inf.u-szeged.hu/mailman/listinfo/gxl-cpp*).

## References


[1] "DATRIX – Abstract semantic graph reference manual, version 1.2," Bell Canada Inc., Montréal, Canada, January 14, 2000.

[2] "The Datrix Home Page," http://www.iro.umontreal.ca/labs/gelo/datrix, last accessed April 30, 2001.

[3] "The GXL Homepage," http://www.gupro.de/GXL, last accessed April 16, 2001.

[4] Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lam, David L. Moore, Brian R. Murphy and Constantine Sapuntzakis, "The Basic SUIF Programming Guide," Computer Systems Laboratory, Standord University, November 1999.

[5] Ira Baxter, "Semantic Designs, personal communication," 2001.

[6] Á. Beszédes, R. Ferenc, F. Magyar and T. Gyimóthy, "Columbus Setup and User's Guide," Nokia Research Center, 1998-2000.

[7] Marat Boshernitsan and Susan L. Graham, "Designing an XML-based Exchange Format for Harmonia," presented at Working Conference on Reverse Engineering, Brisbane, Queensland, Australia, pp. 287-289, November 23-25, 2000.

[8] Ivan T. Bowman, Michael W. Godfrey and Ric Holt, "Connecting Architecture Reconstruction Frameworks," Journal of Information and Software Technology, vol. 42, no. 2, pp. 93-104, 1999.

[9] Thomas Dean, Andrew J. Malton, Ric Holt. "Union Schemas as a Basis for a C++ Extractor," Proceedings of Working Conference on Reverse Engineering, Stuttgart, Germany, October 2-5, 2001.

[10] Rudolf Ferenc. "A short introduction to the Columbus Proposal for a standard C/C++ Schema," http://www.inf.u-szeged.hu/∼ferenc/research/ColumbusSchemaShort.pdf, last accessed April 17, 2001.

[11] R. Ferenc, F. Magyar, Á. Beszédes, Á. Kiss and M. Tarkiainen, "Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems," presented at Symposium on Programming Languages and Software Tools, Szeged, Hungary, pp. 16-27, June 15-16, 2001.

[12] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, "Design Patterns. Elements of Reusable Object-Oriented Software". Reading, Massachusetts: Addison-Wesley, 1995.

[13] Ric Holt, Andreas Winter and Andy Schürr, "GXL: Towards a Standard Exchange Format," presented at Working Conference on Reverse Engineering, Brisbane, Queensland, Australia, pp. 162-171, November 23-25, 2000.

[14] R. C. Holt, "An Introduction to TA: The Tuple-Attribute Language," University of Toronto, Toronto, Draft Mar 24 1997.

[15] International Standards Organization, "Programming languages – C++". ISO/IEC 14882:1998(E)., 1998.

[16] International Standards Organization, "Ada Semantic Interface Specification", ISO/IEC 15291:1999, 1999.

[17] Michael Karasick, "The Architecture of Montana: An Open and Extensible Programming Environment with an Incremental C++ Compiler," presented at Interational Symposium on Foundations of Software Engineering, Orlando, USA, pp. 131-142, November 1-5, 1998.

[18] Rainer Koschke, Jean-Francois Girard and Martin Würthner, "An Intermediate Representation for Integrating Reverse Engineering Analyses," presented at Working Conference on Reverse Engineering, Honolulu, HI, October 12-14, 1998.

[19] E. Mamas and K. Kontogiannis, "Towards Portable Source Code Representations Using XML," presented at Working Conference on Reverse Engineering, Brisbane, Queensland, Australia, pp. 172-182, November 23-25, 2000.

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

[21] Claudio Riva, Michael Przybilski and Kai Koskimies., "Environment for Software Assessment," presented at Workshop on Object-Oriented Architectural Evolution, ECOOP'99, Lisbon, Portugal, 1999.

[22] Georg Sander. "VCG Overview," http://rw4.cs.uni-sb.de/∼sander/html/gsvcg1.html, last accessed April 30, 2001.

[23] Arie van Deursen, CWI, personal communication, 2001.