Chava: Reverse Engineering and Tracking of Java Applets

Jeffrey Korn
Princeton University
Dept. of Computer Science
Princeton, NJ 08544
jlk@cs.princeton.edu

Yih-Farn Chen
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
chen@research.att.com

Eleftherios Koutsofios
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
ek@research.att.com

Abstract

Java applets have been used increasingly on web sites to perform client-side processing and provide dynamic content. While many web site analysis tools are available, their focus has been on static HTML content and most ignore applet code completely. This paper presents Chava, a system that analyzes and tracks changes in Java applets. The tool extracts information from applet code about classes, methods, fields and their relationships into a relational database. Supplementary checksum information in the database is used to detect changes in two versions of a Java applet. Given our Java data model, a suite of programs that query, visualize, and analyze the structural information were generated automatically from CIAO, a retargetable reverse engineering system. Chava is able to process either Java source files or compiled class files, making it possible to analyze remote applets whose source code is unavailable. The information can be combined with HTML analysis tools to track both the static and dynamic content of many web sites. This paper presents our data model for Java and describes the implementation of Chava. Advanced reverse engineering tasks such as reachability analysis, clustering, and program differencing can be built on top of Chava to support design recovery and selective regression testing. In particular, we show how Chava is used to compare several Java Development Kit (JDK) versions to help spot changes that might impact Java developers. Performance numbers indicate that the tool scales well.

1. Introduction

The World Wide Web first started with web servers only presenting static HTML content. Later, Common Gateway Interface (CGI) scripts were introduced to run on web servers to dynamically compose content before presenting them to the clients. Recently, Java applets have been used increasingly on web sites to provide rich user interfaces and perform client-side processing to generate dynamic content. While many web site analysis tools [14, 8] are available to analyze the structure of static HTML content, most of them completely ignore the applet code, which by its nature requires software analysis techniques.

Traditional software repositories [29, 30, 7, 13, 3] apply reverse engineering [12] techniques on the source code to build a central information source for maintaining code in a software system. Repositories are useful to developers as they make it possible to efficiently examine the structure and interaction between components of a system without having to delve through potentially hundreds of thousands of lines of source code. Advanced tools have also been built to perform reachability analysis [7], clustering analysis [20], selective regression testing [10] and even extraction of light-weight object models [28, 19].

This paper presents Chava, a reverse engineering and tracking system for Java [1]. The system presented has several noteworthy features:

- **Data Model for both Byte Code and Source Code:** Like Womble [19] and some recent Java tools, Chava can work on binary class files directly. However, unlike other tools with a single-task focus, Chava aims to have a *complete* data model (as defined in Acacia [7]) at the selected abstraction level – class member declaration – to support a wide range of analysis and tracking tasks. It gets additional information (such as line numbers) from source code when it is available.

  Analysis using only class files is possible primarily due to properties of the Java language. Java does not have a preprocessor, which means that we do not have to deal with constructs such as macros, include files, and templates, whose information would not be available in an object file. Also, Java is an architecture neutral language, so its byte code is the same on all machines. This makes it possible to scan through object code in a machine-independent manner to discover relationships in a program.

- **Program Difference Database:** Chava supports differencing of Java program databases. Similar to the work on change detection in Java from University of Waterloo [25], and in the earlier work of *ciadiff* [5] for C and Cdiff [16] for C++, Chava allows tools to examine what changes have been made in two different versions of a system. However, the approach is quite different: Chava can take two previously-built databases and create a difference database with minimal efforts.

- **Integration with HTML Analysis:** Chava can analyze web pages along with the embedded Java applets by combining its database with HTML analysis results created by WebCiao [8], which also uses an entity-relationship model.

To give a quick idea of the capabilities of Chava, Figure 1 shows a sample diagram generated by our tool from the difference database created for JDK1.0 and JDK1.1. The query was

> Show all the methods that referred to any deleted, protected field member in any Java class.

The diagram shows immediately that only one protected field, `PushbackInputStream.pushBack`, was deleted (shown as a white oval) in JDK1.1, and five methods were affected by this change, all in class `PushbackInputStream`. It also showed that all these references have now been removed (represented by dotted edges) in JDK1.1. By doing a reverse reachability analysis for three layers, we see that the method `DataInputStream.readLine`, which refers to two of those methods that used to access the deleted field, is affected by this change as well and should be retested. Note that solid edges indicate relationships that remain in the new version (JDK1.1). Finding correlations between a new software feature and changed program entities and relationships is frequently useful in helping locate problems should they arise after the introduction of the new feature.

2. A Data Model for Java

Our Java Data model is based on Chen's entity-relationship model [4]. Each Java program is viewed as a set of entities, which may refer to each other. Entities exist for each language construct, such as classes, methods, and fields. Relationships between entities encompass notions such as inheritance and method invocation. This section describes in more detail the composition of the entities and relationships.

A property that our model must satisfy is that of *completeness* as described in Acacia [7]. In order for our model to be complete, it must be the case that if the compilation of an entity *A* depends on entity *B*, a relationship between *A* and *B* is in the model. We satisfy this condition with one notable exception. In Java, classes can be loaded and methods can be invoked dynamically at runtime using the reflection API [26]. Programs that do this may not satisfy completeness. Completeness allows us to perform analyses such as dead code detection and reachability.

In selecting an appropriate model, a level of granularity must be chosen. Not enough granularity will prevent a user from being able to make non-trivial queries. However, too much granularity leads to a database that is too large to handle queries efficiently. Our model handles class member declarations. We create entities for all constructs up to this level of granularity in a program, but do not include information down at the level of statements and expressions. That means detailed control flow analysis or pattern matching on program constructs [23] is not available with this level of abstraction.

We will illustrate the model with an example of a simple Java program. Figure 2 contains the source code for a set of classes that implements circles and rectangles. The base class `Shape` is extended to implement `Circle` and `Rectangle`.

2.1. Entity types

Our model handles the following Java entity types:

- *class*: Contains declarations and definitions of a collection of methods and fields.

- *interface*: Interfaces are similar to classes, but do not contain definitions. Classes implement the declarations of zero or more interfaces.

- *package*: A set of classes.

- *file*: Source code that contains one or more classes

- *method*: A function that is part of a class

- *field*: A variable or constant that is part of a class

- *string*: Strings that are referenced by methods or fields.

For example, in Figure 2, we have the following entities:
Classes: Shape, Circle, Rectangle
Interfaces: Cloneable
Packages: graph
Files: Shape.java
Methods: Shape.printArea, Circle.Circle (constructor), Circle.area, Circle.circumference, Rectangle.Rectangle, Rectangle.area, Rectangle.circumference
Fields: Circle.r, Circle.PI, Rectangle.w, Rectangle.h
Strings: `"Area:"`

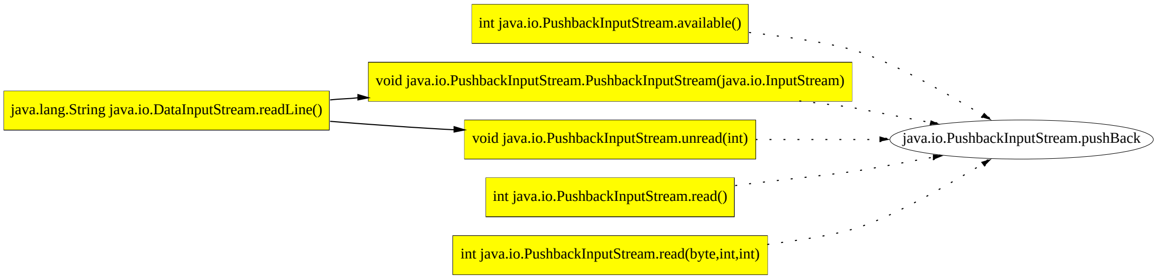
int java.io.PushbackInputStream.available()
void java.io.PushbackInputStream.PushbackInputStream(java.io.InputStream)
java.lang.String java.io.DataInputStream.readLine()
void java.io.PushbackInputStream.unread(int)
java.io.PushbackInputStream.pushBack
int java.io.PushbackInputStream.read()
int java.io.PushbackInputStream.read(byte,int,int)

Figure 1. A Sample Reverse Reachability Diagram from the Difference Database of JDK1.0 and JDK1.1

```
// file name is Shape.java
package graph;

abstract class Shape implements Cloneable
{
        public abstract double area();
        public abstract double circumference();
        public void printArea() {
            System.out.println("Area:"+area());
        }
}

class Circle extends Shape
{
        protected double r;
        protected static double PI = 3.14159265;
        public Circle(double r) { this.r = r; }
        public double area()
                { return PI * r * r; }
        public double circumference()
                { return 2 * PI * r; }
}

class Rectangle extends Shape
{
        protected double w, h;
        public Rectangle(double w, double h)
                { this.w = w;  this.h = h; }
        public double area() { return w * h; }
        public double circumference()
                { return 2 * (w + h); }
}
```

Figure 2. A Simple Java Program

2.2. Attributes

All entities contain the following attributes: *id*, *name*, *kind*, *file*, *begin line*, *end line*, and *chksum*. The *chksum* attribute is a 64 bit integer that can be used to compare two versions of an entity. This attribute is used when comparing two databases that represent different versions of the program.

The model also includes other attributes that only apply to certain kinds of entities. Table 1 summarizes these attributes. They are as follows:

- *parent*: The parent of a method, a field, or a string is its defining class. The parent of a class is the package that contains the class. Packages and files do not have parents.

- *scope*: The scope attribute contains one of private, protected or public. It is used for class entities, which can be public or private, and class members (fields and methods) which can be public, private or protected.

- *flags*: A set of zero or more modifiers for a given object. For methods, modifiers include abstract, final, native, static, and synchronized. For fields, modifiers include final, static, transient, and volatile. For classes and interfaces, modifiers include abstract and final. Flags are represented in the database as a bit vector.

- *params*: For method types, this attribute contains a list of parameters for the method. This attribute is not used by other entity types. It is primarily used to distinguish among overloaded methods under the same name.

- *dtype*: For method types, this attribute contains the return type of the method. For field types, the attribute contains the type name of the field. For classes, this attribute contains the name of the package it is contained in. The full name of a class is constructed by appending the name and dtype attributes of the entity.

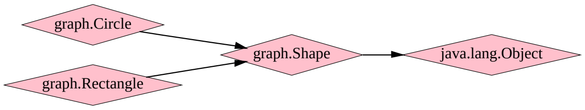

Figure 4. Class inheritance diagram

> The dtype attribute is used in these two differing contexts to preserve space in the database. However, dtype is a package name in one context and a type name in the other, so there is no overlap in the possible set of values for these contexts.

For example, let us consider two entities from our example program. The field `Circle.PI` has as its parent the class `Circle`, its scope is `protected`, its flag is `static`, params is empty and dtype is `double`. The constructor for class `Rectangle` has as its parent the class `Rectangle`, its scope is `public`, flags is empty, params is `(double w, double h)`, and dtype is `void`.

In cases where we are working without source files, some entities will be missing values for the begin and end line numbers. Java class files do not include line numbers for definitions of classes, fields and strings. However, bytecode for methods includes line numbers annotations, which we use to find the begin and end line numbers for this entity type. Entities with missing line numbers do not affect the ability to do most analysis tasks on the code.

2.3. Relationship Types

Our Java data model contains the following relationships:

- *subclass*: Figure 4 shows the subclass relationships that exist in our example program. The class `Shape` has `Object` as a subclass, and both `Circle` and `Rectangle` have `Shape` as a subclass. Arrows go from class to superclass in order to show the direction of dependency and maintain the completeness property.

- *containment*: If a field or method *A* is a member of class *B*, then a containment relationship exists between *B* and *A*.

- *implements*: The implements relationship exists between a class entity *A* and an interface entity *B* if class *A* implements *B*. In the example, `Shape` implements the interface `Cloneable`.

- *field read*: A field read relationship exists between a method entity *A* and a field entity *B* if method *A* reads field *B*.

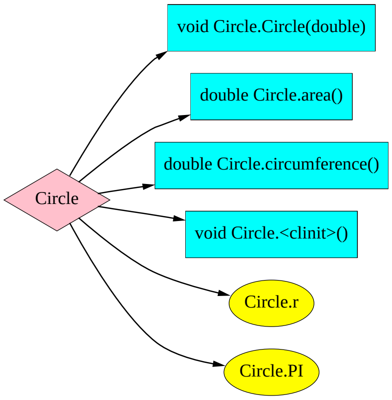

Figure 5. Members of class `Circle`

- *field write*: A field write relationship exists when method *A* field writes to *B*.

- *reference*: Reference relationships exist between two method entities *A* and *B* when *A* invokes *B*.

3. Java Program Analysis

Once we generate a database for a particular Java application, we can then use a number of supporting tools to extract information about the program. This section starts with examples of basic database queries and program visualization, followed by reachability analysis and program differencing.

3.1. Visualization Queries with CIAO

Using a retargetable reverse engineering system called CIAO [6], a user interface can be generated to graph relationships in a Java program. CIAO takes a specification for a language and generates a set of supporting tools for querying and visualizing databases for that language. Instantiations of CIAO exist for a variety of languages including C [9], C++ [7], HTML [8], and ksh.

Going back to our example in the previous section, we can use the Java instance of CIAO to show its class inheritance graph as seen in Figure 4. Another query can be made to the database returning all relationships that exist in the application, as shown in Figure 3. In the graphs, each entity type is drawn with a different shape, color, etc. as defined in the CIAO specification.

If we want to see all of the members of the class `Circle`, we can perform a similar query, returning all relationships that are of the type *containment*. Figure 5 shows the resulting graph.

Type	Parent	Scope	Flags	Params	Dtype
Class	package	public,private	abstract,final	N/A	package name
Interface	package	public,private	abstract,final	N/A	package name
Package	N/A	N/A	N/A	N/A	N/A
File	N/A	N/A	N/A	N/A	N/A
Method	class	public,private,protected	abstract,final,native,static, synchronized	params	return type
Field	class	public,private,protected	final,static,transient,volatile	N/A	type
String	class	N/A	N/A	N/A	N/A

Table 1. Other Attributes

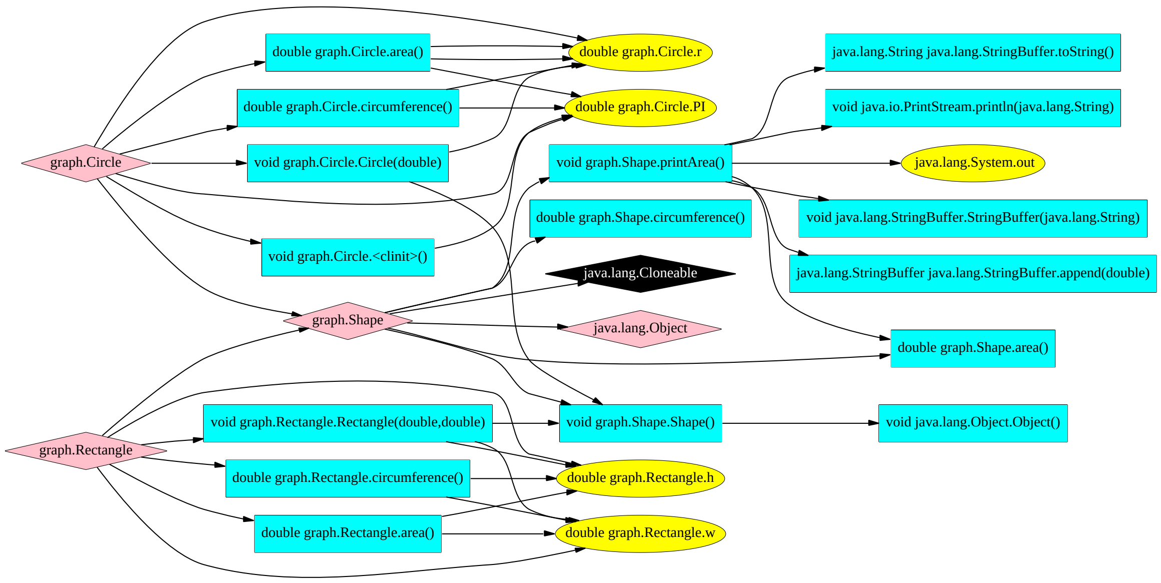
graph.Circle
double graph.Circle.area()
double graph.Circle.circumference()
void graph.Circle.Circle(double)
void graph.Circle.<clinit>()
double graph.Circle.r
double graph.Circle.PI
java.lang.String java.lang.StringBuffer.toString()
void java.io.PrintStream.println(java.lang.String)
void graph.Shape.printArea()
java.lang.System.out
double graph.Shape.circumference()
void java.lang.StringBuffer.StringBuffer(java.lang.String)
java.lang.Cloneable
java.lang.StringBuffer java.lang.StringBuffer.append(double)
graph.Shape
java.lang.Object
double graph.Shape.area()
void graph.Rectangle.Rectangle(double,double)
void graph.Shape.Shape()
void java.lang.Object.Object()
graph.Rectangle
double graph.Rectangle.circumference()
double graph.Rectangle.h
double graph.Rectangle.area()
double graph.Rectangle.w

Figure 3. Relationships in Example Program

3.2. Reachability Analysis

There are two kinds of reachability analysis, *forward reachability* and *reverse reachability*. With forward reachability, we start with an entity and compute all entities that are in the transitive closure of relationships from that entity. This kind of analysis is useful if we have a large set of classes and methods available to us but only need to use a subset for a given program.

Reverse reachability detects entities which depend on a given entity, either directly or indirectly. This type of analysis is useful if we want to make a change to an entity such as a method or field and we need to see what other parts of the program will be affected by the change. Figure 6 shows reverse reachability analysis on the method `java.lang.Number.intValue`. The graph shows that this method is called in the class `Number` as well as from a method called `subParse` in the class `SimpleDateFormat`. Thus, we have learned that any change to `intValue` will require the methods `subParse`, `byteValue`, and `shortValue` to be either modified or retested appropriately.

Reachability analysis along with a difference database and dynamic traces were also used in TestTube [10] to determine what test cases need to be rerun during *selective regression testing*.

3.3. Differencing

Our supporting tools allow two different databases to be compared against each other. Using a set of attributes, all entities in one version of the database are compared against another version. From this, we get a list of added and deleted entities. For entities that occur in both versions, we use the `chksum` field to compare the entities. If the checksum matches, the entities are considered the same, other-

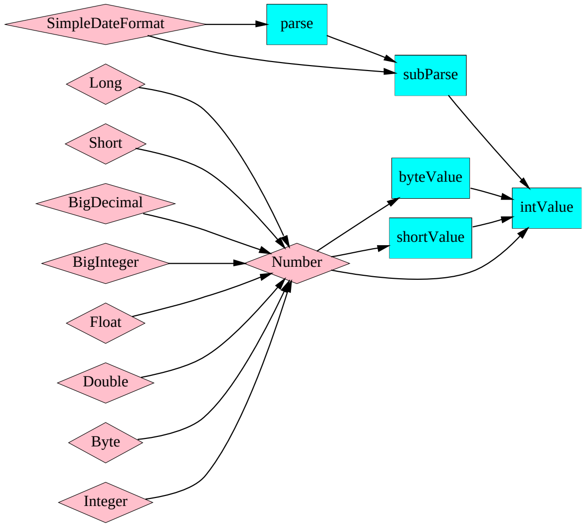

Figure 6. Reverse Reachability Example

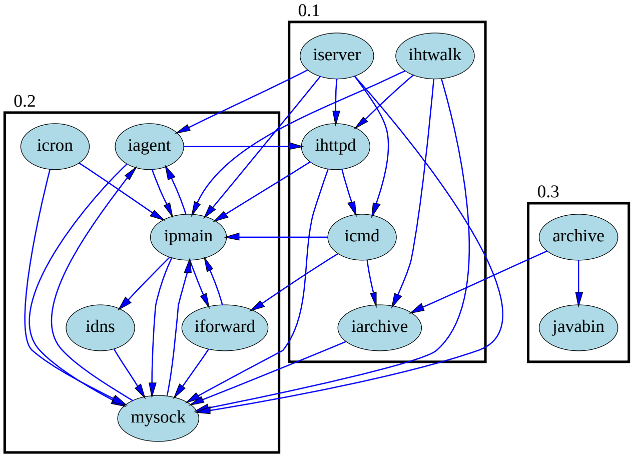

Figure 8. A Cluster Diagram for a Proxy Server Written in Java

wise they are considered changed.

To demonstrate one of the uses of differencing, we have constructed databases for the Java 1.0 standard library and the Java 1.1 standard library. Using differencing, we can easily see what changes were made between these two versions of the libraries. For example, Figure 1 showed differencing combined with reverse reachability.

Figure 7 shows differencing combined with forward reachability. The original program is our example from Figure 2. We compare this to a new version that has removed the method `printArea` from the class `Shape`. We view all relationships in the new program showing difference information. The dotted arrows show relationships that have been deleted. The white box shows the member function that has been deleted.

4. Applications

This section presents some examples of how Chava can be applied to either software systems or analysis of websites.

4.1. Clustering

We have been working with several other researchers to study the use of clustering techniques [21, 18, 27] on software repositories to discover high level software structures from existing code. We applied Bunch [20], the clustering tool developed in this purpose, to the Java software repository we created for a proxy server called iPROXY [24], and obtained the cluster diagram shown in Figure 8. The proxy server consists of 3,600 lines of Java code and we were able to produce the cluster diagram, including the process of creating the database and performing cluster analysis, in less than a few seconds. The resulting clusters are consistent with the developer's view on the high-level structure of the code: the right cluster corresponds to a Java CGI component, the middle cluster represents a web server component, and the left cluster represents the agent component of the proxy server. Such a road map is potentially useful for future software maintainers who did not develop the original source code to quickly identify the logical components in the proxy server.

4.2. Security

Chava analysis can be useful in detecting potential security flaws in an applet or application. If we download a set of binary Java classes off the web, we can use the results of queries to see which potentially dangerous methods and classes are being used in the program. For example, if we run a query asking Chava to find all methods that invoke a method whose parent package is `java.net`, then we can see if the application is making use of the network API. Also, by analyzing changes in the internal structure of a Java applet, we can find out if the dynamic content of a website has been changed in a malicious and not obvious way.

4.3. Object Model Extraction

Similar to Womble [19], we can use the extracted information stored in a Chava database to produce object models of software systems. Object models present a graphical

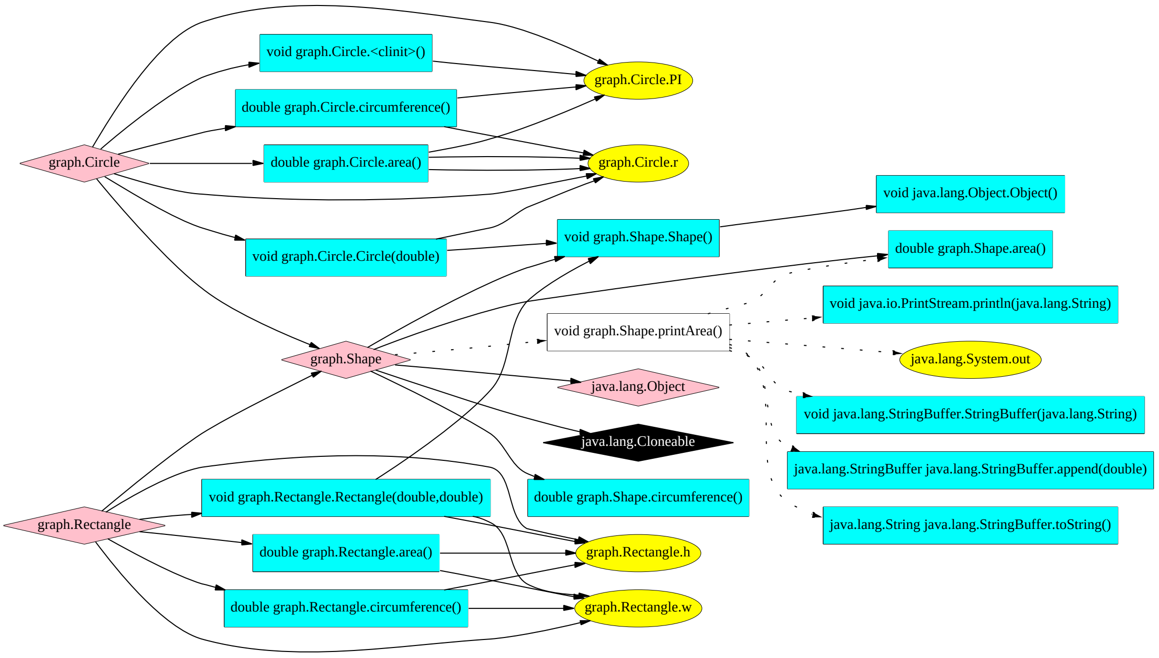
void graph.Circle.<clinit>()
graph.Circle.PI
double graph.Circle.circumference()
graph.Circle
double graph.Circle.area()
graph.Circle.r
void java.lang.Object.Object()
void graph.Shape.Shape()
void graph.Circle.Circle(double)
double graph.Shape.area()
void java.io.PrintStream.println(java.lang.String)
void graph.Shape.printArea()
graph.Shape
java.lang.System.out
java.lang.Object
void java.lang.StringBuffer.StringBuffer(java.lang.String)
java.lang.Cloneable
java.lang.StringBuffer java.lang.StringBuffer.append(double)
void graph.Rectangle.Rectangle(double,double)
double graph.Shape.circumference()
graph.Rectangle
java.lang.String java.lang.StringBuffer.toString()
double graph.Rectangle.area()
graph.Rectangle.h
double graph.Rectangle.circumference()
graph.Rectangle.w

Figure 7. Difference Example

abstraction of the interactions between components in an object-oriented program. By examining the definitions of methods and fields in a class, as well as references to objects in methods of a class, we can find one-to-one, one-to-many, and many-to-one relationships between classes. Graphical tools such as CIAO can be extended to draw the model. The architectural view allows observers to more easily see relationships between program components, which is useful for the design, implementation and debugging of software systems.

4.4. Software Metrics

Using information such as the number of methods in each class and the number of method calls or field references in each method, we can devise software metrics that can be used to suggest how complex a system is. For example, Chava can be used to compute the following Object Oriented Design (OOD) metrics proposed by Chidamber and Kemerer[11]: WMC (weighted methods per class), DIT (depth of inheritance tree), NOC (number of children), and CBO (coupling between object classes).

4.5 Website Analysis

Our tool supports generation of databases for remote applets. When the URL for a web page is specified to the database generation program, it will find any applet tags and process the class files for each applet in the HTML file.

The ability to reverse engineer a remote applet has important uses. A user may wish to reverse engineer an applet on a remote site to get an overview of what it does or analyze potential security problems (see Section 4.2).

Since the author of an applet typically writes the application in one environment and uploads the applet to a server where it is run, there is a possibility that the applet will behave differently in each environment. For example, a different set of classes could be loaded on the server due to its configuration. Our tool eliminates the process of moving the applet and its supporting components to a different machine before doing the analysis.

The CIAO environment works with a variety of languages, one of which is HTML. WebCiao [8] is a system for visualizing and tracking the structures of web sites using an HTML repository. WebCiao generates a database from a set of web pages, and uses a suite of tools that are generated with a CIAO specification.

We have implemented a tool which combines repositories of HTML and Java applets. The combined repository gives web developers an environment for viewing all aspects of web content together, both static and dynamic, including document structure and applet interaction.

Figure 9 shows an example query for the web page `http://www.att.com`. The query shows the reachable set of entities from this url. The web page contains an applet called `AIG_Flipper`, so a relationship exists between the web page and the applet. Thus, this query draws the set of

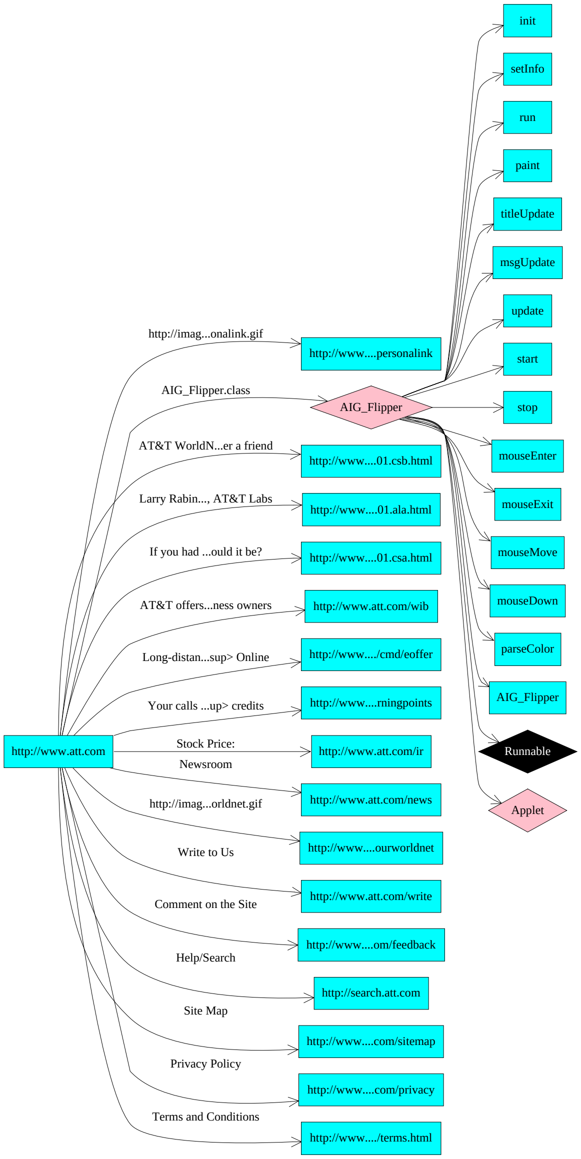
init
setInfo
run
paint
titleUpdate
msgUpdate
update
http://imag...onalink.gif
http://www....personalink
start
AIG_Flipper.class
AIG_Flipper
stop
AT&T WorldN...er a friend
http://www....01.csb.html
mouseEnter
Larry Rabin..., AT&T Labs
http://www....01.ala.html
mouseExit
If you had ...ould it be?
http://www....01.csa.html
mouseMove
AT&T offers...ness owners
http://www.att.com/wib
mouseDown
Long-distan...sup> Online
http://www..../cmd/eoffer
parseColor
Your calls ...up> credits
http://www....rningpoints
AIG_Flipper
http://www.att.com
Stock Price:
Newsroom
http://www.att.com/ir
Runnable
http://www.att.com/news
http://imag...orldnet.gif
Applet
http://www....ourworldnet
Write to Us
http://www.att.com/write
Comment on the Site
http://www....om/feedback
Help/Search
http://search.att.com
Site Map
http://www....com/sitemap
Privacy Policy
http://www....com/privacy
Terms and Conditions
http://www..../terms.html

Figure 9. HTML and Java Integration

URLs referred to by the web page and the methods referred to by the applet together.

5. Implementation

Our implementation consists of two parts. First, there is a program called `chava` that generates a repository for an application. Second, there is a specification file that provides information about the semantics of the database used by external tools.

Chava is implemented as a C program and is around 3,000 lines of source code. A Java-based implementation was considered, but was discontinued after initial performance results were poor. The C-based implementation runs an order of magnitude faster than our Java version.

Chava works analogously to a compiler. It first generates incremental databases on either a per-file or per-class basis (depending whether we are working with source or class files), similar to how a compiler generates object files. Then all of the per-class databases are linked together to generate the complete database, similar to the linking phase of a compiler. The linking phase reads in the per-class databases and resolves entities that reference external entities with their definitions.

5.1. `.class` Files

Chava is able to generate a database for a class given only access to its `class` file (generated by the compiler). The class file contains enough information about the entities in a class that we can fill in most of the attributes for our model. Class files contain a table of methods, fields, classes and strings. Chava goes through each of these tables and converts the table entries into database entities. To determine the relationships of field access and method invocation, Chava goes through the byte code for each method and picks out instructions that call methods or access fields.

5.2. Source Files

Class files do not contain every piece of information that our model provides, but they contain enough to generate a useful database for our tools. Class files are missing some information about line numbers that Chava is able to get by parsing source files. If a source file exists for a processed class, Chava parses it to fill in the missing fields from the database.

If we only have a source file for a class without the compiled class file, Chava invokes the Java compiler to produce the class file and then works with both the source and class files. Thus, Chava is capable of working with only source, only class files, or both.

5.3. Archives

Java allows a set of classes to be combined into a single file as an archive. To generate a database for an archive, the archive is first expanded into its source and/or class files. Then, Chava is run on each file and the set of databases is linked to form a single database from the set of classes. The resulting database can subsequently be linked with other applications.

For example, we have produced a database for the Java default class library. The library is quite large and consists of around 2 megabytes of class files. The generated database contains 15,248 entities and 35,012 relationships. This database can be linked in with Java applications if the user wants to do analysis involving the Java class library. For example, reachability analysis on a Java program would reveal how much of the Java library it actually uses.

5.4. Java Instantiation of CIAO

The query and visualization subsystem for Java is built by constructing a new instance of the CIAO system. CIAO takes a specification file for a language and generates the supporting tools for querying and visualization automatically. The specification consists of the following parts:

- *Schema*. It enumerates the attributes of entities and relationships. Types for attribute fields include integers, strings, and pointers to other entities.

- *Database View*. This section defines how different entity and relationship entries are viewed in text format. For example, a class entity contains the package name as one attribute and the class name as another. When displaying a class entity, we can specify that the name be displayed by combining these fields.

- *HTML View*. This section defines how to format each of the attributes in the database as HTML, making it possible to display query results in a browser window.

- *Source View*. This section defines the fields that are needed to find the source file and line numbers for a given entity in the database.

- *Graph View*. This section defines how to graphically display entities and relationships. Entities are displayed as nodes, and the specification can define what colors, shapes, fonts, etc. should be used for different kinds of entities. Relationships are represented as edges, and can also be represented with various styles and colors.

- *GUI Front End*. The final section defines the appearance of the graphical user interface.

The specification file is less than 300 lines long. The complete suite of query, visualization, and generic reachability analysis tools are generated from this specification file.

6. Performance and Experience

This section looks at the performance of Chava as a function of the number of entities and relationships emitted. It is our goal that Chava scale well with large applications so that it can be useful for real-world software projects.

6.1. Speed and Size

To measure the speed of Chava, we have taken a set of applications ranging in size and generated databases for them. To put the numbers in context, we compare the time that Chava takes to the speed of compilation. We also compare the running time of Chava to javap, which is a Java program in the Java SDK that dumps the contents of a class file. javap can be viewed as a lower bound on performance had we used Java to write Chava.

The three programs we performed experiments on were as follows:

1. *java.**: This is a collection of classes that make up the standard library for Java. All classes are in packages that begin with `java.` The set of classes is stored as an archive in zip format, and source is not available. Thus, our tool does not extract some properties from the archive, such as line numbers.

2. *swing*: Swing is another Java archive distributed from Sun. It contains a set of classes that implement a set of user interface components for use with Java. This is a good example of a large application in which source code is available.

3. *WebDelta*: WebDelta is an AT&T project that implements a Java-based version of the WebCiao interface. This is an example of a medium sized software project. Source is available.

The results are shown in Tables 2, 3 and 4. Running times are from a SPARC station running Solaris 2.5. We see in Table 3 that Chava is significantly faster than compilation with javac. In fact, Chava is also faster than javap, the Java program that dumps the contents of a class file, despite Chava's increased functionality. This is most likely the case because Chava is implemented as a C program instead of a Java program. Nonetheless, these numbers indicate that the performance of Chava is an order of magnitude better than compilation.

Table 4 shows the number of entities and relations generated by each of our test applications. The table also includes

Program	Source size	Class size	Number of classes
java.*	300,000 lines	1,648,508	696
swing	211,640 lines	1,340,364	503
WebDelta	23,951 lines	369,469	92

Table 2. Project information

Program	javac	javap	Compile Database	Link Database
java.*	N/A	1m5.4s	48.6s	22.07s
swing	5m16s	38.29s	30.72s	15.61s
WebDelta	1m8s	8.8s	4.9s	2.52s

Table 3. Performance of Chava

Program	Database size	Number of entities	Number relationships
java.*	1,720,326	15,248	35,012
swing	1,245,338	9,939	30,371
WebDelta	482,594	3,807	14,422

Table 4. Space Requirements

the size of the database file, the content of which is database entries in ASCII format. We see that the generated database size is in the order of the size of the class files, which is quite manageable. Size could be significantly reduced if compression were used on the database. The number of entities and relationships is small enough that queries can be performed efficiently.

7. Summary and Future Work

With the emerging popularity of Java, a growing number of applications are being written which can benefit from tools that assist with software engineering tasks. Our tool makes it possible to work with large applications and perform complex analysis tasks such as reachability analysis and clustering analysis. It also allows visualization of the components of an application to facilitate the understanding of their interaction. Working with object code instead of source has the disadvantage that we cannot extract information that the compiler has removed (such as optimizations or source comments), but is advantageous in that it allows analysis to be applied to a wider variety of applications, such as applets and legacy systems in which source is unavailable.

We have several tasks planned for future development of our Java repository:

- *Handle errors and exceptions*. In Java, methods may throw exceptions and errors. Each method has a set of exceptions that it is allowed to throw, and others must be caught. Our current implementation does not put information in the database about exceptions and errors. Future versions may create separate entities for them and create relationships between methods and the exceptions/errors they throw.

- *Handle compiler optimizations*. Chava works with class files emitted from Java compilers. However, some compilers optimize code. When source files are not available, Chava has difficulty working with optimized code because entities from the original code may not exist in the compiled class file. If source exists, Chava can always recompile the code without optimizations, but it should be possible to deal with some optimizations at the bytecode level.

- *Integration with debuggers*. Our Java database generation tool is the first instantiation of CIAO to work with object files. Much of the information that we extract from applications would be useful to a debugger, as our suite of tools help users to better understand programs. We plan to integrate the CIAO tools into a prototype debugger called deet [17]. Some examples of applications that might be useful are: (1) being able to look at the current location of a program inside its call graph and (2) using queries to find a set of line numbers to use as breakpoints (e.g., set a breakpoint at every line that reads field *X*).

- *3D Visualization*. Feijs and De Jong [15] have applied 3D visualization techniques to software systems with encouraging results. 3D provides more ways to represent program relationships, making use of position and color to describe attributes. We plan to apply some of these techniques to Chava databases.

- *Web-Based Reverse Engineering Service*. Instead of installing CIAO and Chava on each user's machine, we are currently creating a web service for developers who would like to share the understanding of a particular program amongst each other. Users will be able to generate graph views using an applet [2], run database queries through JDBC, and view source code as HTML or XML [22]. Such a service will allow researchers to freely analyze and experiment with public source code.

8. Availability

Chava will soon be available for experimental use. Please visit `http://www.research.att.com/~ciao`.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[2] N. S. Barghouti, J. Mocenigo, and W. Lee. Grappa: A Graph Package in Java. In *Fifth International Symposium on Graph Drawing*, pages 336–343. Springer-Verlag, Sept. 1997.

[3] 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.

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

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

[6] 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.

[7] 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.

[8] Y.-F. Chen and E. Koutsofios. WebCiao: A Website Visualization and Tracking System. In *WebNet97*, 1997.

[9] 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.

[10] Y.-F. Chen, D. Rosenblum, and K.-P. Vo. TestTube: A System for Selective Regression Testing. In *The 16th International Conference on Software Engineering*, pages 211–220, 1994.

[11] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[12] E. H. Chikofsky and J. H. C. II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1), Jan. 1990.

[13] 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.

[14] F. Douglis, T. Ball, Y.-F. Chen, and E. Koutsofios. The AT&T Internet Difference Engine: Tracking and Viewing Changes on the Web. *World Wide Web*, 1(1):27–44, 1998.

[15] L. Feijs and R. D. Jong. 3D visualization of software architectures. *Communications of the ACM*, 41(12):73–78, Dec. 1998.

[16] J. Grass. Cdiff: A Syntex Directed Differencer for C++ Programs. In *Proceedings of the Usenix C++ Conference*, Aug. 1992.

[17] D. R. Hanson and J. L. Korn. A Simple and Extensible Graphical Debugger. In *Winter 1997 USENIX Conference*, pages 173–184, Jan. 1997.

[18] D. Hutchens and R. Basili. System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Software Engineering*, 11:749–757, Aug. 1995.

[19] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. In *Proc. 21st Intl. Conf. Software Engineering*, May 1999.

[20] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In *Sixth International Workshop on Program Comprehension*, June 1998.

[21] H. Müller, M. Orgun, S. Tilley, and J. Uhl. A Reverse Engineering Approach to Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, 1993.

[22] The SGML/XML Web Page. `http://www.oasis-open.org/cover/xml.html`, 1999.

[23] S. Paul and A. Prakash. A Framework for Source Code Search Using Program Patterns. *IEEE Transactions on Software Engineering*, 20(3):463–475, June 1994.

[24] H. Rao, Y.-F. Chen, M.-F. Chen, and J. Cheng. iproxy: An agent-based middleware. In *Poster Proceedings of the Eighth World Wide Web Conference*, May 1999.

[25] D. Rayside, S. Kerr, and K. Kontogiannis. Change and Adaptive Maintenance Detection in Java Software Systems. In *Proc. Fifth Working Conference on Reverse Engineering*, Oct. 1998.

[26] The Java Reflection API. `http://www.javasoft.com/products`, 1998.

[27] R. Schwanke. An Intelligent Tool For Re-Engineering Software Modularity. In *Proc. 13th Intl. Conf. Software Engineering*, May 1991.

[28] J. Seemann and J. W. von Gudenberg. Pattern-based design recovery of java software. In *Proc. Foundation of Software Engineering*, Nov. 1998.

[29] D. Sharon and R. Bell. Tools that Bind: Creating Integrated Environments. *IEEE Software*, 12(2):76–85, Mar. 1995.

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