RC 21596(97394)15NOV99 **Computer Science/Mathematics**

Research Report

An Approach for Extracting Library-Based Object-Oriented Applications

Peter F. Sweeney, Frank Tip
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
pfs@us.ibm.com, tip@watson.ibm.com

Research Division
Almaden · Austin · Beijing · Haifa · T.J. Watson · Tokyo · Zurich

LIMITED DISTRIBUTION NOTICE This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T.J. Watson Research Center [Publications 16-220 ykt] P.O. Box 218, Yorktown Heights, NY 10598. email reports@us.ibm.com
Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home

An Approach for Extracting Library-Based Object-Oriented Applications

Peter F. Sweeney **Frank Tip**
IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
pfs@us.ibm.com tip@watson.ibm.com
+1 (914) 784-6390 +1 (914) 784-6294

Abstract

An *extraction tool* determines redundant program constructs using static analysis, and eliminates these constructs in order to reduce application size. Library-based object-oriented applications typically use only part of the functionality of the class libraries they rely on, and are therefore likely to be excellent candidates for extraction. Thus far, extraction tools have targeted complete applications, and relied on ad-hoc solutions to accommodate "dynamic" language features such as reflection and dynamic loading. We present a general approach for extracting applications ranging from complete programs to extendable class libraries, in which the user supplies additional information about these features using a modular specification language. We implemented our approach in the context of *Jax*, an industrial-strength extractor for Java that has been applied to several commercial applications.

Keywords: Application Extraction, Class Library, Program Optimization, Reflection, Behavioral Specifications.

1 Introduction

Recent years have shown a trend towards library-based software. This is especially true in the case of Java, with its growing base of class libraries that are available in a platform-neutral format. Applications typically rely on class libraries for tasks such as constructing user interfaces, providing secure network communication, or providing efficient data structures. In many cases, libraries are shipped along with applications because one cannot assume the user to have these libraries installed. Since applications often use only part of the functionality of the libraries they use, a substantial portion of the library code tends to be unreachable. Identification and removal of this unreachable code is desirable when size is an issue (e.g., in cases where distribution takes place by download over the internet). An *extraction tool* addresses this issue by removing unused classes, methods, and fields. Many extraction tools perform additional size-reducing program transformations as well.

Jax is an extraction tool for Java that was developed at IBM Research [17]. *Jax* reads in the Java class files that constitute an application, removes unused program constructs, performs several size-reducing program transformations, and generates a zip file containing class files for the extracted application. *Jax* has been available on IBM's alphaWorks web site[1] since June 1998, and has been downloaded over 12,000 times since then. *Jax* has been applied to a number of commercial applications, and will be shipped as a Technology Preview with an IBM product (IBM VisualAge Java 3.0, Enterprise Edition) later this year. We studied the effectiveness of *Jax* on a set of realistic benchmarks ranging from 27 to 2,332 classes, and measured an average reduction in zip file size of 48.7%. We found that for library-based applications, this percentage can easily reach 60% or more.

Dynamic language features such as dynamic loading, reflection, native code, and extension through derivation pose problems for extraction tools because static analysis alone cannot determine the program constructs that are used, and the constructs that can be eliminated. Extractors therefore require additional user input in order to make the appropriate conservative approximations. To this end, the original design of *Jax* provided a mechanism by which the user could specify a set of program constructs (i.e., classes, methods, and fields) that should not be removed or transformed during extraction. Another issue that extraction tools face in practice is the fact that it is often undesirable to analyze *all* of the code that constitutes a library-based application. Consider the case where we need to distribute an application A that uses a library L, and suppose that the user has already installed *a different implementation* of L. Then it is not only unnecessary to ship L with A, but the extractor should not make any assumptions based on specific implementations of L. The standard Java libraries are an obvious example of this situation. In the original design of *Jax*, ad-hoc assumptions were made about the boundary between classes in the application and classes in the Java libraries.

After gaining some practical experience, we found the original design of *Jax* to be inadequate in the following re-

[1] A free evaluation copy of *Jax* can be downloaded from `www.alphaworks.ibm.com/tech/jax`.

spects:

- Different vendors of Java Virtual Machines use different names for packages (and hence classes), and a flexible mechanism is required for distinguishing the classes that should be extracted from the classes that are "external" to the extraction process.
- *Jax* originally only supported the extraction of applications and applets. No mechanisms were provided for the extraction of libraries without accompanying client applications, for the extraction of collections of applets and applications, and for ensuring that certain classes could be subclassed after extraction.
- In our experience, determining the program constructs accessed using reflection and dynamic loading is a fairly easy task for a programmer familiar with the code. However, it can be quite difficult to determine how reflection is used in third-party libraries, especially if the source code for these libraries is unavailable. In the experiments of [17], we relied on instrumenting calls to the reflection API to determine how reflection is used in unfamiliar programs. However, this approach is tedious and error-prone because it requires obtaining complete coverage of the execution paths that involve reflection.
- The use of reflection in a library typically depends on the features that are used, and different clients use different subsets of a library's features. Since the use of reflection or dynamic loading could only be specified unconditionally, it was not possible to create a single reusable specification for a library that allows accurate extraction of multiple client applications.

The approach taken in this paper relies on a specification language for defining how to extract various types of applications, ranging from traditional-style applications for which the usual closed-world assumptions can be made, to class libraries that can be extended after extraction. Our approach has the following salient properties:

- The language is *modular* in the sense that a separate specification is associated with each library. Extraction of a library-based program requires the specification for the program itself, and specifications for the libraries used by the program. An "import" feature is provided to allow modular composition of specifications.
- The language is sufficiently expressive to state that a given program construct R is only accessed using reflection or dynamic loading when a certain library feature is used, by allowing conditions that depend on the reachability of methods.

The combination of modular composition of specifications and a mechanism for conditionally specifying the use of reflection and dynamic loading has important implications. Breaking up specifications into modules enables a separation of responsibilities: Each module can be written by the person most familiar with the associated code. Moreover, the conditional specification mechanism allows the creation of specifications for class libraries that can be reused by clients that use different subsets of the library's features. This suggests a new distribution model for class libraries in which the *vendor* supplies a specification for his library, which enables a *user* to extract his library-based program.

The remainder of this paper is organized as follows. Section 2 presents a number of pragmatic issues that are the motivation of this work. Section 3 introduces a specification language for defining the extraction of various kinds of library-based applications, and Section 4 presents a mechanism for translating specifications to a small set of assertions, and Section 5 discusses how several program transformations performed by extraction tools can be adapted to support these assertions. Section 6 described the implementation of JSL in *Jax*. Section 7 summarizes related work, and Section 8 presents conclusions and directions for future work. .

2 Motivation

We will adopt the non-descriptive term *software unit* to denote any collection of classes that constitutes a logical entity.

Roles of software units

Fundamentally, there is no difference between code in a class library and code in an executable application, and it is only the way in which software units are *used* and *composed* that determines how extraction should be performed. We will illustrate this point by sketching three scenarios involving the extraction of a class library and a client application.

scenario 1: extract a library without assumptions about its clients. Consider the situation where we want to extract a class library L in isolation (i.e., without making assumptions about client applications) by eliminating unreachable methods. In this case, we have to assume that all of L's `public` and `protected` methods[2] may be called by client applications. Consequently, the only methods that can be removed from L are `private` methods that cannot be called (directly or indirectly) by any `public` or `protected` method.

scenario 2: extract a library in the context of a specific application. It is often the case that a class library L has to be distributed with a client application A because it cannot be assumed that the user of A has (the correct version of) L available. In such cases, it is desirable to extract L under the assumption that A is the *only* application that uses L. In this case, we can safely remove any method from L that is not transitively reachable from A's `main()` method.

[2] This assumes that the client application is not in the same `package` as the library. If this assumption is false, methods with package-level access have to be taken into account as well.

scenario 3: extract a client without assumptions about its libraries. There is yet another case to consider: we may want to extract a client application A without making assumptions about the code in library L. This situation may occur when we want to ship a A to a party that has a *different implementation* of L. In this case, determining the set of reachable methods in A requires that worst-case assumptions be made in cases where classes in A override methods defined in L.

The conclusion that can be drawn from the above discussion is that different usage scenarios require that very different assumptions be made by extraction tools. In the remainder of this paper, the term *role* will be used to refer to the way in which a software unit is used. We will consider four roles that frequently occur in practice:

- An *application* is an executable software unit with an external interface consisting of a single `main()` method. It is assumed that classes in applications are not further extended by derivation after extraction.
- An *applet* is an executable software unit that is executed in the context of a browser. An applet extends class `java.applet.Applet` and its external interface consist of a set of methods in `java.applet.Applet` that it overrides. It is assumed that classes in applets are not further extended by derivation after extraction.
- A *library* is not assumed to be executable by itself, but is used as a building block by other units. Classes in libraries may be extended by derivation, and the external interface of a library consists of any method that has `public` or `protected` access rights.
- A *component* is similar to a library in the sense that it is an incomplete program used as a building block by other units but unlike a library it is assumed that classes in a component cannot be extended by derivation. The external interface of a component contains every method with `public` access rights.

Other roles such as JavaBeans [15] and servlets [7] can be modeled similarly.

Specifying the extraction domain

As was mentioned earlier, there is no distinction between classes in different software units at the language level. Consequently, it is necessary to specify the "boundaries" between software units when performing extraction. In our approach, the user selects the set of classes that should be extracted, and worst-case assumptions are made about the behavior of classes that are not selected.

In practice, there are very few situations where *all* classes should be extracted. One can think of the structure of an application as "layered", with the bottom layer consisting of the standard Java libraries, the middle layer consisting of class libraries built on top of the standard libraries (perhaps consisting of sublayers), and the topmost layer consisting of the application itself. It is usually the case that classes below a certain layer do not need to be shipped and extracted because they are already available on the client side. In particular, the standard Java class libraries are generally available, and are usually excluded from the extraction process[3]. Note that this is not merely an issue of avoiding redundant work, but potentially also one of correctness. If an application class contains a call to a method in the Java class libraries, inlining that call on one platform may result in code that does not work on another platform.

```
import java.io.*;
import java.lang.Class;
import java.lang.reflect.Method;

public class Example1 {
  public static void main(String args[]){
    T t = new T();
    Class c = T.getClass();
    Method[] methods = c.getDeclaredMethods();
    for (int i=0; i < methods.length; i++){
      Method m = methods[i];
      String methodName = m.getName();
      System.out.println(methodName);
    }
  }
};

class T {
  void foo(){ ··· };
  void bar(){ ··· };
};
```

Figure 1: A Java program that uses reflection.

Dealing with reflection, dynamic loading, and native code

Java's reflection mechanism allows programs to do various forms of self-inspection. Figure 1 shows an example program that uses reflection. In this program, the class that represents the type `T` of object `t` is retrieved using a call to method `java.lang.Object.getClass()`, and stored in variable `c`. The program then calls method `java.lang.Class.getDeclaredMethods()` to obtain a vector of objects representing the methods in `T`. For each method in this vector, the name is retrieved (by way of a call to `java.lang.reflect.Method.getName()`), and printed to standard output. Hence, the program generates the following output:

```
foo
bar
```

Clearly, program behavior depends on the *presence* and the *name* of the methods in class `T`, even though these methods are not invoked anywhere. It is obvious that the use of reflection in Figure 1 precludes program transformations such

[3] In the case of embedded systems and network PC's that run a fixed set of applications it may be desirable to include the Java class libraries in the extraction domain.

```
import java.io.*;
import java.lang.Class;

public class Example2 {
  public static void baz(String name){
    try {
      Class c = Class.forName(name);
      Object o = c.newInstance();
      I i = (I)o;
      i.zap();
    }
    catch (ClassNotFoundException e){
      System.out.println("Error:  " +
        "Could not find " + name); }
    catch (IllegalAccessException e){
      System.out.println("Error:  " +
        "Illegal access to " + name); }
    catch (InstantiationException e){
      System.out.println("Error:  " +
        "Abstract " + name); }
  }
};

interface I {
  public void zap();
};
```

Figure 2: A Java program that uses dynamic loading.

as the removal or renaming of methods in class `T` because such actions would affect program behavior.

Dynamic loading is a heavily-used[4] mechanism for instructing a Java Virtual Machine to load a class X with a specified name s, and return an object c representing that class. Reflection can be applied to c to create X-objects on which methods can be invoked. The crucial issue is that s is computed at *run-time*. This implies that, in general, static analyses cannot determine which classes are dynamically loaded[5].

In our experience, Java applications use dynamic loading primarily as means to avoid explicit dependences between classes. By partitioning the classes that comprise an application into a core set of classes that are statically loaded, and additional sets of classes that are loaded dynamically on demand, the first screen of the application can be made to appear more quickly. This seems to be the primary motivation for the use of dynamic loading in interactive and web-based environments.

Figure 2 shows a program fragment that exhibits a fairly typical use of dynamic loading. Class `Example2` contains a method `baz` which takes a single argument of type `String`, and dynamically loads a class with that name by calling method `java.lang.Class.forName()`. A reference to the dynamically loaded class is stored

[4] Nine of the thirteen benchmarks studied in [17] use dynamic loading.

[5] In some cases, the type of a dynamically loaded class can be inferred by constant propagation of the string literals that represent the class name. However, in our experience, these names are often read from files or manipulated in non-trivial ways.

```
import java.lang.Class;

public class L {
  public static void f(){
    ...
    Class c = Class.forName("M");
    ...
  }

  public static void g(){
    ...
    Class c = Class.forName("N");
    ...
  }
};

class M { ··· };

class N { ··· };
```

Figure 3: Example class library that uses dynamic loading.

in variable `c`. The program then calls method `java.lang.Class.newInstance()` to create a new object of the dynamically loaded type, casts it down to an interface type `I`, and calls method `zap` on the object. Observe that class instantiation (of the dynamically loaded class) and method invocation (of the default constructor of that class) occur implicitly. This poses problems for optimizations such as dead method removal because the analyses upon which these optimizations are based typically need to know which classes are instantiated, and which methods are invoked.

Java provides a mechanism for implementing methods in a platform-dependent way, typically using C or C++. The mechanism works roughly as follows: The `native` keyword is used to designate a method as being implemented in a different language, and the corresponding method definition is provided in an object file (e.g., a dynamically linked library) associated with the Java application. The native code in the object file may instantiate classes, invoke methods, and access fields in the application. This obviously poses problems for any program transformation that relies on accurate information about class instantiation and method invocation, because object code is notoriously hard to analyze.

It should be evident from the above examples that, without additional information, the use of reflection, dynamic loading, and native methods requires that *extremely* conservative assumptions be made during extraction. The approach taken in this paper will be to have the user specify a list of program constructs (i.e., classes, methods, and fields) that are accessed using these mechanisms, and to make the appropriate worst-case assumptions about these constructs.

Modeling different usage contexts

In order to create configuration files that are *reusable* in different contexts, it is often desirable to specify that a given program construct is only accessed using reflection or dynamic loading under certain conditions. To illustrate this

JSLScript	::=	Item*
Item	::=	DomainSpecifier | Statement | Import
DomainSpecifier	::=	ClassPath | Include
ClassPath	::=	PATH <Directory> | PATH <ZipFile>
Include	::=	INCLUDE <Class> |
		INCLUDE <PackageName>
Statement	::=	RoleStmt | PreserveStmt
RoleStmt	::=	APPLICATION <Class> |
		APPLET <Class> |
		LIBRARY <Class> |
		COMPONENT <Class>
PreserveStmt	::=	SimplePreserveStmt |
		CondPreserveStmt
SimplePreserveStmt	::=	PRESERVE <Class> |
		PRESERVE <Method> |
		PRESERVE <Field>
CondPreserveStmt	::=	SimplePreserveStmt
		WHEN REACHED <Method>
Import	::=	IMPORT <FileName>

Figure 4: BNF Grammar of for the core of JSL

.

issue, Figure 3 shows a small class library consisting of three classes `L`, `M` and `N`. Class `L` has two methods: `f` and `g`. A call to method `f` results in the dynamic loading of class `M`, and a call to method `g` results in the dynamic loading of class `N`. Note that a client that calls `f` but not `g` will only dynamically load `M`, and a client that calls `g` but not `f` will only dynamically load `N`. A specification of the library's behavior that states that *any* client of `L` dynamically loads both `M` and `N` would clearly be overly conservative.

Section 3 introduces a mechanism that allows conditional specifications of the form "program construct X should be preserved when method m is executed". This allows one to express how dynamic loading or reflection is dependent on the part of a software unit's functionality that is *used*. Consequently, it enables the creation of a single, reusable configuration file for a software unit that can be used to extract that unit accurately in the context of different clients.

3 A Specification Language

Figure 4 presents a BNF grammar for the core of JSL (*Jax* Specification Language), a simple specification language that allows users to specify at a high level how to extract a library-based application. The semantics of the various features in JSL are closely related to the discussions in Section 2. A JSL script comprises:

1. A *domain specification*, consisting of a class PATH where classes can be found, and a set of `include` statements that specify which classes should be optimized. Any class not listed in an INCLUDE statement is considered external to our analyses in the sense that it will not be extracted, and that worst-case assumptions will be made about its behavior.

2. A set of *statements*. There are two kinds of statements. First, ROLE statements serve to designate the role of some or all of the classes included in the extraction domain as APPLICATION, APPLET, COMPONENT, or LIBRARY. The semantics of these roles were discussed earlier in Section 2. Second, PRESERVE statements are used to specify that program constructs (i.e., classes, methods, and fields) should be preserved because they are accessed using reflection, dynamic loading or from native code, and that worst-case assumptions should be made about these constructs. Following the discussion of Section 2, program constructs can be conditionally preserved depending on the reachability of a specified method using a conditional PRESERVE statement.

3. A list of imported configuration files. The semantics of the IMPORT feature consist of textual expansion of the imported file into the importing file.

```
public class A {
  public static void main(String args[]){
    ...
    L l = new L();
    l.g();
    ...
  }
};
```

Figure 5: Example application that uses the library of Figure 3.

```
PATH ...
INCLUDE L
LIBRARY L
PRESERVE M WHEN REACHED L.g()
PRESERVE N WHEN REACHED L.f()
```

Figure 6: Specification *lib.jsl* for the class library of Figure 3.

```
PATH ...
INCLUDE A
APPLICATION A
IMPORT lib.jsl
```

Figure 7: Specification *app.jsl* for the application of Figure 5.

Figure 5 shows an example application `A` that uses the library of Figure 3. Observe that `A`'s `main()` routine creates an `L`-object and invokes `L`'s method `g()`. Figures 6 and 7 present JSL specifications *lib.jsl* and *app.jsl* for the library of Figure 3 and the application of Figure 5, respectively. The conditional PRESERVE statements in *lib.jsl* ensure that class `M` is preserved if method `L.g()` is REACHED, and that class `N` is preserved if method `L.f()` is reached. Since `A` only calls method `L.g()`, class `N` will not be extracted.

4 Implementation Strategy

Our approach for extracting a collection of software units consists of two steps. First, the information contained in a JSL specification is translated to a form more suitable for use in extraction tools. This is accomplished by adding a number of lower-level assertions to JSL, and translating the user-level

Statement	::=	Assertion | ConditionalAssertion
Assertion	::=	SimpleAssertion
Assertion	::=	EXTENDABLE <Class>
Assertion	::=	OVERRIDABLE <Method>
SimpleAssertion	::=	INSTANTIATED <Class>
SimpleAssertion	::=	REACHED <Method>
SimpleAssertion	::=	ACCESSED <Field>
SimpleAssertion	::=	PRESERVEIDENTITY <Class>
SimpleAssertion	::=	PRESERVEIDENTITY <Method>
SimpleAssertion	::=	PRESERVEIDENTITY <Field>
ConditionalAssertion	::=	SimpleAssertion WHEN REACHED <Method>

Figure 8: BNF grammar for assertions in JSL.

ROLE and PRESERVE statements to these lower-level features. This step is presented below in Section 4. Then, in Section 5, we discuss how these "lowered" JSL specifications are taken into account by a number of specific optimizations.

Translation of JSL specifications

The specification language presented in Figure 4 was designed to make it easy for programmers to specify how a collection of software units should be extracted. However, the algorithms used by extraction tools typically require low-level information such as methods that are potentially executed, and classes that are potentially instantiated. To bridge the gap between user-level and extractor-level nformation, we add a number of assertion constructs to JSL, and provide a translation from user-level statements to these assertions. An important benefit of this approach is that all roles and usage scenarios can be treated uniformly by the extractor.

Figure 8 shows a BNF grammar for JSL assertions. The INSTANTIATED, REACHED, and ACCESSED assertions are provided for expressing that a class is instantiated, a method is reached, or a field is accessed, respectively. The PRESERVEIDENTITY assertions express that a program construct may be accessed from outside the extraction domain, which implies that the construct's name or signature should not be changed. The EXTENDABLE and OVERRIDABLE assertions serve to express that a class may be extended, and that a method may be overridden after extraction, respectively. In Section 5, we discuss the impact of the latter two types of assertions on the closed-world assumptions made by optimizations such as call devirtualization and method finalization.

The REACHED, ACCESSED, INSTANTIATED, and PRESERVEIDENTITY assertions also have a conditional form. Conditional assertions are used to model the conditional PRESERVE statements that specify situations where reflection or dynamic loading is used in a specific method.

Table 1 shows how statements are translated to assertions. The table contains a row for each type of JSL statement in which the rightmost column shows the assertions generated for that statement. The translation process for roles can be summarized as follows:

- Worst-case assumptions are made to determine a set of methods that can be invoked from outside the extraction domain. Each such method is assumed to be REACHED, and receives the PRESERVEIDENTITY assertion to indicate that external references may rely on its name and signature. Different roles require different treatment. For example, for APPLICATIONs, only the `main()` method is referenced externally and needs to be added to the set. However, for classes that play a LIBRARY role all `public` and `protected` methods are added.

- For each ROLE of a class, the appropriate assumptions are made to determine the fields that may be accessed from outside the extraction domain, and all such fields are asserted to be ACCESSED. For example, all `public` fields of COMPONENTS are assumed to be accessed.

- Any class that plays an APPLET role is instantiated by the JVM when the applet is loaded by a browser. We model this by asserting that each applet class is INSTANTIATED.

- For classes that play a LIBRARY role, we have to assume that further subclassing and method overriding may take place after extraction. To preserve this behavior, we assert that the class should be EXTENDABLE and all of its virtual methods should remain OVERRIDABLE.

The translation of PRESERVE statements into assertions assumes that the identity of *any* program construct accessed using reflection or dynamic loading should be preserved. Hence, any program construct that is referenced in a PRESERVE statement receives the PRESERVEIDENTITY assertion. For reflectively accessed classes, we make the conservative assumption that they are instantiated if they are not `abstract`. Each reflectively accessed method that is referenced in a PRESERVE statement is assumed to be invoked, and is therefore asserted to be REACHED. Similarly, each reflectively accessed field that is referenced in a PRESERVE statement is assumed to be ACCESSED. The translation of conditional PRESERVE statements involves carrying over the condition from the statement to the assertion, but is otherwise completely analogous.

5 Extraction of Library-Based Applications

This section presents how a number of program transformations and optimizations that we implemented in *Jax* can be adapted to operate on various kinds of library-based applications by taking into account JSL assertions. Many of these optimizations were originally presented as whole-programs optimizations, by making the "closed world" assumption that the entire program is available at analysis time. A number of these optimizations can be classified as *time* optimizations that increase execution speed, and *space* optimizations that decrease an application's space requirements at run-time, and can be found in traditional optimizing compilers. However, we also consider a number of *application size* optimizations

statement	derived assertions
APPLICATION C	PRESERVEIDENTITY C REACHED C`.main(java.lang.String[])` PRESERVEIDENTITY C`.main(java.lang.String[])`
APPLET C	INSTANTIATED C PRESERVEIDENTITY C PRESERVEIDENTITY $C.m$, when $C.m$ overrides `java.applet.Applet.`m REACHED $C.m$, when $C.m$ overrides `java.applet.Applet.`m
COMPONENT C	PRESERVEIDENTITY C PRESERVEIDENTITY $C.m$, when $C.m$ is a `public` method REACHED $C.m$, when $C.m$ is a `public` method PRESERVEIDENTITY $C.f$, when $C.f$ is a `public` field ACCESSED $C.f$, when $C.f$ is a `public` field
LIBRARY C	PRESERVEIDENTITY C EXTENDABLE C REACHED $C.m$, when $C.m$ is a `public` or `protected` method PRESERVEIDENTITY $C.m$, when $C.m$ is a `public` or `protected` method OVERRIDABLE $C.m$, when $C.m$ is a `public` or `protected` virtual method ACCESSED $C.f$, when $C.f$ is a `public` or `protected` field PRESERVEIDENTITY $C.f$, when $C.f$ is a `public` or `protected` field
PRESERVE C	INSTANTIATED C, when C is not an interface or an abstract class PRESERVEIDENTITY C
PRESERVE $C.m$	REACHED $C.m$ PRESERVEIDENTITY $C.m$
PRESERVE $C.f$	ACCESSED $C.f$ PRESERVEIDENTITY $C.f$
PRESERVE C WHEN REACHED $D.n$	INSTANTIATED C WHEN REACHED $D.n$ PRESERVEIDENTITY C WHEN REACHED $D.n$
PRESERVE $C.m$ WHEN REACHED $D.n$	REACHED $C.m$ WHEN REACHED $D.n$ PRESERVEIDENTITY $C.m$ WHEN REACHED $D.n$
PRESERVE $C.f$ WHEN REACHED $D.n$	ACCESSED $C.f$ WHEN REACHED $D.n$ PRESERVEIDENTITY $C.f$ WHEN REACHED $D.n$

Table 1: Translation of statements into assertions.

for reducing the size of the application itself that are specific to extraction tools.

Call graph construction

Since all of the optimizations under consideration rely directly or indirectly on the construction of a call graph, we will first discuss how call graph construction algorithms can be adapted to take into account JSL assertions. We will use Rapid Type Analysis (RTA), an efficient call graph construction algorithm, as a specific example. Other call-graph construction algorithms (see e.g., [8, 10]) can be adapted similarly.

RTA [5, 4] is a popular algorithm for constructing call graphs and devirtualizing call sites that only requires class hierarchy information and global information about instantiated classes, and that has been demonstrated to scale well in practice [17]. RTA is most easily implemented as an iterative algorithm that uses three worklists containing (i) reached methods, (ii) reached call sites[6], and (iii) instantiated classes. Initially, the worklist of reached methods is initialized to contain the set of methods called from outside the application (e.g., an application's `main()` method), and the other two worklists are initialized to the empty set. Then, following steps are performed repeatedly:

- The body of a reached method is scanned. Any call sites and class instantiations that were not previously encountered are added to the appropriate worklist.
- Each call to a method $C.f$ is resolved with respect to each instantiated class D, if D is a subclass of C. This involves performing a method lookup for f in class D. If the lookup resolves to a method that was not previously reached, it is added to the worklist of reached methods, and the call graph is updated with edges that reflect the flow of control between caller and callee.

This iterative process continues as long as additional methods, call sites, and instantiated classes are found. In cases where a class C in the extraction domain overrides a method f in a class outside the extraction domain, we make the worst-case assumption that there is a call to this method on any object of any instantiated class.

In order to adapt RTA to take into account JSL assertions, we first need to adapt the initialization of the worklists. The worklist of reached methods is initialized to contain any method m for which an assertion REACHED m was generated. The worklist of reached call sites is initialized to contain the empty set. Finally, the worklist of instantiated classes is initialized to contain any class C for which an assertion INSTANTIATED C was generated.

[6] Since all calls to any given method f are resolved similarly, any reasonable implementation combines them.

Then, in the iterative part of the algorithm, we add the following additional steps, which are executed when a method m is added to the worklist of reached methods.

- Whenever a method m is added to the worklist of reached methods for which an assertion INSTANTIATED C WHEN REACHED m exists, class C is added to the worklist of instantiated classes if it does not already occur in that list.
- Whenever a method m is added to the worklist of reached methods for which an assertion REACHED m' WHEN REACHED m exists, method m' is added to the worklist of reached methods if it does not already occur in that list.

Dead method removal

Dead Method Removal [17] is an optimization that removes redundant method definitions. This optimization relies on the information gathered during call graph construction to determine situations where a method can be removed completely, as well as situations where a method's body can be removed but where its signature needs to be retained. The latter situation arises in the following cases:

- There is a reached virtual call site that refers statically to method $C.m$, but $C.m$ is not the target of any dynamic dispatch or direct call.
- There is a class C that (i) contains an unreached method $C.m$, and (ii) implements an interface I containing a declaration $I.m$ of the same method that is called elsewhere in the application.

Note that, in the latter case, method $C.m$ cannot be removed because the resulting class file would be syntactically invalid. In both cases, no additional information is necessary beyond the information determined during call graph construction.

Call devirtualization

Call devirtualization [6, 3] transforms virtual method calls into direct method calls. This transformation can be applied to a virtual call site x to a method $C.m$ if (i) there is only one method that can be reached from x, and (ii) method $C.m$ cannot be overridden after extraction of the application. The first condition can be verified by inspection of the call graph, and the second condition is met if there is no assertions OVERRIDABLE $C.m$ or EXTENDABLE C, where $C.m$ is the method invoked at call site x.

Dead field removal

Dead field removal [16] removes fields that are not accessed, as well as fields that are write-accessed but not read-accessed. This optimization requires that the bodies of all reached methods are scanned for read and write operations to fields[7]. Fields that neither read nor written can simply be removed. Fields that are only written are also removed along with the write-operations that access these fields. Dead field removal can be adapted to handle JSL assertions by considering a field $C.f$ to be read-accessed if there exists a ACCESSED $C.f$ assertion. Conditional ACCESSED assertions can be treated in the same way as conditional REACHED assertions.

Name compression

Name compression reduces application size by replacing the names of classes, methods, and fields with shorter names. The names of a class or field x can be changed if x is not instantiated or accessed outside the extraction domain, respectively. The conditions under which methods can be renamed are a bit more complicated. Certain methods such as constructors, class initializers, and class finalizers cannot be renamed. Virtual methods require that the method does not override a method outside the extraction domain, and if one virtual method overrides another, both must be renamed correspondingly[8]. In the presence of JSL assertions, a number of additional constraints have to be imposed on the renaming of program constructs. Any program construct for which there exists an assertion PRESERVEIDENTITY x cannot be renamed, and any method m for which there exists an assertion OVERRIDABLE m cannot be renamed.

Method finalization and class finalization

Marking classes and methods `final` is a mechanism to inform compilers (especially Just-in-Time compilers) that no further extension of the class hierarchy will take place. JSL assertions are accommodated by not finalizing any class C for which there is an assertion EXTENDABLE C, and by not finalizing any method m for which there exists an assertion OVERRIDABLE m.

Method call inlining

Inlining [14] replaces a method call by a customized version of the callee's body that is obtained by substituting actual parameters for formal parameters[9]. Inlining relies on call graph information (usually, only call sites that have exactly one target are candidates for inlining) as well as heuristics (inlining a large method with many call sites may result in too much code growth). In the presence of JSL assertions, a call to a virtual method $C.m$ for which there exists an assertion OVERRIDABLE $C.m$ cannot be inlined. This is the case because the method may be overridden after extraction has taken place, and we conservatively assume that the call site may resolve to these methods.

Class hierarchy transformations

[7] This is most easily done during call graph construction when method bodies have to be traversed anyway.

[8] Actually, the situation is slightly more complex. Consider a situation where a class C extends a class B and implements an interface I, and where a method f is declared in I, defined in B, but not defined in C itself. Then, the occurrences of f in I and B are related and should be renamed correspondingly.

[9] In the presence of exception handling, inlining a method call also involves combining the exception tables of the caller and the callee.

Removal of unused classes, and merging of a derived class C with its base class B reduce application size. The latter transformation involves moving the methods[10] and fields from C to B, and updating the references to these methods accordingly. The main benefit of class merging has to do with the fact that in Java class files, each class is a self-contained unit with its own set of literals, referred to as its *constant pool*. Classes that are adjacent in the hierarchy typically have many literals in common, and merging such classes reduces the duplication of literals across the different class files. Class merging may also enable the transformation of virtual method calls into direct method calls. Space limitations do not permit a complete discussion of class merging here, and we refer the reader to [17, 18] for details. In order to take into account JSL assertions, any class C for which there exists an assertion PRESERVEIDENTITY C should not be removed, or merged into its base class.

6 Implementation

We implemented JSL in the context of *Jax* [17], and version 6.0 of *Jax* (released in August 1999) supports JSL in its full generality, although the syntax of the JSL keywords in the system differs slightly from the syntax used in this paper. The implementation also permits users to specify JSL assertions directly, and has mechanisms for specifying the name of the generated zip file, and for selectively disabling some of the optimizations. *Jax* provides two mechanisms to support JSL. In "batch mode", a JSL script is read from a file, and the application is processed accordingly. We also provide an Graphical User Interface (GUI) that allows users to create JSL scripts interactively. Figure 9 shows a snapshot of the main window of the GUI. This window contains a tree-like view of the class hierarchy's packages, classes, methods, and fields. The leftmost column of this view is a checkbox that allows a user to select whether or not a given class or package is **include**d in the extraction domain. The **type** column allows the user to select the role(s) of a class from a menu (shown in the figure), and the **preserve** column enables users to **preserve** a class, method, or field by selecting "yes", "no", or "conditionally preserved" from a pull-down menu. Selecting "conditionally preserved" in the **preserve** column will spawn another window containing a view of hierarchy in which the user can select the method(s) of the WHEN REACHED part of the statement.

7 Related Work

The extraction of applications was pioneered in the Smalltalk community, where it is usually referred to as "packaging" [11, 9, 13], Smalltalk packaging tools typically have mechanisms for excluding certain standard classes and objects from consideration, and for forcing the inclusion of objects and methods. While the latter mechanism is sufficient to handle programs that use reflection, we are not aware of any Smalltalk extractor that models different types of applications, or that provides a feature to preserve certain program constructs conditionally.

Agesen and Ungar [2, 1] describe an application extractor for the Self language that eliminates unused slots from objects (a slot corresponds to a method or field). In his PhD thesis [1, page 146], Agesen writes that there is no easy solution to dealing with reflection other than "rewriting existing code on a case by case basis as is deemed necessary" and suggests "encouraging programmers writing new code to keep the limitations of extraction technology in mind".

In the context of Java, we are aware of a number of several commercially available extraction tools. DashO-Pro[11] and Condensity[12] are tools with similar goals as *Jax*. We are not aware of any published work on the algorithms used by these tools, or on the internal architecture of these tools.

There is a large body of work on reverse engineering that attempts to extract designs or object models from applications (see e.g., [12]). This work could benefit from application extraction technology because, by eliminating dead code, more precise designs could be extracted, and spurious relationships between classes or program constructs would not appear in the extracted designs. Similar to application extractors, design extraction tools face problems with dynamic language features such as reflection, in the sense that additional information from the user is needed to perform extraction.

8 Conclusions and Future Work

We have presented an approach in which different types of library-based applications can be accurately extracted in the presence of dynamic language features such as reflection and dynamic loading. Our approach relies on a simple, modular specification language, and supports a model where each module is written by the programmer who is most familiar with the code. We recently completed an implementation of our approach in the context of *Jax*, a widely used extraction tool [17], and initial feedback has been positive.

As future work, we intend to support the extraction of other widely used library types such as JavaBeans [15]. We also intend to study mechanisms for specifying additional properties of a library that would enable a tool to verify if a client application uses the library correctly.

Acknowledgements

We are grateful to John Field and Gregor Snelting for comments on drafts of this paper. Chris Laffra, David Streeter, Lisa Martin, and Sarvamangala Jagadeesh contributed to the development of *Jax*.

REFERENCES

[1] AGESEN, O. *Concrete Type Inference: Delivering Object-Oriented Ap-*

[10] A minor practical issue that comes up here is that constructor methods need to be made unique. At the Java class file level, this can be accomplished by adding additional dummy arguments.

[11] DashO-Pro is a trademark of preEmptive Solutions, Inc. See `www.preemptive.com`.

[12] Condensity is a trademark of Plumb Design, Inc. See `www.condensity.com`.

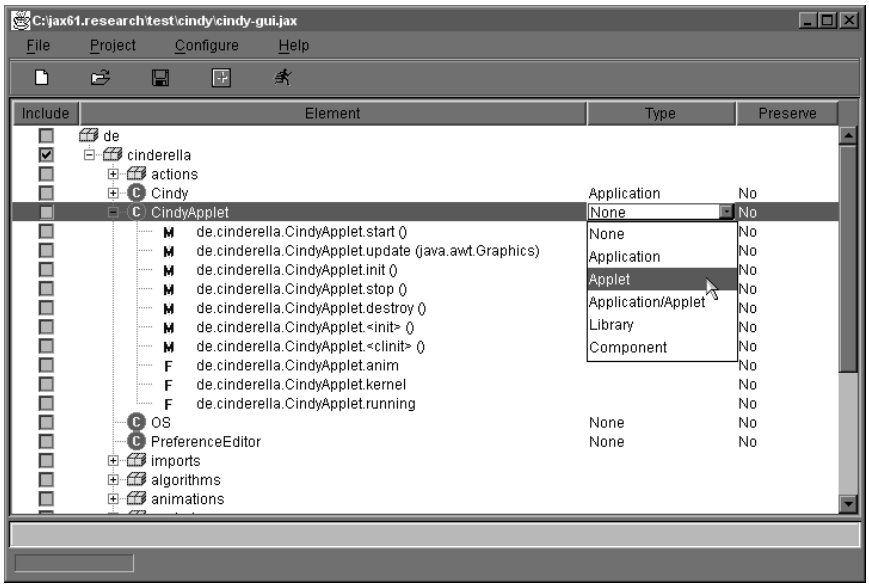

Figure 9: Selection of roles in *Jax*

.

plications. PhD thesis, Stanford University, December 1995. Appeared as Sun Microsystems Laboratories Technical Report SMLI TR-96-52.

[2] AGESEN, O., AND UNGAR, D. Sifting out the gold: Delivering compact applications from an exploratory object-oriented programming environment. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)* (Portland, OR, 1994), pp. 355–370. *ACM SIGPLAN Notices* 29(10).

[3] AIGNER, G., AND HÖLZLE, U. Eliminating virtual function calls in C++ programs. In *Proceedings of the Tenth European Conference on Object-Oriented Program (ECOOP'96)* ((Linz, Austria), July 1996), pp. 142–166.

[4] BACON, D. F. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, Computer Science Division, University of California, Berkeley, Dec. 1997. Report No. UCB/CSD-98-1017.

[5] BACON, D. F., AND SWEENEY, P. F. Fast static analysis of C++ virtual function calls. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, 1996), pp. 324–341. *SIGPLAN Notices* 31(10).

[6] CALDER, B., AND GRUNWALD, D. Reducing indirect function call overhead in C++ programs. In *Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages* (1994), pp. 397–408.

[7] CALLAWAY, D. R. *Inside Servlets: Server-Side Programming for the Java Platform*. Addison-Wesley, 1999.

[8] DEAN, J., GROVE, D., AND CHAMBERS, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95)* (Aarhus, Denmark, Aug. 1995), W. Olthoff, Ed., Springer-Verlag, pp. 77–101.

[9] DIGITALK INC. *Smalltalk/V for win32 Programming*, 1993. Chapter 17: "Object Libraries and Library Builder.

[10] DIWAN, A., MOSS, J. E. B., AND MCKINLEY, K. S. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, 1996), pp. 292–305. *SIGPLAN Notices* 31(10).

[11] IBM CORPORATION. *IBM Smalltalk User's Guide*, version 3, release 0 ed., 1995. Chapter 36: Introduction to Packaging, Chapter 37: "Simple Packaging, Chapter 38: "Advanced Packaging.

[12] JACKSON, D., AND WAINGOLD, A. Lightweight extraction of object models from bytecode. In *Proceedings of the International Conference on Software Engineering (ICSE '99)* (Los Angeles, CA), 1999).

[13] PARCPLACE SYSTEMS. *ParcPlace Smalltalk*, objectworks release 4.1 ed., 1992. Section 16: Deploying an Application, Section 28: Binary Object Streaming Service.

[14] SCHEIFLER, R. W. An analysis of inline substitution for a structured programming language. *Commun. ACM 20*, 9 (Sept. 1977), 647–654.

[15] SUN MICROSYSTEMS. *JavaBeans*, version 1.01 ed. 2550 Garcia Avenue, Mountain View, CA 94043, July 1997.

[16] SWEENEY, P. F., AND TIP, F. A study of dead data members in C++ applications. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Desigen and Implementation (PLDI '98)* (Montreal, Canada, June 1996), pp. 324–332. *ACM SIGPLAN Notices* 33(6).

[17] TIP, F., LAFFRA, C., SWEENEY, P. F., AND STREETER, D. Practical experience with an application extractor for Java. In *Proceedings of the Fourteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)* (Denver, CO), 1999). To appear.

[18] TIP, F., AND SWEENEY, P. F. Class hierarchy specialization. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)* (Atlanta, GA, 1997), pp. 271–285. ACM SIGPLAN Notices 32(10).