Extracting Library-Based Object-Oriented Applications

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

Frank Tip
IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
tip@watson.ibm.com

ABSTRACT

In an increasingly popular model of software distribution, software is developed in one computing environment and deployed in other environments by transfer over the internet. Extraction tools perform a static whole-program analysis to determine unused functionality in applications in order to reduce the time required to download applications. We have identified a number of scenarios where extraction tools require information beyond what can be inferred through static analysis: software distributions other than complete applications, the use of reflection, and situations where an application uses separately developed class libraries. This paper explores these issues, and introduces a modular specification language for expressing the information required for extraction. We implemented this language in the context of *Jax*, an industrial-strength application extractor for Java, and present a small case study in which different extraction scenarios are applied to a commercially available library-based application.

1. INTRODUCTION

In an increasingly popular software distribution model, software is developed in one computing environment, and deployed in other environments by transfer over the internet. Because the time required to transfer an application is generally proportional to the transferred number of bytes, it becomes important to make applications as small as possible. Application *extractors* are tools that reduce application size by determining unused functionality that can be removed from the application without affecting program behavior.

Previously, extractors have been designed primarily with complete applications in mind. Such whole-application extractors require one to specify an application's entry point(s), and rely on a static whole-program analysis to determine functionality that can be removed without affecting program behavior. However, the extraction of software distributions other than complete applications raises several issues:

- Modern object-oriented applications typically rely on one or more independently developed class libraries. With the advent of virtual machine technology, library code is amenable to the same analyses as application code, because the same representation is used in each case. When an application is distributed separately from the libraries it depends upon, an extraction tool needs to be aware of the *boundary* between the two.
- Different kinds of software distributions (e.g., complete applications, web-based applications that execute in the context of a browser, and extensible frameworks) have different sets of entry points, and require the application extractor to make different assumptions about the deployment environment. In fact, the same unit of software may even play different roles, depending on the deployment scenario.
- The use of dynamic features such as *reflection*[1] poses additional problems for extraction tools, because a static analysis alone is incapable of determining the program constructs that are used, and hence the program constructs that can be removed.
- There are also some interesting interactions between the above issues. For example, consider a situation where an application A is to be distributed together with an independently developed class library L in which reflection is used. In general, the use of reflection in L may depend on the *features* in L that are used by A. We will discuss how this observation affects extraction.

Each of these issues requires information that cannot be obtained using static analysis alone, and has to be provided to the extraction tool by the user. This paper explores the above issues in detail, and provides a uniform solution in the form of a small, modular specification language MEL (Modular Extraction Language) for providing the information required to extract various kinds of programs. MEL's features are essentially language-independent, with the exception of some Java-specific syntax used to refer to program constructs such as classes, methods and fields. In order to validate our approach, we implemented MEL in the context of *Jax*, an industrial-strength application extractor for Java developed at IBM Research [18]. We discuss how several of

SIGSOFT 2000 (FSE-8) 11/00 San Diego, CA, USA
© 2000 ACM ISBN 1-58113-205-0/00/0011...$5.00

[1] For convenience, we will henceforth use the term "reflection" to refer to all mechanisms for loading and accessing program constructs by specifying their name as a string value, and for examining program structure.

the program transformations and optimizations performed by *Jax* are adapted to take into account MEL scripts, and present a small case study in which different extraction scenarios are applied to a commercially available library-based Java application.

The remainder of this paper is organized as follows. In Section 2, we present the requirements on extraction tools in the presence of class library usage. Section 3 introduces a specification language for defining the extraction of various kinds of library-based applications. Section 4 presents a mechanism for translating specifications to a small set of assertions. Section 5 discusses an implementation of MEL, and reports on a small case study. Section 6 summarizes related work, and Section 7 presents conclusions and directions for future work.

2. REQUIREMENTS

In this section, we analyze a number of frequently occurring distribution scenarios, and determine what information is required by extraction tools beyond what can be obtained through static analysis.

2.1 Distribution scenarios

Figure 1 shows several distribution scenarios that may occur in the presence of: a library vendor l responsible for creating and distributing a class library L, an application vendor a responsible for creating and distributing an L-based application A, and two users, u and v, of application A.

It is reasonable to assume that library vendor l will want to make library L as small as possible, in order to reduce the download times experienced by customers, but also to reduce the load of the server from which the library is downloaded. Hence, l creates an *extracted version* L_{ext} of L, and distributes L_{ext} instead of L. Clearly, L_{ext} should offer the same functionality as L, but size-reducing optimizations can still be applied to parts of L that are not exposed to users.

Application vendor a presumably downloads L_{ext} for use during development of application A. When application A is ready for distribution, there are two options, depending on whether or not a user already has the prerequisite library L installed. Figure 1 shows a user u who does not have (the correct version of) L. Assuming that u does not expect to download or create other L-based applications, it is desirable for u to download a distribution AL_{ext} that comprises the functionality of A and the parts of L used by A, but that *omits* the parts of L that are not used by A. Because applications typically use only a small part of the functionality of libraries they rely on[2], the removal of the parts of L not used by A is likely to significantly reduce the size of the distribution.

There are also scenarios where it is preferable to keep the distributions of L and A separate. Figure 1 shows another user v of application A, who has downloaded L_{ext} directly from l, because he is planning to deploy multiple applications that rely on the library. Because v already has L_{ext}, he only needs to download the application itself from vendor a. To this end, a creates an extracted version A_{ext} of A that can be downloaded by v. It is important to realize that keeping the distributions of A and L separate has repercussions for the extraction of A itself. If we want to accommodate scenarios where v obtains a different *version*[3] of L, then the extractor should derived A_{ext} from A without making assumptions about the specific version of L that happens to be available in a's development environment. The standard Java libraries are an obvious example of this situation.

We will now investigate the issues related to the use of reflection. In essence, reflection allows one to access a program construct by specifying a run-time string value that represents the construct's name, and to examine the structure of the classes used in a program. Such features are problematic for extraction tools because, in general, a static analysis cannot determine which program constructs are accessed using reflection, and should therefore not be removed or transformed. Thus, extractors require additional information from the user that specifies which program constructs are accessed using reflection. In our experience, determining the program constructs that may be accessed using reflection 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 example of Figure 1, the extraction of AL_{ext} from A and L by application vendor a requires additional information about the use of reflection in L. This can be difficult to determine from distribution L_{ext} alone, because it does not contain the source code for the library. To complicate matters further, the set of program constructs in L accessed using reflection may *depend on the features in L that are used* by A. In general, different L-based applications may cause different usage of reflection within L. Our solution to these problems (discussed in detail below) will be to have library vendor l distribute a script along with L_{ext} that contains the information required to extract *any* L-based application. Our scripts allow l to specify that a program construct is only accessed using reflection under certain conditions (e.g., when a certain method is reachable).

We have only discussed a few example distribution scenarios. Other likely scenarios include:

- Extracting a library together with multiple applications that use it.
- Extracting a library in the context of another library that uses it. We believe that such situations, where multiple layers of libraries exist and where only the topmost layer is exposed to an application, is likely to become increasingly common.

2.2 Roles of software units

We will adopt the non-descriptive term *software unit* in the sequel to denote any collection of classes that constitutes a logical entity. Recall that 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. 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 the context of Java:

[2] In previous work on whole-application extraction [18], we reported that up to 60% of the methods in several library-based benchmark applications is unreached.

[3] This could either be an earlier version of L that was obtained from library vendor l, or a completely different implementation of the library from a different vendor.

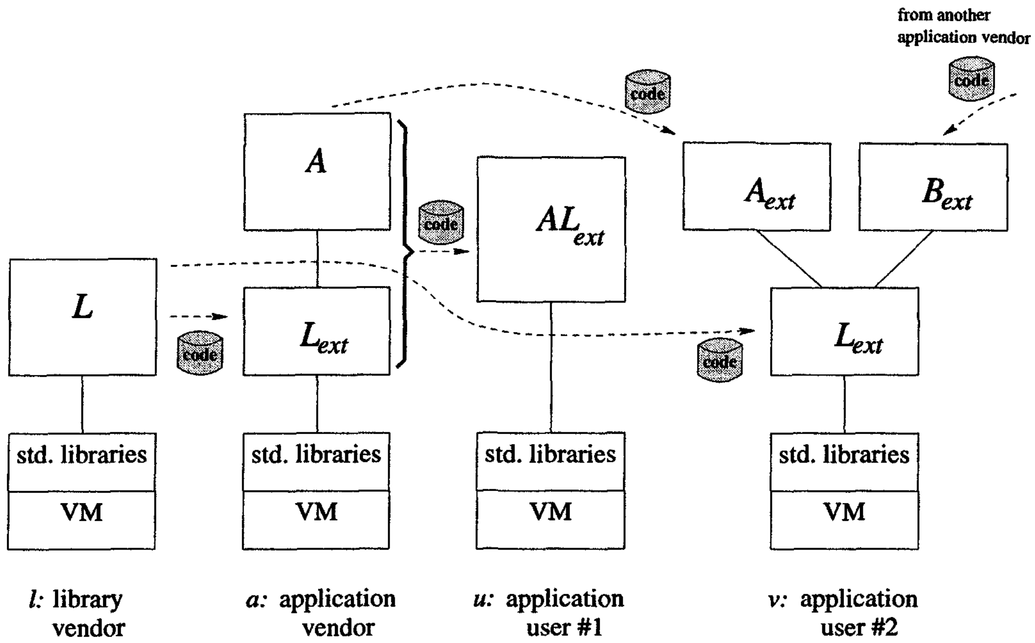

Figure 1: Illustration of different distribution scenarios.

- 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. 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 [16] and servlets [7] can be modeled similarly. For example, in the case of JavaBeans, all of the JavaBean's methods that may be invoked by client applications are contained in its external interface.

2.3 Specifying the extraction domain

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 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 extracted and shipped because they are already available in the deployment environment. In particular, the standard class libraries are generally available and are usually excluded from the extraction process.[4] It is important to realize that this is not merely an issue of avoiding redundant work and shipping redundant code, but potentially also one of correctness. If an application class contains a call to a method in the standard libraries, inlining that call on one platform may result in code that does not work on another platform.

2.4 Dealing with dynamic features

Java's reflection mechanism allows programs to do various forms of self-inspection. Figure 2 shows an example program that uses structural reflection (sometimes also referred to as introspection). 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:

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

```
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 2: A Java program that uses reflection.

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

Dynamic loading, another form of reflection, is a heavily-used[5] 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.[6]

Figure 3 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 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.

```
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 3: A Java program that uses dynamic loading.

Java provides a mechanism for implementing methods in a platform-dependent way, typically using 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: It would essentially be impossible to remove, rename, or transform any program construct. The approach taken in this paper relies on the user to 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.

2.5 Modeling different usage contexts

Section 2.1 already alluded to issues related to the use of third-party libraries in which reflection is used. In order to create MEL scripts that are *reusable* in different contexts, it is often desirable to specify that a given program construct is only accessed using reflection under certain conditions. To illustrate this issue, Figure 4 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 access M, and a client that calls g but not f will only access N. A specification of the library's behavior

[5] Nine of the thirteen benchmarks studied in [18] use dynamic loading.

[6] 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, we have observed that 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 4: Example class library that uses dynamic loading.

that states that *any* client of L accesses 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.

We conclude this section with an observation. In the above discussion, we have sketched two very different scenarios involving library L. In one example (the distribution of L_{ext} by l), all externally accessible L-methods should be treated as entry points in determining which methods are reachable. In the other scenario, (the distribution of AL_{ext} by a), only L-methods invoked from A and methods transitively reachable from those methods should be preserved. Hence, the decision on which methods to preserve requires information not present in the code of L. This precludes an approach based on annotating the code of L with additional information, unless different annotations are used to support different scenarios.

3. A SPECIFICATION LANGUAGE

Figure 5 presents a BNF grammar for a simple specification language, MEL (Modular Extraction Language), that allows users to specify at a high level how to extract a library-based application. The semantics of the various features in MEL are closely related to the discussions in Section 2. A MEL script comprises:

1. A *domain specification*, consisting of a class path where classes can be found, and a set of include statements that specify the extraction domain. 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. *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 2.2. *Preserve* statements are used to specify that program constructs (i.e., classes, methods, and fields) should be preserved because they are accessed either outside of the extraction domain or through reflection, and that worst-case assumptions should be made about these constructs. Following the discussion of Section 2.5, 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.

```
MELScript        ::=  Item*
Item             ::=  DomainSpecifier |
                         Statement | Import
DomainSpecifier  ::=  ClassPath | Include
ClassPath        ::=  path <Directory> |
                         path <ZipFile>
Include          ::=  include <Class> |
                      include <PackageName>
Statement        ::=  Role | Preserve
Role             ::=  application <Class> |
                      applet <Class> |
                      library <Class> |
                      component <Class>
Preserve         ::=  SimplePreserve |
                      CondPreserve
SimplePreserve   ::=  preserve <Class> |
                      preserve  <Method> |
                      preserve  <Field>
CondPreserve     ::=  SimplePreserve
                         when reached <Method>
Import           ::=  import <FileName>
```

Figure 5: BNF Grammar for the user-level information in MEL

```
import L;

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

Figure 6: Example application that uses the library of Figure 4.

Figure 6 shows an example application A that uses the library of Figure 4. Observe that A's main() routine creates an L-object and invokes L's method g(). Figures 7 and 8 present MEL scripts *L.mel* and *A.mel* for the library of Figure 4 and the application of Figure 6, respectively. The conditional preserve statements in *L.mel* 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.

```
path ···
include L
library L
preserve M when reached L.g()
preserve N when reached L.f()
```

Figure 7: Specification *L.mel* for the class library of Figure 4.

```
path ···
include A
application A
import L.mel
```

Figure 8: Specification *A.mel* for the application of Figure 6.

```
Statement        ::=  Assertion | ConditionalAssertion
Assertion        ::=  SimpleAssertion
Assertion        ::=  extendible <Class>
Assertion        ::=  overridable <Method>
SimpleAssertion  ::=  instantiated <Class>
SimpleAssertion  ::=  reached <Method>
SimpleAssertion  ::=  accessed <Field>
SimpleAssertion  ::=  preserveIdentity <Class>
SimpleAssertion  ::=  preserveIdentity <Method>
SimpleAssertion  ::=  preserveIdentity <Field>
CondAssertion    ::=  SimpleAssertion
                       when reached <Method>
```

Figure 9: BNF grammar for the extractor-level information in MEL.

4. IMPLEMENTATION STRATEGY

The specification language presented in Figure 5 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 information, we add a number of assertion constructs to MEL, 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 9 shows a BNF grammar for MEL 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 or accessed through reflection, which implies that the construct's name or signature should not be changed. The `extendible` 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.

The `instantiated`, `reached`, `accessed`, and `preserveIdentity` assertions also have a conditional form. Conditional assertions are used to model the conditional preserve statements that specify situations where reflection is used in a specific method.

Table 1 shows how statements are translated to assertions. The table contains a row for each type of MEL 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 its identity is preserved 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 allow this behavior, we assert that the class should be `extendible` and all of its methods should remain `overridable`.

The translation of `preserve` statements into assertions assumes that the identity of *any* program construct accessed outside the extraction domain or through reflection should be preserved. Hence, any program construct that is referenced in a `preserve` statement receives the `preserveIdentity` assertion. For preserved classes, we make the conservative assumption that they are instantiated if they are not `abstract` or an `interface`. Each preserved method is assumed to be invoked, and is therefore asserted to be `reached`. Similarly, each preserved field 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.

It is hard to make any completeness arguments about MEL. In our design of the high-level MEL statements, we have attempted to make it easy for the user to specify commonly occurring extraction scenarios. In addition, the low-level MEL assertions are sufficient to ensure that a program construct will not be affected by an extractor. In our implementation, we have given the user direct access to the lower-level MEL assertions as a fall-back option for extraction scenarios that are not supported by high-level MEL statements.

5. IMPLEMENTATION

In order to validate our approach, we implemented MEL

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 $C.m$ for every $C.m$ that overrides java.applet.Applet.m reached $C.m$ for every $C.m$ that overrides java.applet.Applet.m
component C	preserveIdentity C preserveIdentity $C.m$ for every public method $C.m$ reached $C.m$ for every public method $C.m$ preserveIdentity $C.f$ for every public field $C.f$ accessed $C.f$ for every public field $C.f$
library C	preserveIdentity C extendible C reached $C.m$ for every public or protected method $C.m$ preserveIdentity $C.m$ for every public or protected method $C.m$ overridable $C.m$ for every public or protected instance method $C.m$ accessed $C.f$ for every public or protected field $C.f$ preserveIdentity $C.f$ for every public or protected field $C.f$
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.

in the context of *Jax* [18].[7] The implementation also permits users to specify MEL assertions directly, and has mechanisms for specifying the name of the generated zip file, and for selectively disabling optimizations. *Jax* provides two mechanisms to support MEL. In "batch mode", a MEL script is read from a file, and the application is processed accordingly. A Graphical User Interface (GUI) that allows users to create MEL scripts interactively is also provided.

We will discuss how a number of program transformations and optimizations performed by *Jax* can be adapted to operate on various kinds of library-based applications by taking into account MEL assertions. These optimizations were originally presented as whole-programs optimizations, by making the "closed world" assumption that the entire program is available at analysis time.

5.1 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 MEL assertions. We will use Rapid Type Analysis (RTA) [5, 4], an efficient call graph construction algorithm, as a specific example. Other call-graph construction algorithms (see e.g., [9, 11, 19]) can be adapted similarly.

RTA 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 [18]. RTA is most easily implemented as an iterative algorithm that uses three worklists containing (i) reached methods, (ii) reached call sites[8], and (iii) instantiated classes. 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, where 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 MEL assertions, we first need to adapt the initialization of the work-

[7] Version 6.0 of *Jax* (released in August 1999) supports MEL in its full generality, although the syntax of the MEL keywords in the system differs slightly from the syntax used in this paper.

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

lists. 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.

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.

5.2 Dead method removal

Dead Method Removal [18] 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 call site that refers statically to an instance 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.

5.3 Call devirtualization

Call devirtualization [6, 3] transforms run-time dispatch calls into direct calls. This transformation can be applied at a call site x that calls 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 `extendible` C, where $C.m$ is the method invoked at call site x. Other optimizations that rely on closed-world assumptions (e.g., inlining [15] and call devirtualization) can be adapted similarly.

5.4 Dead field removal

Dead field removal [17] 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.[9] Fields that are 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 MEL 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.

5.5 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. A method that overrides a method outside the extraction domain cannot be renamed. Finally, if one method overrides another, both must be renamed correspondingly.[10] In the presence of MEL assertions, a number of additional constraints have to be imposed on the renaming of program constructs. Any program construct x for which there exists an assertion `preserveIdentity` x cannot be renamed, any method m for which there exists an assertion `overridable` m cannot be renamed, and any class c for which there exists an assertion `extendible` c cannot be renamed.

5.6 Class hierarchy transformations

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[11] 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 polymorphic calls into direct method calls. Space limitations do not permit a complete discussion of class merging here, and we refer the reader to [18, 20] for details. In order to take into account MEL assertions, any class C for which there exists an assertion `preserveIdentity` C should not be removed, or merged into its base class.

5.7 A Case Study

We now present a small case study in which different extraction scenarios are applied to *Cinderella*[12], an interactive geometry tool used for education and self-study in schools and universities. Cinderella consists of an application, which can be used for constructing interactive geometry exercises, and an applet in which students can attempt to solve these

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

[10] 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.

[11] 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.

[12] See `www.cinderella.de`.

distribution	zip file	classes	methods	fields
Original	664,826	337	3057	2391
Antlr (orig.)	226,648	130	1392	684
Antlr	181,535	130	1369	677
Applet	172,486	154	1241	789
Application	380,299	263	2382	1733
Both	390,463	268	2424	1743
Applet+Antlr	184,486	177	1355	842
Application +Antlr	391,813	285	2490	1784
Both + Antlr	403,327	293	2541	1797

Table 2: Results of multiple distribution scenarios for "Cinderella".

exercises. Two interesting observations can be made about Cinderella. First, the application and the applet are derived from the same code base, which is contained in a single zip file. Second, Cinderella relies on a class library called "ANTLR" for parsing.

Table 2 shows different distribution scenarios for Cinderella. The first two rows, labeled `Original` and `Antlr (orig.)` are concerned with the original distributions of Cinderella and ANTLR, respectively. The columns of the table show the size of the zip file, and the numbers of classes, methods and fields, respectively. The next row, labeled `Antlr` shows the result of extracting ANTLR as a stand-alone library. The reduction in size was obtained by removing several methods and fields that are only accessible inside the library. The next three rows, labeled `Applet`, `Application`, and `Both` shows the size of extracting the application, the applet, and their combination without ANTLR. Finally, the last three rows, labeled `Applet + Antlr`, `Application + Antlr`, and `Both + Antlr` show the results of extracting the application, the applet, and their combination together with the parts of ANTLR that they use.

The following observations can be made:

- The applet's functionality is (roughly) a subset of the application's functionality, because adding the applet to the distribution does not increase size by much.
- On the other hand, the size of the applet is significantly smaller than the combined distribution. Hence, users that only require the applet will prefer the distribution containing only the applet.
- From the fact that the distributions that include ANTLR are not much bigger than the distributions without ANTLR, we can infer that Cinderella uses only a small subset of ANTLR's functionality.
- Extracting ANTLR by itself results in a nontrivial (about 20%) reduction of distribution size. This confirms that extracting stand-alone class libraries is worthwhile.

6. RELATED WORK

We will begin this section with a brief historical perspective on this work. The approach taken in this paper was motivated by our experiences with *Jax*, an application extractor for Java [18]. *Jax* was developed as tool for extracting applications, and initially relied on ad-hoc solutions for several of the problems we study in this paper. For example, there was a fixed "boundary" between applications and the standard libraries and based on the names of classes, and a simple, low-level mechanism was provided to specify that certain program constructs accessed using reflection should be preserved. As a result, *Jax* was only suitable for distribution scenarios in which an application is shipped by itself, or where an application and a library are extracted and shipped together. The benchmarks studied in [18] are all instances of one of these scenarios. The work in this paper was motivated by our goal to accommodate other distribution scenarios such as independently shipped libraries, and to unburden the developer of a library-based applications from having to specify information (e.g., the use of reflection) about the library.

The extraction of applications was pioneered in the Smalltalk community, where it is usually referred to as "packaging" [12, 10, 14]. 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 contrast, we allow the user to specify where reflection occurs, so that applications that use reflection can be extracted.

Chen et. al [8] describe Acacia, an extraction tool for C/C++ based on a repository that records several relationships between program entities. Several types of reachability analyses can be performed, including a forward reachability analysis for determining entities that are unused. Chen et al. identify several issues that make extraction difficult such as the use of libraries for which code is unavailable, and situations where functionality should be preserved because source modules are shared with other applications. Unlike our work, Acacia is an analysis tool aimed at providing information to the user, and does not actually perform any program transformations such as dead code elimination. A number of issues that we study such as the use of reflection are not discussed, and no mechanism appears to be available for supplying additional information to the extractor.

In the context of Java, we are aware of a number of several commercially available extraction tools. DashO-Pro[13] and Condensity[14] 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., [13]). 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

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

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

not appear in the extracted designs. Similar to application extractors, design extraction tools require additional information from the user in the presence of dynamic language features such as reflection.

7. CONCLUSIONS AND FUTURE WORK

We have identified a number of situations where the extraction of software requires information that cannot be obtained by static analysis techniques alone. This includes software distributions other than complete applications, the use of reflection in applications, and situations where library-based applications are extracted and distributed separately.

To address these issues, we have proposed a small, modular specification language, MEL, that allows one to specify the information required for extraction in a uniform manner. We have argued that the modular nature of MEL scripts allows for a useful separation of responsibilities: each module of a MEL script can be written by a programmer who is familiar with the code, and extraction of an application that relies on third-party libraries only requires a MEL script for that library. We have discussed how several whole-program transformations performed by extractors can be adapted to various other kinds of software units by taking into account the information contained in MEL specifications. Our approach was implemented in the context of Jax, an application extractor for Java [18], and we presented a small case study that involves several realistic extraction scenarios.

We intend to support the extraction of other widely used library types such as JavaBeans [16]. Other topics for ongoing research include adding more sophisticated conditional features to MEL such as conditions based on *paths* in call graphs, and boolean conjunction and disjunction of conditions. Furthermore, we are considering "safety" features such as the insertion of run-time checks to ensure that the information in a MEL script is correct and complete.

Acknowledgements

We are grateful to John Field, Harold Ossher, Gregor Snelting, and the anonymous FSE referees for comments on drafts of this paper.

8. REFERENCES

[1] Agesen, O. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Stanford U., 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 *Proc. of the Ninth Annual Conf. 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 *Proc. of the Tenth European Conf. 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, U. 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 *Proc. of the Eleventh Annual Conf. 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 *Proc. 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] Chen, Y.-F., Gansner, E. R., and Koutsofios, E. A C++ data model supporting reachability analysis and dead code detection. *IEEE Transactions on Software Engineering 24*, 9 (Sept. 1998), 682–694.

[9] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. of the Ninth European Conf. on Object-Oriented Programming (ECOOP'95)* (Aarhus, Denmark, Aug. 1995), W. Olthoff, Ed., Springer-Verlag, pp. 77–101.

[10] Digitalk Inc. *Smalltalk/V for win32 Programming,* 1993. Chapter 17: "Object Libraries and Library Builder.

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

[12] 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.

[13] Jackson, D., and Waingold, A. Lightweight extraction of object models from bytecode. In *Proc. of the International Conf. on Software Engineering (ICSE '99)* (Los Angeles, CA), 1999).

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

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

[16] Sun Microsystems. *JavaBeans,* version 1.01 ed. 2550 Garcia Avenue, Mountain View, CA 94043, July 1997.

[17] Sweeney, P. F., and Tip, F. A study of dead data members in C++ applications. In *Proc. of the ACM SIGPLAN'98 Conf. on Programming Language Desigen and Implementation (PLDI '98)* (Montreal, Canada, June 1996), pp. 324–332. *ACM SIGPLAN Notices* 33(6).

[18] Tip, F., Laffra, C., Sweeney, P. F., and Streeter, D. Practical experience with an application extractor for Java. In *Proc. of the Fourteenth Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)* (Denver, CO), 1999), pp. 292–305. *SIGPLAN Notices* 34(10).

[19] Tip, F., and Palsberg, J. Scalable propagation-based call graph construction algorithms. In *Proc. of the Fifteenth Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)* (Minneapolis, MN), 2000). To appear.

[20] Tip, F., and Sweeney, P. F. Class hierarchy specialization. In *Proc. of the Eleventh Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)* (Atlanta, GA, 1997), pp. 271–285. ACM SIGPLAN Notices 32(10).