Support for Maintaining Object-Oriented Programs

Moises Lejter, Scott Meyers, and Steven P. Reiss

Abstract—**In this paper, we explain how inheritance and dynamic binding make object-oriented programs difficult to maintain, and we give a concrete example of the problems that arise. We show that the difficulty lies in the fact that conventional tools are poorly suited for work with object-oriented languages, and we argue that semantics-based tools are essential for effective maintenance of object-oriented programs.**

We then describe a system we have developed for working with C++ programs. This system comprises a relational database system for information about programs and an interactive database interface integrated with a text editor. We describe our system architecture, detail the database relations, provide informal evidence on the system's effectiveness, and compare it to other research with similar goals.

Index Terms— **C++, object-oriented languages and environments, programming environments, semantic analysis of code, software maintenance, software representation in relational databases.**

I. Introduction

OBJECT-ORIENTED languages are growing in popularity. They introduce features not available in conventional programming languages, such as function and operator overloading, inheritance, and dynamic (late) binding. Although these features offer advantages, they also have some drawbacks.

Questions such as "what is the definition of this function?" or "where is the declaration for this symbol?" are asked often enough during the development cycle that special-purpose tools have been created to enhance or supplement editors, easing the process of answering these questions. While these tools have proven their usefulness when used with conventional programming languages, they typically lack enough flexibility to cope with the features mentioned above. For example, using a function name as a key to identify which function the programmer is referring to is a strategy commonly used by software development tools. However, overloading function names renders the strategy ineffective: the name alone is no longer enough to uniquely identify a function.

Manuscript received April 26, 1992; revised August 3, 1992. This work was supported by the NSF under grant DCR 8605567; by DARPA under contract N00014-83-K-0146, ARPA order 6320; by the Digital Equipment Corporation under agreement 393; by AFOSR contract F49620-88-C-0132, ARPA order 6426; by an NSF Presidential Young Investigator Award IRI-8957 601; by the Air Force and DARPA under contract F30602-91-C-0041; and by the NSF in conjunction with DARPA under Contract IRI-8905436. An earlier version of this paper was presented at the 1991 Conference on Software Maintenance [1]. Recommended by V. Rajlich.

The authors are with the Department of Computer Science, Brown University, Providence, RI 02912.

IEEE Log Number 9204091.

To help programmers deal with problems such as these, we have developed the XREF editing module and its underlying relational database XREFDB. XREF is a set of interactive routines integrated into the GNU Emacs text editor [2]. These routines allow programmers to obtain information about programs they are working on through special editor commands. (Equivalent functionality is available through annotedit, the primary text editor in the FIELD programming environment [3], [4]. However, because Emacs is more familiar to most readers than is annotedit, in this paper we focus on the Emacs interface to XREFDB.) XREFDB is a relational database whose relations were specifically designed to support queries about programming languages. The relations it offers are sufficient to capture most features present in commonly used programming languages, including object-oriented languages such as C++.

II. Object-Oriented Software and Conventional Tools

An important problem facing developers who create and maintain large software systems is that of simply capturing the overall structure of the system. Understanding the relations between functions in a program, definitions and uses of variables, even finding the definition of a function given its name or a call site, are problems that developers and maintainers commonly face. Larger, more complex systems naturally exacerbate these problems.

Strategies for attacking these problems are fairly straightforward for conventional languages. Languages like C, for example, are block-structured, without function overloading. For these languages, it is enough to get the name of a symbol and use it to look the symbol up in the current lexical scope, any enclosing lexical scopes (which are usually in the same file), or the global scope. In addition, the usual programming practice is to break the software system down into files by functionality. This has the effect of keeping the number of different files down, and makes it more likely that the definitions for functions declared in a given file will also be in that file.

The increasing use of object-oriented languages makes the need for software development tools far more acute. Languages like C++ support function and operator overloading, inheritance, dynamic and static binding, and powerful data abstraction facilities. These features defeat simple approaches such as lexical symbol lookup; it is now necessary to understand the *semantic* structure of the system when providing software development tools—the context in which a particular symbol appears determines which specific code object (variable or function) is actually being referenced.[1]

Inheritance, Dynamic Binding, and Code Comprehension

Inheritance and dynamic binding are two of the most common features of object-oriented languages; the benefits of these features have been widely reported. When it comes to the nuts and bolts of actually editing object-oriented programs, however, these features complicate operations that would be straightforward in conventional procedural languages, and, acting in concert, they can turn usually simple tasks into major investigative undertakings.

Inheritance gives rise to distributed class descriptions. That is, the complete description for a class C can only be assembled by examining C as well as each of C's superclasses.[2] Because different classes are described at different places in the source code of a program (often spread across several different files), there is no single place a programmer can turn to get a complete description of a class. This problem can be mitigated somewhat by class browsers and similar tools [5], [6], but such tools are, at this point, neither well developed nor commonly available. The end result is that a programmer working on object-oriented software can spend a substantial amount of time searching through various class descriptions in an attempt to find the information s/he needs.

Dynamic binding makes it impossible to statically determine which function body will be invoked at a given call site in a program. If an object O is statically declared to be of type T, then at runtime it may be bound to an object of type T or any of T's subtypes.[3] If a class function f is then invoked on O, it is the dynamic type of O that determines which class's f is invoked. For a programmer examining an unfamiliar object-oriented program, the fact that dynamically bound calls can't be statically traced can pose a serious barrier to her/his understanding of the program.

An Example

One of the software systems currently under development at Brown University is used to model directed hypergraphs [7], [8]. In these graphs, hyperarcs are sets of source and destination branches connected to source nodes and destination nodes, respectively. Source (destination) branches are attached to nodes at output (input) ports. The design takes advantage of the inheritance and dynamic binding features provided in C++ to capture semantic constraints on the different objects in the model.

For example, arcs in the hypergraph consist of source and destination branches, which share a certain set of features. This relationship is expressed by the presence of a

[1] In C++, for example, a reference to the variable named x within a class member function can only be resolved to the specific variable meant after examining the scopes of the function and class in which it appears and those of all of that class's superclasses.

[2] Details vary from language to language. In the case of languages like C++, classes can inherit data and functions from arbitrarily distant superclasses, so the search for a complete class description is recursive.

[3] Again, details vary. In C++, for example, dynamic binding is performed only on pointers to or references to objects, not on objects themselves.

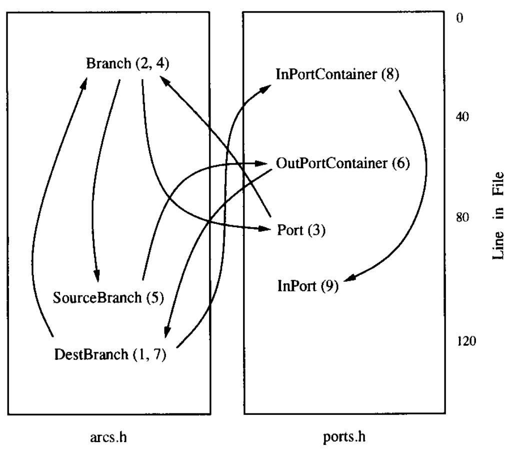


Fig. 1. Route taken by programmer search for body of putToken.

generic class `Branch`, which contains the structure common to all branches, and subclasses `SourceBranch` and `DestBranch`, containing the details specific to those particular types of branches.

Class `DestBranch` includes the following function:

```
virtual void
DestBranch::putToken(Token *token)
{ port.putToken(token); }
```

The implementation of this function merely invokes another function named `putToken`, namely the one declared as a member of the class of which the object `port` is an instance. When reading this code, a programmer is likely to want to find the definition (i.e., code body) of this second `putToken` function, but it does not suffice to simply to look up functions with that name, as the name could conceivably be defined by any of the classes used in the program. It is necessary to determine both the type of the object on which the function is being called, `port` in this case, as well as the types of the arguments to the function, in order to uniquely determine which particular function is being invoked. It turns out that in the system from which this example is drawn, the function `putToken` is dynamically bound, and a substantial search effort is required to determine which function is called by looking only at the source code.

Meyers has published the details of the search elsewhere [9]. For our purposes, it suffices to note that the programmer had to examine nine classes (including two duplicates) just to find out what code would be executed for this call site: the search involved the determination of the static type of `port` (potentially involving all the supertypes of type `DestBranch`), its definition for `putToken`, the dynamic type of `port` (potentially requiring a search for all possible assignments to `port`), and the definition that type provided for `putToken`.

Yet this example understates the complexity involved in the programmer's search, because C++ programs are typically broken up into multiple files. In this case, the files of relevance are `arcs.h` and `ports.h`. Fig. 1 depicts the route the programmer took through these files as s/he tried to

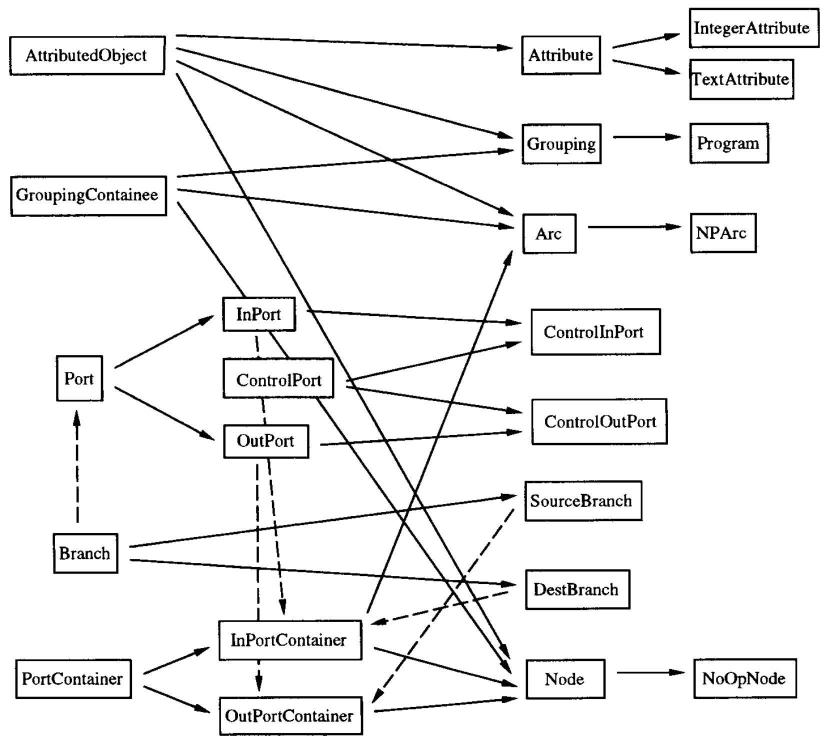


Fig. 2. Part of a real inheritance graph. Arrows run from superclasses to subclasses. Broken arrows indicate some of the "uses" relations present in the classes pictured.

track down the code that was to be executed by the call to `DestBranch::putToken` shown above. The figure shows classes that had to be examined, the locations of the class declarations in the files, and the order in which the declarations were visited during the search for the appropriate function body.

The first class visited was `DestBranch`, declared near the end of `arcs.h`, followed by `Branch`, declared near the beginning of the same file, followed by `Port`, declared near the middle of `ports.h`, etc. The files `arcs.h` and `ports.h` actually define many more classes than are shown in the figure, but those not directly related to the problem at hand have been omitted. If nothing else, Fig. 1 demonstrates that programmers can hardly rely on locality of reference to assist them in their work of understanding object-oriented programs.

In fact, the problem might well have been much harder. Had there been intervening classes in the inheritance hierarchy between `DestBranch` and `Branch`, the programmer would have had to examine each of them in her/his quest for the declaration of `port`. Had multiple inheritance been present, the search could have been substantially more extensive. Had the classes been spread out over more files (which is almost always the case with C++ programs), the task would have been that much more difficult. As real-world examples go, this one is fairly tame.

To better appreciate the complexity that can exist in inheritance hierarchies in real programs, consider Fig. 2, which shows a portion of the inheritance graph (23 of 37 classes) from which the example was taken. This hierarchy, which is declared in five different files (and implemented in five more), is probably more tightly coupled than most. It contains classes with up to four superclasses, classes with up to four subclasses, and inheritance paths of up to length three. In the presence of this kind of graph, determining the class relationships affecting function calls is rarely straightforward, yet it is a task that must be performed frequently.

III. The XREF/XREFDB System

Any solution to the problems encountered when attempting to develop and maintain large software systems written using object-oriented languages has to fulfill two important requirements: it has to be able to describe the structure of the system being supported, and it has to provide an efficient and user-friendly interface.

To address the first of these requirements, we decided to go back to an old idea: using a database to store and retrieve descriptions of the structure of a system. We defined a set of database relations that is comprehensive enough to store cross-referencing and declarative information about systems written in several software languages (including object-oriented languages), and we developed program analyzers that can extract that information from program source code.

The `FIELD` programming environment includes a cross-reference database system, known as `XREFDB`, which we chose as the underlying information repository for the query tools we developed to interact with the C++ programmer.

XREFDB itself merely maintains a database containing all the cross-reference information for a given software system and manages user queries about that information. The actual gathering of the cross-reference information necessary to describe the system is done by front-end programs that can be designed in a language-specific fashion.

To address the second of our requirements, we incorporated an interface to this database into the GNU Emacs text editor. Using this interface, programmers can obtain information about the symbol or function call they are looking at directly, or they can specify a particular symbol on which a particular item of information is desired. The architecture of our system is shown in Fig. 3.[4]

Editing Support for C++

Work on the development of several software systems in C++ suggested a series of queries that programmers are likely to make. These queries include the following.

- Static type lookup: The user can query XREF for the static type of any symbol in the code (variable or function); the result of the query is display in the Emacs mode (status) line, as shown in Fig. 4.
- Declaration lookup: The user can query XREF for the declaration site for any symbol in the program (variable, function, or class); XREF will either display that site in a separate buffer, or switch the editor's current buffer to that site.
- Definition lookup: The user can query XREF for the definition site for any symbol in the program (variable, function, or class). Fig. 5 shows the result of asking XREF for the definition site for the function under the cursor (which was `eligible()` in this case).
- Cross-reference information: The user can query XREF for information about call sites for a given function, assignments to a given variable, or uses of a given variable.
- Inheritance information: The user can query XREF for the declarations of any of the superclasses (or any of the derived classes) of any class in the program. When more than one possible superclass (or derived class) exists, the user is given a choice among all possibilities, as shown in Fig. 6.
- Class information: The user can ask XREF for a summary description of any class in the program; XREF will traverse the class hierarchy and present a view of the class that includes all of the data and function members accessible to that class, listing for each one the (base) class that provides that member, its type, and the file and line where the member can be found (as shown in Fig. 7).
- Polymorphic functions: The user can query XREF for the set of redefinitions for any polymorphic function in the program, or alternatively, the set of classes that rely on the definition provided by a particular class in the hierarchy.

[4]The architecture for the system using `annotedit` is quite similar.

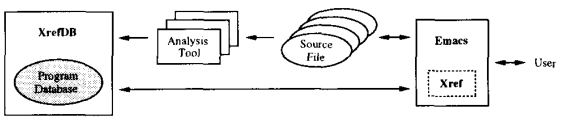
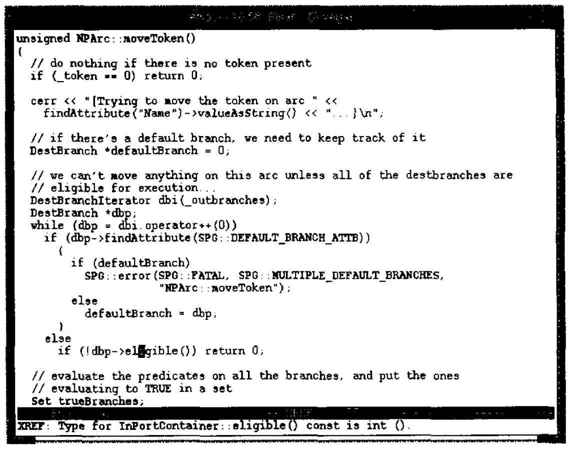
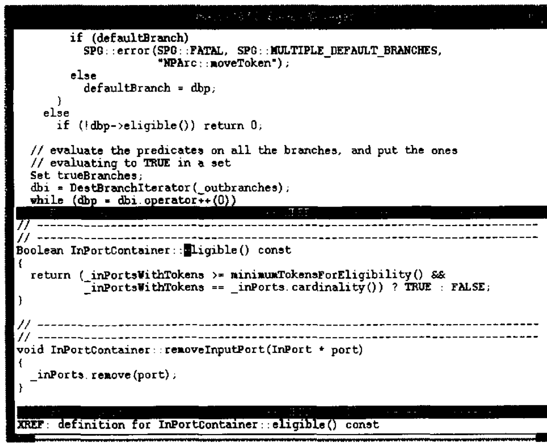


Fig. 3. The architecture of XREF/XREFDB.



```
unsigned NPArc::moveToken()
{
  // do nothing if there is no token present
  if (_token == 0) return 0;

  cerr << "[Trying to move the token on arc " <<
    findAttribute("Name")->valueAsString() << "...]\n";

  // if there's a default branch, we need to keep track of it
  DestBranch *defaultBranch = 0;

  // we can't move anything on this arc unless all of the destbranches are
  // eligible for execution...
  DestBranchIterator dbi(_outbranches);
  DestBranch *dbp;
  while (dbp = dbi.operator++(0))
    if (dbp->findAttribute(SPG::DEFAULT_BRANCH_ATTB))
      {
        if (defaultBranch)
          SPG::error(SPG::FATAL, SPG::MULTIPLE_DEFAULT_BRANCHES,
                     "NPArc::moveToken");
        else
          defaultBranch = dbp;
      }
    else
      if (!dbp->eligible()) return 0;

  // evaluate the predicates on all the branches, and put the ones
  // evaluating to TRUE in a set
  Set trueBranches;
XREF: Type for InPortContainer::eligible() const is int ().
```

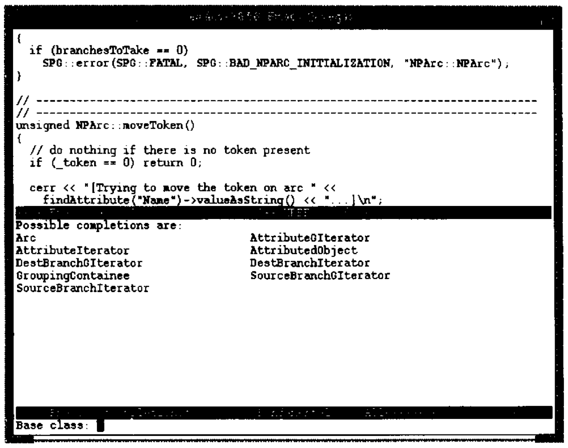
Fig. 4. Querying the type of the function `eligible()`.



```
        if (defaultBranch)
          SPG::error(SPG::FATAL, SPG::MULTIPLE_DEFAULT_BRANCHES,
                     "NPArc::moveToken");
        else
          defaultBranch = dbp;
      }
    else
      if (!dbp->eligible()) return 0;

  // evaluate the predicates on all the branches, and put the ones
  // evaluating to TRUE in a set
  Set trueBranches;
  dbi = DestBranchIterator(_outbranches);
  while (dbp = dbi.operator++(0))
// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
Boolean InPortContainer::eligible() const
{
  return (_inPortsWithTokens >= minimumTokensForEligibility() &&
          _inPortsWithTokens == _inPorts.cardinality()) ? TRUE : FALSE;
}

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
void InPortContainer::removeInputPort(InPort * port)
{
  _inPorts.remove(port);
}

XREF: definition for InPortContainer::eligible() const
```

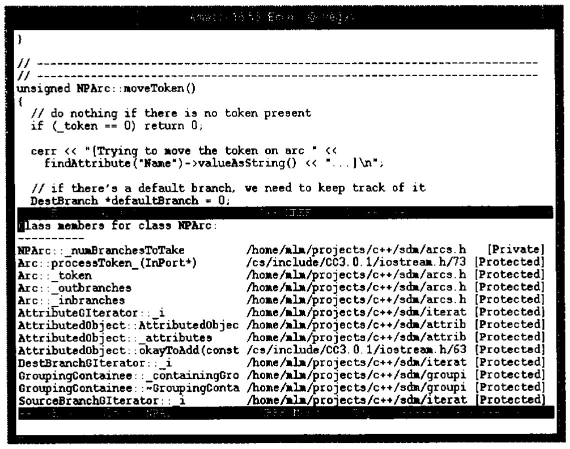
Fig. 5. Showing the definition for the function `eligible()`.

Whenever XREF presents the user with a list of items, it presents it as a list that the user can use to switch automatically to the file and line containing any one of the entries in the list displayed. In addition to these queries, XREF also supports utility functions such as determining the currently active class scope, and scope-specific symbol completion (the ability to do



```
{
  if (branchesToTake == 0)
    SPG::error(SPG::FATAL, SPG::BAD_NPARC_INITIALIZATION, "NPArc::NPArc");
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
unsigned NPArc::moveToken()
{
  // do nothing if there is no token present
  if (_token == 0) return 0;

  cerr << "[Trying to move the token on arc " <<
    findAttribute("Name")->valueAsString() << "...]\n";
Possible completions are:
Arc                                  AttributeGIterator
AttributeIterator                    AttributedObject
DestBranchGIterator                  DestBranchIterator
GroupingContainee                    SourceBranchGIterator
SourceBranchIterator

Base class:
```

Fig. 6. Possible base classes of NPArc to display.



```
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
unsigned NPArc::moveToken()
{
  // do nothing if there is no token present
  if (_token == 0) return 0;

  cerr << "[Trying to move the token on arc " <<
    findAttribute("Name")->valueAsString() << "...]\n";

  // if there's a default branch, we need to keep track of it
  DestBranch *defaultBranch = 0;
Class members for class NPArc:
----------
NPArc::_numBranchesToTake           /home/mlm/projects/c++/sdm/arcs.h   [Private]
Arc::processToken_(InPort*)         /cs/include/CC3.0.1/iostream.h/73 [Protected]
Arc::_token                         /home/mlm/projects/c++/sdm/arcs.h [Protected]
Arc::_outbranches                   /home/mlm/projects/c++/sdm/arcs.h [Protected]
Arc::_inbranches                    /home/mlm/projects/c++/sdm/arcs.h [Protected]
AttributeGIterator::_i              /home/mlm/projects/c++/sdm/iterat [Protected]
AttributedObject::AttributedObjec   /home/mlm/projects/c++/sdm/attrib [Protected]
AttributedObject::_attributes       /home/mlm/projects/c++/sdm/attrib [Protected]
AttributedObject::okayToAdd(const   /cs/include/CC3.0.1/iostream.h/63 [Protected]
DestBranchGIterator::_i             /home/mlm/projects/c++/sdm/iterat [Protected]
GroupingContainee::_containingGro   /home/mlm/projects/c++/sdm/groupi [Protected]
GroupingContainee::~GroupingConta   /home/mlm/projects/c++/sdm/groupi [Protected]
SourceBranchGIterator::_i           /home/mlm/projects/c++/sdm/iterat [Protected]
```

Fig. 7. Showing the complete specification of class NPArc.

symbol completion based on the currently active class scope, including all applicable superclasses).

These queries were implemented as a set of routines in Emacs Lisp [10], collectively known as the XREF module, that allow users to ask these questions from within GNU Emacs. XREF translates user requests into XREFDB queries, communicates them to an XREFDB process running alongside Emacs, and then uses the answers provided by the database to take appropriate actions within Emacs. These actions could be simply to display the results, or to open new edit buffers on the selected file/line, given the answer to the user query.

The routines determine the symbol in the text file the user is asking about by selecting the identifier under the edit cursor. If that identifier has a type compatible with the query being made, it is used as the object of the query; otherwise, the user is asked for an identifier explicitly. On those queries where a list of results is possible, the user is presented the results as a menu, and s/he can choose which entry to display/edit by selecting the corresponding entry in that menu. For example, when the user asks for a description of a class, XREF displays in a buffer all the members belonging to that class (both those defined in the class itself and those defined in all its base classes). The user can then choose which member to obtain more information about.

XREFDB *Relations*

XREFDB is a relational database designed to store useful information about programs, independent of the language they are written in. The following general-purpose relations are supported by XREFDB.

Scope: Relation used to describe scopes. Each tuple in this relation specifies the kind of scope, its id, the file it is in, and the set of lines it encloses.

Ref: Relation used to describe references to symbols. Each tuple in this relation specifies the name of the symbol being referenced, the function within which the reference occurs, the file and line number in which the reference occurs, and whether the reference is an assignment to the symbol.

Decl: Relation used to describe symbol declarations. Each tuple in this relation specifies the name of the symbol being declared, the file, line and function in which the declaration occurs, the "kind" of declaration (its visibility and storage type), the type of the symbol being declared, the scope in which the declaration occurs, and a unique id for the declaration.

Call: Relation used to describe function calls. Each tuple in this relation specifies the function being called, the function it is being called from, and the file and line in which the call occurs.

Fct: Relation used to describe function definitions. Each tuple in this relation specifies the name of the function being defined, the number and names of the formal arguments, the scope in which the function is defined, and the file and line where the definition of the function begins.

file: Relation used to describe file names. Each tuple in this relation specifies the name of one of the source files, the complete pathname for that file, and a unique file id.

In order to store all the information necessary to represent object-oriented languages, the following relations were added to XREFDB.

Hier: Relation used for describing inheritance and friendship[5] relations between classes. Each tuple in this relation includes the name of a class; the name of one of its parent classes (for inheritance relationships); the file and line where the inheritance/friendship relationship is declared; for inheritance, whether the base class (superclass) is virtual or nonvirtual, public, protected, or private; and whether the relationship is one of friendship or inheritance.

Memb: Relation used to describe friends and members of C++ classes. Each tuple in this relation includes the name of the member or friend, the class it belongs to, the file and line in which its declaration can be found, its access level (public, protected, or private), whether it is a data or function member,

[5]C++ uses *friend* declarations to selectively bypass data abstraction. If a class *A* extends friendship to a class or function *B*, then *B* has complete access to the internal members of *A*, regardless of their level of general accessibility.

and whether it is declared to be inline, friend, virtual, static, pure virtual or const.

mDef: Relation used to store supplementary information about class members. Each tuple in this relation includes the name of the member, the name of the class it belongs to, the file and line in which the member is defined, and the type of the member.

CLIENT: Relation used to indicate client-supplier relationships. Each tuple consists of two types T_1 and T_2, where T_1 contains one or more instances of T_2.

Front-end processors for XREFDB*:* The tasks of generating the tuples that reflect the structure of the program and that of manipulating that information were decoupled, to increase the flexibility of our system. As long as the set of relations defined is powerful enough to represent the semantic relations present in a programming language, all that is required for the database to support that language is a language-specific program analyzer to obtain the necessary cross-reference information from the source files. So far, program analyzers have been written for C, Pascal, and C++.

All these analyzers begin by examining an executable file to determine which source files were used to generate it. They then parse each of the source files, adding the desired information to the XREFDB database.

In the case of the C and Pascal analyzers, they simply read the user's original source files. In the case of C++, the situation is more complex. Originally, instead of duplicating a large part of the behavior of Cfront (the C++-to-C translator used in AT&T's implementation of C++), our C++ analyzer read both the original C++ source files and the C files generated from Cfront, trusting it to perform all the necessary disambiguation, and then using the results. While this simplified the design of the C++ analyzer, it also had the drawback that information about calls to virtual functions was lost in Cfront's translation to C, and it turned out to be difficult to recover the missing information from the original C++ source files.

After struggling for some time with this approach, we concluded that there was no alternative to modifying a C++ compiler to output the appropriate database information as it compiles each source file. We proceeded to modify both Cfront and the GNU C++ compiler to generate this information.

IV. Experiences with XREF

Because C++ programs are difficult to work with using conventional software development tools, there has never been a shortage of programmers willing to employ XREF on real programs, even when XREF existed as only a very preliminary prototype. As a result, the functionality of XREF has been strongly influenced by the pragmatic concerns of day-to-day C++ programmers. For example, the first version of XREF supported only "relative" symbol lookup, i.e., the ability to get information on a symbol that the edit cursor was on top of. Very quickly we discovered that "absolute" symbol lookup, i.e., the ability to get information on arbitrary symbols typed in from the keyboard, was also essential. Similarly, we now provide some support for "trains of thought" in XREF, whereby programmers are able to string together a series of queries through XREF, along with their attendant changes in edit buffers, so that they can "go back to the beginning" of a complex string of queries. This kind of capability is particularly useful for programmers attempting to understand dynamically bound function calls, since a programmer needing to find out which function would be called at a particular call site might first want to embark on a train of thought to discover what the dynamic type of an object was likely to be at the time of the call. We have been pleased with the ability of XREF to smoothly accommodate the addition of such unanticipated features.

Although the usefulness of XREF in the development and maintenance of software systems has yielded encouraging results, the fact that XREF and its database are still in the prototype stage means that it is not yet practical to use them for really large systems. Creation of the database is not particularly fast—on the order of minutes for systems with thousands of lines of code, tens of minutes for systems with many tens of thousands of lines of code—and some queries take many seconds to process. At the same time, however, our efforts so far have focused on providing functionality, not on tuning performance, and, as noted later, there are fairly simple modifications that could be expected to dramatically improve the speed of the system.

We have no quantitative information on the impact XREF has on the ability of developers to understand the structure of the systems they are working on, but the general feeling among those programmers who have used XREF is that the time and effort required to understand a system drops significantly when the programmer can rely on XREF to do the tedious work of locating functions and the types of objects in the system. Similarly, users claim that the ability to follow a series of function calls through the source code by simply pointing at a function and switching to an edit window on the definition of that function reduces some of the drudgery of maintaining a system. It allows the programmer to concentrate on the semantics of the system, rather than the uninteresting details of what is where in the system's source code.

Perhaps more important than these subjective impressions is the observation that some C++ programmers have freely chosen to use XREF even when the package was incomplete, was known to contain nontrivial bugs, and/or suffered from rather lackluster performance. Such voluntary use under less than ideal conditions increases our confidence in the need for such a package.

V. Related Work

The Trellis programming environment [11] provides Trellis/Owl programmers with a set of tools similar to the ones presented in this paper. Like our system, Trellis is built around a program database that includes cross-referencing and type information about the objects and functions present in the program under development. Since it is an integrated system, it also supports user queries directly from the editor. It appears to be less flexible than the system we describe here, both in terms of the kinds of queries supported by the interface and in terms of the ease with which it could be adapted to other programming languages. Other systems that include a database integrated with a text editor include MasterScope for Interlisp [12] and the R^n environment for FORTRAN [13]. None of these systems supports C++.

The C and C++ Information Abstractors (CIA and CIA++) [14], [15] are systems in which a database is used to store and manipulate information describing the structure of a program. The information stored in the database in this approach is comparable to the information stored in our system, and in both systems it is obtained by program analyzers that collect this information from the original source files. However, CIA and CIA++ do not provide an interface to the database integrated into any other tools (in particular the text editor): the approach taken by CIA and CIA++ is to make the database a stand-alone tool.

Other systems consisting of a stand-alone database containing information on program structures include FAST [16], Cscope [17], and OMEGA [18]. Other approaches built around a central database include ENCORE [19] and the Harvard Programming Development System [20].

Recently, Rosenblum and Wolf have developed an internal representation for the semantics of C++ programs [21], and this representation could be used in place of a formal database for C++ development tools. The implementation is not yet complete, however, so it is too early to evaluate the success of this approach. At any rate, their representation is very much specific to C++.

VI. Status

We have implemented prototype versions of XREF, XREFDB, and program analyzers to generate the appropriate database information for C++, as well as C and Pascal. The prototypes are currently being tested on several systems under development at Brown University with encouraging results.

Our original prototype implementation of XREFDB suffered from poor response times on large systems, but we have since added a series of query optimizations that have increased performance considerably. As things stand now, fairly sophisticated queries over a database corresponding to the FIELD source (comprising some 250 000 lines of code) can be evaluated in just a few seconds.

The prototype implementations of the front-end processors used to generate the C++ cross-referencing information stored in XREFDB have some shortcomings, which limit the power of XREF. For example, to statically resolve calls to virtual member functions, the dynamic type of the object on which the function is being called must be determined. There is currently no way of differentiating cases in which the dynamic and static types are the same from those in which they may be different. This problem is undecidable in general, and two approaches seem plausible given this restriction: we can either try to determine what the dynamic type of an object will be at the point of a query, or we can assume that the dynamic and static types are always the same. XREF currently assumes that they are the same.

The version of the C++ analyzer that relies on Cfront has an additional shortcoming in that bugs in the layout of the C code generated by Cfront sometimes result in erroneous line numbers in the database. This can eventually lead to query failures due to the resulting mismatch between the information in the database and the information used by XREF to index into that database.

VII. Summary

Some of C++'s most important features—classes, inheritance, and dynamic binding—can interfere with the ability of software developers and maintainers to understand the code they work with. The difficulty lies in the tools generally available to work with C++ programs. Conventional tools such as text editors are designed to work in purely lexical units, and while this suffices for procedural languages, it becomes woefully inadequate when applied to C++, with its penchant for overloaded names, interdependent scopes that are not lexically nested, and both static and dynamic object types.

The solution to the problem lies in the creation of tools specifically geared to the requirements of C++ software development. Foremost among these are editors capable of "understanding" the semantics of class hierarchies and dynamically bound functions, so they can offer programmers the ability to view and edit programs in terms of the meaningful units of the language being used. We have implemented a prototype system that addresses this need, one that is compatible with traditional software development tools, and have demonstrated that it provides a qualitative improvement in the environment available to C++ programmers.

Acknowledgment

The United Stated government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

References

[1] M. Lejter, S. Meyers, and S. P. Reiss, "Support for maintaining object-oriented programs," in *Proc. Conf. Software Maintenance*, pp. 171–178, Oct. 1991. (Also available as Brown University Computer Science Department *Technical Report CS-91-52*, Aug. 1991. This paper is largely drawn from two other papers [9], [22].)

[2] R. M. Stallman, "EMACS: The extensible, customizable, self-documenting display editor," in *Proc. ACM SIGPLAN/SIGOA Symp. Text Manipulation*, pp. 147–156, June 1981.

[3] S. P. Reiss, "Connecting tools using message passing in the FIELD program development environment," *IEEE Software*, pp. 57–67, July 1990. (Also available as Brown University Computer Science Department *Technical Report CS-88-18*, "Integration mechanisms in the FIELD environment," 1988).

[4] ——, "Interacting with the FIELD environment," *Software: Practice and Experience*, vol. 20, pp. 89–115, June 1990.

[5] S. P. Reiss and S. Meyers, "FIELD Support for C++," in *USENIX C++Conf. Proc.*, pp. 293–299, Apr. 1990.

[6] R. Raghavan, N. Ramakrishnan, and Ş. Strater, "A C++ class browser," in *1987 USENIX C++Papers*, pp. 274–281, 1987.

[7] S. Meyers and S. P. Reiss, "A system for multiparadigm development of software systems," in *Proc. Sixth Int. Workshop on Software Specification and Design*, Oct. 1991. (Also available as Brown University Computer Science Department *Technical Report CS-91-50*, Aug. 1991.)

[8] S. Meyers, "Representing software systems in multiple-view development Environments," Ph.D. dissertation, Brown University Department of Computer Science, 1993.

[9] ——, "Working with object-oriented programs: The view from the trenches is not always pretty," in *Proc. Symp. Object-Oriented Program-*

ming Emphasizing Practical Applications (SOOPPA), pp. 51–65, Sept. 1990.

[10] B. Lewis and D. LaLiberte, *The GNU Emacs Lisp Reference Manual.* The Free Software Foundation, 1990.

[11] P. D. O'Brien, D. C. Halbert, and M. F. Kilian, "The Trellis programming environment," in *Proc. 1987 Conf. Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pp. 91–102, Oct. 1987.

[12] W. Teitelman and L. Masinter, "The Interlisp programming environment," *IEEE Computer*, vol. 14, pp. 25–34, Apr. 1981.

[13] R. T. Hood and K. Kennedy, "A programming environment for Fortran," *Tech. Rep. TR84-1*, Rice University, June 1984.

[14] Y.-F. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy, "The C information abstraction system," *IEEE Trans. Software Eng.*, vol. 16, pp. 325–334, Mar. 1990.

[15] J. E. Grass and Y.-F. Chen, "The C++ information abstractor," in *USENIX C++Conf. Proc.*, pp. 265–277, 1990.

[16] J. Browne and D. B. Johnson, "FAST: A second generation program analysis system," in *Proc. Second Int. Conf. Software Engineering*, pp. 142–148, 1977.

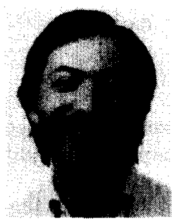
[17] J. L. Steffen, "Interactive examination of a C program with CScope," in *Proc. USENIX Association Winter Conf.*, pp. 170–175, 1985.

[18] M. A. Linton, "Implementing relational views of programs," in *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. Practical Software Development Environments*, pp. 132–140, Apr. 1984. P

[19] S. B. Zdonik and P. Wegner, "A database approach to languages, libraries, and environments," *Tech. Rep. CS-85-10*, Brown University Department of Computer Science, 1985.

[20] T. E. Cheatham, Jr., "An overview of the Harvard program development system," in *Software Engineering Environments*, H. Hunke, Ed. New York: North-Holland, 1981.

[21] D. S. Rosenblum and A. L. Wolf, "Representing semantically analyzed C++ code with reprise," in *USENIX C++Conf. Proc.*, pp. 119–134, Apr. 1991.



Moises Lejter received the B.A. degree in computer science and linguistics from Brandeis University in 1986, and the M.S. in computer science from Brown University in 1988.

He is a graduate student in computer science at Brown University and is scheduled to receive his Ph.D. in May 1993. His research interests include distributed planning and control, robotics, and software development environments.

Scott Meyers will be awarded his Ph.D. in computer science from Brown University, Providence, RI, in May 1993.

He is currently a postdoctoral Research Associate at Brown University. His research interests include programming environments, multiple-view software development, and tools and techniques for effective programming in C++.

Steve Reiss received the Ph.D. degree from Yale University in 1977.

He is currently a Professor at Brown University. He is interested in the effective use of workstations for programming. His recent work includes two projects that provide experimental insight into this area. GARDEN is a conceptual programming environment based on an object-oriented programming system. It provides a framework whereby programmers can quickly develop and use new languages that match the way they conceptualize a program. FIELD is an integrated approach to traditional programming. It provides integration mechanisms for broadcast message-based communication between various UNIX tools and for consistent access to source files. These, combined with tools for visually displaying data structures and for program animation, offer a powerful and unique programming environment that is used both for teaching and for research.