An Approach for Modeling the Name Lookup Problem in the C++ Programming Language

James F. Power
National University of Ireland, Maynooth
Computer Science Dept
County Kildare, Ireland
James.Power@may.ie

Brian A. Malloy
Clemson University
Computer Science Dept
Clemson, SC USA
malloy@cs.clemson.edu

ABSTRACT

Formal grammars are well established for specifying the syntax of programming languages. However, the formal specification of programming language semantics has proven more elusive. A recent standard, the Unified Modeling Language (UML), has quickly become established as a common framework for the specification of large scale software applications. In this paper, we describe an approach for using the UML to solve the name lookup problem for the recently standardized C++ programming language. We apply our approach to C++ because a solution to the name lookup problem is required for parser construction and the development of analysis and testing tools.

General Terms

Name lookup, parser, BNF, Unified Modeling Language, UML, class diagram, sequence diagram, object diagram.

1. INTRODUCTION

Formal grammars are well established for specifying the syntax of programming languages. Moreover, extended Backus Naur Form (EBNF) is a popular representation for language syntax; furthermore, many tools have been developed that accept regular expressions and context free grammars as a basis for automating the early stages of compiler construction. However, the formal specification of programming language semantics has proven more elusive. No single formal technique for semantics specification has gained the wide acceptance that formal grammars have attained for syntax specification. Indeed, formal semantics specification has yet to yield convincing solutions for problems associated with modularity, scalability and automatic implementation of semantic analyzers[5].

The Unified Modeling Language (UML) has quickly become established as a common framework for the specification of large scale software applications[2; 9]. The UML is widely accepted by software developers and many UML CASE tools are now available. In this paper, we describe an approach for using the UML to specify aspects of the static semantics of programming languages. In particular, we describe a technique for solving the *name lookup* problem for the recently standardized C++ programming language[3].

The *name lookup problem* is defined as follows: given the use of a name in a program, find the corresponding declaration of that name. We present a structural model that captures syntactic information about *where* a name occurs in a program together with semantic information about *how* the name occurs in the program. We discuss various options for placement of a lookup method that, given a program name, finds the corresponding declaration for that name; we arrive at a placement of the lookup method that properly places responsibility for lookup in the semantic hierarchy of the structural model. We include a *program processor* in the model that assembles an object that contains contextual information about a name encountered in a program. The program processor than passes the newly created object to the semantic hierarchy that performs the lookup in the enclosing scope and searches other relevant scopes if the name is not found.

We apply our approach to C++ because a solution to the name lookup problem is required for parser construction and C++ parsers are difficult to construct[1; 4; 7; 8]. There is no public domain parser currently available that accepts the language described in the C++ standard. Our use of UML to specify the semantics of C++ will provide documentation of the language to UML-knowledgeable developers.

The remainder of this paper is organized as follows. Section 2 presents the options that we considered in our design of the structural model and Section 3 provides object and interaction diagrams that elucidate aspects of the structural model. Moreover, Section 3 presents an example of the name lookup problem together with a solution using our approach; the interested reader may consult reference [6] for a solution to virtually all of the name lookup examples listed in the C++ standard. In Section 4 we draw conclusions.

2. EVOLUTION OF THE MODEL

In this section we describe the design options that we considered in our approach to solving the name lookup problem. Clause three of the recently adopted C++ standard enumerates the procedures for accomplishing name lookup. We present two different approaches to the construction of a class diagram that realizes the lookup in clause three and

SAC 2000 Como, Italy

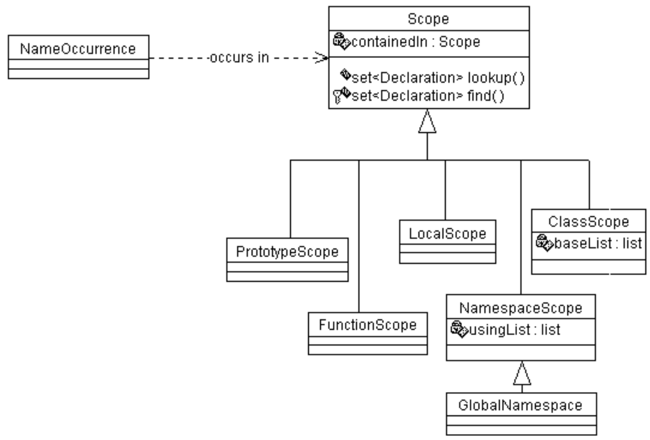

Figure 1: *Class Diagram representing the scope hierarchy.* This figure enumerates the different scoping constructs in the C++ language and the relationships between them. The *lock* symbol indicates a private class member and the *key* symbol indicates a protected class member; all other class members are public.

captures the static semantics of C++ with an explanation of the advantages and disadvantages of each approach. In the next section we overview the important classes in each approach and Section 2.2 follows with a presentation of an approach that places the burden of lookup on the syntax of the language. Finally, Section 2.3 presents an approach that places the burden of lookup on the semantics of the language.

2.1 Overview of the Approach

The class diagram in Figure 1 illustrates the important classes in the structural model that we use to solve the name lookup problem, and the associations between the classes. The class on the left side of the Figure, NameOccurrence, represents the physical occurrence of a name in the program text, where the occurrence can be a declaration, definition or any usage of a name. Each such occurrence of a name occurs in the context of a logically enclosing scope, which is represented by the hierarchy on the right side of the figure. For the name lookup problem, a scope is a list of names declared and therefore available in a logical region of the program. As part of a typical implementation of a program parser, these scopes are usually represented by a *symbol table*. Further consideration of the representation of a name in a symbol table is not important for the problem that we consider here. We include two methods, find, a private method that simply searches through those names defined in the current scope and lookup that drives the overall search procedure in the context of the current scope, and provides a public interface to find. Since every scope other than global scope may be logically nested within an enclosing scope, we provide a containedIn attribute to capture this relationship; the containedIn attribute is shown as a private data member of class Scope in Figure 1.

The C++ standard distinguishes five kinds of scope that we choose to represent through subtyping. Thus, base class Scope in Figure 1 has five subclasses, LocalScope, ClassScope, NamespaceScope, FunctionScope and PrototypeScope. The first three of these classes correspond to local scope in a block of the program, class scope and namespace scope. Both ClassScope and NamespaceScope have specialized mechanisms for referring to other scopes that we represent using attributes: a list of base classes for ClassScope and a list of imported namespaces in namespace scope represented by baseList and usingList respectively. Since every C++ program contains an implicit namespace for global attributes, we represent this implicit namespace with the singleton class GlobalNamespace derived from NamespaceScope. Finally, classes PrototypeScope and FunctionScope, also derived from Scope, represent two less familiar scope levels. Function prototype scope refers to the fact that parameters in a function declaration are in scope for the duration of that declaration. Function scope refers specifically to the use of label declarations and goto statements within a function.

In summary, the NameOccurrence class captures local information about *where* a name occurs in the program; this class represents a syntactic view of the program that incorporates contextual information into the model. The Scope hierarchy captures global information about *how* a name occurs in the program; this hierarchy represents a semantic view of the program that incorporates information about the organization of the scopes.

Since the goal of name lookup is to associate a name in the program with its corresponding declaration, an important consideration is where in the model is the burden of lookup placed. In the sections that follow, we explore two options for the placement of lookup.

2.2 A Context-Centered Approach to Lookup

Our first approach to modeling name lookup adopts a context-centered view of the problem that places a lookup method in class NameOccurrence. In clause three of the recently adopted C++ standard, the name lookup problem is clearly defined including a detailed enumeration of the procedures for accomplishing lookup. The procedural enumeration in clause three is organized by context where the list of contexts

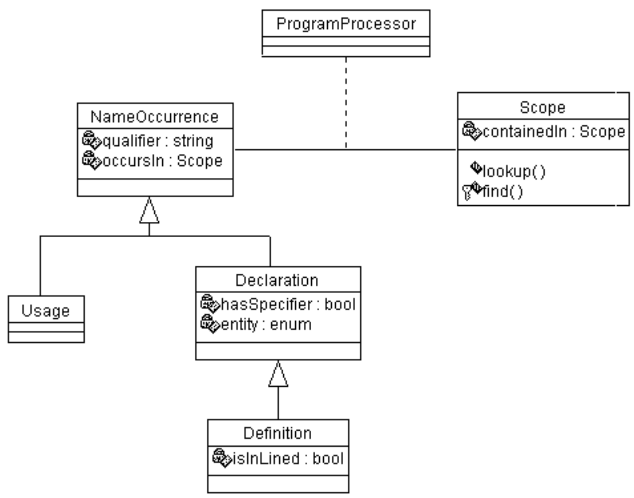

Figure 2: *Class diagram for the semantic-centered approach.* This figure illustrates how the diffusion of control to the `Scope` and `ProgramProcessor` classes allows. The Program-Processor class, illustrated at the top of the figure, establishes the relationship between NameOccurrence and Scope classes by passing the former as a parameter to the lookup method of the latter. Since every Declaration or Usage of a name in a program is a NameOccurrence, we use generalization to expresses this association.

include (1) an unqualified name, (2) an argument-dependent name, (3) a qualified name, (4) a name that includes an elaborated type specifier, (5) a class member name, or (6) a name that's a target of a using directive or namespace alias. We incorporate context into the structural model by subtyping the class NameOccurrence with the classes enumerated in clause three of the C++ standard.

However, there are several problems in the context-centered approach. The first problem is that partitioning control based on the structure of clause three of the C++ standard induces methods that manifest a high degree of coupling between classes. This coupling results from the fact that the single instance of a name occurrence can incorporate features from many other classes. For example a lookup of member name m in `o.Q::m(a)` combines instances of `QualifiedName` based on qualifier `Q`, `ArgDepName` based on argument a and ClassMemberName for the object `o`.

A second problem with this approach is that once a lookup is initiated from a particular object in the NameOccurrence hierarchy, the logic of control proceeds similarly for all such objects since it effectively mirrors the structure of the Scope hierarchy. By restricting the context to initiation of control and permitting the Scope object to complete the search process we avoid the duplication of search logic in each of the NameOccurrence objects and this reorganization permits coalescing and considerable simplification of the NameOccurrence hierarchy.

Furthermore, once the context of an occurrence has been resolved and control passed to the relevant Scope object, the object can either complete the lookup itself or it can nominate a Scope object to continue the search until the lookup is complete. This process of initially determining syntactic context and then passing control to semantic processing closely parallels the actions of a processor based on an attribute-grammar that selects an appropriate grammar rule, determines syntactic context, and then performs the corresponding semantic action.

2.3 A Semantic-Centered Approach to Lookup

Figure 2 illustrates an alternative, semantic-centered approach to solving the name lookup problem. Here, the lookup function that controls the search strategy has been transferred to class Scope. The NameOccurrence class and its subclasses are now information carriers and do not encapsulate any search capabilities of their own. That part of the search strategy concerned with assembling context information is now delegated to a new class ProgramProcessor, which we may regard as an abstraction of the parser.

The class hierarchy represented by NameOccurrence, has been reorganized based around the kind of context information required by the lookup method of the Scope class. We are now in a position to model the categorization of name occurrences in terms of declarations, definitions and uses as described early in clause three of the C++ standard. The distinction between qualified and unqualified names is now represented by an attribute in the NameOccurrence class while more specific information, such as the presence of an elaborated type specifier, is represented as an attribute of the Declaration class.

As the ProgramProcessor encounters a name, since it knows the context of the name from the grammar, it assembles this information in an instance of the NameOccurrence class and then passes this object, through the lookup method, to the enclosing scope. The enclosing scope can now use the context incorporated in the object to make suitable decisions during the lookup process, e.g., ignoring enclosing namespaces during a qualified name lookup.

3. BEHAVIORAL MODELING OF NAME LOOKUP

While the class diagrams of the previous section provide a general framework within which name lookup takes place, we still have not specified the actual rules used in deciding the sequence of scopes that are searched, and which would be encoded in the `lookup` method for these scopes.

As might be expected with such a complex language as C++, there are many cases to consider, and indeed, several examples are presented in the C++ standard. In this section we present two specific cases, namespace lookup and class-based lookup, which capture many of the fundamental decisions involved in the lookup process. For further information, the interested reader may consult reference [6] for solutions to the name lookup problem for virtually all of the examples listed in the C++ standard.

The examples presented below are based on some of those presented in the C++ standard. These examples typically present a series of declarations or definitions, and then some usages, and then describe the name lookup for these usages. We observe here that the standard UML modeling techniques of object diagrams and sequence diagrams fit smoothly onto this pattern, since the former are ideal for describing the environment created by a declaration sequence, and the latter are ideal for describing a particular invocation of the name lookup procedure.

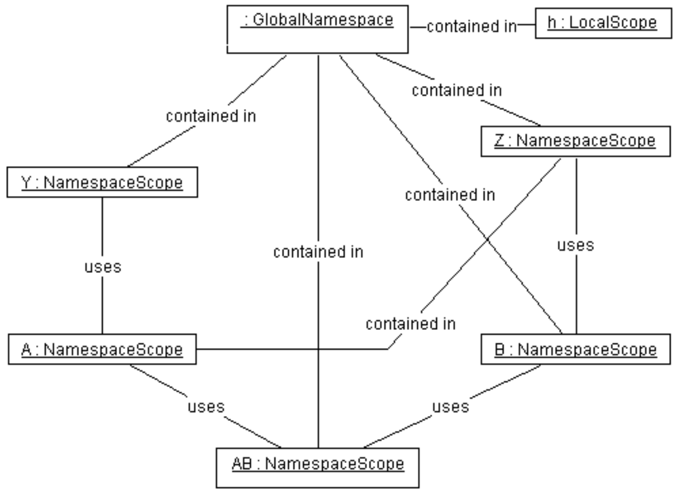

Figure 4: *Object diagram for the namespace example.* This figure represents the two kinds of relationships between namespaces: those established by a using directive and those established by nesting.

3.1 Name Lookup for Namespaces

Namespaces act as a modularization construct in C++, allowing the programmer to partition the names used in a program to prevent them from interfering with each other. Thus, given some variable `x` declared in namespace `A`, once outside namespace `A` we may refer to the variable using explicit qualification, as in `A::x`.

Namespaces may be nested inside each other, in which case name occurrences inside the inner namespace may refer to those already declared at the outer level without the need for qualification with this process continuing recursively, eventually reaching the global namespace where all namespaces are ultimately nested.

In addition to this textual relationship achieved through qualification or nesting, we may establish a logical relationship between namespaces by importing one into another with a `using` directive. The declarations in a namespace that are imported in this way are treated as though they were originally declared in the importing namespace.

In Figure 3 we list a program segment to illustrate name lookup for namespaces. Here we declare five namespaces, with various using relationships and one nesting relationship; we also define a single function `h()`.

Our first step in representing this example is to construct an object diagram, illustrated in Figure 4, showing how each scope instantiates a corresponding subclass of our model's Scope class, illustrated in Figure 1 and discussed in Section 2.1. The object diagram also represents the nesting and import relationships between the namespaces. For example, all namespaces are contained in an implicitly declared global namespace; thus, namespaces Y, AB, B and Z of Figure 3 are implicitly contained in an unnamed global namespace. This nesting relationship is captured in Figure 4 by the unnamed class at the top of the figure that is an instance of GlobalNamespace, and classes Y, AB, B and Z related to the unnamed instance of GlobalNamespace by the contained in connector. Similarly, namespaces A and B of Figure 3 contain named using directives for Y and Z respectively. This import relationship is captured in Figure 4 by the uses connector between instances of Y and A, and between Z and B. Similar uses relationships are shown in Figure 4 between A and AB and between B and AB.

At the end of the program segment illustrated in Figure 3 we have function `h()` at global scope, which contains a usage of the name `f`, with an explicit qualification by the namespace `AB`. To illustrate name lookup for `f`, we present a sequence diagram, illustrated in Figure 5, indicating the scopes that are searched and the order that they are searched. The lookup proceeds as follows:

- Namespace `AB` is first searched unsuccessfully; we must next search its imported namespaces `Z::A` and then `B`.
- The search of namespace `Z::A` yields no declaration, and so we search its imported namespace `Y`. Our search of `Y` is successful.
- Going back to `AB` we are now directed to search namespace `B` where we also find a declaration, thus precluding the search of its imported namespace `Z`.

The search terminates, returning the two possible definitions of `f` to the program-processor, which will then proceed with the subsequent stages of processing such as overload resolution.

Note that we do not consider the namespace `Z` in which `A` is nested in this part of the search, nor do we search global namespace, in accordance with the lookup rules for qualified names. The presence of a qualifier is made known to the Scope's `lookup` method through its parameter, the NameOccurrence instance corresponding to `f`.

In the sequence diagram we are able to explicitly represent the roles and responsibilities of each scope level. For example, it is the namespace `AB` that causes a search of `Z::A` and then `B` (in that order), and which is responsible for combining the results from each of these searches.

4. CONCLUDING REMARKS

```
void f(float);
namespace Y {
  void f(char);
  void h(double);
}
namespace Z {
  void f(float);
  void h(int);
  namespace A {
    using namespace Y;
    int i;
  }
}
namespace B {
  using namespace Z;
  void f(int);
  void g(int);
}
namespace AB {
  using namespace Z::A;
  using namespace B;
  void g();
}
void h() {
  AB::f(0);
}
```

Figure 3: *Namespace example.* This program segment is used to demonstrate name lookup for namespaces. We search a series of nested namespaces for the declaration corresponding to name `f`.

In this paper we have presented an approach for modeling the name lookup problem for the C++ programming language. We have presented a structural model that captures the syntactic and semantic aspects of name occurrence in two separate but related class hierarchies. We have included interaction diagrams that elucidate aspects of the structural model by describing examples of name lookup closely modeled on those found in the C++ standard. We presented two possible views of the structural model corresponding to a context-centered approach and a semantic-centered approach to the solution of the name lookup problem. The semantic-centered approach provides advantages over the context-centered approach that was based on the structure described in the C++ standard.

5. ACKNOWLEDGEMENTS

We would like to thank the anonymous referees who provided guidance in revising this paper. We especially thank the referee who suggested the title of the paper.

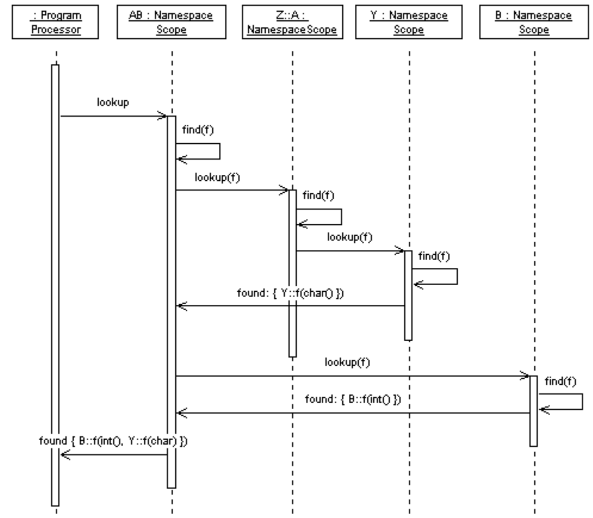

Figure 5: *Sequence diagram for the namespace example.* This figure captures the flow of control as messages are passed through the namespace hierarchy.

6. REFERENCES

[1] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *OON-SKI*, pages 122–136, Oregon, USA, 1994.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, 1999.

[3] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. American National Standards Institute, first edition, September 1998.

[4] J. Lilley. PCCTS-based LL(1) C++ parser: Design and theory of operation. Version 1.5, February 1997.

[5] P. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 11, pages 575–631. Elsevier, 1990.

[6] J. F. Power and B. A. Malloy. Exploiting the UML to Specify the Static Semantics of an Object-Oriented Programming Language. Technical Report Revision 1.0, Clemson University, September 1999.

[7] S. Reiss and T. Davis. Experiences writing object-oriented compiler front ends. Technical report, Brown University, January 1995.

[8] J. Roskind. A YACC-able C++ 2.1 grammar, and the resulting ambiguities. Independent Consultant, Indialantic FL, 1989.

[9] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison Wesley Longman, Inc, 1999.