Detection of Unreachable C++ Code

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}

July 28, 1997

Abstract

Detecting unreachable code in C++ programs is frequently necessary 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 while remaining practical for large systems. We describe such a C++ model with the granularity of top-level declarations and discuss our experiences in applying reachability analysis tools based on this model to industrial C++ code.

1 Introduction

Reachability analysis and dead code detection are two tasks frequently needed in large software projects. Such analyses are used to remove excess software baggage[8], select regression tests[5], and support software reuse studies[4]. We assert that designing the underlying data model is crucial to how effectively these tasks can be performed. In particular, the model must be broad enough to capture all of the relevant dependencies, but not so broad as to make the analysis of very large systems intractable.

Complex, object-oriented languages such as C++ further complicate the construction of adequate data models for reachability analysis. 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 and subtle ways. We present the essentials of a C++ data model and examples to show how effective the model supports dead code detection.

2 A C++ Data Model

Our C++ data model is formulated using Chen's entity-relationship modeling[1] and it supports both the 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 more complete 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[5].

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 $R(e)$ 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 $D(e)$ is simply the difference between $R(e)$ and S, where r is the entity that serves as the starting point of the program execution. $D(e)$ 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. The granularity can be as coarse as a source module or as fine as expressions and tokens.

Our model uses the granularity of top-level declarations. This includes entities for types, functions, variables, macros and files. It captures the principal structural artifacts of a program, especially those used across modules and classes, and allows us to perform the vast majority of reachability analyses pertaining to issues of software engineering. On the other hand, any finer granularity typically imposes a cost of one or two orders of magnitude on the size of the resulting databases, making such models much less useful for large software systems.

Since the focus on this paper is on reachability analysis, we'll discuss only relevant relationships in the model. For a more complete treatment of the model, refer to the ESEC paper[3]. There are several possible relationships in C++: inheritance, friendship, containment, instantiation, and reference relationships. We examine each relationship briefly and explain how it affects reachability analysis.

2.1 Inheritance Relationship

If class A inherits from class B, then class A depends on class B for compilation. An inheritance relationship can be *private*, *protected*, or *public* and each inheritance relationship can be *virtual*. These two pieces of information are also recorded in the database and are useful for some variants of reachability analysis (such as visibility analysis).

2.2 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.

2.3 Containment Relationship

There is a containment relationship between every parent class or struct and each member that it contains. 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.

2.4 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)`.

2.5 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.

3 Detection of Unreachable Code

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.
- *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[7]. 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;
};
```

Now 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 and copy constructor. On the other hand, if we replace the test driver with the following piece of code:

```
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)
```

In the case of dead code detection, computing the simple transitive closure (excluding containment relationships) is appropriate; however, in some other tasks, such as packaging existing components for software reuse, the task also requires computing certain indirect relationships 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, our model explicitly or implicitly contains complete reachability information, so that indirect relationships can be generated using appropriate queries over the database.

Note also 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 our model would consider the function as needed.

We have implemented a system called Acacia that implements such a data model for C++. This system consists of a collection of tools for analyzing C++ source, plus an instantiation of the CIAO software visualization system[2] based on our C++ model. To find out how scalable Acacia is in supporting reachability analysis, we applied it to a C++ software subsystem of a telecommunications project. 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 only 1.02 MB. The database generation time from the Acacia parser is roughly equivalent to that of our C++ complier; both are based on EDG[6]'s compiler frontend.

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. On the other hand, all other entities are considered irrelevant for the purpose of alarm transmission. The closure computation took only 1.58 CPU seconds to run on a desktop SGI Indy (150 MHZ R4400 processor) running IRIX 5.3.

Based on this and our previous experiences on *incl* [8] and *TestTube* [5], two C analysis tools using a similar model and applied to million-line telecommunications projects, the granularity of top-level declarations appears to be reasonable for typical software enginering tasks and allows fast analysis even on large and complex code.

4 Summary and Future Work

The growing number and size of C++ projects have been presenting challenging maintenance tasks in the software engineering community. Reachability analysis is a fundamental process that supports many complex maintenance tasks such as dead code detection, software reuse, and selective regression testing. We have presented a C++ data model that has shown to be effective in performing reachability analysis on large, real-world C++ projects.

Future work will involve adopting the evolving features of the C++ language, such as namespaces and exceptions. In addition, we feel this approach can be applied to similar object-oriented languages such as Java, Eiffel and Ada 95.

References

[1] P. P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, Mar. 1976.

[2] 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.

[3] Y.-F. Chen, E. Gansner, and E. Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Sept. 1997.

[4] Y.-F. Chen, B. Krishnamurthy, and K.-P. Vo. An Objective Reuse Metric: Model and Methodology. In *Fifth European Software Engineering Conference*, 1995.

[5] 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.

[6] Edison Design Group. http://www.edg.com.

[7] A. Koenig. An Example of Dynamic Binding in C++. *Journal of Object-Oriented Programming*, 1(3), Aug. 1988.

[8] K.-P. Vo and Y.-F. Chen. Incl: A Tool to Analyze Include Files. In *Summer 1992 USENIX Conference*, pages 199–208, June 1992.