# A C++ Data Model Supporting Reachability Analysis and Dead Code Detection

Yih-Farn R. Chen, Emden R. Gansner, Eleftherios Koutsofios

AT&T Labs - Research, 180 Park Ave., Florham Park, NJ 07932, USA
{chen,erg,ek}@research.att.com
http://www.research.att.com/info/{chen,erg,ek}

**Abstract.** A software repository provides a central information source for understanding and reengineering code in a software project. Complex reverse engineering tools can be built by analyzing information stored in the repository without reparsing the original source code. The most critical design aspect of a repository is its data model, which directly affects how effectively the repository supports various analysis tasks. This paper focuses on the design rationales behind a data model for a C++ software repository that supports reachability analysis and dead code detection at the declaration level. These two tasks are frequently needed in large software projects to help remove excess software baggage, select regression tests, and support software reuse studies. The language complexity introduced by class inheritance, friendships, and template instantiations in C++ requires a carefully designed model to catch all necessary dependencies for correct reachability analysis. We examine the major design decisions and their consequences in our model and illustrate how future software repositories can be evaluated for completeness at a selected abstraction level. Examples are given to illustrate how our model also supports variants of reachability analysis: impact analysis, class visibility analysis, and dead code detection. Finally, we discuss the implementation and experience of our analysis tools on a C++ software project.

## 1 INTRODUCTION

There has been a growing trend [25][1][24] in building software repositories to help maintain structure information of existing legacy code. A software repository provides a central information source for understanding and reengineering code in a software project. While many variants of repository-based systems have been constructed, there is not a clear agreement on how the repository should be organized. One popular approach is to store variants of abstract syntax trees in the repository, such as those used in Reprise[23], ALF[20], Genoa[9], Cobol/SRE[21], PRODAG[22], Aria[10], and Rigi[18] in the IBM program understanding project[2]. Because of the nature of the representations, tree traversal routines are frequently used to generate various abstractions.

The other popular approach, which was adopted in the construction of the C/C++ Information Abstraction Systems[4][7][15] and in XREFDB[17], is to

structure the repository as a relational database so as to reuse the large body of existing database technology. Complex reverse engineering tools can be built by composing queries without reanalyzing the original source code. In this approach, the most critical design component of a repository is its data model, which directly affects how effectively the repository supports various analysis tasks.

This paper focuses on the design of a data model for a C++ software repository that, among other goals, supports reachability analysis and dead code detection, two tasks that are frequently needed in large software projects. They serve as the basis for various reverse engineering tasks, such as detecting unnecessary include files[27], performing selective regression testing[8], and computing objective software reuse metrics[6]. This paper also examines how we use our model to implement reachability analysis and its variants (including dead code detection) in the context of the C++ programming language.

Researchers have found it frustrating to compare reverse engineering tools even on simple criterion such as how well they extract function call graphs[19]. The difficulty arises because different underlying models of these tools give different interpretations on what a function call means. On the other hand, the ability for a repository to support complete reachability analysis as defined in this paper is an objective criterion, for a selected abstraction level, that programmers and researchers can use to compare different repository implementations.

A complex, object-oriented language such as C++ makes constructing a data model adequate for reachability analysis a complicated and delicate process. In addition to the entities and relationships found in typical procedural languages, C++ introduces such additional relationships as inheritance, friendship, access adjustments and template instantiation, which affect the analysis in various ways. As an example, Figure 1 shows what we expect to obtain from a simple query like

*Find all C++ entities reachable from the class* `Pool`.

`Pool` is a class in a C++ components library developed in AT&T that manages a set of same-size memory blocks. In Figure 1, boxes are used for functions, diamonds for types, and ovals for variables. Class inheritance relationships are shown in dashed lines, friendships in dotted lines, and all other relationships in solid lines. This picture reveals several key relationships used in C++ reachability analysis:

- *containment relationship*: between `Pool` and `Pool::alloc()`
- *friendship relationship*: between `Pool` and `Pool_element_header`
- *inheritance rationship*: between `Pool` and `Block_pool_ATTLC`
- *reference relationship*: between `Pool::purge` and `Pool::head`

These relationships and the template instantiation relationship will be examined in detail when we discuss our C++ model.

This paper is organized as follows. We start by presenting our C++ data model and explain the rationales behind many design decisions. We then discuss how the model supports various flavors of reachability analysis in C++

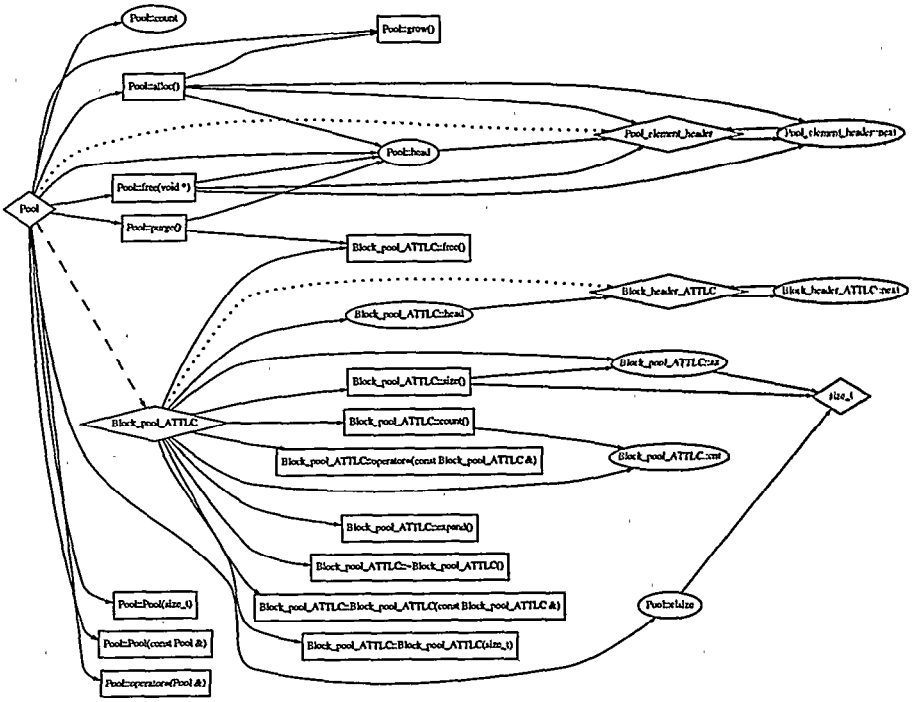


**Fig. 1.** The reachability graph of the class `Pool`

programs, followed by a description of our implementation and an experience report on some sample C++ code we collected from a C++ software project. Finally, we discuss our future plans for the C++ software repository and research opportunities.

## 2 A C++ DATA MODEL

Our C++ data model is formulated using Chen's entity-relationship modeling[3] and it supports both C and C++ programming languages. We consider a C or C++ program as a collection of source entities referring to each other, an entity representing a static, syntactic construct such as a macro, a type, a function or a variable. Since our focus is on creating a *complete* data model that supports reachability analysis and dead code detection, we need to provide a clear definition on *completeness*:

> *Completeness:* A data model $M$ on a programming language $L$ is considered *complete* if, for any two entities $a$ and $b$ in the model, a dependency

relationship $a \rightarrow b$ also exists in $M$ when one of the following two conditions holds:

- $C1$ : if the *compilation* of the entity $a$ depends on the existence of a declaration of the entity $b$.
- $C2$ : if the *execution* of the entity $a$ depends on the existence of the entity $b$.

For example, if $a$ is a source file that includes a header file $b$, then $a \rightarrow b$ should be captured according to $C1$. Similarly, if $a$ is a variable initialized with a macro $b$, or a class that inherits from class $b$, or a template class instantiated from class template $b$, then $a \rightarrow b$ should exist as well because $a$ cannot be compiled without a declaration of $b$.

On the other hand, if a function $a$ calls or refers to a function $b$, even if $b$ is not declared (as is allowed in some C programs and shell scripts), then $a \rightarrow b$ should exist in the model according to $C2$. For a discussion on conditions required (*well-defined memory* and *well-bounded pointer*) for static analysis tools to capture such relationships, directly or indirectly, refer to the TestTube paper[8].

A model that satisfies the completeness criterion allows us to define *reachable entity set* and *dead entity set* in the following way:

*Reachable Entity Set:* A reachable entity set $R(a)$ is the set of entities reachable from an entity $a$ through standard closure computations on the dependency relationships in the model.

*Source Entity Set:* A source entity set $S$ is simply the set of all entities in a program according to the model.

*Dead Entity Set:* A dead entity set $D(r)$ is simply the difference between $R(r)$ and $S$, where $r$ is the entity that serves as the starting point of the program execution. $D(r)$ is the set of program entities that are not needed for the compilation or execution of the program.

The first design choice we have to make in designing a *complete* model is the entity granularity. There are several possibilities, moving from coarser to finer granularity:

- *file:* This is the granularity used by most source code control systems such as RCS[26] and configuration management tools such as *nmake*[12].
- *top-level declaration:* This creates entities for all constructs not defined within function bodies, *plus* the nested components of any entity representing a type. This level could be expanded to include entities for all declarations.
- *atomic:* This models all program information, down to the level of statements and expressions. It captures the complete static syntactic and semantic information of the program. This is used by syntax-directed editors, and is the basis of many commercial software browsers and debuggers.

Each choice has its own consequences. For example, the file granularity does not allow dead entity declarations in a source file to be detected, while the declaration granularity implies that questions concerning detailed flow control cannot be answered. In addition, as the granularity becomes finer, the size of the repository and the time of queries can be prohibitive for real software projects. Whatever entity granularity is selected, all relationships among entities at that level of abstraction must be captured in the repository in order for reachability analysis to be complete and accurate.

Our model uses the granularity of top-level declarations. This includes entities for types, functions, variables, macros and files. We feel that this level provides adequate information for the vast majority of analyses pertaining to issues of software engineering, while avoiding the excessive overhead of finer granularities. It captures the principal structural artifacts of a program, especially those used across modules and classes.

Our C++ model significantly expands and cleans up an earlier model proposed in [15]. In particular, we have added support for template-related entities and relationships, and enforce consistent reference relationships in nested class declarations. Both are required for complete reachability analysis.

In the following, we first discuss the basic attributes shared by all C++ entities and then discuss additional attributes that are required for each entity kind. Many of these attributes are used to modify the behaviors of variants of reachability analysis.

### 2.1 Common Entity Attributes

The attributes that are shared by all C++ (and C) entities in our model include *unique id*, *entity kind*, *entity name*, *source file*, *location*, *definition/declaration flag*, and *checksum*, a numeric value associated with the entity's contents. All C++ entities, including functions, variables, types, macros, and files, have these common attributes.

Note that even such a simple model, without additional attributes discussed later, already implies that several queries are possible:

- count entities: we can count the number of entities of each *kind*.
- search entities: we can find out if an entity with a certain pattern exists in the database.
- retrieve entity source: we can retrieve and view the source code of each entity by using the *source file* and *location* attributes. This is useful if software entities are to be rearranged or packaged for reuse.
- detect entity changes: With two versions of a database, we can find out the lists of entities that are deleted, added, or changed from the old version to the new by examining their corresponding checksums.

For example, a query like "*count the number of deleted function definitions from version 1 to version 2*" can easily be handled by a difference database, which has an additional entity attribute that specifies whether an entity is deleted, added, or changed.

## 2.2 Principal Entities

The entities for types, functions and variables form the basis for most significant program analyses. In addition, C++ allows the constructs to be declared as members inside classes and structs. We call these entities *principal entities*, which require three additional attributes:

- *scope*: A member can be either *private*, *protected*, or *public*. The scope of a non-member entity is either *extern* (global scope) or *static* (file scope). The scope of an entity affects *visibility analysis* discussed later.
- *parent*: This attribute records the parent class or struct of a member, if any.
- *subkind*: In C++, a type entity can be a class template, template class, typedef, struct, class, union or an enum. A function entity can be a function template, template function, or just a regular one. Instead of creating an entity kind for each of these variations, we use this attribute to distinguish among them.

Since a type can be a class template or an instantiated template class, an additional attribute *type param* is necessary to store the formal parameters in the former case and actual parameters in the latter case. For example, the following shows a class template Map and an instance of it retrieved from one of our C++ databases for standard C++ headers:

```
<class S, class T>Map
Map<Set_or_Bag_hashval, unsigned int>
```

Similarly, a function can be a function template, an instantiated template function, or a regular function. We need an additional attribute *function param* to store the template or actual arguments. The following example shows both the function template declaration and a template function of `remove` instantiated with a `String` type:

```
<class T>remove(T * arry, int sz, const T & val)
remove(String *, int, const String &)
```

The definition of C++ limits what a model can provide concerning templates. The macro nature of templates in the language, with non-lexical scoping of free identifiers, means that some relationships simply cannot be resolved at the point of definition. Given this, our model supports various attributes, such as formal template parameters, and containment relationships in templates, such as those between a parent class template and its member function templates, but largely forgoes more complete analysis of template definitions[1]. On the other hand, when a template is used, i.e. instantiated, all the relationships involving the template class or function are captured.

Note that the kind and subkind fields are useful in determining whether some operations are applicable. For example, a tool that detects unnecessary include

[1] We could extend the model to provide entities for the components (e.g., member functions and variables) of a class template, but even this is problematic, as the types associated with these entities may well involve unresolvable type identifiers.

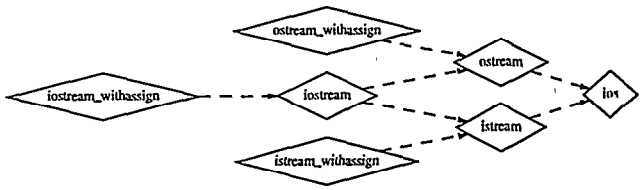


**Fig. 2.** Class inheritance structure of the `iostream` library

files need only be applied to file entities, while *visibility analysis*, discussed later, can only be applied to classes and their members.

### 2.3 C++ Relationships

There are several possible relationships in C++: inheritance, friendship, containment, instantiation, and reference relationships. We examine each relationship in detail and explain how it affects reachability analysis.

**Inheritance Relationship** Figure 2 shows the inheritance structure of the C++ `iostream` library. Note that if an entity refers to `iostream_withassign`, then it also depends on `iostream`, `ostream`, `istream`, and `ios` for compilation. An inheritance relationship can be *private*, *protected*, or *public*. Also, an inheritance relationship can be *virtual*. Two additional attributes are created to handle these variations: *protection kind* and *virtual flag*. The *protection kind* attribute affects the visibility analysis described later.

**Friendship Relationship** There is a friendship relationship from *class A* to *class B* if *class B* declares *class A* as a friend. The relationship direction is set this way because members in *class A* may access members in *class B* and therefore depend on *class B* as far as the direction of reachability analysis is concerned. For example, `ios` depends on `Iostream_init` in the following piece of code because members of `ios` are allowed to access members of `Iostream_init`.

```
static class Iostream_init {
        static int        stdstatus ;
        static int        initcount ;
        friend class      ios ;
public:
        Iostream_init() ;
        ~Iostream_init() ;
} iostream_init ;
```

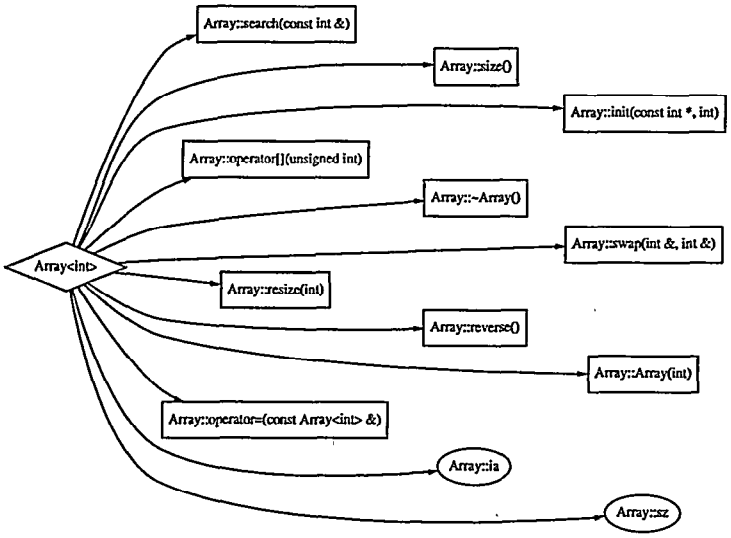


**Fig. 3.** Containment relationships between `Array<int>` and its members

**Containment Relationship** There is a containment relationship between every parent class or struct and a member. For example, Figure 3 shows that there are 10 member functions (boxes) and two member variables (ovals) contained in the template class `Array<int>`.

Containment relationships may or may not be walked through depending on the purpose of the reachability analysis . We shall elaborate on this in the next section.

**Instantiation Relationship** An instantiation relationship exists if entity A is an instance of template B. A depends on B for compilation and linking. For example, the template class `set<int>` is an instance of the class template `<class T>set` and the template function `sort(String *,int)` is an instance of `<class T>sort(T* arry, int sz)`.

**Reference Relationship** Formally, a reference relationship exists between entity A and entity B if (a) it is not one of the above relationships, and (b) entity A refers to entity B in its declaration or definition. Again, entity A cannot be compiled and linked without the declaration or definition of entity B. The following example shows several reference relationships:

```
class ios {
    ...
    enum        { skipping=01000, tied=02000 } ;
```

```
    streambuf*      bp;
    void        setstate(int b)
                {       state |= (b&0377) ;
                        ispecial |= b&~skipping ;
                }
    ...
};
```

- variable `ios::bp` to class `streambuf`
- function `ios::setstate` to member variable `ios::state`
- function `ios::setstate` to enum constant `ios::type-230-9::skipping`[2]

# 3 REACHABILITY ANALYSIS

Reachability analysis can have different slants depending on the purpose of the analysis. We shall discuss several variants and describe how our model and the implementation, *Acacia*, supports each one of them.

Note that the reachability analysis supported by our model is conservative, in that the set of entities returned may be a superset of the minimal closure based on the actual source. This follows from the fact that we are only capturing static syntactic information concerning top-level declarations. For example, an analysis of a program at the expression level may indicate that a call to a function only occurs in a branch that is never executed, and hence the function is never called, whereas Acacia would report the function as needed. Based on our experience, this appears to be reasonable for typical software engineering tasks.

## 3.1 Forward Reachability Analysis

Forward reachability analysis, which computes *Reachable Entity Set* as defined earlier, is the basis for detecting dead code, packaging reusable software entities, and computing software reuse metrics.

For many tasks, computing the simple transitive closure is sufficient. In some cases, such as software reuse, the task also requires computing certain indirect relations by doing selective reverse reachability computations. For example, class member declarations cannot exist on their own for compilation and therefore we must also capture the *containing* parent declarations. In general, the model explicitly or implicitly contains complete reachability information, so that indirect relations can be generated using appropriate queries over the database.

Figure 4 shows the first three layers of C++ entities transitively reachable from the `main` function in a sample program. The complete closure set includes 134 functions, 17 types, and 72 variables and is too large to fit on a single page.

Note that there is an *instantiation relationship* between `Array<int>` and `<class Type>Array`. Also, while `ostream::operator <<(const char*)` is considered referenced by `main`, the class `ostream` itself is not. We have two choices, depending on the purpose of reachability analysis:

[2] `type-230-9` is a surrogate name created for the anonymous enum type.

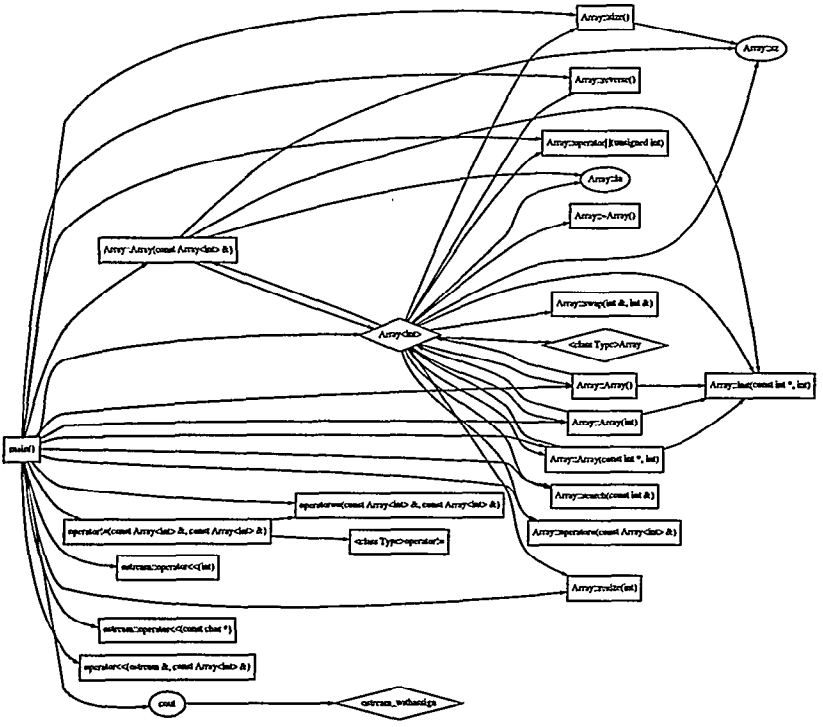


Fig. 4. The first three layers of C++ program entities reachable from `main`

- *software reuse*: In this case, the parent of `ostream::operator <<(const char*)`, `ostream`, is considered referenced. We then include the forward reachability analysis on `ostream` in the closure. While this process may include some class members that are not used in a particular application, we assume that a class should be treated as a unit[3].
- *dead code detection*: This is discussed next.

### 3.2 Dead Code Detection

Many large software projects suffer from a syndrome called *excess baggage* that has one or more of the following symptoms:

- *unnecessary include files:* Many declarations in the header files are never used, but are compiled repeatedly for the source file that includes them.

[3] If space is an issue, the analysis can be modified to only include class member functions that are used in the given program.

– *dead program entities:* Due to program evolution, many program entities usually become obsolete, but programmers either cannot locate them or are afraid to delete anything because they cannot predict the consequences.

To remove excess baggage, we start from the entry points of a program and find the closure set of entities reachable. Containment relationships do not have to be expanded if we want to detect dead member entities for a particular application. This is sometimes critical for applications with strict memory requirements. As described previously in the definition of *dead entity set*, by comparing the closure set against the complete set of program entities in the database, we get a list of unused program entities. Usually, the user is only interested in dead entities in their own code and ignore dead ones in system header files. Our dead code detection tool creates a database of dead program entities; queries can be used to filter out or focus on particular subsets.

As an example, we applied our analysis tool to a C++ program written by Andrew Koenig that illustrates the concept of dynamic binding[16]. One of the key classes is `Tree`:

```
class Tree {
public:
    Tree(int);
    Tree(char*,Tree);
    Tree(char*,Tree,Tree);
    Tree(const Tree& t){ p = t.p; ++p->use; }
    ~Tree() { if (--p->use == 0) delete p; }
    void operator=(const Tree& t);
private:
    friend class Node;
    friend ostream& operator<< (ostream&, const Tree&);
    Node* p;
};
```

We would like to determine if the sample test program (shown below) exercises all member entities in the `Tree` class.

```
main()
{
  Tree t = Tree ("*", Tree("-", 5), Tree("+", 3, 4));
  cout << t << "\n";
  t = Tree ("*", t, t);
  cout << t << "\n";
}
```

While it may not be immediately obvious for some users, this small test program does exercise all member functions of `Tree`, including the destructor, which is called implicitly on the local variable `t` just before the function exits.

On the other hand, if we replace the test driver with the following piece of code:

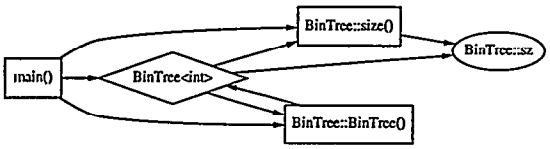


Fig. 5. Impact Analysis: C++ entities that depend on `BinTree:sz` directly or indirectly

```
main()
{
  Tree t = Tree (5);
  cout << t << "\n";
}
```

then the dead code analysis tool reports that the following three member functions of `Tree` are not exercised by the new test driver:

```
Tree::Tree(const Tree &)
Tree::Tree(char *, Tree)
Tree::Tree(char *, Tree, Tree)
```

### 3.3 Impact Analysis

Before software changes are made, it is frequently desirable to find all program entities that potentially can be affected. Impact analysis, or reverse reachability analysis, allows programmers to find all program entities that depend on an entity directly or indirectly. For example, Figure 5 shows that if `BinTree::sz` is changed, then all the other entities in the graph potentially can be affected. Such an impact analysis is the basis for selecting regression tests[8] after a change is made in the source code.

### 3.4 Visibility Analysis

It is frequently necessary to determine what member variables and functions in a class inheritance hierarchy are visible to a derived class. For example, to find all member functions, variables, and types visible to class `BinaryNode` in Koenig's example[16], we can perform a reachability analysis on the containment relationships in the inheritance tree starting from `BinaryNode` and limit the search to members of the proper scope. All members in `BinaryNode` are obviously visible to itself; all *public* and *protected* members from `Node` are also visible because `BinaryNode` has a *public* inheritance relationship with `Node`, but the *private* member variable of `Node` (`Node::use`) is not included. On the other hand, `Node` is a friend of `Tree` and therefore all members of `Tree` are visible to `Node`, but none of them are visible to `BinaryNode` because friendship cannot be inherited. Figure 6 shows the result obtained.

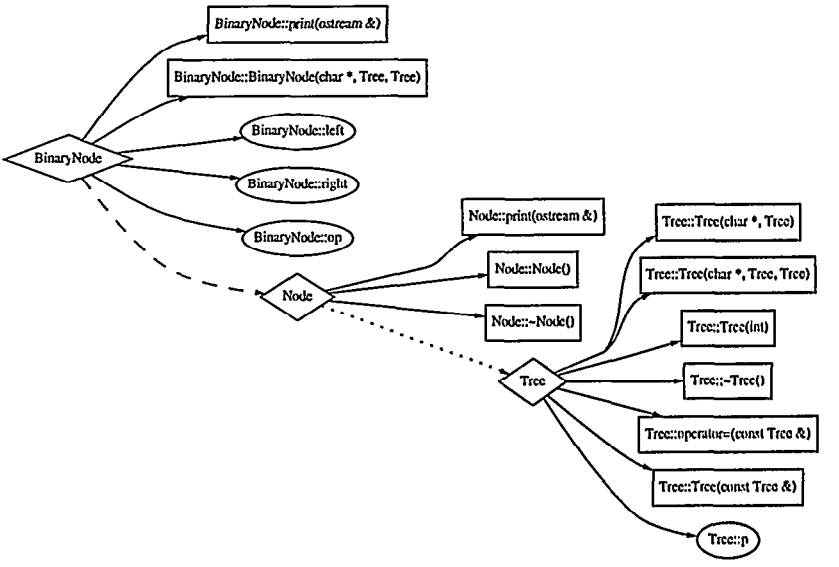


**Fig. 6.** Visibility analysis of `BinaryNode` and `Node`. The subgraph in the lower right is visible only to members of `Node`, not `BinaryNode`. A dashed edge represents an inheritance relationship, while a dotted edge represents a friendship relationship.

# 4 IMPLEMENTATION

We have implemented a system called Acacia that implements the data model described above. This system consists of a collection of tools for analyzing C++ source, plus an instantiation of the CIAO software visualization system[5] based on our C++ model. Acacia uses `cql`[13] for query and closure computations, and `dot`[14] for automatic graph layouts. In this section, we briefly describe the implementation of the major components in Acacia.

## 4.1 Repository Creation

We built the C++ database extraction tool using the Edison Design Group's (EDG)[11] compiler front end. The front end preprocesses, parses, and type-checks the source, producing a fairly detailed representation, essentially corresponding to a high-level abstract syntax tree, in an intermediate language. In addition to dynamic semantic information, this representation also contains declaration, file and source data. This latter detail is crucial for constructing a source level view of the code.

Given the intermediate language representation, the extraction tool traverses the data structure to generate entities for all source items that are not nested within a function scope. After creating the entities, the extraction tool then performs a second pass over the intermediate language representation to generate the required relationships. For C++, this analysis must record implicit uses, such as calls to copy constructors, destructors, and assignment or cast operators, that are not syntactically evident but are generated due to C++ semantics.

The extraction tool produces a repository based on the information within a single source file, analogous to a compiler's producing a single object file corresponding to a single source file. A second tool in the Acacia suite plays a role analogous to the linker, which combines multiple object files into a single executable image. In Acacia, this tool combines the individual repositories into a single repository representing an entire program or subsystem. This integration largely involves replacing references to declarations with references to the corresponding definition, if found.

Since the analysis in Acacia is performed at the source code level, the use of an imported library, without access to its source code, limits the completeness and accuracy of the analysis. Library interfaces, typically specified in source include files, can provide much of the relevant entity and relation information. By the nature of a library, most direct relationships involve external code using something in a library, and not the other way around. When the library does access an entity from a higher level, such as a call-back function, this usually involves the higher level providing a pointer to this entity. Since we cannot analyze how this entity is used within the library, we can only make the conservative assumption that it is used. In the cases when external libraries are involved for which Acacia repositories are not available, definition entities will not exist. In this case, all references to differing declarations of a single entity are replaced by a reference to a single, representative declaration entity.

### 4.2 Instantiation of CIAO for C++

The query and visualization subsystem of Acacia is built by constructing a C++ instance of the CIAO system[5] using an *instance compiler* that takes a specification file for a new language or document type and generates the complete query and visualization environment automatically.

The specification file has five sections:

- *schema*: It maps our data model to the physical *cql*[13] database schema by enumerating the entity and relationship fields. Typically, a field is either an integer or string, but a data type of *entity pointer* allows an entity record to refer to another entity. For example, the *parent type* field of a member entity stores the entity id of its parent class.
- *database view*: This section defines how different entity and relationship records are to be presented as a query result in CIAO's database mode. Each entity kind can have a customized format. For example, the printed name of a member entity is *parent_name::member_name*. If an entity is a template

instance, the template arguments are attached after the name. Otherwise, the name is just the plain name of the entity. This is important in C++ since it is very common to have members of the same or different classes to share a common name due to operator overloading, redefining of inherited methods, etc.

- *source view*: The third section defines what fields are needed to locate the source file and position within that file where an entity appears. A standard CIAO source view tool can use these pointers to output the actual text.
- *graph view*: The fourth section defines how to represent each entity and relationship when CIAO displays the results of a query as a graph. Entities are represented as nodes of various shapes, colors, and fonts. Relationships are represented as edges of various edge styles and colors.
- *GUI front end*: The fifth section defines the appearance and functionality of the graphical front end. It also defines which queries are appropriate for each kind of entity. For example, in the C instance of CIAO, the query *incl* is marked as only being appropriate for file entities.

Our specification file for C++ consists of only 284 lines. The complete suite of query, visualization, and generic reachability analysis tools in Acacia was generated from this specification file. Only dead code detection and visibility analysis tools require special *cql*[13] query code to handle customized closure computations.

# 5 PERFORMANCE AND EXPERIENCE

This section examines speed and storage requirements of our tools on some sample test code and reports our experience in applying Acacia to a C++ software project.

## 5.1 Storage Requirements and Speed

To evaluate the storage requirements and speed of our C++ database generator, we compared it to our local C++ compiler, which is also based on EDG's front end.

The sample C++ source program consists of 1525 lines of C++ code (including header files), with a total size of 42.3 KB. It took 1.47 CPU seconds (sys time + user time) to compile this program on an SGI Challenge L server (with four 200 MHZ MIPS R4400 processors)running IRIX 5.3. Our C++ database generator spent 1.51 seconds on the same piece of code. So its speed is roughly comparable to the C++ compiler. The database consists of 756 entity and 796 relationship records with a total size of 94.9 KB, which is roughly 2.2 times the size of the program source. This storage and speed overhead is rather low compared to many CASE vendor tools that we have seen and is acceptable to the software projects that we have been working with inside AT&T.

### 5.2 Experience on a C++ Software Subsystem

This C++ software subsystem is part of a telecommunications system. The software subsystem was merged from two previous and similar projects and is expected to have a significant amount of unnecessary code.

The system consists of 202 source files and a total of 41,821 lines of C++ code. The C++ database we generated consists of 2,878 C++ functions, 3,208 variables, and 791 types, which include 276 C++ classes. There are no templates used in this project. The program database consists of 7,649 entity records (definitions and declarations), and 9,260 relationships. The size of the database is 1.02 MB.

We picked a user-defined class that deals with alarm transmission and ran the reachability analysis tool to see how large its closure set can be. The result shows that it can reach 241 entities, including 92 functions, 20 types, and 117 variables, defined or declared in 11 separate source files. All these entities must be collected just for the alarm transmission class to compile if it is to be reused in a different project. The closure computation took only 1.58 CPU seconds to run on a desktop SGI Indy (150 MHZ R4400 processor) running IRIX 5.3.

As an example of impact analysis, we ran a reverse reachability analysis to find out how many C++ entities would be directly or indirectly affected if we change the implementation of a function that gets an application message. The reverse closure consists of 332 entities, including 217 functions, 35 variables, and 39 types distributed in 40 different files. The reverse closure computation took only 2.50 cpu seconds to run on the same SGI desktop.

These results show how difficult it might be, without the assistance of automatic analysis tools, for a programmer to reuse a software component or to track down how far-reaching the impact of a software change could be.

## 6 SUMMARY AND FUTURE WORK

The growing body of C++ code and its language complexity have been presenting challenging maintenance tasks and generating research opportunities in the software engineering community. Reachability analysis is the fundamental building block that supports many complex analysis tasks such as dead code detection, software reuse, and selective regression testing. This paper presents the data model of a C++ software repository and discusses how design decisions made in our C++ model affect variants of reachability analysis. It is crucial that the model be complete at the selected level of abstraction so that the analysis can be performed accurately. Efforts in our implementation of Acacia were greatly reduced due to two factors: the use of an EDG's mature compiler front end and the new instance compiler that generates the complete query and visualization environment from a small specification file. Due to the entity granularity and tradeoffs we selected, the performance and storage overhead of our implementation is quite acceptable to real software projects. We feel our work and experience on the C++ model can benefit future repository builders for other object-oriented languages such as Java, Eiffel or Ada 95.

# 7 AVAILABILITY

Acacia is available for experiments to educational institutions. Please visit

*http://www.research.att.com/sw/tools/Acacia*

for information on how to obtain the package.

## References


1. R. S. Arnold. Software Reengineering: A Quick History. *Commun. ACM*, 37(5):13–14, May 1994.
2. E. Buss, R. D. Mori, W. Gentleman, J. Henshaw, J. Johnson, K. Kontogianis, E. Merlo, H. Müller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. TIlley, J. Troster, and K. Wong. Investigating Reverse Engineering Technologies for the CAS Program Understanding Project. *IBM Systems Journal*, 33(3):477–500, 1994.
3. P. P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, Mar. 1976.
4. Y.-F. Chen. Reverse engineering. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 6, pages 177–208. John Wiley & Sons, New York, 1995.
5. Y.-F. Chen, G. S. Fowler, E. Koutsofios, and R. S. Wallach. Ciao: A Graphical Navigator for Software and Document Repositories. In *International Conference on Software Maintenance*, pages 66–75, 1995.
6. Y.-F. Chen, B. Krishnamurthy, and K.-P. Vo. An Objective Reuse Metric: Model and Methodology. In *Fifth European Software Engineering Conference*, 1995.
7. Y.-F. Chen, M. Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334, Mar. 1990.
8. Y.-F. Chen, D. Rosenblum, and K.-P. Vo. TestTube: A System for Selective Regression Testing. In *The 16th Internation Conference on Software Engineering*, pages 211–220, 1994.
9. P. Devanbu. Genoa—a language and front-end independent source code analyzer generator. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 307–317, 1992.
10. P. Devanbu, D. Rosenblum, and A. Wolf. Generating Testing and Analysis Tools with Aria. *ACM Trans. Software Engineering and Methodology*, 5(1):42–62, 1996.
11. Edison Design Group. http://www.edg.com.
12. G. Fowler. A Case for make. *Software – Practice and Experience*, 20:35–46, June 1990.
13. G. Fowler. cql – A Flat File Database Query Language. In *USENIX Winter 1994 Conference*, pages 11–21, Jan. 1994.
14. E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering*, pages 214–230, Mar. 1993.
15. J. Grass and Y. F. Chen. The C++ Information Abstractor. In *The Second USENIX C++ Conference*, Apr. 1990.
16. A. Koenig. An Example of Dynamic Binding in C++. *Journal of Object-Oriented Programming*, 1(3), Aug. 1988.

17. M. Lejter, S. Meyers, and S. P. Reiss. Support for Maintaining Object-Oriented Programs. *IEEE Transactions on Software Engineering*, 18(12):1045–1052, Dec. 1992.
18. H. Müller, M. A. Orgun, S. TIlley, and J. S. Uhl. A Reverse Engineering Approach to Subsystem Structure Identification. *Journal of Software Maintenance*, 5(4):181–204, 1993.
19. G. Murphy, D. Notkin, and E.-C. Lan. An Emprical Study of Static Call Graph Extractors. In *The 18th International Conference on Software Engineering*, pages 90 – 99, 1996.
20. R. Murray. A Statically Typed Abstract Representation for C++ Programs. In *Proceedings of the USENIX C++ Conference*, pages 83–97, Aug. 1992.
21. J. Q. Ning, A. Engberts, and W. Kozaczynski. Automated Support for Legacy Code Understanding. *Commun. ACM*, 37(5):50–57, May 1994.
22. D. Richardson, T. O'Malley, C. Moore, and S. Aha. Developing and Integrating PRODAG in the Arcadia Environment. In *Fifth ACM SIGSOFT Symp. Software Development Environments*, pages 109–119, Dec. 1992.
23. D. Rosenblum and A. Wolf. Representing Semantically Analyzed C++ Code with Reprise. In *USENIX C++ Conference Proceedings*, pages 119–134, Apr. 1991.
24. D. Sharon and R. Bell. Tools that Bind: Creating Integrated Environments. *IEEE Software*, 12(2):76–85, Mar. 1995.
25. I. Thomas. PCTE Interfaces: Supporting Tools in Software-Engineering Environments. *IEEE Software*, 6(6):15–23, Nov. 1989.
26. W. F. Tichy. RCS-a system for version control. *Software – Practice and Experience*, 15(7):637–654, July 1985.
27. K.-P. Vo and Y.-F. Chen. Incl: A Tool to Analyze Include Files. In *Summer 1992 USENIX Conference*, pages 199–208, June 1992.