Ciao: A Graphical Navigator for Software and Document Repositories

Yih-Farn R. Chen
Glenn S. Fowler
Eleftherios Koutsofios
Ryan S. Wallach

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
{chen,ek,gsf,rsw}@research.att.com

Abstract

Programmers frequently have to retrieve and link information from various software documents to accomplish a maintenance task. Ciao is a graph-based navigator that helps programmers query and browse structural connections embedded in different software and document repositories. A repository consists of a collection of source documents with an associated database that describes their structure. Ciao supports repositories organized in an architecture style called Aero, which exploits the duality between a class of entity-relationship (ER) databases and directed attributed graphs (DAG). Database queries and graph analysis operators in Aero are plug-compatible because they all take an ER database and produce yet another ER database by default. Various presentation filters generate graph views, source views, and relational views from any compatible ER database. The architecture promotes the construction of successively more complex operators using a notion of virtual database pipelines. Ciao has been instantiated for C and C++ program databases, and program difference databases. The latter allows programmers to explore program structure changes by browsing and expanding graphs that highlight changed, deleted, and added entities and relationships. The unifying ER model under ciao also allows users to navigate different software repositories and make necessary connections. We have linked program difference databases and modification request (MR) databases so that users can investigate the connections between MRs and affected entities. Ciao has been applied to several large communications software projects and we report experiences and lessons learned from these applications.

1 Introduction

Software maintainers frequently face the need to retrieve and link information from various software documents to accomplish a task. The task is often accomplished without adequate tools to explore or preserve the original software structure. The situation is analogous to maintaining a complex building with missing blueprints. This paper describes *ciao*, a graph-based navigator that helps large software projects to regenerate their software blueprints and trace architecture evolution.

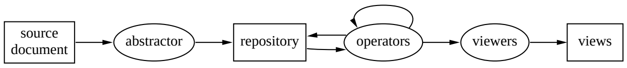

Figure 1: Repository-Based Reverse Engineering

Ciao uses a repository-based reverse engineering approach[6]. A software repository consists of a collection of source documents[1] and an associated database that describes the components and relationships in those documents. The repository allows various abstractions to be generated directly without repeated processing of the original software documents. Figure 1 shows a high-level view of the repository-based reverse engineering process.

The abstractor maps a software document to a repository according to a data model. Each operator retrieves necessary information in the repository to compute a particular abstraction, which is either passed to viewers to generate views, stored back in the repository, or passed to yet another operator for further computation. The feedback links from operators to the repository and other operators allow programmers to build successively higher levels of abstractions. The separation of operators and viewers makes it easy to generate multiple views on the same abstraction.

Section 2 briefly describes *Aero*, an architecture style[26][17] that follows this approach with an em-

[1] In this paper, a source document refers to any software document, including, but not limited to, source code, configuration management files, modification requests (MRs), and manuals.

phasis on converting source documents to a database using the Entity-Relationship model[4]. We shall use instantiations of *Aero* repositories for C and C++ to demonstrate the capabilities and interoperability of *Aero* operators.

Ciao is a navigator that can be instantiated to work with various *Aero* repositories. Software reuse at this architecture level simplifies the construction of graphical navigators for different repositories. Section 3 uses a sample *ciao* session to demonstrate how users formulate queries, generate graphs, interact with graph nodes, and perform various graph analysis tasks. Section 4 describes instantiations of *ciao* to examine program structure differences. It also discusses how the information can be linked to modification request (MR) databases to establish connections between MRs and program entities.

Integrating software tools with an underlying repository has been recognized as a promising approach[30][31]. However, there has been a lack of experience reports on applying repository-based reverse engineering tools to large legacy code, where the tools are needed most. Section 5 describes our experience in applying *ciao* to several software projects. Some of these projects involve millions of lines of code and have existed for nearly a decade. Finally, Section 6 concludes with a summary and outlines our plan to apply *ciao* to other application domains.

2 Aero: An Architecture Style for Repositories

While repository-based systems have been available for examining or reengineering software written in various languages[1], there is not a clear agreement on how the repository should be organized. One popular approach is to store variants of complete parse trees in the repository, such as those used in Reprise[28], ALF[23], Genoa[12] Cobol/SRE[24], and the IBM program understing project[3]. Because of the nature of the representations, tree traversal routines are frequently used to generate various abstractions.

The other popular approach, which we adopted in the construction of the C Information Abstraction System (CIA)[7][9] and its extension for C++[18], and elsewhere in XREFDB[22], is to structure the repository as a relational database so as to reuse the large body of existing database languages and tools to manage the repository. Figure 2 shows an architecture style called *Aero* that characterizes our system as an expansion of the general repository-based reverse engineering architecture shown in Figure 1.

The word *Aero* stands for four primary concepts in the architecture style: *Attribute, Entity, Relationship*, and *Operator*. The style emphasizes an open architecture with reusable operators that consume and produce sets of entities and relationships in standard forms. Such a *plug-compatible* architecture allows Unix pipes and shell scripts[2] to easily connect operators recursively to provide different levels of abstractions. Moreover, new databases are created when necessary to store computation results so as to use a new level of query, analysis, and visualization services.

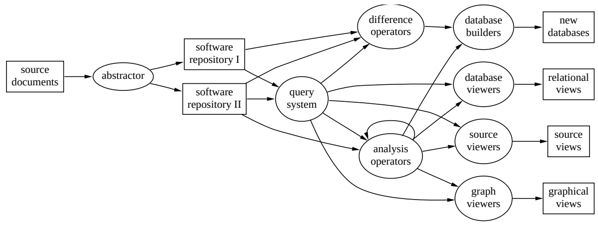

Figure 2: Aero: An Architecture Style for Software and Document Repositories

This architecture is language-independent and consists of three major components: Repositories (with associated abstractors and query systems), Operators, and Viewers. We examine each component in the following sections and introduce output representations of the operators and viewers as they appear in *ciao*.

2.1 Repository

A repository consists of a collection of source documents and their associated database. The abstractor maps source documents to the database according to their corresponding data model. Our C and C++ versions of the abstractor act like a compiler except that they create a database rather than an object file. The similarity has allowed us to maintain the repository automatically using most configuration management systems such as *nmake*[14]. The abstractors store five kinds of program entities: functions, global and static variables, macros, types, and files, and their interrelationships in the databases. Each entity carries a set of attributes to describe, among other things, its data type, storage class, location, and a checksum computed from its token sequence. The checksum attribute allows us to perform fast program comparison at the entity level and constructs the difference database described in Section 4.

The query system associated with the repository consists of queries that retrieve a subset of entities and relationships from the database, respectively, according to a given specification. We use *cql* [15], an efficient interpretive query language, to implement all our queries. A query specification may involve logical connectives and regular expressions to constrain the attribute values. For example, a typical maintenance query with the following specification can be run on the database created for a graph editor:

> *Retrieve functions defined in* `menu.c` *that match the name pattern "*`menu*`*", refer to the variable* `display`*, and return a data type of* `int`.

The query can be formulated as a command with selection clauses that limit entity and relationship attribute values. Alternatively, it can be represented in a simple *ciao* query table shown in Figure 3. The query returns three function entities `menu_align`, `menu_middle`, and `menu_left` and their associated relationships with `display`. The query result is struc-

general	queries	db mode	help

name:	menu*	display
kind:	function	variable
file:	menu.c	
dtype:	int	
sclass:		
def:		
id:		
rkind:		

Figure 3: A query formulated in a *ciao* query table

tured as a *virtual database*, an archive format supported by *cql*. The virtual database consists of an entity section, a relationship section, and a directory section that describes the offset, length, protection mode, and timestamp of each section (portions of the database, represented in "...", have been omitted for brevity):

```
;vdb;CIAO for cc: 1.0
ENTITY
2348;menu.c;f;2348;;n;1;9133;1554;df;51ae87fc;n
4491;display;v;4435;Display *;g;189;0;189;df;517a8141;y
2421;menu_left;p;2348;int;g;804;805;871;df;ed661571;y
...
RELATIONSHIP
p;2421;v;4491;862@863@866;;;
...
DIRECTORY
ENTITY;13;320;MODE=0444;DATE=795475170
RELATIONSHIP;346;94;MODE=0444;DATE=795475170
DIRECTORY;0000000450;0000000116
```

By combining both the entity set and relationship set in a single archive file, the virtual database becomes a standard data exchange form that can be sent on a pipeline. The retrieved virtual database is a subset of the original database and can be presented by various viewers, including the database viewer, the source viewer, and the graph viewer, all explained in the next section.

2.2 Viewers

There are three classes of basic viewers that correspond to the three major representations of the document information: relational views, source views, and graph views.

A database viewer simply takes a virtual database and presents its relational view. Piping the query result in the last section to a database viewer gives us the following view, in which `p` stands for function and `v` stands for variable:

```
$ query_op | database_viewer

k1 file1    name1         k2 file2    name2
== ======== ============= == ======== =========
p  menu.c   menu_align    v  main.c   display
p  menu.c   menu_middle   v  main.c   display
p  menu.c   menu_left     v  main.c   display
```

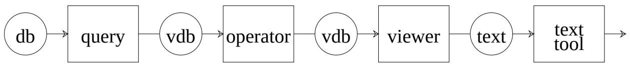

Figure 4: A pipeline with virtual databases sent on the pipe

Here we introduce the notion of a *virtual database pipeline*, where an operator pipes its output database to another operator to create new views or abstractions. In this paper, we use the Unix shell prompt character "`$`" to specify the beginning of a pipeline and the "`|`" character to denote the pipe connector. Both queries and operators can be applied to either the original database or a virtual database on the pipe. A sequence of operators and queries, mixed in any order, can be connected on the pipe, as long as they both take and generate a virtual database. Output generated by viewers can be piped further to regular text processing tools to get line counts or other information. Figure 4 shows a typical virtual database pipeline.

Piping a virtual database to a relationship source viewer gives us the source lines where relationships occur. Alternatively, an entity source viewer can be used to view the source of the function or variable definitions. The following is an example of a relationship source view, which shows all the source lines where the matched functions refer to the variable `display`. Note that each function may refer to the same variable in multiple places.

```
$ query_op | source_viewer

<menu.c:menu_left>
000862; XSetFunction(display, menu_gc, GXcopy);
000863;
XCopyPlane(display, save_just_pm.pm, menu_sw, ...
000866;
XCopyPlane(display, dot_pm.pm, menu_sw, menu_gc, ...
<menu.c:menu_align>
...
<menu.c:menu_middle>
...
```

The same virtual database can also be visualized as a graph. There are two steps involved. First, a graph builder maps the ER database to a graph description, then a layout tool maps that description to an actual graph drawing. In our system, we use *dagger*[5] to map each C or C++ program entity to a node and each relationship to an edge, both decorated with associated display attributes. We then pass the graph description to *dot*[16], an automatic layout tool, or *dotty*[21], a graph browser. Figure 5 shows the graphical view of the same relationships above and is generated with such a pipeline:

```
$ query_op | graph_builder | layout_tool
```

The above examples demonstrate the advantage of separating *information retrieval* from *information presentation*: multiple views on the same set of data are obtained through connections of plug-compatible operators and data viewers. The same database pipeline

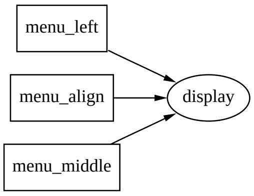

Figure 5: A simple graphical view

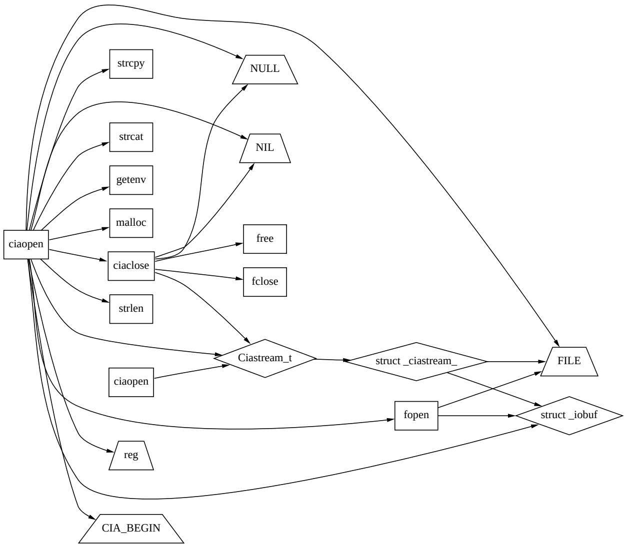

Figure 6: A Reachability Graph rooted at the C function `ciaopen`

can be used to generate a large variety of graphs simply by changing the query specification.

2.3 Operators

Each analysis operator provides an abstraction that can be visualized or used by other operators to build more complex abstractions. One abstraction that is reused by many other tools is the reachable set provided by our *closure* operator. It computes all the entities and relationships that are reachable starting from a specified set of *root* entities. *Closure* also emits a virtual database, so the same set of queries, operators, and viewers can be applied to its output. For example, Figure 6 is a reachability graph generated with the following pipeline:

```
$ closure_op | graph_builder | layout_tool
```

The picture shows all program entities reachable from a C function `ciaopen`. To distinguish between different entity kinds, functions are mapped to boxes, variables to ovals, types to diamonds, macros to trapezoids, and files to parallelograms.

Several operators are built on top of *closure*: The *deadobj* operator uses *closure* to detect unreachable program entities. *TestTube*[10], a system for selective regression testing, uses *closure* and another abstraction computed by the differencing operator (see Section 4) to help find test cases that need to be rerun after a program change. The *focus* operator uses *closure* to retrieve only selected levels of reachable sets in both the forward and reverse reference directions of an entity.

If we pipe the output of the closure operator to a source viewer, followed by the application of an NCSL (non-commented source lines) tool, then we get a new tool that counts the NCSL lines of all source code that a root entity depends on directly or indirectly. Such a tool has been used in constructing objective reuse metrics[8]:

```
$ closure_op | source_viewer | ncsl_tool
```

Sometimes an analysis operator builds an internal graph structure from the ER database so as to apply graph algorithms to the data efficiently. One such operator is *incl*[32], a tool that analyzes an include hierarchy to detect header files that are unnecessary for the compilation of a source file. *Incl* first constructs an internal graph directly from the database and then walks the entity reference paths to determine which files are not needed. Such information has been used in certain projects to restructure their header hierarchies. Again, the analysis results can be saved in a virtual database and displayed graphically.

Many analysis tools, like *incl*, generate abstractions that, when stored in a database or fed back to the original one, will speed up future queries that compute more complex abstractions. After a new database is built, we can then recursively apply the architecture design in Figure 2 to instantiate query and visualization operators for the new database.

We have examined all major components in the Aero architecture style and discussed the use of database pipelines to construct new abstractions or views. The difference operator is discussed in detail in Section 4. While our system was initially built for C and C++ programs, the underlying architecture has been applied to other forms of software documents, including the difference database and the modification request (MR) database described in Section 4. In the next section, we show how the various operators and views presented in this section are intergrated in *ciao*.

3 Ciao: A Graphical Navigator for Aero Repositories

Ciao is a graphical navigator that allows users to query and navigate *Aero* repositories easily. *Ciao* is built on top of *dotty*[21] and inherits all its graph manipulation capabilities and its extensibility through customizable *lefty*[20] scripts. Figure 7 shows the snapshot of a *ciao* session. Users first interact with *ciao* through a main query table shown on the right. A query is formulated by specifying constraints on attribute values. Each formulated query retrieves a set of entities and relationships. The user then decides

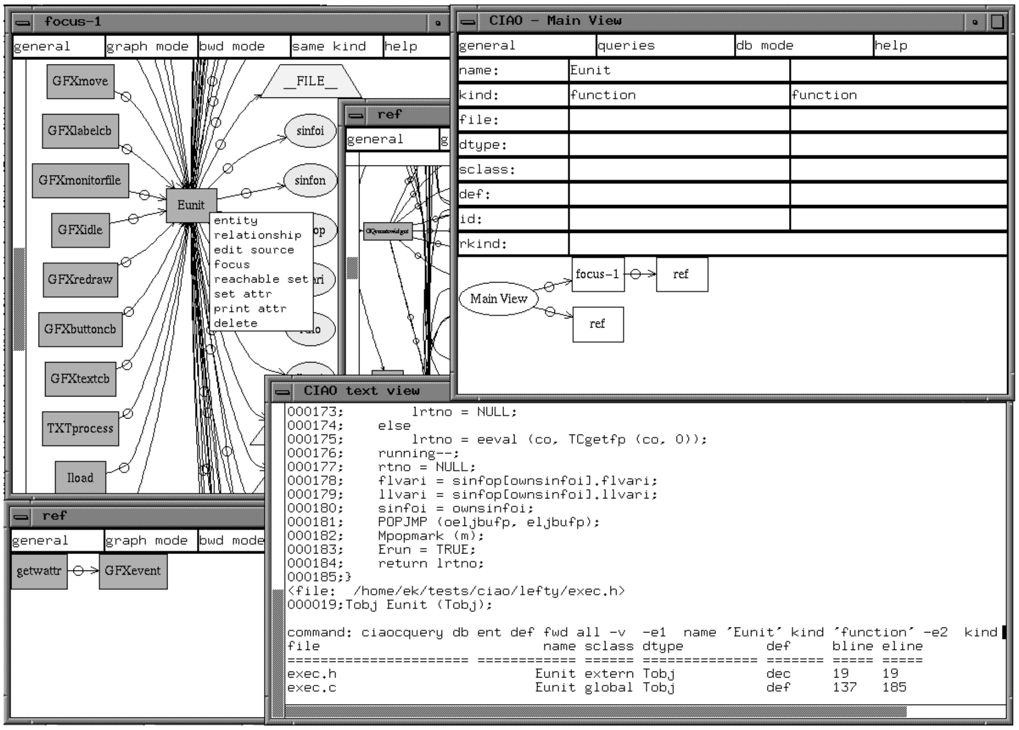

Figure 7: A Snapshot of Ciao for C

what *operator* and *viewer*, determined by the three presentation modes (graph, text, and database), to apply to that set to produce a desired view. This allows users to generate multiple views on the same query result. Several views were generated on the entity `Eunit` in this picture: the leftmost graph shows all entity interactions with `Eunit` by using the *focus* operator under the graph mode, the `text view` window in the center of the snapshot shows both the scrollable source text and formulated database records of `Eunit`. The *focus* graph identifies all the entities that refer to `Eunit` and all entities that `Eunit` depends on. Focus graphs are used heavily during the navigation of a program's static structure because they provide quick access to entities related to a focal point.

After a directed graph is created, users can invoke queries and operators directly from each node in the graph. A pop-up menu is attached to each node listing only the legal operators that can be applied to the corresponding entity. For example, the *incl* operator (see Section 2), can only be applied to file nodes, while the *relationship* operator that shows references to and from a particular entity, can be applied to any entity. Operators applied to an entity may generate more graph windows for further interactions. Direction of a relationship query can be specified as either *forward* or *backward*, indicating that the entity is considered a *parent*, or a *child*, respectively, in a relationship. The graph window in the bottom left of the snapshot was created by selecting a *reverse relationship* operator under the graph mode at a `GFXevent` node in the focus graph.

As many graphs can be generated easily from *ciao* and the user may lose control over what has been generated, a *navigation* graph, similar to the Meta-Graph in ICUE (also based on CIA)[29], under the query table shows the derivation of all the open graph views. Menus attached to the nodes in this graph allow the user to bring the corresponding graphs to the foreground or delete them directly. *Ciao* also maintains a cache of all the queries formulated and their associated graphs. The cache allows previously generated graphs to be retrieved instantly.

The architecture of *ciao* is language-independent and can be instantiated to work with languages other than C. Figure 8 shows a C++ instantiation with a type inheritance graph. Note that the main query table has more entity and relationship attributes, but the style of the interface remains the same.

4 Examining Structure Differences with Ciao

A difference operator of an *Aero* repository (see Section 2) takes two ER databases or corresponding subsets and performs a structure comparison. Unlike text differencing tools, which produce output that does not permit easy association with program entities, our C program differencing tool[7] detects which entities and relationships have been deleted, added, or changed, and produces a difference database. A C++ version of the program differencing tools has also been implemented[19]. Moreover, all differencing tools follow the same style of query, closure, and visualization tools of their corresponding operators for C. For ex-

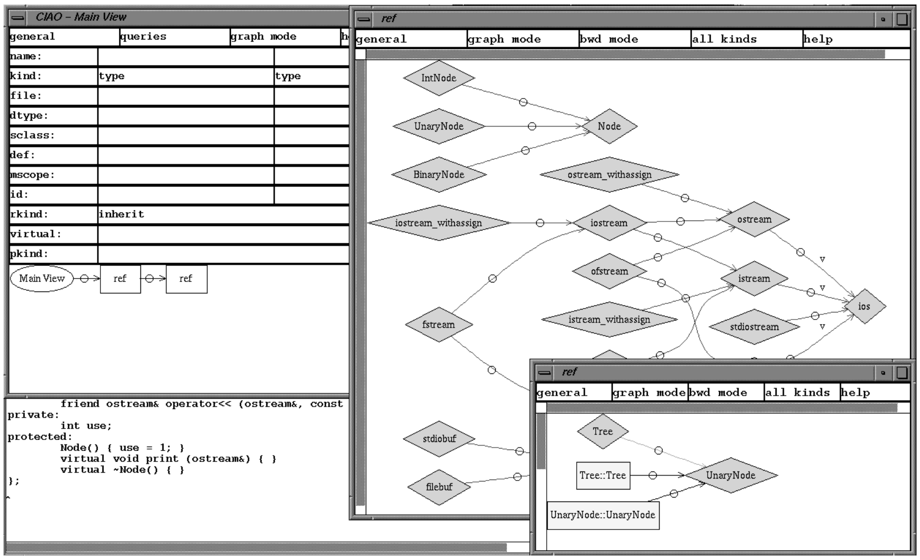

Figure 8: A Snapshot of Ciao for C++

ample, the following query result shows how entities in the file `incl.c` were changed, deleted, and added from one version to the other.

```
$ diff_query_op | diff_database_viewer
    tag kind             file             name
======= ======== ============ ================
changed file           incl.c           incl.c
changed function       incl.c           subsys
deleted function       incl.c         dagprint
changed function       incl.c          exprint
  added macro          incl.c           DELETE
changed macro          incl.c          DEQUEUE
  added macro          incl.c           INSERT
   same function       incl.c        symbolcmp
   same variable       incl.c           Maxlev
  added variable       incl.c         Nodelist
deleted variable       incl.c            Queue
...
```

Figure 9 shows an example *ciao* session in progress on a difference database. The query table has been set so that the relationships between `mkgraph` and all other functions in the program are displayed. The window at the bottom right of the figure shows a portion of the output of this query in the form of formatted records from the difference database. Relationships for both versions of the program are shown; the `tag` indicates whether the relationship was deleted in the new version, added to the new version, or exists in both versions. The window on the left side of the figure is the result of the same query in graphical form. Here, differences in the entities between versions of the program are indicated by the color or shading of the graph nodes. Green (or light grey) indicates the function was unchanged, yellow (or medium grey) means the function changed between versions, and red (or dark grey) indicates that the function was added in the new version. The relationship differences are indicated by the kind of edges connecting the nodes. A solid line indicates the relationship is present in both versions, a dotted line indicates the relationship is present only in the first version, and a dashed line indicates the relationship was added in the second version. Once a graph has been displayed, queries may be run from any of the nodes. The figure shows how the user can select the *reachable set* (closure) query from the node for the function `ciaopen()`. The reachability graph generated by this query can be automatically merged with the original graph if the query is run under inplace mode. The merged graph, generated from another window showing the functions reachable from `mkgraph()`, is shown in the window on the right. Relationships between all types of entities are shown. The white color (or shading) of function `t_search` indicates that it was deleted in the second version of the program. By gradually expanding graphs using the *inplace* mode, programmers can effectively explore and visualize the structure changes of selected entities and relationships. An operator "*diff old & new*" is also provided so that users can visualize the textual differences of the two versions of an entity.

In industry, most large software development projects use change management systems to control access to source files and to provide version control. One method of providing version control for a project is to use a system such as SCCS (Source Code Control System)[27], which stores an original version of a file and each incremental change (or delta) made to the file. SCCS requires developers to check out a file for editing and check it in when they have made their changes. Each checked-in version of a file is assigned a version ID number by the system. When a particular

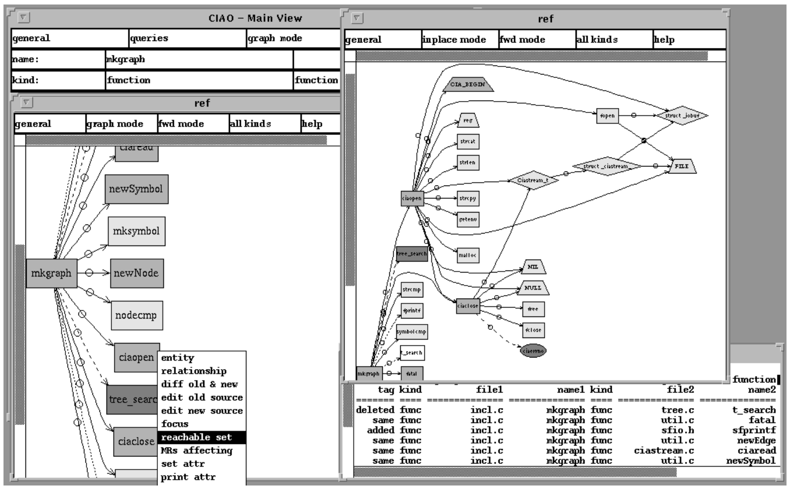

Figure 9: A Snapshot of *Ciao* for a sample relationship query on program differences

version of the file is desired, the system retrieves the original version and applies the appropriate deltas to it.

SABLIME[11], a change control system used in AT&T, provides a front end to the SCCS system whereby problem descriptions can be linked to the software changes used to fix them. Each Modification Request (MR) stored in SABLIME contains user-defined fields to specify why the software change occurred, the names of the files which were changed in conjunction with it, and the information needed for SCCS to retrieve the version of the file that was created as a result.

In *ciao*, program databases and MR databases are linked through the "*MRs affecting*" query, which maps the selected entity onto its resident file, then queries the change management system for MRs which affected that file. Figure 10 shows how *ciao* can be used this way to browse entities and relationships in a software change control system. In this graph, MRs, source files, and versions of those files stored in SCCS files are represented as graph nodes, and relationships between the entities are shown. Three modification requests have affected the file `src/cmd/put_info.c`, and the system has created three SCCS versions in two parallel branches (SABLIME keeps an official branch, in which new SCCS versions are created only after an MR has been placed in a particular state in the database, and an MR branch, in which new SCCS versions are created when the individual files are checked in after editing). The SCCS version ID and the MR that created it are shown in each SCCS version node. Menus are associated with each entity, allowing the user to add information about related entities to the graph. The menu associated with an MR is shown. The user can view the text of the MR from the change control system's database, add other files touched by the MR to the graph, or show the MR's relationship to

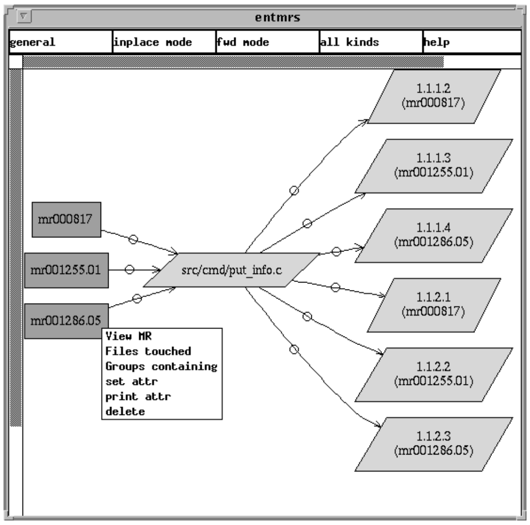

Figure 10: Graphical view of information stored in a change control system.

user-defined groups of MRs in the system. For example, users can find other files that are touched by MR `mr001255.01` and expand the graph further by selecting the "*Files touched*" menu item under the inplace graph mode.

The program difference technology is a critical first step in building a *software history* database that helps us study the *aging* of legacy code. By associating changes in program entities and relationships to feature enhancements or bug fixes, we have a better hope of detecting *ignorant surgeries* to the code[25] and to locate entities that are likely to be affected (again) when a new MR is submitted.

5 Experience and Performance

In this section, we describe the set of design decisions that either have contributed to *ciao*'s success or hindered its development. We also provide performance figures to demonstrate the feasibility of our implementation of the *Aero* architecture style.

5.1 The Entity-Relationship Model

The design of our system has benefited repeatedly from the entity-relationship model, which not only helped us cut through the complex C and C++ syntax to get a clean structure model, but also helped us to unify all the tool interfaces. The duality between our entity-relationship databases and directed attributed graphs allowed us to visualize and apply graph algorithms to the code structure through simple transformation steps. Moreover, operating at the ER database level allows us to treat different languages in a uniform way in their corresponding query and analysis operators. For example, instead of creating new and separate operators to compute the C++ inheritance hierarchy (or friendship) graph shown in Figure 8, the same pipeline that creates a C type dependency graph is reused. Only a *selection clause* is added in the query specification to limit the type to type relationship to inheritance (or friendship). The only language-specific processing is the mapping of virtual inheritance relationships to edges properly labeled with "v", This can be done through the specification of a small graph decoration database. Our experience shows that reusing existing operators with slight adaptations requires a carefully-designed data model that leaves most of the language-specific tasks to the abstractor.

5.2 Database Overhead and Query Performance

Although hardware performance has improved significantly in the past decade, our experience strongly suggests that effective use of storage space and adequate query performance remain critical to the acceptance of repository-based reverse engineering tools in large software projects.

To help evaluate the database overhead, we consider the following metrics for the C language: source size (S), database and index size (D), entity set size (E), relationship set size (R), expansion factor (D/S), and connectivity factor (R/E). Our metrics tool, *ciastat*, collects these numbers and other statistics automatically. As an example, Table 1 compares the metrics obtained from three software projects. Note that the expansion factor, the major metrics value concerning the database overhead, varies widely depending on the structure of the program, but it usually stays well below 150%. A relatively low connectivity factor is usually an indication of a large number of unused entities – as we have observed in project C.

One key decision that contributes to the compactness of our program database is a careful tradeoff between entity granularity and the database size. The current database keeps complete dependency information among externally and statically visible program

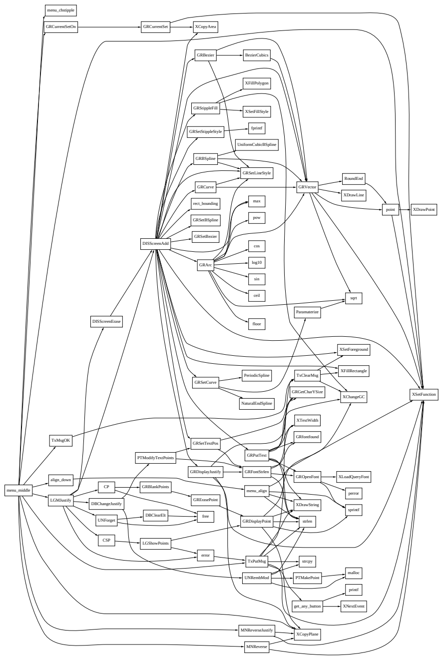

Figure 11: A Closure Graph from Project B

entities while ignoring local declarations and their internal relationships – such information is usually not essential in the context of large software projects.

With plug-compatible query and analysis operators, the shell script language *ksh* became our primary language for building new reverse engineering tools. For example, both *deadobj* and *focus* are simple shell scripts that use queries and the closure operator to perform their tasks. Since basic queries are used as fundamental building blocks in many scripts, they must run efficiently. As an example, we ran three typical queries (entity, relationship, closure) for each of the three databases listed in Table 1 on an SGI Indy R4400 workstation running IRIX 5.2 and listed the execution time (user + system time) of each query in Table 2.

As the table indicates, even for project C, which involves over 46,000 entities and 47,000 relationships, basic entity and relationship queries return results instantly. The closure computations, while more complex, also return results within seconds. The layout tool *dot* is also very efficient. It took only 1.12 seconds on the same machine to map the closure set (85 entities and 142 relationships) retrieved for project B to the layout shown in Figure 11.

5.3 Virtual Database Pipeline

The virtual database representation used in our current implementation of *ciao* allows us to treat data in a uniform way. Initially, we had two possibilities on a pipeline: an entity (E) set or a relationship (R) set. For example, a closure operator may output ei-

Proj	number of files	number of lines	source size	database size	ent. set size	rel. set size	exp. factor	conn. factor
A	111	20,724	487 KB	356 KB	2,380	3,558	73.1%	149%
B	147	24,415	617 KB	722 KB	4,840	6,635	117%	137%
C	6,228	861,939	13.4 MB	6.37 MB	46,299	47,879	47.5%	103%

Table 1: Source and database size metrics from three software projects

Proj	entity query	relationship query	closure query	closure set size
A	0.13s	0.28s	1.23s	180
B	0.12s	0.19s	0.59s	85
C	0.13s	0.18s	2.80s	531

Table 2: Performance of sample query and analysis tools

ther a reachable E set or R set. It depends on what the next operator expects. By unifying the input and output representations to a virtual database, operators become completely plug-compatible. The default result of a closure operator becomes a virtual database that stores the complete set of reachable E and R sets. Users can then apply any operator, including an entity query, a relationship query, or even another closure operator starting at a different root entity, to the resulting virtual database to retrieve desired information.

5.4 Software Visualization

Selectivity is essential to successful visualization of complex software structures. We frequently deal with graphs that are hundreds of times more complex than the one shown in Figure 11[2]. Fortunately, the virtual database pipeline allows us to selectively retrieve different facets of a software structure using a variety of query, analysis, and visualization operators.

Initially, we used pre-defined functions to map entity and relationship attributes in C and C++ to display attributes such as colors, shapes, and edge styles. This approach was not satisfactory because a programmer might decide to change the mapping to emphasize certain aspects of a program graph. For example, in a general C++ type dependency graph involving inheritance, friendship, and reference relationships, the edge styles (solid, dashed, dotted) may be used to represent these three types of relationships rather than the three kinds of access specifications (public, protected, and private) of inheritance relationships. We have generalized the graph decoration process by using a small, customizable decoration database that specifies the mapping from selected entity attributes to decoration attributes.

While we have presented only database, source, and graph viewers in this paper, additional visualization services such as that provided by *SeeSoft*[13] (by changing *SeeSoft*'s granularity from a line to an entity) can also be built on top of a software database.

[2]An example from an operations systems project can be found on the cover of the July/August 1994 issue of the AT&T Technical Journal.

6 Summary and Future Work

Ciao is an extensible and customizable graphical interface for querying and browsing *Aero* repositories, which store source documents and their associated ER databases. We have demonstrated the flexibility of the *ciao* interface in the navigation of C, C++, program difference, and MR databases. The virtual database is the standard data exchange format between plug-compatible *ciao* operators. Many complex applications and graphical views are built by simply connecting *ciao* operators on a pipeline. Exploring software structure changes and their impact is particularly effective with the graphical navigation facilities provided by *ciao*. Linking the program difference database with the MR database has allowed us to start making connections between software features and actual program entities. Since the *Aero* architecture style is strongly language-independent, plans are under way to build different instantiations of *ciao* for many other types of repositories, including those of shell scripts, World-Wide-Web (WWW) home pages, and business information.

7 Acknowledgements and Availability

Thanks to all those colleagues in the Software Engineering Research Department and Communication Information Systems Research Department who have provided well-designed and reusable software components to make this work possible. Many of the *ciao*-related tools described in this paper can be obtained by visiting the World-Wide-Web site *http://www.research.att.com/orgs/ssr/book/reuse.*

References

[1] Robert S. Arnold. Software Reengineering: A Quick History. *Communications of the ACM*, 37(5):13–14, May 1994.

[2] Morris I. Bolsky and David G Korn. *The Korn-Shell – Command and Programming Language.* Prentice Hall, 1988.

[3] E. Buss, R. De Mori, W.M. Gentleman, J. Henshaw, J. Johnson, K. Kontogianis, E. Merlo, H.A. Müller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S.R. TIlley, J. Troster, and K. Wong. Investigating Reverse Engineering Technologies for the CAS Program Understanding Project. *IBM Systems Journal*, 33(3):477–500, 1988.

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

[5] Yih-Farn Chen. Dagger: A Tool to Generate Program Graphs. In *Proceedings of the USENIX Unix Applications Development Symposium*, pages 19–35, 1994.

[6] Yih-Farn Chen. Repository-Based Reverse Engineering: An Experience Report. In *The Fourth Reengineering Form*, pages 33–1–33–10, 1994.

[7] Yih-Farn Chen. Reverse engineering. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 6, pages 177–208. John Wiley & Sons, New York, 1995.

[8] Yih-Farn Chen, Balachander Krishnamurthy, and Kiem-Phong Vo. An Objective Reuse Metric: Model and Methodology. In *Fifth European Software Engineering Conference*, 1995.

[9] Yih-Farn Chen, Michael Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[10] Yih-Farn Chen, David Rosenblum, and Kiem-Phong Vo. TestTube: A System for Selective Regression Testing. In *The 16th International Conference on Software Engineering*, pages 211 – 220, 1994.

[11] S. Cichinksi and G. S. Fowler. Product Administration through Sable and Nmake. *AT&T Technical Journal*, 67(4):59–70, July 1988.

[12] 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.

[13] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner. SeeSoft – tool for visualizing line oriented software. *IEEE Transactions on Software Engineering*, 11(18):957–968, 1992.

[14] Glenn Fowler. A Case for make. *Software – Practice and Experience*, 20:35–46, June 1990.

[15] Glenn Fowler. cql – A Flat File Database Query Language. In *USENIX Winter 1994 Conference*, pages 11–21, January 1994.

[16] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering*, pages 214–230, March 1993.

[17] David Garlan, Robert Allen, and John Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, December 1994.

[18] Judith Grass and Y. F. Chen. The C++ Information Abstractor. In *The Second USENIX C++ Conference*, pages 265–275, April 1990.

[19] Judith E. Grass. Cdiff: A Syntax Directed Differencer for C++ Programs. In *USENIX C++ Conference Proceedings*, pages 181–193, August 1992.

[20] Eleftherios Koutsofios and David Dobkin. LEFTY: A Two-view Editor for Technical Pictures. In *Graphics Interface '91, Calgary, Alberta*, pages 68–76, 1991.

[21] Eleftherios Koutsofios and Steve North. Applications of Graph Visualization. In *Proceedings of Graphics Interface*, pages 235–245, May 1994.

[22] Moises Lejter, Scott Meyers, and Steven P. Reiss. Support for Maintaining Object-Oriented Programs. *IEEE Transactions on Software Engineering*, 18(12):1045–1052, December 1992.

[23] Rob Murray. A Statically Typed Abstract Representation for C++ Programs. In *Proceedings of the USENIX C++ Conference*, pages 83–97, August 1992.

[24] Jim Q. Ning, Andre Engberts, and W. Kozaczynski. Automated Support for Legacy Code Understanding. *Communications of the ACM*, 37(5):50–57, May 1994.

[25] David L. Parnas. Software Aging. In *The 16th International Conference on Software Engineering*, pages 279 – 287, 1994.

[26] Dewayne Perry and Alexander Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT*, 17(4):40–52, October 1992.

[27] Marc J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering, Vol.SE-1*, 1(4):365–370, December 1975.

[28] David Rosenblum and Alexander Wolf. Representing Semantically Analyzed C++ Code with Reprise. In *USENIX C++ Conference Proceedings*, pages 119–134, April 1991.

[29] Peter G. Selfridge and George T. Heineman. Graphical Support for Code-Level Software Understanding. In *The Ninth Knowledge-Based Software Engineering Conference*, 1994.

[30] David Sharon and Rodney Bell. Tools that Bind: Creating Integrated Environments. *IEEE Software*, 12(2):76–85, March 1995.

[31] Ian Thomas. PCTE Interfaces: Supporting Tools in Software-Engineering Environments. *IEEE Software*, 6(6):15–23, November 1989.

[32] Kiem-Phong Vo and Yih-Farn Chen. Incl: A Tool to Analyze Include Files. In *Summer 1992 USENIX Conference*, pages 199–208, June 1992.