Source-Level Linkage: Adding Semantic Information to C++ Fact-bases

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

ABSTRACT

Facts extracted from source code have been used to support a variety of software engineering activities, ranging from architectural understanding, through detection of design patterns, to program exploration. Several fact extractors have been developed and published in the literature, but most of them extract facts only from individual compilation units. Linking multiple fact-bases is largely overlooked. Source-level linkage is different from compilation linkage. Its goal is to assist a software engineer, not to produce an executable program. Thus a source-level linker needs to collect as many as possible facts that may be potentially helpful to a software engineer's task, many of which are not available from a compiler linker. We present the design of a source-level linker for C++. This linker has been used to analyze a dozen of Microsoft Foundation Classes (MFC) programs and over 200 C++ programs that cover an extensive subset of C++ features, including templates from the Standard Template Library (STL). As a further validation, we design a Structural Constraint Language, SCL, to express and machine-check a wide range of constraints on the Abstract Semantics Graph (ASG) produced by the linker.

0.1. Keywords

source-level linkage, fact extraction, name resolution, type analysis, C++, Datrix

1. Introduction

Facts extracted from source code have been used to recover software architecture [1, 2], detect instances of design patterns [3], identify potential problems at the levels of both design and implementation [4, 5, 6, 7], and assist software engineers in investigating programs [8, 9]. Typically, such analyses exploit not only syntactic facts available in an Abstract Syntax Tree (AST), such as *"class C defines method m,"* but also semantic information in an Abstract Semantics Graph (ASG) [10, 11], for example, the binding of an identifier to a definition (e.g., a variable, a function, or a type).

The kinds of semantic information required and ways of obtaining them vary among tools. For example, architecture recovery and mapping [1, 4] use facts about function calls and the use of variables. More specifically, [1] uses a parser-based extractor while [4] is lexically-based. As another example, Chen et al.'s reachability analysis and dead code detection requires such facts as containment, friendship, inheritance, and instantiation, from a C++ program, but their data model does not supports facts within a procedure body [6], which are often necessary for program exploration, e.g. SCA [8], and flow analyses.

Adding semantic information to a fact-base will not only enable additional software tasks but also improve existing ones. For example, during program exploration, we can search for all the references to an identifier. This is especially useful when searching for references to short names such as operators or copy constructors, for which a tool like *grep* is less effective. The added facts may also be used as a basis for further analyses, such as pointer analysis, and to build up call graphs, both of which can help improve the accuracy of tasks like the detection of design patterns [3].

Much engineering effort has been devoted to the development of fact extractors [6, 11, 12, 13], but the issue of linking multiple separately extracted fact-bases into a global one has been largely overlooked in the literature. There does not exist a detailed discussion on how to systematically perform a *source-level* linkage. While source-level and binary linkage share some fundamental notions like scoping and symbol tables, much source information, for example, the various annotations and local variable names, is not available in a binary linker. Some information, like typedefs and default arguments, are relevant only to a software engineer. They are of no interest to a compiler writer, and thus discarded. The linkage process described in a compiler text like [14] does not take into account C++ features such as the use directives, default arguments, and templates. Thus source-level linkage is worth of investigation from the software engineering perspective.

This paper presents the design and implementation of a source-level linker for C++ (*dxlinker*) and our experience with it. We address two general problems: name resolu-

tion and the removal of redundant facts. The design should be applicable to any fact extractors that support the Tuple Attribute format [15] or its offsprings, and to any languages that adopt the separation compilation model, such as C and C++. Different from the description in [14], which mixes functional and performance concerns, our design is described in terms of high-level abstractions (Section 5). Our implementation is based on the Datrix model and the C++ fact extractor *dxparscpp* [11].

1.1. Paper organization

The rest of this paper is structured as follows. Section 2 summarizes related work. Section 3 presents an overview of the Datrix model. Section 4 introduces the problems that *dxlinker* solves. Section 5 describes its design. Section 6 summarizes our experience with *dxlinker* and *dxparscpp*. Finally, Section 7 concludes the paper.

2. Related work

A literature survey reveals that source-level linkage is only briefly touched on in [6, 12]. The study in [16] indicates that linkage anomalies at the program level can have a negative and noticeable impact on the quality of extracted system models at various levels of abstraction.

Different tasks have different requirements on the quantity and quality of the facts in a fact-base. For example, tools for design recovery and program understanding [1, 4, 8] can tolerate not only certain noises in the fact-base, but also the omission of some facts like expressions and flow information. Lin et al. [17] define four levels of completeness and the notion of relative completeness for fact extractors, and validate *CPPX* by comparing the two binaries generated from the original source and the source recovered from ASG, respectively. Ferenc et al. [18] distinguish three kinds of source information: lexical structure, syntax, and semantics. *dxlinker* is aimed to provide the semantic information required by many program analyses but absent from existing C++ fact extractors.

CPPX is aimed at providing accurate and "complete" facts [13]. It dumps facts of a C++ compilation unit into a fact-base in the Datrix schema. The facts are obtained from the GCC C++ compiler and include some but not all semantic analysis information. In particular, *CPPX* does not perform external linkage [19] yet. Since *CPPX* also supports the Datrix model, it should be possible to port *dxlinker* to link the output of *CPPX*. Finally, *CPPX* cannot handle the non-standard Microsoft Visual C++ features presented in MFC programs.

An important work is to validate the result of fact extraction. Sim et al. advocate the use of benchmarks in software engineering research and report on their experience with a benchmark suite for C++ fact extractors [20]. Lin et al. [17] try to automate the validation of *CPPX*. We rely on manual examination of ASGs generated from small but focused test cases. In this paper we also report on our experience with *dxparscpp*.

The concept of ASG originates from Reprise (*Repr*esentation *i*ncluding *se*mantics) [10], an early schema for representing a C++ program. Reprise views ASG as AST plus additional semantics information from name and type analyses. More recent schemas include Datrix and Columbus [18]. They are intended to serve as general schemas for a broad range of software engineering tasks, and as an exchange medium between toolsets. GXL [21] is an effort towards a generic format for exchanging graph information.

Base.h

```
class Base {
public:
  virtual void m(int x);
private:
  int d;
}
```

Base.cpp

```
#include "Base.h"

void Base::m(int x)
{. . .}
```

Sub.h

```
#include "Base.h"

class Sub: public Base
{
public:
   void m(int x);
}
```

Sub.cpp

```
#include "Sub.h"

void Sub::m(int x)
{
  Base::m(x);
  . . .
}
```

Figure 1. Example C++ source

3. Overview of Datrix model

In this section we give a brief overview of the Datrix model using the code of Figure 1. Base.cpp and Sub.cpp are parsed by *dxparscpp* into two ASGs, shown in Figure 2 and Figure 3, respectively.

The Datrix model is based on the Tuple–Attribute language (TA) [15]. TA is designed to encode graphs such as the structure of large computer programs. Such a graph contains nodes and edges. Program entities such as files, namespaces, classes, functions, statements, expressions, and templates, are represented as nodes. The relationships between these entities are captured by edges.

Both nodes and edges may have attributes. For example, a class member can have a visibility attribute, and a global variable may be "external." For another example, an inheritance relationship can both be "virtual" and have a visibility attribute, and a statement contained by a block has an "order" attribute indicating the position of the statement. For example, in Figure 2, the "ScopeGlb" node has three children: the Base class, the builtin int type, and the member function m, in this order. The order reflects the relative positions of the Base class and the member function m in

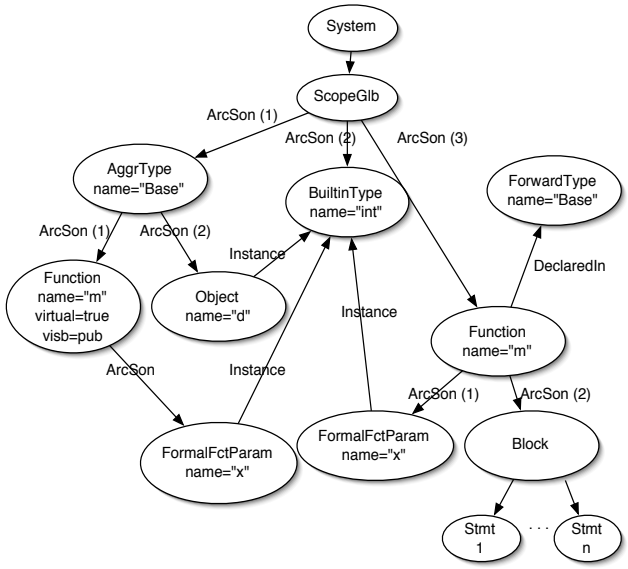


Figure 2. Datrix ASG for Base.cpp

Figure 3. Datrix ASG for Sub.cpp

Base.cpp, and is important for further analysis like linkage to function correctly.

The Datrix model is designed for C/C++ style of programming languages. It has been used to represent C, C++, and Java programs. The model is a union of language features. 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.

3.1. Nodes and edges

The root of each ASG is a "System" node that represents the whole system. (The "System" node in the ASG for a compilation unit is merely a placeholder.) The only child of the system node is a "ScopeGlb" node for the global scope.

Data types are represented by various nodes. Builtin types are children of the global scope, including signed char, char, long double, double, float, unsigned long long, unsigned long, unsigned int, unsigned short, long long, long int, int, short, and void. For clarity our example shows only the "BuiltinType" int. 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.

Function bodies are represented by a "Block" node. Both Figures 2 and 3 contain "Block" nodes. A "Block" node may have statement nodes and expression nodes as its children. Statements and expressions are represented by nodes of their own [11].

Name references are represented by "NameRef" nodes. *dxparscpp* does not resolve name references. For example, Figure 3 contains two name references: one for the function `m` and the other for the variable `x`.

Edges are also typed. An "ArcSon" edge represents the containment relationship, such as that between a class and its members, a function and its parameters and body. "ArcOpd" edges represent the relationship between an expression and its operands.

Several other types of edges are used to represent relationships such as inheritance, friendship, and that between a method definition and the class to which it belongs ("DeclaredIn"; see Figures 2 and 3 for examples).

4. Requirements for *dxlinker*

Given a set of ASGs, *dxlinker* performs two tasks: (1) resolving name references and expression types, and (2) removing redundant facts from the final ASG. In the output ASG, information about each program entity must appear once and only once, which normally should be its definition. *dxlinker*, however, should be able to handle incomplete input as well. For example, if an ASG for the compilation unit that defines a variable is not provided, as in the case of libraries, a declaration may remain in the output.

4.1. Name resolution and type analysis

Datrix uses "NameRef" nodes to represent the use of names, such as variables, the invocation of functions, and types. The use of the variable `x` and the invocation of the member function `m` in Figure 3 are two examples. In particular, *dxparscpp* generates "NameRef" nodes for type names

when they are used as arguments to a template instantiation, or as the target type of a type cast expression. The keyword this is also modeled with a "NameRef" node.

In name resolution, *dxlinker* links all uses of names to their definitions. Essentially this is the standard name resolution, but it is more complex for C++ than most conventional languages [22]. After name resolution, static types for expressions are inferred and added to the final ASG.

4.2. Linking multiple ASGs and removing redundant facts

The separation compilation model and other features of C++ cause redundancy in the extracted facts. For example, there can be multiple declarations for the same function in multiple compilation units, but at most one of them can be a definition. Similarly, both a forward type and the concrete type which it refers to may appear in the fact-base.

Therefore, when linking ASGs, *dxlinker* needs to remove redundant nodes while at the same time maintaining the correctness of the resulting graph. In particular, relationships involving a removed node must be transferred to the one which the removed one is resolved to.

4.2.1. Removing redundant nodes The following are situations where nodes need to be removed:

- A node for a namespace should be removed if the namespace has been seen.
- A node for an aggregate type and its subtree should be removed if the type has been seen.
- 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 as these may be available only in the declaration.
- A forward type node should be removed once its concrete type is found.
- A variable declaration (extern) should be removed and replaced by a definition once the definition is found.

The final ASG of Figure 4 is the result of linking the two ASGs of Figures 2 and 3. It demonstrates the removal of both the whole subtree for a class and the declaration of a member function. Specifically, since the subtree for class Base appears in both ASGs, only one of them may stay in the final graph. Also note that once the definition of Base::m is found, its declaration is removed.

4.2.2. Preserving relationships When one node matches another, one of them is considered redundant, and thus must be removed. But all edges of the removed node must be correctly transferred to the remaining node. Figures 5 and 6 illustrate two special cases, one has to do with forward types, and the other with nested types.

Figure 4. Final factbase

Figure 5 shows one case of preserving relationships where forward types are involved. *dxparscpp* assigns each program entity a node regardless whether it is a definition. For example, a forwarded type is allocated a "ForwardType" node, which is redundant with regard to the concrete type it is actually bound to. In Figure 5 (b), the node "X*" points to the node "ForwardType X", which is shown by a dashed arrow line between them. Since the forward type node would be of no further use as far as the semantics of the program is concerned, both the node and its associated edges can 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 the other case due to nested types. As shown in Figure 6 (c), when merging the two ASGs for ex1.cpp and ex2.cpp, only one of the two subtrees for the class X will remain. Particularly, the "AliasType INT" node is deleted, to which the node "b" points previously. *dxlinker* must ensure that there be an edge from "b" to the remaining "AliasType INT".

5. An Object-Oriented design for *dxlinker*

dxlinker takes as input a set of ASGs produced by *dxparscpp* and links them into a final ASG as output. Specifically, it performs a top-down (in the direction of the program text), depth-first traversal of each input ASG, to generate the output ASG. Initially, the output ASG is empty. When traversing the current ASG, *dxlinker* constructs a

symbol table for each new scope encountered. A program entity may be added, deleted, or retrieved to/from a symbol table.

dxlinker assumes that all of its input compilation units be well-formed according to the semantics of C++ [22]. 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) guarantees that once the definition of an entity is inserted into its symbol table, it will never be replaced by any other entities in the future.

The rest of this section outlines the key abstractions in the design of *dxlinker*. Section 5.1 describes the data structure for ASG, which enables the removal of a redundant node and the preservation of the relationships of a deleted node. Section 5.2 introduces symbol tables and the identifier context. Section 5.3 discusses function resolution. Section 5.4 summarizes *dxlinker* features.

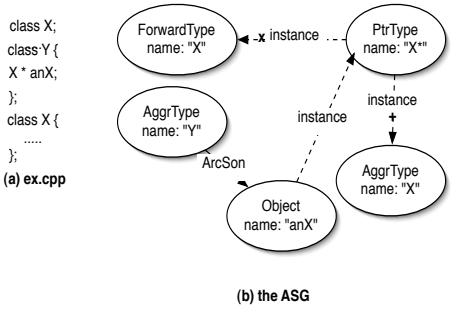


Figure 5. Preserving relationships: forward types

5.1. Data structure and main control

An ASG is a directed graph. Nodes and edges are modeled by the classes Node and Edge, respectively (Figure 7).

5.1.1. Classes Node, Edge, and Asg The GraphElement class captures the commonalities of nodes and edges: both are typed entities, have name-and-value pairs as attributes, and can be marked as deleted through setting the attribute useful to false.

Node and Edge are subclasses of GraphElement. Each node has a unique id. An edge records the ids of its source and target nodes.

If a node is redundant, then its useful field must be set to false to indicate that this node will be removed from the output. However, if this node is also connected to other nodes, in order to preserve the edges connecting to this node, its refId will be used to record the id of the node that this node is resolved to.

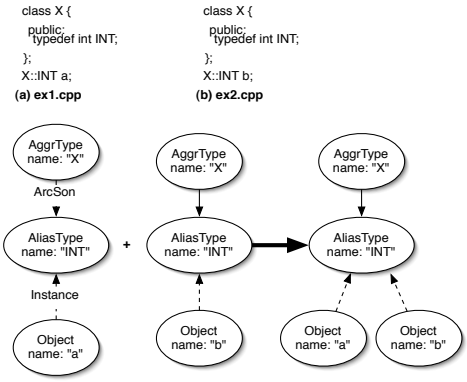


Figure 6. Preserving relationships: nested types

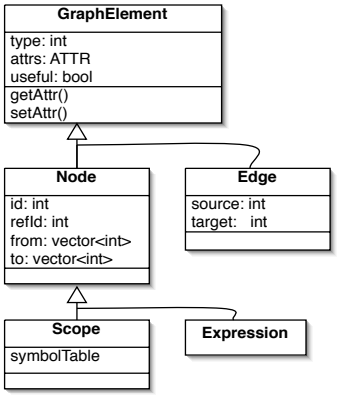


Figure 7. Classes Node and Edge

The class Asg (Figure 8) implements the ASG based on the classes Node and Edge. It has two data members, the map nodeMap and the vector edgeVec. nodeMap maps node ids to nodes, and edgeVec stores edges.

To flexibly traverse the ASG, the Node class maintains two sets of indices to the vector edgeVec, from and to, for outgoing and incoming edges, respectively.

The class Node is further specialized into subclasses such as Scope and Expression. The class Scope represents scopes such as classes and functions. Each scope maintains its own symbolTable for name resolution.

5.1.2. Main control The class Asg contains the main control of *dxlinker*. It works in three phases: loading all ASGs into memory, performing name resolution and type analysis, and streaming the final ASG.

The first task for Asg is to read in all the ASGs from the file system and re-assign ids for all the nodes.

Once all the ASG files are loaded, name resolution is done through the method nameResolution. Based on the

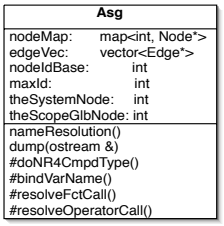


Figure 8. Class Asg

type of the current node, this method delegates the task to an appropriate method such as doNR4CmpdType (processing compound types), bindVarName (resolving variable references), resolveFctCall, and resolveOperatorCall.

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

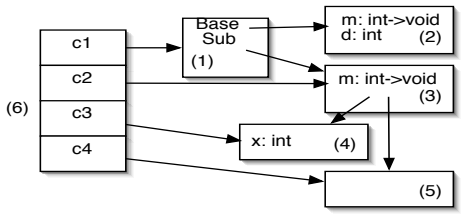
When loading ASGs, two details need attentions. First, the ids used by two input ASGs may overlap. In order for a node to still have a unique id in the final ASG, it must be assigned a new id when merged into the final ASG, which is done with the two fields nodeIdBase and maxId. Specifically, nodeIdBase records the largest id in all the ASGs that have been loaded in, and maxId records the largest id used by the current ASG. Second, each compilation unit has a pair of "System" and "ScopeGlb" nodes, but the final ASG needs only one pair. The fields theSystemNode and theScopeGlbNode keep track of the ids of the pair that will remain.

5.2. Symbol tables and identifier context

C++ supports three primary scopes, that is, namespaces, classes, and local scopes. A scope contains program entities such as variables, functions, and types. Entities defined in a given scope may be referenced from within the scope or other scopes. To resolve name references, each scope maintains a symbol table for the program entities defined within it. The symbol table essentially maps the name of a program entity to other information about it, for example, the signature for a function. A symbol table can be further divided into sub-tables, for variables, functions, types, and namespaces, respectively.

Each identifier has a context. The context of an identifier consists of a sequence of symbol tables, one of which defines the identifier. As an example, Figure 9 depicts the context for Base::m(x) in Figure 1. Boxes (1) to (5) are symbol tables. Box (6) chains them together to form the context. An identifier, like x and Base in Base::m(x), is searched in the symbol table pointed to by c4 first, if not found, then c3, c2, and so on. Note that parameters have a separate symbol table ((4)), and each block also has a symbol table (the empty (5)).

The rich scoping rules of C++ can be supported by customizing this basic data structure. For example, the name



(1) global scope (2), (3) Symbol tables for Base and Sub
(4), (5) parameters and body of Sub::m (6) context of Base::m(x)

Figure 9. Context for Base::m(x) in Figure 1

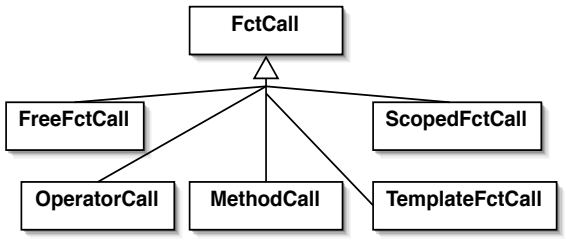


Figure 10. Function calls

Base is bound to the base class of Sub at c2 (if class Base were not the base class of Sub, then the name Base would have been bound at c1). For another example, a "using declaration" like using NS::f will cause the function f from the namespace NS to be added into the current symbol table. Finally, normally only functions within the same scope may overload each other. An exception is due to namespaces as functions from two namespaces can overload each other. This feature can be supported by adding these functions into the current symbol table.

5.3. Function call resolution

Once an identifier is bound to a definition, its static type is known. The type of an identifier can then be used to deduce the type of the expression of which the identifier is an operand. Function calls are a special kind of expressions. In this section, we focus on resolving function calls.

There are three steps involved in resolving a function call: (1) collect all the candidate functions from a scope, (2) select the viable functions from the candidates, and (3) decide the best one.

5.3.1. Collecting candidates This is done by looking into the symbol tables for functions that share the same name with the function call. As shown in Figure 10, there are several syntactical forms for function calls, each requiring a different way of looking up. For example, S::f() (ScopedFctCall) is looked up by first resolving S, which may be either a class or a namespace, and then searching for f inside S. Note that if S is a class, then the resolution of S::f

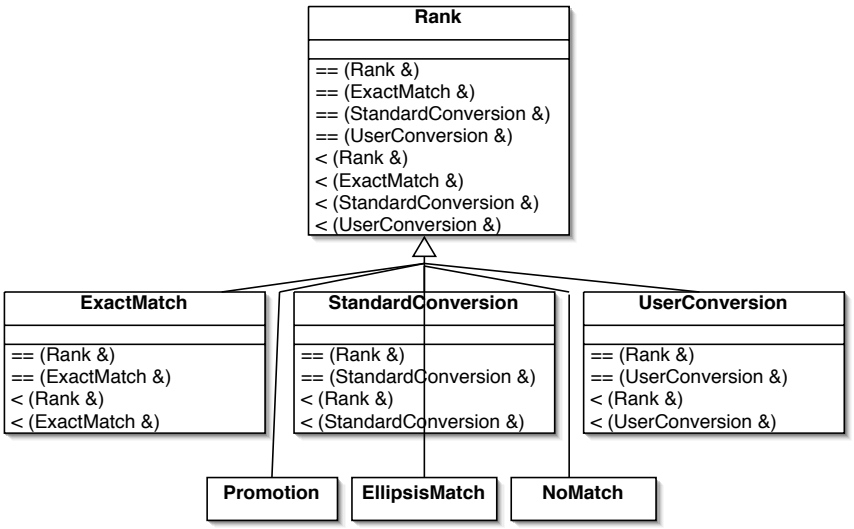


Figure 11. Ranking arguments and parameters

may involve searching transitively the base classes of S in a breadth-first order. On the other hand, o.m() (MethodCall) is resolved by first resolving the static type of the receiver object o. Finally, the resolution of a op b (OperatorCall) may involve both class-level and top-level operators .

5.3.2. Selecting viable candidates The signature of a function comprises its name and the sequence of parameter types. A parameter type is compared with the argument type of the function call, resulting in a rank. If all the ranks are pre-defined ones (Figure 11), then the function is considered as *viable*. The type information for arguments and parameters is represented by the class TypeInfo (Figure 12).

5.3.3. Determining the best match Given two viable functions a and b, if one rank of a is better than the corresponding rank of b, and all other ranks of a are no worse than b, then a is considered better than b.

The type information for a parameter or argument includes the base type, whether it has a const modifier, whether it is pointer type, function pointer type, reference type, or ellipsis. The base type of a type removes all modifiers from the type. For example, the base type for const int & is int. For an argument type, it also includes whether it is a literal constant, and if so, whether it is the constant 0. Two special kinds of TypeInfo are FctNameTypeInfo and TemplParamTypeInfo. FctNameTypeInfo is for function names that are passed as arguments to function calls, and TemplParamTypeInfo is for the type information of template parameters. Both require different ways of ranking: the former requires comparing all the viable functions with the parameter to find an *exact match*, and the latter requires template argument deduction.

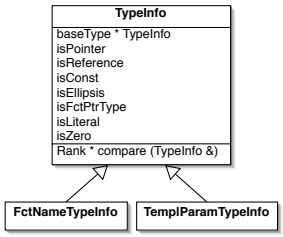


Figure 12. Type information for arguments and parameters

Comparing an argument type with a corresponding parameter type may yield one of the six results: *exact match*, *promotion*, *standard conversion*, *user-defined conversion*, *ellipsis*, and *no match*. These ranks are implemented as classes shown in Figure 11. 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 sub-ranks. For example, the *exact match* category comprises 5 situations 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*, further comparison is required.

5.4. Summary of features

Features of *dxlinker* are summarized as follows:

- *dxlinker* merges a set of ASGs into one ASG, removing all redundant nodes and edges.
- *dxlinker* resolves forward types to their concrete types.
- *dxlinker* resolves the reference to a variable to its declaration. This includes references to both global and local variables, parameters, data members from both classes and base classes, and data members of template classes.
- *dxlinker* resolves a function call to its declaration. This includes not only ordinary functions, but overloaded functions, overloaded operators, implicit conversion functions, and template functions.
- *dxlinker* resolves the reference to a function name to its declaration.
- *dxlinker* infers the types of all expressions and adds that information into the final ASG by adding a new *Instance* edge between an expression and its type node. If the type is not in the factbase, one is created in the appropriate scope.
- *dxlinker* supports scopes such as namespaces and nested classes, and related operations such as namespace imports and scoped name references.
- *dxlinker* handles C++ specifics such as typedefs, default arguments, initializer expressions, and initialization lists for constructors.

6. Experience

dxlinker has been used to analyze a dozen of MFC programs and regularly tested against over 200 test cases during development and maintenance. Some of the test cases were obtained from the Datrix manual [11] and books such as [23], and others were written by us to test specific C++ features. These test cases cover a substantive set of C++ features, including namespaces and templates, class definition and inheritance, struct and union, enum, function overloading and resolution, operator resolution, name importing, default arguments, and more. A Perl program is used to automate the testing process. It compares the output from each test case with a previously approved result and outputs a warning whenever there is an inconsistency between the two. The test cases are run regularly and very useful in catching regression errors.

6.1. Characteristics of test cases

Table 1 shows the number of test cases per C++ feature used to test *dxlinker*. Test cases tend to be focused on particular features, and thus short, and most of them are less than 20 lines. The following is a sample test case for function resolution:

```
#include <iostream>

void F(int a, char b) { cerr<<"Fii\n";}
void F(char a, char b) { cerr<<"Fcc\n";}

void main()
{
        F((short)'a', 'y'); // Matches Fii
}
```

C++ features	number of tests
Function resolution	22
Templates	17
Namespaces	13
STL	12
Pointers	9
Function pointers	5
Const	5
Array init, default args, construction, aliases	4 each

Table 1. Number of test cases per C++ feature

Files	#lines	preprocessed	ASG
ComboBox.cpp	74	395 257	935
ComboBoxDlg.cpp	197	395 380	939
IDCombo.cpp	189	395 243	938
StdAfx.cpp	8	395 009	932
total (KB)	468	1 186 275	3 744
linked ASG (KB)			1 178

Table 2. Space data for one MFC program

6.2. MFC test cases

Some MFC header files contain a vast amount of information, and use a rich set of language features. Correctly handling MFC programs gives us some confidence on the correctness of our tool. The sizes of header files also affect performance. After preprocessing, the headers generate nearly 400 KLOCs, which are then included in many compilation units. Table 2 shows a set of data from one MFC program. We can see that the linked ASG saves 68% of the total space used by the 4 compilation unit ASGs.

Microsoft Visual C++ deviates from standard C++ in certain aspects. For example, it is less strict than standard C++ when assigning one pointer to a member function to another, which is used in implementing event handlers. Another example is about accessing fields of a base structure, as illustrated by

```
struct  tagVARIANT {
    union {
        struct {
            [4 fields elided]
            union {
                [12 fields elided]
                SAFEARRAY  *parray;
                [26 fields elided]
```

```
                struct {
                    PVOID pvRecord;
                    IRecordInfo  *pRecInfo;
                };
            };
        };
        DECIMAL decVal;
    };
};
```

This code is from an MFC header file. A subclass inherits struct tagVARIANT and accesses the field parray directly. By default this would be an error since tagVARIANT itself does not have a field named parray. We modified *dxlinker* to allow for such accesses.

6.3. Another validation: SCL

The output of *dxlinker* has been used as the foundation of a static analyzer SCL (Structural Constraint Language) [24]. SCL views a program as a graph. The key idea is that many design intent can then be expressed as constraints on the graph and machine-checked. SCL has been used to express and check coding idioms, naming conventions, C++ programming rules [25], and framework rules [7]. These studies have exercised a wide range of C++ facts, so we are reasonably confident in the design of *dxlinker*. In the following, we use an example to briefly introduce SCL.

For certain kinds of classes, good class design recommends maintaining the principle of substitution for derived classes. When a derived class extends the behaviour of its base class, it should do so in a way that preserves base class behaviour. That is, a derived class should act like its base class when used in a base class setting. In addition, derived classes should be as loosely coupled to their ancestors as possible. In a particular context like Figure 1, this principle may be translated into a requirement that *"all overrides of Base::m must contain Base::m."* This requirement can be expressed in SCL as follows:

```
1  for D: subclasses(class("Base")) holds
2  [def m_Base as method(class("Base"), "m");
3   def m_D as method(D, "m")]
4    exists e: exprs(m_D) holds
5    method(e) = m_Base
```

To check rules like this, *dxlinker* adds to the graph the set of subclasses of Base so that they can be accessed straightforwardly. Such facts are seldom directly available in existing C++ fact extractors.

At evaluation time, the graph is mapped to an object-oriented representation, where source code entities are strongly-typed objects and relationships are implemented by methods on these objects. SCL specifications are evaluated on this representation. The SCL evaluator recursively descends through the structure of the SCL formula. At each quantifier it constructs the finite set that forms the domain of the quantifier. Then it evaluates the sub-formula against each element of the domain. Finally primitive functions are evaluated directly against facts in the model. Figure 13 illustrates the evaluation process.

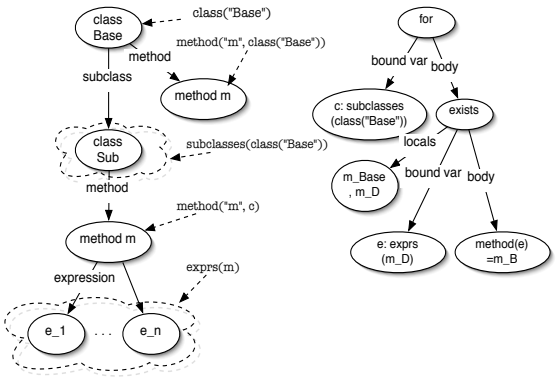


Figure 13. Evaluating SCL constraints. Left: program facts as a graph; dotted areas indicating sets of entities. Right: parse tree for the SCL constraint

6.4. Limitations of *dxparscpp*

We have identified some problems with *dxparscpp*. Some of these problems are fixed by *dxlinker*, and others tend not to be on the critical path of our analysis. In the following, we summarize these problems so that other extractors can avoid them.

- Arrays are treated as pointer types.
- The object creation expression new X(...) has type X instead of X*.
- bool is not recognized as a builtin type.
- The const modifiers in both types and functions are not captured. This has an 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 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, but otherwise distinct operators $>>$ are treated as the same.

7. Conclusion

This paper presents the design and implementation of the program *dxlinker* that performs semantic analysis and external linkage on the Datrix ASGs. *dxlinker* has been used to analyze both MFC programs and C++ programs that use rich C++ features such as STL. Our tool was regularly tested by over 200 test cases during development and maintenance. As a further validation, we build a constraint language SCL and use it to express and machine-check a variety of assertions on the resulting graph of *dxlinker*. Since the Datrix model is adopted by all popular fact extractors, the general design outlined in this paper can be useful to other extractors as well.

dxlinker is available upon requests from the authors.

ACKNOWLEDGMENTS

Bell Canada provided us with a free research license for the Datrix toolset. Kenny Wong introduced us to the Datrix project. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Alberta Sciences and Research Authority (ASRA), and the Alberta Ingenuity Fund (AIF).

References

[1] I. T. Bowman, R. C. Holt, and N. V. Brewster, "Linux as a Case Study: Its Extracted Software Architecture," in *proceedings of ICSE'99*, 1999.

[2] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey, "Structural Redocumentation: A Case Study," *IEEE Software*, vol. 12, no. 1, pp. 46–54, 1995. [Online]. Available: citeseer.ist.psu.edu/article/wong95structural.html

[3] Z. Balanyi and R. Ferenc, "Mining Design Patterns from C++ Source Code," in *proceedings of ICSM 2003*. IEEE Computer Society, Sept. 2003, pp. 305–314.

[4] G. C. Murphy and D. Notkin, "Lightweight Lexical Source Model Extraction," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 3, pp. 262–292, July 1996.

[5] P. T. Devanbu, "GENOA–a Customizable, Front–End Retargetable Source Code Analysis Framework," *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 2, pp. 177–212, April 1999.

[6] Y.-F. Chen, E. R. Gansner, and E. Koutsofios, "A C++ Data Model Supporting Reachability Analysis and Dead Code Detection," *IEEE Transactions on Software Engineering*, vol. 24, no. 9, pp. 682–693, September 1998.

[7] D. Hou, H. J. Hoover, and P. Rudnicki, "Specifying Framework Constraints Using FCL," in *proceedings of CASCON 2004*, Toronto, Canada, October 2004.

[8] S. Paul and A. Prakash, "A Query Algebra for Program Databases," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, pp. 202–217, March 1996.

[9] R. C. Holt. (2002, May 5) Introduction to the Grok Language. [Online]. Available: http://plg.uwaterloo.ca/ holt/papers/grok-intro.html

[10] D. Rosenblum and A. Wolf, "Representing Semantically Analyzed C++ Code with Reprise," in *proceedings of the Third C++ Technical Conference*. Berkeley, CA: USENIX Assoc., 1991, pp. 119–134.

[11] R. C. Holt, A. E. Hassan, B. Lague, S. Lapierre, and C. Leduc, "E/R Schema for the Datrix C/C++/Java Exchange Format," in *proceedings of WCRE 2000*, 2000, pp. 349 – 358.

[12] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy, "Columbus – Reverse Engineering Tool and Schema for C++," in *proceedings of ICSM 2002*. IEEE Computer Society, Oct. 2002, pp. 172–181.

[13] A. J. Malton, T. Dean, and R. C. Holt, "Union Schemas as the Basis for a C++ Extractor," in *proceedings of WCRE 2001*, Stuttgart, Germany, Oct. 2–5 2001, pp. 59 – 67.

[14] A. V. Aho, R. Sethi, and J. D. Ullman, *Compiler Design: Principles, Techniques, and Tools*. Addison–Wesley, 1986.

[15] R. C. Holt. (1997) An Introduction to TA: the Tuple-Attribute Language. [Online]. Available: http://plg.uwaterloo.ca/ holt/papers/ta.html

[16] J. Wu and R. C. Holt, "Resolving Linkage Anomalies in Extracted Software System Models," in *proceedings of IWPC 2004*, 2004, pp. 241–245.

[17] Y. Lin, R. C. Holt, and A. J. Malton, "Completeness of a Fact Extractor," in *proceedings of WCRE 2003*, Victoria, Canada, Nov. 13–16 2003.

[18] R. Ferenc, S. E. Sim, R. C. Holt, R. Koschke, and T. Gyimóthey, "Towards a Standard Schema for C/C++," in *proceedings of WCRE 2001*, Stuttgart, Germany, Oct. 2–5 2001.

[19] A. J. Malton. personal communcation.

[20] S. E. Sim, S. Easterbrook, and R. C. Holt, "Using Benchmarking to Advance Research: A Challenge to Software Engineering," in *proceedings of ICSE 2003*, Portland, USA, May 2003.

[21] R. C. Holt, A. Schurr, S. E. Sim, and A. Winter. Graph eXchange Language (GXL). [Online]. Available: http://www.gupro.de/GXL

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

[23] S. B. Lippman and J. Lajoie, *C++ Primer*. Reading, MA: Addison Wesley, 1998, third Edition.

[24] D. Hou and H. J. Hoover, "Using SCL to Specify and Check Design Intent in Source Code," *IEEE Transactions on Software Engineering*, June 2006.

[25] D. Hou, H. J. Hoover, and P. Rudnicki, "Specifying the Law of Demeter and C++ Programming Guidelines Using FCL," in *proceedings of SCAM 2004*, Chicago, USA, 2004.