dxlinker: Generating Real C++ ASGs

Daqing Hou, H. James Hoover, and Piotr Rudnicki
Department of Computing Science
University of Alberta
Edmonton, AB Canada T6G 2E8
{daqing, hoover, piotr}@cs.ualberta.ca

ABSTRACT
Different program analysis tasks need different information from the source code. For some applications, an AST (Abstract Syntax Tree) representation of the source may be sufficient. However, we want to be able to specify assertions on the structure of C++ programs directly in terms of semantic entities and their relationships, rather than syntactical ones such as the names of source files, thus ASGs (Abstract Semantics Graph) rather than ASTs are needed. This paper documents the design of a program called *dxlinker* that can generate ASGs. *dxlinker* is based on Bell Canada's Datrix schema and the tool *dxparscpp*. It has been used to analyze both MFC (Microsoft Foundation Classes) source code and C++ programs that use STL (Standard Template Library). Difficulty with *dxparscpp* are summarized.

Keywords
AST, ASG, C++, type analysis

1 Introduction

Many software analysis tasks involve analyzing source programs. Typically, the analyses involve parsing the code and extracting facts from it. A parser is usually used to generate an AST (Abstract Syntax Tree) from which many facts such as "*class X has method m*" can be extracted.

An ASG (Abstract Semantics Graph) [16, 2], compared with an AST, embodies more semantics. Examples include relations between the use of a variable and its definition, a function call and the definition of the function, the use of a type name such as the target type in a type cast, and its definition, and so on.

For our purpose, we wanted to be able to perform whole–program queries of the source programs, checking assertions about their structures [9]. To achieve that, we need an intermediate representation of the source that is more suitable for the task than the source itself. Furthermore, because the kinds of queries that we want to perform are still evolving, ideally we would also like the model to retain from the source as much information as possible. For example, we do not want to prematurely exclude expressions from the model; in fact, it has turned out that they are often needed when we specify constraints.

Bell Canada's Datrix model [2] and the associated C++ tool, *dxparscpp*, are adopted. The tool *dxparscpp* works on compilation units. It processes one compilation unit at a time and generates a textual representation for it, which is based on the Datrix model.

```
int v;

class X {
public:
       void foo() { bar();};
       void bar() { v++;};
private:
       int v;
};
```

Figure 1: Sample C++ Source: ex1.cpp

The C++ code of Figure 1 will be used as an example to introduce the Datrix model. *dxparscpp* is used to parse the code and a textual model is generated. Due to the size of the generated model (32 nodes and 37 edges), only an abbreviated version is provided, as visualized in Figure 2.

As shown in Figure 2, the output of *dxparscpp* is not a pure AST, since some leaf nodes, specifically those representing types, have outgoing edges that lead back to higher level nodes of the tree. Type information for the global variable *v* and the data member *v* of the class *X*, both of which are *int*, are presented. The figure also shows the return types of the methods *foo* and *bar*, both of which are *void*.

dxparscpp, however, can only generate *partial* ASGs. Although the tool provides type information for both the definitions of variables and the return type of functions, it does not resolve any function calls and variable uses. For example, in Figure 2, the invocation of *bar* by *foo* is only represented as a "NameRef" node, so is the use of the data member *v* in *bar*.

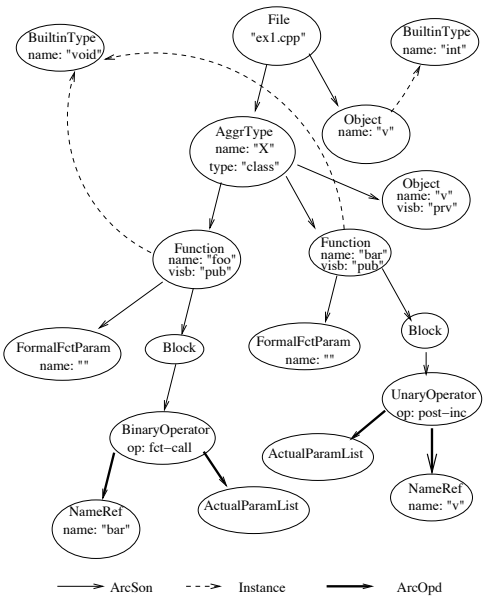


Figure 2: *dxparscpp*–generated ASG for ex1.cpp

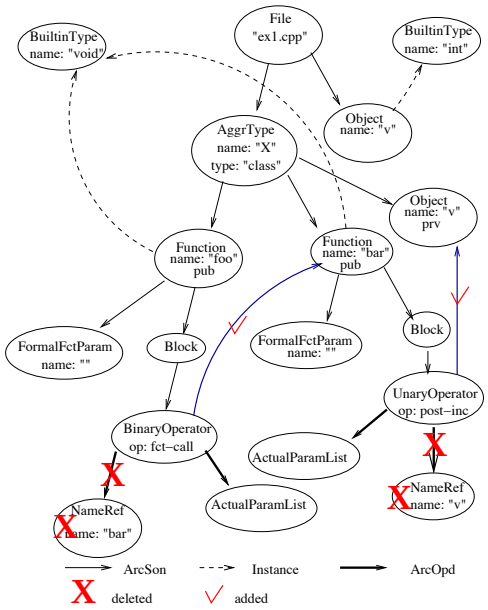


Figure 3: Desired ASG for ex1.cpp

ASTs, or the partial ASGs like what *dxparscpp* produces, can make certain types of analyses inconvenient, if not impossible. For example, if we ask from Figure 2 who uses the *global* variable *v*, based only on the ASG, a tool would have to do name resolution first so as not to confuse with the data member *X::v*. But doing queries in such an ad hoc way is not deemed to be good.

In contrast, we want an ASG like that of Figure 3. The difference is that the two "NameRef" nodes, one for the call to *b*ar and the other for the reference to the data member *v*, are now resolved: the "NameRef" nodes are deleted and the edges that point to them previously are redirected to their definitions. Given such an ASG, the above question can be immediately answered by merely examining all the incoming edges of the global variable *v*.

Our goal is to create a linkage program, *dxlinker*, that takes a set of *dxparscpp*–generated ASGs as input, and converts them into one single system–wise ASG.

Overview of Paper

The rest of this paper is structured as follows: Section 2 presents an overview of the Datrix model. Section 3 introduces the types of problems that our tool has to deal with. Section 4 describes the design of *dxlinker*. Section 5 briefly summarizes some problems with the *dxparscpp* tool. Section 6 describes some related work. Section 7 concludes the paper.

2 Overview of the Datrix Model

The Datrix model is based on TA (Tuple–Attribute language) [7]. TA allows recording of information about certain types of graphs. The information includes (1) nodes and edges in the graph, and (2) attributes of nodes and edges. Program entities such as files, namespaces, classes, functions, statements, expressions, templates and so on, are represented as nodes. The relations between these entities are captured by edges that connect the nodes.

Both nodes and edges can have attributes. For example, a class member can have a visibility attribute, and a global variable can have the "external" attribute, or a storage attribute like "register" or "volatile". For another example, an inheritance relation can have both "virtual" and/or visibility attributes, the ownership relation between a block and its statements can have the attribute "order" to tell the order of the statements within the block.

The Datrix model is designed for C/C++ style of programming languages. It has been used to represent C, C++, and Java programs. Overall, the model is a union of the features of the languages. For example, it includes nodes for templates, which is only available for C++ but not Java and C, and similarly the synchronization mechanisms for Java, which are not available elsewhere.

Nodes and Edges

At the system level, a C++ program consists of a set of compilation units. *dxparscpp* generates a partial ASG for each compilation unit. The ASGs are independent

of each other.

The root of each ASG is an artificial "cSystem" node that is meant to represent the whole system. The only child of the system node is the global scope, which is represented by a "cScopeGlb" node. The compilation unit itself is represented as a "cScopeCompil" node, which is currently treated as a child of the global scope. A compilation unit may contain more than one file through header file inclusion. Each file is represented as a "cScopeFile" node. The entities within each file are treated as its child nodes.

Data types are represented by different types of nodes. The builtin types include signed char, char, long double, double, float, unsigned long long, unsigned long, unsigned int, unsigned short, long long, long int, int, short, and void. Other types include aggregate types, enumeration, array, pointer type, reference type, function pointer, template type, template parameter type, template generated type, forward type, and alias type. Builtin types are children of the global namespace. Currently, pointer type, reference type, forward type and function pointer type are treated as children of the global namespace, which is inappropriate, see section 5 for more discussion.

Function bodies are represented by "cBlock" nodes. A "cBlock" node may have statement nodes and expression nodes as its children. There are also various types of nodes designed to represent statements and expressions. See the Datrix reference manual [2] for complete reference.

Currently, *dxparscpp* does not resolve any name references, thus "cNameRef" nodes are used to represent name references.

Edges are also typed. For example, "cArcSon" edges are used to represent all scope relations, such as those between a class and its members, a function and its parameters and body. "cArcOpd" edges represent the relation between an expression and its operands.

There are also several other types of edges representing relations such as inheritance, friendship, and that between a non-inline method definition and its class. See the Datrix reference manual [2] for other types of edges.

3 Requirements for *dxlinker*

Given a set of Datrix ASGs, *dxlinker* should (1) merge them into one ASG, and (2) resolve all the referenced names. In the output ASG, information about each program entity should appear once and only once, which normally should be its definition. *dxlinker*, however, should be able to handle incomplete input too. For example, if the ASG that defines a variable is not provided, as in the case of libraries, the declaration may be reserved in the output.

Merging Multiple ASGs

Due to the way that the C++ programming language organizes source code and its compilation model, viewed from the whole program standpoint, information about one program entity can redundantly appear within and/or across multiple compilation units. One example is the separation of function declarations and definitions; a function may have multiple declarations appearing in multiple compilation units but one definition. Another is forward types and the concrete types that they refer to. A third one is external declarations and their corresponding definitions.

Therefore, when merging ASGs, *dxlinker* needs to remove the redundant nodes. At the same time, care must be taken to maintain the correctness of the resulting graph. We call this the *type node reference problem.*

Eleminating Redundant Nodes

The following is the list of situations where one needs to remove nodes and subtrees:

- The subtree rooted at a header file should be deleted if it has been seen in a previous compilation unit.
- The subtree for a namespace should be deleted if it has been seen in a previous compilation unit.
- The subtree for an aggregate type should be removed if its definition has been seen previously.
- A forward type node should be removed once its concrete type is found.
- The declaration of a member function should be removed and replaced by its definition once the definition is found. The definition node should also "inherit" from the declaration node attributes such as visibility and virtuality.
- A variable declaration should be removed and replaced by its definition once the definition is found.

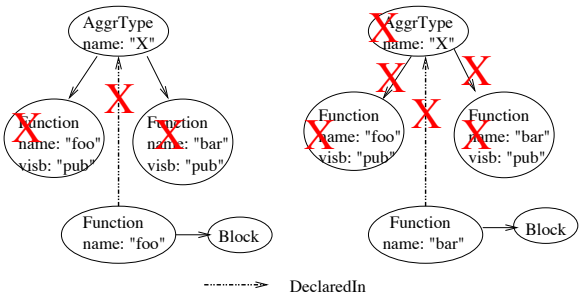
The example in Figure 4 demonstrates the elimination of both the whole subtree for a class and the declarations of member functions. As visualized in Figure 4 (c), the whole subtree of class X in `ex22.cpp` is marked as *deleted*. Also note that all the declarations of member functions are also marked as *deleted*. And their corresponding definitions are connected to the class X (note that member function definitions are connected to their classes by "DeclaredIn" edges). Furthermore, the visibilities of the deleted declaration nodes, which are *public*, should also be transferred to their corresponding definition nodes. Figure 4 (d) shows the desired out-

come.

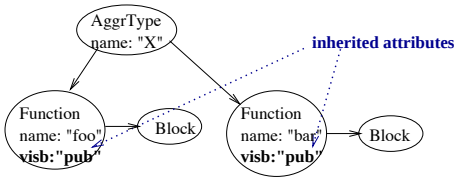
```
class X{          class X{
public:           public:
   void foo();        void foo();
   void bar();        void bar();
};                };

X::foo(){         X::bar(){
.....              .....
}                 }
```

a) ex21.cpp b) ex22.cpp



c) ASGs for ex21.cpp (left) and ex22.cpp (right)



d) the final ASG

Figure 4: Example of Eliminating Redundant Nodes and Subtrees

Type Node Reference Problem

Generally, when removing a type node, one has to make sure that the old edges that point to the node are properly reconnected. Figures 5 and 6 are two examples that illustrate two special cases, one has to do with forward types, and the other with nested types.

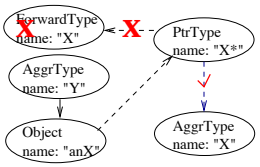
Figure 5 shows one case of the problem caused by forward types. Datrix model assigns each program entity a node regardless whether it is a definition. For example, a forward type name is allocated a "ForwardType" node, which will become redundant and need to be removed once the concrete type is found.

In Figure 5 (b), the node "X*" points to the node "ForwardType X", which is shown by the dashed arrow line between them. Since the forward type node would be of no further use in terms of the semantics of the program, both the node and its associated edges should be removed. At the same time, *dxlinker* must create a new edge from the "X*" node to the "AggrType X" node, to preserve the meaning that the former is a pointer type to the latter.

Figure 6 shows another case of the problem caused by nested types. As shown by Figure 6 (c), when merging the ASGs of `ex41.cpp` and `ex42.cpp`, only one of the two subtrees for the class X should be kept. Particularly, the "AliasType INT" node is deleted, to which the node "b" points previously. *dxlinker* has to ensure that there be an edge from "b" to the reserved "AliasType INT".

```
class X;
class Y{
X * anX;
};

class X{
......
};
```



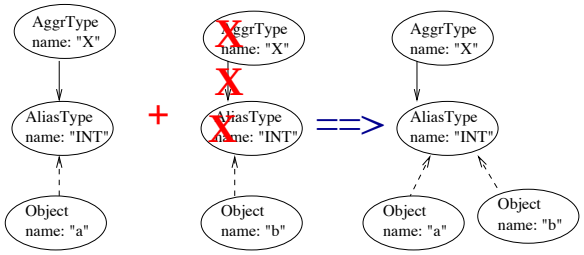
a) ex3.cpp b) the ASG

Figure 5: Type Node Reference Problem: Forward Type

```
class X{              class X{
public:               public:
typedef int INT;      typedef int INT;
};                    };

X::int a;             X::int b;
```

a) ex41.cpp b) ex42.cpp



c) merge ASGs for ex41.cpp and ex42.cpp

Figure 6: Type Node Reference Problem: Nested Type

Name Resolution

In Datrix model, the "NameRef" nodes are used to represent the uses of names. Such nodes can represent the uses of variables and types, the invocation of functions, and so on. Two examples from Figure 2 are the use of the variable *v* and the invocation of the member function *bar*.

A type name can be used not only as the type of variables, but also as the argument to template instantiations, or as the target type of type cast expressions. *dxparscpp* generates "NameRef" nodes for all these uses of type names.

The uses of the keyword *this* are also treated as "NameRef" nodes.

In name resolution, *dxlinker* needs to resolve all uses of

names to their definitions. Essentially this is the standard name resolution problem, with the added complexities of C++ [10, 11, 20, 19].

4 Design of *dxlinker*

dxlinker takes a set of Datrix ASGs produced by *dxparscpp* and merges them into a full ASG. It performs a top down (in the direction along which that the program text goes), depth first traversal of each input ASG, merging its nodes and edges into the output ASG. During the traversal, *dxlinker* adds, deletes, replaces, and/or retrieves entities to/from the corresponding symbol tables.

Therefore, the structure of *dxlinker* can be characterized as the repository architecture style [17]. The ASG is the centralized data, and the traversal algorithms are the computational processes that mutate the data.

dxlinker assumes that all its input compilation units are well-formed according to the semantics of C++ [10]. This assumption is necessary for the correct operation of the tool. For example, the DBU (Declaration Before Use) rule guarantees that the use of a variable will eventually be linked to a declaration in some symbol table. Similarly, ODR (One Definition Rule) can guarantee that once the definition of an entity is inserted into its symbol table, it will not be replaced by any other entities in the future.

A list of features of *dxlinker* are summarized as follows:

- *dxlinker* merges a set of ASGs into one ASG, removing all duplicated nodes and edges.
- *dxlinker* resolves forward types to their concrete types.
- *dxlinker* can resolve the reference to a variable to its declaration. This includes both global and local variables, parameters, data members from both classes and base classes, and data members of template classes.
- *dxlinker* can resolve a function call to its declaration. This includes not only ordinary functions, but overloaded functions, overloaded operators, implicit conversion functions, and template functions.
- *dxlinker* can resolve the reference of a function name in an expression to its declaration.
- *dxlinker* infers the types of all expressions and adds that information into the ASG.
- *dxlinker* supports scopes such as namespaces and nested classes, and related operations such as namespace imports and scoped name references.
- *dxlinker* can handle C++ specifics such as default arguments, initializer expressions, and initialization lists for constructors.
- *dxlinker* fixes some problems of *dxparscpp*. For example, *dxparscpp* uniformly treats all forward types as the sons of the global namespace, which is inappropriate since a forward type defined within a class should be the son of that class instead. Even worse, the global namespace might have a forward type with the same name, which will confuse with the forward type of the class. Similar things happen with both pointer and reference types.

The rest of this section outlines the design of *dxlinker*, with an emphasis on function resolution, since it is probably the most complex task.

The Data Structure For ASG

An ASG is a directed graph that consists of nodes and edges, which are modeled by the classes *Node* and *Edge* respectively.

Class Node and Class Edge

The *GraphElement* class captures the common properties of both nodes and edges: both are typed entities, both have attributes that are name-and-value pairs, and they all can be marked as *deleted* through setting the attribute *useful false*.

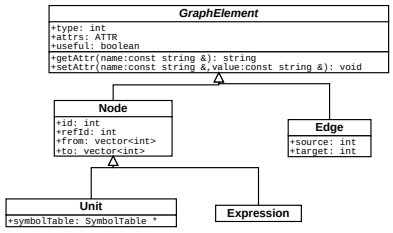


Figure 7: Node and Edge

Both *Node* and *Edge* are subclasses of *GraphElement*. Each node is assigned a unique id to identify itself. Each edge records the ids of its source and target nodes.

If a redundant node represents types, and there are other nodes referencing it, simply setting *useful* to *false* is not enough: then the reference nodes will lose its type information. The *refId* field is added so that the redundant type node can use it to record the id of the node that it is resolved to.

The class *Asg* (Figure 8) implements the ASG. It has two important data members, the map *nodeMap* and the vector *edgeVec*. *edgeVec* stores edges, which can be accessed through their positions in the vector. *nodeMap* is used to store nodes, whose ids are used as the map keys.

To flexibly traverse the ASG, the implementation of the Node class uses a slight variant of the adjacency list

data structure. Each node maintains two sets of indices to *edgeVec*, *from* and *to*; *from* stores the indices for all the edges that leave the current node, and *to* stores the indices for all the incoming ones.

The class *Node* is further specialized into subclasses such as *Unit* and *Expression*. The class *Unit* represents scopes such as classes and functions. In order to do name resolutions, it has a data member *symbolTable* to access its symbol table.

Class Asg
The behavior of the class *Asg* can be divided into three phases: loading ASGs, name resolution, and persistence. It comprises the main body of *dxlinker*.

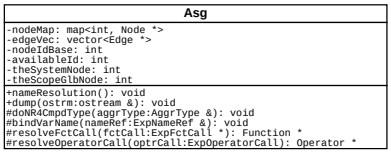


Figure 8: Class Asg

The first thing that the class *Asg* does is to read and merge all the ASGs that belong to the same system into one single final ASG. Two details have to be taken care of here. One is to correctly assign new ids for the nodes of the final ASG, and the other is to correctly merge the duplicated *cSystem* and *cScopeGlb* nodes.

A system may comprise more than one *independent* ASG, each of which corresponds to a compilation unit. This causes the first problem, that is, the sets of ids of the ASGs overlap. In order for the new ASG to maintain the invariant that each node has a unique id, the class *Asg* has to assign new ids to all the nodes of all the ASGs but the first one. This is done together through two fields, *nodeIdBase* and *availableId*. The field *nodeIdBase* records the id for the last node of the last ASG, that is, the largest id so far, and *availableId* records the next available id.

Similarly to the shared ids problem, each compilation unit has a *cSystem* node and a *cScopeGlb* node, but the final ASG only needs one pair of them. The fields *theSystemNode* and *theScopeGlbNode* are used to record the pair. All other pairs are deleted.

Once all ASG files are loaded, name resolution is done through the method *nameResolution*. Based on the type of the current node, this method delegates the task to other appropriate methods such as the protected methods *doNR4CmpdType* (process compound types such as classes), *bindVarName* (resolve variable references), *resolveFctCall*, and *resolveOperatorCall*, and so on.

Finally, the ASG is printed to the standard output using the Datrix format.

Symbol Tables and Symbol Table Stack
C++ supports three primary categories of scopes, namespace scopes, class scopes, and local scopes, and C++ programs are organized into tree-structured hierarchies with scopes as nodes and the global namespace as roots. The outermost namespace scope of a program is called the global namespace scope or global scope. Each class definition introduces a distinct class scope. A local scope can be either the portion of a program that defines a function, a compound statement, or a block. A fourth type of scope, the file scope, is due to the legacy of the C programming language; If a file defines any static variables, then the file forms a file scope.

A scope can contain program entities such as variables, functions, and types, and so on. Each entity of a given scope can be referenced from both within the scope and any of the other scopes that are enclosed within it.

To resolve name references, each scope needs a symbol table to maintain the program entities that are defined within it. The symbol table is essentially a map from identifier names to the corresponding program entities.

Symbol tables can be further divided into subtables, with one for each of variables, functions, types, and namespaces. Each type of scope can only have the subtables appropriate for it. For example, a namespace symbol table can have all four subtables whereas the symbol table of a class cannot have the namespace subtable.

The implementation is straightforward. Both the variable and type subtables are implemented as maps from names to node ids. Giving a name, the corresponding node information can be retrieved from the ASG. The function subtable is a bit complex; Since function names can be overloaded, the table is implemented as a map from a function name to the set of signatures for the overloaded functions.

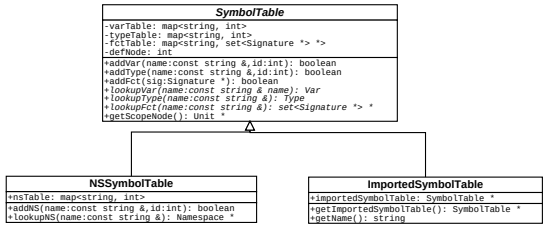


Figure 9: Class SymbolTable

C++ allows to import a namespace and a name, using "using directive" and "using declaration", respectively.

The class *ImportedSymbolTable* can distinguish an imported scope from a regular scope.

Each identifier can have a context. The context of an identifier consists of a sequence of symbol tables whose scopes enclose the identifier. At runtime, a *searchable* symbol table stack is needed to maintain the context information. The class *SymbolTableStack* implements the abstraction. *push*, *pop*, and *top* implements the usual stack operations. *size* and the operator "[]" allow the iteration of the stack. *lastScope*, *globalScope*, and *fileScope* allow access to the respective symbol tables. *numberOfScopes* returns the number of scopes that enclose the current point. *getFileNode* returns the node that represents the current file. *hasTemplate* tells whether the current point is within the body of a template.

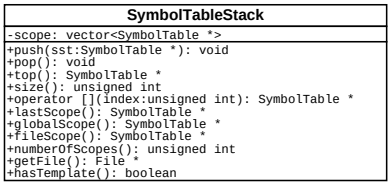


Figure 10: Class SymbolTableStack

Function Call Resolution

Function calls, as shown in Figure 11, are special expressions. Resolving a function call needs 3 steps: (1) collect all the candidate functions, (2) select the viable functions from the set of candidates, (3) decide the best one.

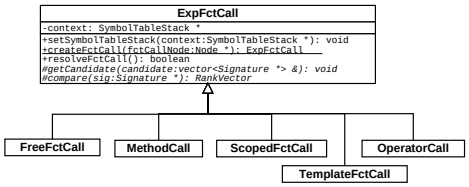


Figure 11: Function call expressions

The method *resolveFctCall* implements the three steps as an abstract algorithm. It invokes two other virtual functions, *getCandidate* and *compare*. As their names suggest, function *getCandidate* collects all the candidate functions for the current function call, and function *compare* evaluates the current function call against a function signature *fctSig* and returns a vector of ranks (see section 4). The one with the best vector of ranks is then chosen as the match.

A function call needs its symbolic context to resolve itself. *setSymbolTableStack* can be used to set up the context.

There are several syntactical forms for function calls, for example, *f(...)*, *ns::f(...)*, *o.m(...)*, and *a op b*, and so on. Different forms require different ways of function resolutions, which are implemented in the corresponding subclasses. *createFctCall* is a static method that constructs objects of the subclasses.

Both function declarations and definitions are modeled with the class *Signature* (Figure 12). Each function may have more than one declaration but at most one definition, and there may exist multiple *Signature* objects for the same function at certain moment. But eventually the ASG will allow one and only one of them to exist. The equality operators and *replaceWith* member function help ensure this: the equality operators decide whether two *Signature* objects are for the same functions, and the *replaceWith* function replaces the current signature with the parameter. Together with its subclasses, the class *Signature* uses "double dispatch" to implement the equality operations.

Each function has a return type and a sequence of parameters. The member functions, *size*, *iTh*, and *getReturnType*, collectively implement an interface to query the type information of the function.

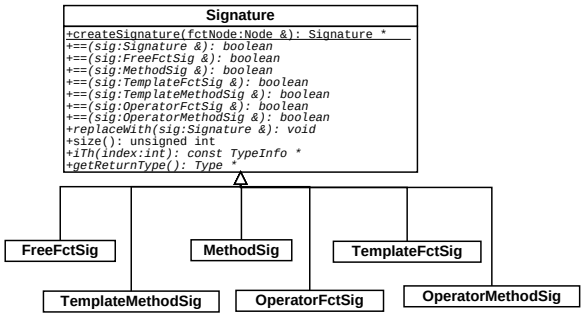


Figure 12: Function Signatures

Ranking Argument–to–Parameter Conversion

Comparing the type of an argument with the type of its corresponding parameter can yield one of the following six results: *exact match*, *promotion*, *standard conversion*, *user-defined conversion*, *ellipsis*, and *no match*. These ranks are implemented as classes shown in Figure 13.

One can compare two ranks, r_1 and r_2, with the operators $==$ and $<$. "$r_1 == r_2$" means that both r_1 and r_2 are at the same rank. "$r_1 < r_2$" means that r_1 ranks better than r_2. Therefore, a *no match* is "greater" than all the other ranks.

Some ranks can be further refined into subranks. For example, the *exact match* category comprises 5 situa-

tions that can be considered as *exact match*, i.e., *exact match*, *lvalue-to-rvalue conversion*, *array to pointer conversion*, *function to pointer conversion*, and *const qualification*, with *const qualification* considered greater than the other four.

Thus even if two rank objects are both *exact match*, one cannot say that they are equal, and further comparison between them is required. Again, this is also implemented as a "double dispatch".

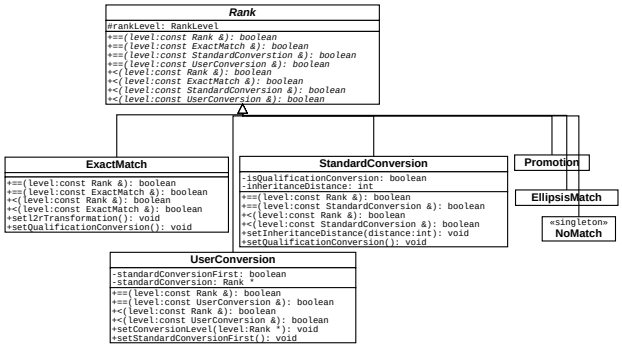


Figure 13: Ranking Arguments and Parameters

Comparing a function call with a function declaration thus yields a sequence of ranks. A comparison is considered a match if there is no *no match* in the sequence. The class *RankVector* of Figure 14 models the result of the comparison. The operation *isMatch* tells whether it is a match. In case of resolving overloaded functions, one function is considered better than another if each rank of the former is no worse than that of the latter. The operator $<$ implements this.

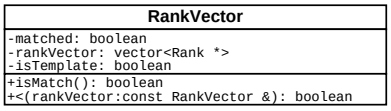


Figure 14: Rank Vectors

Type Information

Type information serves as the basis of comparing arguments types of function calls with the parameter types of functions. The *TypeInfo* class represents the type information for both arguments and parameters (see Figure 15).

The type information for a parameter or an argument includes its base type (that is, the type with all modifiers removed. For example, the base type for "const int &" is "int".), whether it has a *const* modifier, whether it is pointer type, function pointer type, reference type, or ellipsis, and for an argument, whether it is a literal constant, and particularly, if so, whether it is the constant 0.

The method *compare* of the class *TypeInfo* ranks a *TypeInfo* object for an argument against the *TypeInfo* object of its corresponding parameter, according to the rules defined by C++ [10]. It calls *SCS* (Standard Conversion Sequence) and *UCS* (User–Defined Conversion Sequence) to implement this.

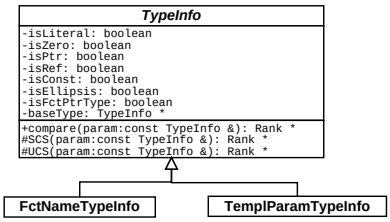


Figure 15: Type Information for Arguments and Parameters

Two special kinds of *TypeInfo* are *FctNameTypeInfo* and *TemplParamTypeInfo*. *FctNameTypeInfo* is for function names that are arguments to function calls, and *TemplParamTypeInfo* is for the type information of template parameters. Both need different ways of ranking: the former will have to compare all the viable functions with the parameter to find an *exact match*, and the latter has to perform template argument deduction.

5 Limitation of *dxparscpp*

Although *dxparscpp* has been well enough for our purpose, we do find some problems with it, which is listed out as follows:

- The *const* modifiers in both types and functions are not captured. This has impact on the accuracy of function resolution.
- *dxparscpp* cannot correctly parse function pointer declarations. For example, in *int (*FUNC)(int, int);*, instead as a function pointer, *FUNC* is recognized as a function with a return type of *int **.
- *dxparscpp* cannot correctly parse conversion functions whose target types are template parameters. For example, in a template with the parameter *TYPE*, the conversion function *operator TYPE *(){...}* is treated as a function with the name *operator TYPE* that returns *int*. This is wrong.
- *dxparscpp* does not distinguish *char* and *unsigned char*. This problem is found when parsing the following code of the class *istream*:

```
# 32 "/usr/include/g++-3/iostream.h" 2 3
    ...
    class istream: virtual public ios{
    ....
    istream & operator>>(char & c);
```

```
istream & operator>>(unsigned char & c){
 return operator>>((char &)c);}
...
}
```

As a result, the two overloaded operators >> are treated as the same, which in fact are different.

- Arrays are treated as pointer types.
- The creation expression "new X(...)" has the type "X" instead of "X*".
- *bool* is not recognized as a builtin type.

Scope–related Problems

- *dxparscpp* cannot properly process scoped names that involve more than one scope. For example, in "Namespace1::Namespace2::Class", both "Namespace1" and "Namespace2" are treated as "ForwardType" nodes, and "Class" as declared in "Namespace2". Conceivably, we should be able to use these information to look up the definition of "Class". However, the fact that "Namespace2" is a nested namespace within "Namespace1" is not captured, which makes it impossible for *dxlinker* to look up the definition of "Class". Similar problem also happens when the referred scopes are template parameters.
- *dxparscpp* only accepts *using* clauses appearing at the global scope. Those at class scopes and local scopes are treated as errors.
- Pointer types that point to types defined in user–defined scopes such as namespaces or classes, are treated as children of the global namespace. This is inappropriate. Rather, they should have been the children of the corresponding user–defined scopes. Similar mistakes happen for reference types and forward types.

Template–related Problems

- *dxparscpp* cannot handle template default arguments.
- *dxparscpp* does not recognize the initialization of the static data members of templates.
- *dxparscpp* cannot correctly process template specialization [20]. This problem was spotted when we tried to link a C++ program that makes use of the standard *string*, which is actually an alias type of a partial specialization of the *basic_string* template.

6 Related Work

The concept of ASG originates from Reprise (*Repr*esentation *i*ncluding *se*mantics) [16], an early work on developing a schema for representing C++ program. Reprise views ASG as AST with additional semantics information from name analyses. Reprise uniformly represents all semantic information as strongly typed expressions, that is, applications of operators to arguments. The simple conceptual model can then be realized in a graph data structure.

Two more recent schemas are Datrix and Columbus [4]. They are intended to serve as general schemas for a broad range of software engineering tasks, and as an exchange medium between toolsets.

Different tasks seem to have different requirements on the program database, in terms of both the kinds of information needed and the precision of the information. The authors of [4] distinguish three levels of abstraction for source information: lexical structure, syntax, and semantics. Transformational tools such as refactorings would need lexical information such as the position of an identifier so that source code can be reproduced in a form close to the original. Syntax concerns the structure of program elements such as types, functions, templates, and so on, in a form more close to the concrete syntax of the language. Examples of semantics are which entity an identifier refers to, which function a function call really uses, and from which template class a generated type is instantiated, and so on. Most work handles syntax better than semantics, and *dxlinker* is an attempt to provide the semantic information that is absent from the output of *dxparscpp*.

CPPX [3] is an open source project that dumps the C++ parse tree into some text formats conforming to the Datrix schema. The key idea is to view the problem of transforming the GCC schema to the Datrix schema as a succession of relatively simple and independent transformations. However, at the moment, it seems that *CPPX* does not link multiple compilation units as *dxlinker* does.

Many tools have been built for reverse engineering tasks, from producing and manipulating architectural views [8, 14], through defining intermediate schemas for connecting architecture reconstruction frameworks [1], to supporting program comprehension [5, 6, 13, 12, 15]. These tasks can all, to a degree, be tolerant to not only certain amount of noises generated by the tools, but also the omission of some information, whereas our work, similar to compilers, requires the absolute correctness and completeness of the information.

7 Concluding Remarks

This paper documents the development of the program *dxlinker* that performs type analysis on top of the Datrix ASGs and links them. *dxlinker* has been used to analyze both MFC programs and C++ programs that use STL.

Difficulty with *dxparscpp* are also summarized.

C++ is a feature–rich language. This makes it rather challenging to process it and to design an appropriate intermediate representation. It takes us quite some effort to build a usable prototype. And generally, this sort of work is deemed to be of "little research value" and "less rewarding" [4, 3]. We disagree with this view. Admittedly, many techniques can be adopted from the theory and practice of compiler construction, but they are not enough; applications in software engineering demand new methods and algorithms. Experience should be solicited and research should be encouraged in this field so that effort can be reused and key problems can be identified, as described in [18].

ACKNOWLEDGEMENTS

We sincerely thank Bell Canada for providing us a free research license for the Datrix toolset.

This work was supported by the Natural Sciences and Engineering Research Council of Canada, and the Alberta Science Research Authority.

REFERENCES

[1] Ivan T. Bowman, Michael W. Godfrey, and Ric Holt. Connecting architecture reconstruction frameworks. *Journal of Information and Software Technology*, 42(2):93–104, 1999.

[2] Bell Canada. Datrix abstract semantic graph: Reference manual, version 1.4. *http://www.casi.polymtl.ca/casibell/datrix/refmanuals/asgmodel-1.4.pdf*, May 1 2000.

[3] Thomas Dean, Andrew Malton, and Richard Holt. Union schemas as the basis for a c++ extractor. In *WCRE 2001: Working Conference on Reverse Engineering*, Stuttgart, Germany, Oct. 2–5 2001.

[4] Rudolf Ferenc, Susan Elliott Sim, and Richard Holt. Towards a standard schema for c/c++. In *WCRE 2001: Working Conference on Reverse Engineering*, Stuttgart, Germany, Oct. 2–5 2001.

[5] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM System Journal*, 36(4):564–593, November 1997.

[6] Richard Holt. Software bookshelf: Overview and construction. *http://swag.uwaterloo.ca/pbs/papers/bsbuild.html.*

[7] Richard Holt. An introduction to ta: the tuple-attribute language. *http://plg.uwaterloo.ca/~holt/papers/ta.html*, 1997.

[8] Richard Holt. Structural manipulation of software architecture using tarski relational algebra. In *WCRE 1998: Working Conference on Reverse Engineering*, Honolulu, Hawaii, USA, Oct. 12–14 1998.

[9] Daqing Hou and H. James Hoover. Towards specifying constraints for object–oriented frameworks. In *CASCON 2001*, Toronto, Canada, November 2001.

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

[11] Stanley B. Lippman and Josee Lajoie. *C++ Primer, Third Edition.* Addison Wesley Longman, Inc., Reading, MA, 1998.

[12] Alberto Mendelzon and Johannes Sametinger. Reverse engineering by visualizing and querying. *Software–Concepts and Tools*, 16:170–182, 1995.

[13] Hausi A. Mueller, O. A. Mehmet, S. .R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem identification. *Software Maintenance and Practice*, 5:181–204, 1993.

[14] Gail C. Murphy and David Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, July 1996.

[15] Santanu Paul and Atul Prakash. A query algebra for program databases. *IEEE Transactions on Software Engineering*, 22(3), March 1996.

[16] D. Rosenblum and A. Wolf. Representing semantically analyzed c++ code with reprise. In *Proceedings of the Third C++ Technical Conference*, pages 119–134, Berkeley, Calif., 1991. USENIX Assoc.

[17] Mary Shaw. Patterns for software architectures. In *First Annual Conference on the Pattern Languages of Programming*, 1994.

[18] Susan Elliot Sim, Steve Easterbrook, and Ric Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of ICSE 2003*, Portland, USA, May 2003.

[19] Bjarne Stroustrup. *The Design and Implementation of C++.* Addison Wesley Longman, Inc., Reading, MA, 1998.

[20] Bjarne Stroustrup. *The C++ Programming Language, Third Edition.* Addison Wesley Longman, Inc., Reading, MA, 1999.