Practical Extraction Techniques for Java

FRANK TIP and PETER F. SWEENEY
IBM T. J. Watson Research Center
CHRIS LAFFRA and ALDO EISMA
Object Technology International
and
DAVID STREETER
IBM Toronto Laboratory

Reducing application size is important for software that is distributed via the internet, in order to keep download times manageable, and in the domain of embedded systems, where applications are often stored in (Read-Only or Flash) memory. This paper explores extraction techniques such as the removal of unreachable methods and redundant fields, inlining of method calls, and transformation of the class hierarchy for reducing application size. We implemented a number of extraction techniques in *Jax*, an application extractor for Java, and evaluated their effectiveness on a set of large Java applications. We found that, on average, the class file archives for these benchmarks were reduced to 37.5% of their original size. Modeling dynamic language features such as reflection, and extracting software distributions other than complete applications requires additional user input. We present a uniform approach for supplying this input that relies on MEL, a modular specification language. We also discuss a number of issues and challenges associated with the extraction of embedded systems applications.

Categories and Subject Descriptors: F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*; D.3.4 [**Programming Languages**]: Processors—*Compilers; Optimization*

General Terms: Algorithms, Languages, Performance, Experimentation, Measurement

Additional Key Words and Phrases: Application extraction, call graph construction, class hierarchy transformation, whole-program analysis, packaging

This paper borrows material from three previously published conference papers [Tip et al. 1999; Sweeney and Tip 2000; Tip and Palsberg 2000]. More information about the *Jax* project can be found at: www.research.ibm.com/JAX. A free evaluation copy of *Jax* (along with documentation and examples) can be downloaded from: www.alphaWorks.ibm.com/tech/JAX.

Authors' addresses: F. Tip and P. F. Sweeney, IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598; email: tip@watson.ibm.com; pfs@us.ibm.com. C. Laffra and A. Eisma, Object Technology International, Burg. Haspelslaan 131, 1181 NC Amstelveen, The Netherlands; email: {Chris_Laffra; Aldo_Eisma}@oti.com; D. Streeter, IBM Toronto Laboratory, 1150 Eglinton Ave. East, Toronto, Ontario, Canada; email: daves@ca.ibm.com.

1. INTRODUCTION

Application size is an important limiting factor for software distribution over the internet, because the time required to download an application is proportional to that application's size. In the embedded systems domain, where memory is a scarce resource and where applications are often stored in Read-Only or Flash memory, it is also desirable to keep applications as small as possible, in order to keep costs and power-consumption low. At odds with these size requirements is the increasing use of third-party class libraries and components in both PC-based and embedded applications. Vendors of applications that rely on third-party libraries often cannot assume that the user has installed (the correct version of) these libraries, and are therefore forced to include such prerequisite libraries in their distribution, resulting in a significant size increase.

Extraction tools reduce the size of applications and other types of software distributions by removing unused functionality and applying size-reducing program transformations. The contributions of this paper are as follows:

—We study several compiler optimizations and program transformations that can be used as extraction techniques (e.g., removal of unreachable methods, and transformations of the class hierarchy), and investigate the pragmatic issues that arise when applying these techniques to a full-scale programming language.

—We implemented these extraction techniques in *Jax*,[1] an application extractor for Java[2] [Joy et al. 2000], and determined their effectiveness on a set of large Java benchmarks ranging from 45 to 2,326 classes (with corresponding class file archives of 55,765 to 3,810,120 bytes). We measured that, on average, the class file archives for these benchmarks were reduced to 37.5% of their original size.[3] More dramatic size reductions are not uncommon for large, library-based applications in which much unused library functionality can usually be pruned away.

—The extraction of applications that use reflection and dynamic loading, and of software distributions other than complete applications requires information that cannot be determined with static analysis alone. We analyze these issues in detail, and present a modular specification language that allows a user to specify the extraction of various kinds of software distributions, while allowing an extractor to treat all types of distributions uniformly. A small case study is presented in which a number of extraction scenarios is applied to one of our benchmarks.

—A number of the extraction techniques performed by *Jax* have also been implemented in SmartLinker, a packaging tool that is incorporated into IBM's

[1] *Jax* has been available from IBM's alphaWorks web site (`www.alphaWorks.ibm.com/tech/jax`) since June 1998, and is one of the most popular tools there, having received over 30,000 downloads. Several commercial software products (developed both inside and outside IBM) have been processed with *Jax* before being shipped.

[2] Java and all Java-based marks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

[3] All average ratios reported in this paper are computed using the geometric mean.

WebSphere Studio Device Developer (WSDD), an environment for developing Java applications for embedded systems. SmartLinker shares a significant amount of infrastructure with *Jax*, but also incorporates functionality that is specific to the embedded systems domain, such as support for partially pre-linked libraries.

The remainder of this paper is organized as follows. Section 2 surveys the extraction techniques implemented in *Jax*. Many of these techniques rely on call graphs, and Section 3 presents a number of call graph construction algorithms that we experimented with. Section 4 presents the results of applying *Jax* to set of large benchmark applications. Section 5 is concerned with MEL, a modular language for specifying various kinds of software distributions. Section 6 examines issues that arise in the embedded systems domain, and discusses the extraction techniques incorporated in SmartLinker. Related work is presented in Section 7. Finally, Section 8 presents conclusions and directions for future work.

2. EXTRACTING APPLICATIONS

This section presents a high-level overview of the transformations and optimizations that *Jax* employs as extraction techniques. We will use the example Java application of Figure 1(a) to illustrate the transformations performed by *Jax*. This application comprises classes `AA`, `BB`, `CC`, and `Example`, and an interface `II`. `II` declares a method `foo()` that is overridden in `AA`, `BB`, and `CC`, and `AA` defines a method `bar()` that is overridden in `BB`. Moreover, `AA` declares three fields `xx`, `yy`, and `zz`. The program contains two direct call sites (the calls to methods `ff()` and `gg()`), and three dynamically dispatched call sites (the calls `i1.foo()`, `i2.foo()`, and `this.bar()`). Figures 1(b-d) show several steps in the transformation process that will be discussed shortly. Although shown as source-to-source transformations here, in reality these transformations are performed at the class file level.

2.1 Loading the Application

Jax begins by reading in the application from the original archive(s), and constructing an in-memory representation of the Java class files for that application. *Jax* only loads classes that contain the application's entry points, and any classes that are directly or indirectly referenced from those classes.

Java provides a mechanism (in the Java literature referred to as *dynamic loading*) that allows an application to load a class (and create an object of that type) by providing the class name as a string. Because these strings are run-time values, it is in general not possible for a static analysis to determine which classes are dynamically loaded by an application. Therefore, *Jax* relies on the user to specify all classes that are dynamically loaded (the specification language that is used to provide this information will be presented in Section 5). Dynamically loaded classes are treated as additional entry points for the class loading process.

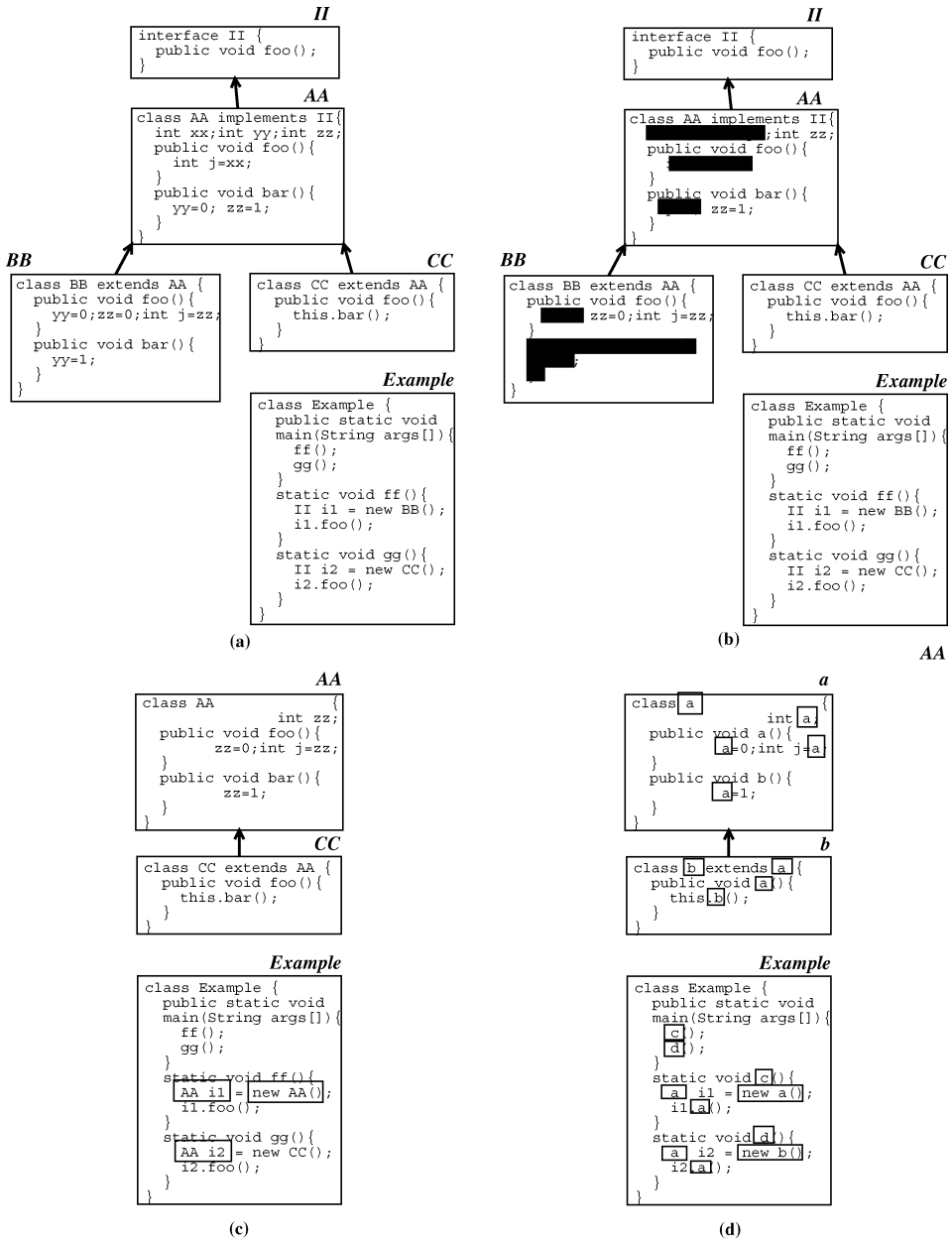

Fig. 1. **(a)** Example program. **(b)** View of the program after *Jax* has removed: unreachable method `BB.bar()`, the body of unreachable method `AA.foo()`, unaccessed field `AA.xx`, and write-only field `AA.yy`. Removed code fragments have been blacked out. **(c)** The program after merging class `BB` into class `AA`, and interface `II` into class `AA`. The affected code fragments are shown boxed. **(d)** The program after name compression has been applied. The affected code fragments are shown boxed.

2.2 Removal of Redundant Class File Attributes

Bytecode attributes that are unnecessary for *executing* the program such as local variable name tables and line number tables are discarded during loading.

2.3 Call Graph Construction and Removal of Unreachable Methods

In general, only a subset of the methods in the loaded classes are required by an application. Unreachable methods may arise for many reasons. One reason is that, as systems grow, programmers tend to lose track of the methods that are used. More importantly, breaking up applications into components that are designed and implemented separately leads to unreachable methods, because the creators of a component cannot always anticipate what functionality is needed. The scenario where applications use general-purpose class libraries that are developed elsewhere is a well-known example of this situation. Section 3 presents several call graph construction algorithms that have been incorporated in *Jax* to detect unreachable methods. After applying one of these algorithms, unreachable methods are removed. As we shall see shortly, there are cases where an unreachable method cannot be removed for syntactic reasons. In such cases, *Jax* can still remove the body of this method and replace it with a **return** statement.

Figure 1(b) shows the program after removing unreachable method `BB.bar()`, and removing the body of unreachable method `AA.foo()` (method `AA.foo()` is unreachable, but it cannot be removed because class `AA` implements interface `II`, which declares method `foo()`).

2.4 Redundant Field Elimination

Any field that is only accessed from unreachable methods can be removed from the application. Fields that are only written to (but not read) can also be removed because their value cannot affect the program's behavior [Sweeney and Tip 1998]. In addition to removing the field itself, this involves removal of the instructions that store the value in the field.

In Figure 1(b), unaccessed field `AA.xx` and write-only field `AA.yy` have been removed (along with the assignments to field `yy` in methods `AA.bar()` and `BB.foo()`).

2.5 Class Hierarchy Transformations

Jax applies a number of semantics-preserving transformations to the class hierarchy. These transformations reduce archive size by eliminating classes entirely, and by merging adjacent classes in the hierarchy. Merging classes may enable the transformation of virtual method calls into direct method calls, and may reduce the duplication of literals across the constant pools of different classes (this will be discussed in more detail in Section 2.8). The class hierarchy transformations used in *Jax* are an adaptation of the ones in Tip and Sweeney [1997, 2000], where they serve as a simplification phase after the generation of specialized class hierarchies (the relationship with this work is discussed in Section 7).

One of the simplest transformations is the removal of an uninstantiated class that does not have any derived classes, and that does not contain any live methods or fields. More interesting is the transformation where a base class X and a derived class Y are merged if there is no live non-`abstract` method f that occurs in both X and Y, and: (1) X is uninstantiated, or (2) Y does not contain any live non-static fields. By requiring that (1) or (2) holds, we ensure that no object created by the application becomes larger (i.e., contains more fields) as a result of the merge.

Merging a base class X with a derived class Y involves a number of steps. All live methods and fields of Y are moved to X. Cases where X and Y have fields or static methods with identical names pose no problem, because we can simply rename any field or static method in cases where name conflicts occur. Constructors require special treatment because X and Y may have constructors with identical signatures, and constructors cannot be renamed. In such cases, a new signature for the constructor is synthesized by adding dummy arguments, and constructor calls are updated accordingly by pushing `null` elements on the stack. If X and Y both have static initializer methods, we turn the static initializer for X into an ordinary static method, and insert a call to that method at the beginning of the initializer for Y.[4] If X and Y both contain an instance method f, then at least one of these methods must be `abstract` due to the preconditions stated above. If both methods f are `abstract`, $Y.f$ is simply removed. Otherwise, the non-`abstract` method is preferred over the `abstract` method. Finally, all references to Y, as well as methods and fields in Y are updated to reflect their new "location" in class X.

Other class hierarchy transformations include the merging of classes with interfaces, and are very similar to the transformation described above. For more details on class hierarchy transformation, we refer the reader to Tip and Sweeney [2000]. However, a few more issues pertaining to class merging should be mentioned. In order to allow the merging of classes across `package` boundaries, classes, methods and fields need to be made `public`. Finally, classes that are explicitly referenced using reflection cannot be merged with other classes.

Figure 1(c) shows the example program after merging `BB` into `AA`, and `II` into `AA`. Note that the `new BB` statement in `Example.main()` has changed to `new AA`, and that the types of variables `i1` and `i2` have changed from `II` to `AA`. The affected declarations are shown in a box. Observe that merging `BB` and `AA` was *not possible in the original class hierarchy*, because classes `AA` and `BB` both contained non-abstract methods `foo()` and `bar()`. Hence, this class merging operation was *enabled* by the fact that methods `AA.foo()` and `BB.bar()` were found to be unreachable.

2.6 Name Compression

A Java class file is a self-contained unit of executable code. References to other classes, methods, and fields are made through string literals. For example, if a class contains a method call `Thread.sleep(500)`, the constant pool for that class

[4]This assumes that programs do not rely on the execution order of static initializers of different classes.

contains the strings "`java/lang/Thread`", "`sleep`", and "`(J)V`", representing the fully qualified class name, the method name, and a string representation of the method's signature (one argument of type `long`; returning `void`). Because all linking information is represented in string form, and is replicated in each class file, it is obvious that replacing class, method, and field names by shorter ones will result in smaller archives. *Jax* currently renames classes, methods, and fields `a`, `b`, `c`, ... but more ambitious naming schemes, in which methods with different signatures get the same name are possible. Certain names cannot be compressed. This includes any reference to a class, method, or field in an external library, methods that override methods in external libraries, any program component accessed using reflection, the name of class containing the `main` routine, and and the names of constructors and static initializers.

Figure 1(d) shows the program after name compression has been applied. Here, classes `AA` and `CC` have been renamed to `a` and `b`, respectively, and all references to these types have been updated correspondingly. Moreover, methods `foo()`, `bar()`, `ff()`, and `gg()` have been renamed to `a()`, `b()`, `c()`, and `d()`, respectively, and field `zz` was renamed to `a`. Note that class `Example` and method `Example.main()` cannot be renamed because this would affect the external entry point to the application.

2.7 Performance Optimizations

Thus far, we have primarily focused on archive size reduction; optimization was only a secondary goal. However, a few simple and easy-to-implement optimizations have been implemented in *Jax*. Non-overridden methods are inlined in cases where this does not increase application size. Moreover, in cases where a virtual dispatch has only one potential target, we "devirtualize" the call by replacing an `invokevirtual` with an `invokespecial` bytecode [Calder and Grunwald 1994; Aigner and Hölzle 1996]. Unfortunately, the Java Virtual Machine Specification [Lindholm and Yellin 1997] only permits this in a very limited number of situations: the callee has to occur in a superclass of the caller class, or has to be a private method. *Jax* marks non-overridden virtual methods `final` so that a just-in-time compiler can inline these calls where appropriate.

The example program of Figure 1 illustrates how transformations enable each other. We have already seen how elimination of (the body of) unreachable methods enables class merging. Note that, in the resulting program of Figure 1(d) only a single method `b()` remains in the entire class hierarchy. This implies that the call to `b()` can be inlined.

2.8 Constant Pool Compression

Removing unreachable methods and redundant fields may render constant pool entries unnecessary. In the in-memory representation of class files constructed by *Jax*, references to constant pool entries are replaced by explicit references to objects representing the classes, methods, fields, and constants normally contained in the constant pool. After the transformations described above have been performed by *Jax* the class is written out again, and a new constant pool is created from scratch. Only the classes, methods, fields, and constants that are

actually referenced will be added to this constant pool. The resulting constant pool has minimal size and is typically much smaller than the constant pool originally found in the class file.

Class merging has interesting repercussions for the size of constant pools. Adjacent classes in the hierarchy are likely to share many literal values, which are *duplicated* in their constant pools. Merging these classes allows us to eliminate this duplication. We will see in Section 4.5 that the contribution of class hierarchy transformation to archive size reduction can be significant.

3. CALL GRAPH CONSTRUCTION

3.1 Call Graph Construction Algorithms

The key step in call graph construction is to conservatively approximate the "target" methods that can be invoked by a dynamic dispatch. Various algorithms have been proposed to determine the potential targets of a dynamic dispatch, including Class Hierarchy Analysis (CHA) [Dean et al. 1995], 0-CFA [Shivers 1991; Grove et al. 1997], VTA [Sundaresan et al. 2000], and algorithms for alias or points-to analysis [Pande and Ryder 1996; Shapiro and Horwitz 1997; Steensgaard 1996].

Jax initially used CHA and Rapid Type Analysis (RTA) [Bacon and Sweeney 1996; Bacon 1997]. Later, we adopted the more precise XTA algorithm [Tip and Palsberg 2000]. Ignoring a number of details,[5] each of these call graph construction algorithms relies on the following steps to compute a set of reachable methods N:

(1) The set N is initialized with a set of initially reachable methods, such as the `main()` method of an application.
(2) The body of each method n in N is analyzed, and the call sites that occur in the body of n are stored in a set $CallSites(n)$.
(3) For each call site s in $CallSites(n)$, the set of method definitions that may be reached from that call site is determined, and added to N.

It is important to realize that steps 2) and 3) must be performed iteratively until no additional reached methods can be found. The final value for N represents the set of nodes in the call graph. The CHA, RTA, and XTA algorithms differ in the way a virtual call site $v.m(\cdots)$ is resolved in step 3:

—CHA only uses information that is available in the class hierarchy, and assumes that all methods $W.m(\cdots)$ may be reached, where W is any subtype of the declared type V of receiver v.
—RTA keeps track of the classes that have been instantiated, and stores them in a set S. To this end, the analysis of method bodies in step 2) above is extended. In particular, for each instruction "`new C`" that is encountered, class C is added to S. In order to determine the set of methods reachable from

[5]In particular, we ignore the construction of the *edges* of a call graph here. For an in-depth comparison of these algorithms, the reader is referred to Tip and Palsberg [2000].

call site $v.m(\cdots)$, RTA uses the set S to approximate the possible run-time types of variable v. Specifically, for any type $X \in S$ that is a subtype of V, RTA determines class $W = \textit{static-lookup}(X, m(\cdots))$[6] and adds method $W.m(\cdots)$ to N.

—XTA is similar to RTA, in the sense that it tracks allocation sites in order to approximate dynamic dispatch behavior. However, whereas RTA uses a single set S that represents types that may be available anywhere in the application, XTA uses a distinct set of types S_m for each method m, representing the types of objects that may occur in m, and a distinct set S_f for each field f representing the types that may be stored in field f. XTA computes these sets of types by iteratively propagating types between callers and callees (modeling the flow of objects passed through parameters), between callees and callers (modeling object flow via return values) from methods to fields (by examining which fields are written to by methods), and from fields to methods (corresponding to fields read by methods). In each case, information about declared types is used to restrict the flow of types between sets. For example, when a method calls another method, the declared types of the callee's parameters are used to determine which types may flow from caller to callee.

3.2 Example

To better understand the workings of these algorithms, we will examine the call graphs constructed for the program of Figure 1(a) by CHA, RTA, and XTA:

—CHA determines that `AA.foo()`, `BB.foo()`, and `CC.foo()` can be reached from call sites `i1.foo()` and `i2.foo()`, because classes `AA`, `BB`, and `CC` each provide an overriding definition of `I.foo()`. Following a similar argument, CHA determines that both `AA.bar()` and `BB.bar()` can be reached from call site `this.bar()`.

—RTA uses the fact that no object of type `AA` is created *anywhere in the program* to rule out `AA.foo()` as a potential target of the calls `i1.foo()` and `i2.foo()`, and determines that only `BB.foo()` and `CC.foo()` can be reached. RTA is unable to detect that method `BB.bar()` is unreached, because it finds that both `AA.bar()` and `BB.bar()` may be reached from call site `this.bar()`.

—XTA finds that the `BB`-object created in method `ff()` cannot reach method `gg()`, by analyzing how objects are passed around via parameter passing and reads/writes to fields. Likewise, XTA determines that the `CC`-object created in `gg()` cannot reach `ff()`. Hence, XTA finds that the call site `i1.foo()` in `ff()` can only invoke `BB.foo()`, and that the call site `i2.foo()` in `gg()` can only invoke `CC.foo()`. Moreover, XTA determines that no objects of type `BB` reach call site `this.bar()`, and finds that method `BB.bar()` is unreached. This situation was illustrated earlier in Figure 1(b).

[6]Here, $\textit{static-lookup}(X, m(\cdots))$ is a function that determines the definition of method $m(\cdots)$ that is invoked if the receiver of the method call has run-time type X.

Table. I. Characteristics of the benchmark applications used to evaluate *Jax*: For each benchmark, the initial number of classes, methods, and fields is shown. The size of the initial archive shown here is in bytes and excludes any resource files contained in the shipped archives

benchmark	# classes	# methods	# fields	archive
Hanoi	45	378	233	55,765
Jax	302	2,900	1,274	534,658
javac	210	1,512	1,107	452,125
bloat	282	2,677	1,255	506,736
mBird	2,050	17,946	6,739	2,950,543
Jinsight	264	2,974	1,875	393,857
JavaFig	160	2,108	1,526	394,432
CindyApplet	467	4,449	3,075	881,555
Cinderella	467	4,449	3,075	881,555
eSuite Sheet	588	5,590	4,305	1,251,765
eSuite Chart	733	8,302	5,448	1,570,569
Hyper/J	921	8,776	2,733	1,523,670
Res. System	2,326	21,495	12,487	3,810,120

3.3 Pragmatic Issues

Implementing a call graph construction algorithm for the full Java language requires that a number of pragmatic issues be addressed.

Class initializer methods are executed upon the first active use of a class (i.e., when the class is instantiated, when a `static` method in the class is called, or when a non-`final` `static` field in the class is accessed [Joy et al. 2000, Section 12.4]). Our analysis adds an initializer method to the set of reached methods when an active use of its class is observed.

When applications make use of class libraries, methods in the application may be invoked from callbacks within those libraries. Consider a situation where a class C in the application extends a library class L, and suppose that C provides an overriding definition for a method $L.f()$. Then, a virtual dispatch inside L can resolve to method $C.f()$ in the application's code. If the code for the library is unavailable, worst-case assumptions have to be made regarding such callbacks (for details, see Tip and Palsberg [2000]).

Java's reflection mechanism [Arnold et al. 2000] allows one to invoke a method by specifying as a string (computed at run-time) its name and signature. In general, it is not possible to predict all possible run-time values of such strings at analysis time. Therefore, *Jax* relies on the user to specify the methods that are invoked via reflection and dynamic loading. This will be discussed at length in Section 5.

4. RESULTS

4.1 Overview of the Benchmarks

Table I lists the Java applications used to evaluate *Jax*. The benchmarks cover a wide spectrum of programming styles and are publicly available (except for *mBird* and *Reservation System*). For each benchmark, the initial number of classes, methods, and fields are shown, as well as the initial size of the archive.

Hanoi is an interactive applet version of the well-known "Towers of Hanoi" problem, and is shipped with *Jax*. An earlier version of *Jax* itself (version 7.1) was used as a benchmark. *javac*[7] is the SPEC JVM 98 version of Sun's source to byte-code compiler. *bloat*[8] is a byte-code optimizer developed at Purdue University. *mBird* is a proprietary IBM tool for multi-language operability. It relies on, but uses only limited parts of, several large class libraries (including Swing, now part of the standard libraries, and IBM's XML parser). *Jinsight*[9] is a performance analysis tool developed at IBM Research. *JavaFig*[10] (version 1.43 (22.02.99)) is a Java version of the `xfig` drawing program. *Cinderella*[11] is an interactive geometry tool used for education and self-study in schools and universities. *CindyApplet* is an applet that allows users to solve geometry exercises interactively. It is contained in the same class file archive as *Cinderella*. *Lotus eSuite Sheet*[12] is an interactive spreadsheet applet, which is part of the examples shipped with Lotus' *eSuite*, a productivity suite. *Lotus eSuite Chart* is an interactive charting applet, another example shipped with Lotus *eSuite*. *Hyper/J*[13] is a system for advanced separation of concerns developed at IBM research. *Reservation System* is an interactive front-end for an airline, hotel, and car rental reservation system developed by an IBM customer.

4.2 Measurement Issues

For a number of the benchmarks, the shipped version of the initial archive contains resource files such as properties files and image files. Since our techniques only address the transformation of class files, we moved all resource files to a separate archive. This "resources archive" is unaffected by *Jax*, and its contents should be added to the archive produced by *Jax* in order to run the compressed application. Currently, our techniques do not address the issue of determining which resources are actually used by an application, and we have observed cases where archives contained many unneeded resources.

Another issue is that different implementations of *zip* and *jar* tend to produce slightly different results. In order to give a consistent evaluation, all archives mentioned in this paper have been unzipped, and subsequently rezipped (into a single archive) using WinZip 7.0.[14]

4.3 Reductions in Archive Size, Classes, Methods, and Fields

Table II shows the overall size reductions obtained by applying *Jax* (version 7.4) to the benchmarks of Table I, as well as the time required by *Jax* to process the

[7]See `www.specbench.org`.
[8]See `www.cs.purdue.edu/homes/hosking/pjama.html`.
[9]See `www.research.ibm.com/jinsight`.
[10]See `tech-www.informatik.uni-hamburg.de/applets/javafig`.
[11]See `www.cinderella.de`.
[12]See `www.esuite.lotus.com`.
[13]See `www.research.ibm.com/hyperspace`.
[14]See `www.winzip.com`.

Table. II. The number of classes, methods, and fields and the archive size for the benchmark applications of Table I after processing by *Jax*. The rightmost column shows the time (in seconds) required by *Jax* to process the application

benchmark	# classes	# methods	# fields	archive	time
Hanoi	21	179	97	21,055	7.0
Jax	272	2,128	739	333,268	22.8
javac	198	1,342	496	231,605	21.1
bloat	245	2,266	635	251,822	23.9
mBird	230	1,934	743	276,884	19.6
Jinsight	222	2,469	1,036	317,166	22.0
JavaFig	114	1,443	1,055	223,913	15.1
CindyApplet	171	1,325	831	179,702	21.5
Cinderella	280	2,456	1,773	385,900	28.7
eSuite Sheet	240	2,378	939	330,677	28.0
eSuite Chart	398	4,472	2,129	598,985	38.9
Hyper/J	428	3,584	924	424,074	80.0
Res. System	1,432	11,540	5,004	1,725,421	256.4

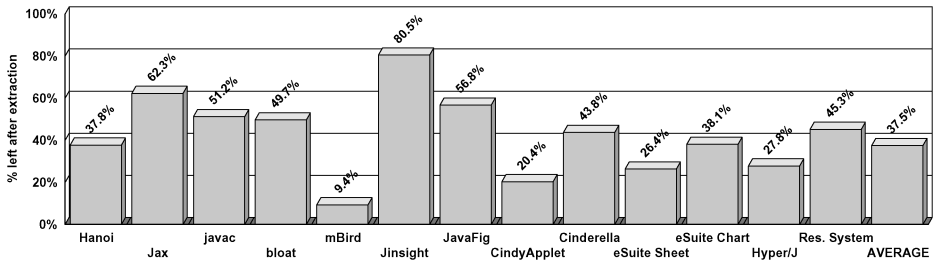

Fig. 2. Size of the extracted benchmark applications of Table I as a percentage of the size of the original archive. Resource files are excluded from both the initial and processed archives.

benchmarks.[15] *Reservation System*, the largest benchmark, was processed in about 4.5 minutes. We consider these processing times to be quite acceptable, especially since application extraction is typically an infrequent activity that is only performed when applications are shipped.

Figure 2 depicts the size of the extracted archives as a percentage of the size of the original archive. As can be seen from the figure, the size of the extracted archives ranges between 9.4% and 80.5% of the original size (37.5% on average). Figure 3 depicts the number of classes, methods, and fields in the extracted benchmarks, as a percentage of the original number of classes, methods, and fields, respectively. As is shown in the figure, the number of classes in the extracted benchmarks ranges between 11.2% to 94.3% (53.9% on average) of the original number of classes. Moreover, the number of methods in the extracted benchmarks ranges between 10.8% to 88.8% (50.3% on average) of the original number of methods, and the number of fields in the extracted benchmarks

[15]All measurements were taken on a Pentium III/800Mhz PC with 1 processor, a 64K L2-cache and 512MB memory running Windows 2000. All measurements were conducted using Sun JDK 1.2.2, in combination with a Just-In-Time compiler developed at IBM [Ishizaki et al. 1999].

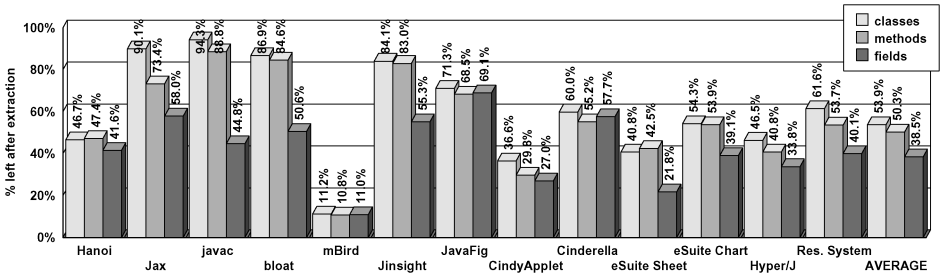

Fig. 3. Number of classes, methods, and fields in the extracted benchmark applications of Table I, shown as a percentage of the original number of classes, methods, and fields, respectively.

ranges between 11.0% to 69.1% (38.5% on average) of the original number of methods.

We have conducted measurements of the reductions in download times over 56K modem connections and fast LAN connections and found that, in general, the reduction in download time is similar to the reduction in archive size (within a few percentage points).

4.4 Evaluation

A number of observations can be made about the results reported above.

The benchmark for which we measured the smallest reduction in archive size is *Jinsight*. A discussion with the developers revealed that this benchmark does not rely on any class libraries other than the standard libraries, and that most of the removed methods correspond to future extensions that were never fully implemented, and to functionality that had become obsolete.

The benchmark for which we measured the greatest reduction in archive size, *mBird*, is a special case. *mBird* consists of two distinct components: a tool with an interactive GUI and a command-line "batch" tool. These tools are usually shipped together as a single class file archive. In our evaluation, we extracted *only* the batch component from this archive. The main reason for the very large size reduction is that *Jax* is very effective in removing the unused GUI-related library classes. Other benchmarks for which we measure large reductions such as *Hanoi* (37.8% of the original size remaining), *Lotus eSuite Sheet* (26.4% remaining), *Lotus eSuite Chart* (38.1% remaining), and *Hyper/J* (27.8% remaining) either rely on class libraries, or are structured as a class library with a client application. The large size reductions we measure for these benchmarks are in agreement with the general perception that applications typically use only a small fraction of the functionality in class libraries that they rely on, and it shows that our techniques are quite successful in eliminating redundant library functionality.

The *Cinderella* and *Cinderella Applet* benchmarks are another interesting case because they are derived from the same original archive. In this case, *Cinderella Applet* contains (roughly) a subset of the *Cinderella*'s functionality. For development purposes, it is desirable to have the two applications share the same archive, but for distribution purposes it is undesirable to ship the entire archive for *Cinderella Applet*. A common solution to such problems consists

Table. III. Detailed Measurements for the *Hyper/J* Benchmark

Hyper/J	classes	methods	fields	archive
original size	921	8,776	2,733	1,523,670
unreferenced classes	789	7,674	2,602	1,309,962
redundant attributes	789	7,674	2,602	1,021,698
dead methods	789	3,870	2,602	711,678
redundant fields	789	3,870	924	650,461
inlining/devirtualizing	789	3,844	924	648,646
class transformations	428	3,584	924	535,606
name compression	428	3,584	924	424,074

of splitting the classes into a package with "core functionality", and separate packages with additional functionality that is used by different components. This approach has some obvious organizational drawbacks. In such cases, an application extractor can extract the desired functionality for each application. The fact that *Jax* is capable of eliminating unused functionality is evident from the fact that we see a much larger reduction for *Cinderella Applet* (20.4% remaining) than for *Cinderella* (43.8% remaining). Section 5.5 explores different distribution scenarios for *Cinderella* in more detail. Encouraged by these results, the creators of *Cinderella* decided to use *Jax* to create their various distributions.

4.5 Breakdown of the Results

Measuring the individual contributions of each transformation performed by *Jax* is complicated by the fact that each transformation's effectiveness strongly depends on the preceding transformations. For example, the removal of useless fields is performed after the removal of unreachable methods (otherwise, we would not be able to remove fields that are only referenced within unreached methods). Hence, correlating the contributions of dead fields or methods *in isolation* to the reduction in archive size would be meaningless. Another argument along these lines is that the number of classes that can be merged is strongly dependent on removal of unused fields and methods in a previous step. Consequently, what we will study in the remainder of this section is the *cumulative* effect of each step. By selectively disabling steps performed by *Jax*, we measure the additional impact of each step.

Table III shows detailed statistics gathered for the *Hyper/J* benchmark, indicating the size of the archive, and the numbers of classes, methods, and fields after each step. Figures 4 and 5 show the contributions of the successive steps in graphical form. These figures reveal several interesting facts. A substantial number of classes in the initial archive (132 out of 921) are not loaded. Removal of these unreferenced classes reduces the archive to 86.0% of its original size. Most of these unused classes are library classes that are shipped with, but not used by the application. Removal of redundant attributes such as line number tables and local variable name tables contributes 18.9% to the reduction in archive size, reducing the archive to 67.1% of its original size. Removal of unreachable methods results in an additional reduction of 20.4%, further reducing the archive to 46.7% of its original size. The contribution of useless field removal

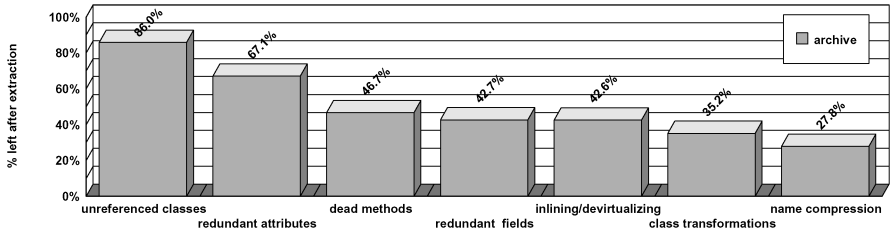

Fig. 4. Archive size for *Hyper/J* as a percentage of the original archive size after (1) loading, (2) removal of unreachable methods, (3) removal of useless fields, (4) method inlining/devirtualizing, (5) class hierarchy transformations, and (6) name compression.

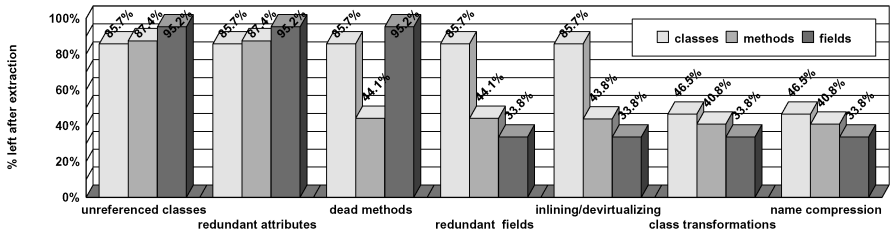

Fig. 5. Number of classes, methods, and fields for *Hyper/J* as a percentage of the original version after (1) loading, (2) removal of unreachable methods, (3) removal of useless fields, (4) method inlining/devirtualizing, (5) class hierarchy transformations, and (6) name compression.

is relatively small: 4.0%. Many of the removed fields are static final fields. Java compilers apply constant propagation and replace each occurrence of such a field by its value. The original field remains, even though it is now redundant. In the case, of *Hyper/J*, almost 2,800 fields are removed from the application. Method inlining and devirtualizing have only a small effect on the result (0.1%). The contribution of class hierarchy transformations is a noticeable 7.8%, reducing the archive to 35.2% of its original size. The *Hyper/J* benchmark is written in a highly object-oriented style, with heavy use of interfaces. Large sections of the class hierarchy are flattened by *Jax*, as is evidenced by the fact that the number of classes is reduced from 789 to 428. From Table III and Figure 5 it can be seen that the class hierarchy transformations remove an additional 3.0% of the methods (260 methods). These are `abstract` methods that disappear as a result of class merging. Name compression reduces the resulting archive by another 7.4%, resulting in a final archive that is only 27.8% of its original size.

4.6 The Impact of Call Graph Precision

All previously discussed results are based on the use of the XTA algorithm [Tip and Palsberg 2000] for identifying unreachable methods (see Section 3). In order to determine the impact of call graph precision, we also extracted *Hyper/J* using the CHA and RTA algorithms. Table IV shows a comparison of the archive size, and number of classes, methods, and fields we obtained for *Hyper/J* using CHA, RTA, and XTA. These results are depicted in Figure 6. As can be seen from the chart, using CHA results in an extracted application containing 52.9% of the methods in the original application. RTA and XTA are far more effective

Table. IV. Comparative Measurements for *Hyper/J* using Class Hierarchy Analysis and Rapid Type Analysis

Hyper/J	classes	methods	fields	call graph edges	archive
original size	921	8,776	2,733	N/A	1,523,670
processed with CHA	551	4,640	1,141	30,791	554,091
processed with RTA	434	3,597	924	20,754	429,153
processed with XTA	428	3,584	924	19,826	424,074

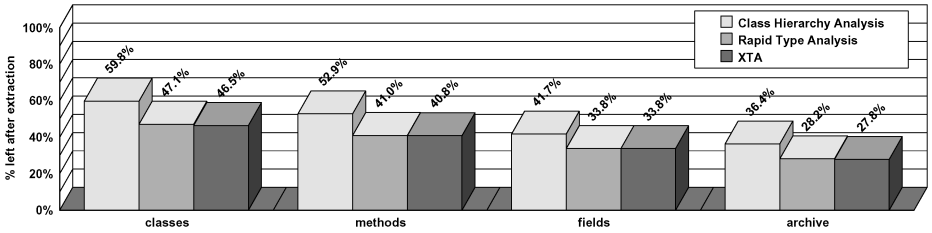

Fig. 6. Effect of different call graph construction algorithms (Class Hierarchy Analysis, Rapid Type Analysis, and XTA) on the the number of classes, methods, fields, and archive size of the extracted *Hyper/J* application. The results are shown as a percentage of the original size of the application.

and reduce the number of methods to 41.0% and 40.8%, respectively. The use of CHA results in an archive that is 36.4% of its original size, as opposed to 28.2% for RTA, and 27.8% for XTA. Observe that the removal of additional methods has a measurable impact on the number classes that can be merged or removed. Extracting *Hyper/J* using CHA results in an archive with 551 classes, compared with 434 classes using RTA, and 428 classes using XTA.

It is clear from Table IV and Figure 6 that XTA is only marginally more precise than RTA as far as detecting unreached methods is concerned. However, XTA can be noticably more effective than RTA when it comes to detecting redundant call graph edges. A closer look revealed that XTA constructs a call graph with 19,826 edges compared to an RTA call graph with 20,754 edges. Upon further examination, we found that the RTA call graph contained 4,592 monomorphic call sites and 2,385 polymorphic call sites. XTA found a unique target for 380 of these RTA-polymorphic call sites: in 16.1% of all cases. Although *Jax* could not devirtualize many of these call sites due to the constraints imposed on Java byte codes, XTA may enable more devirtualization and inlining when a target representation other than class files is used or when class file annotations are used in conjunction with a JIT. Section 8.2 discusses annotations as future work.

4.7 Execution Time Speed-Up

Table V shows the running time for the four non-interactive benchmarks (*Jax*, *javac*, *mBird*, and *Hyper/J*) before and after applying *Jax*. The other benchmarks are all interactive GUI-based applications, so that direct speedup measurements are difficult to conduct. For *Jax*, *javac* and *mBird* the speedups are small: 3.1%, 3.9%, and 3.5%, respectively. For *Hyper/J*, the speedup is a more noticeable 14.3%.

Table. V. Speed-Up Measurements for the Noninteractive Benchmarks. Times shown are in seconds

benchmark	execution time (original)	execution time (processed)	speedup
Jax	7.43	7.20	3.1%
javac	20.37	19.58	3.9%
mbird	2.02	1.95	3.5%
Hyper/J	5.58	4.78	14.3%

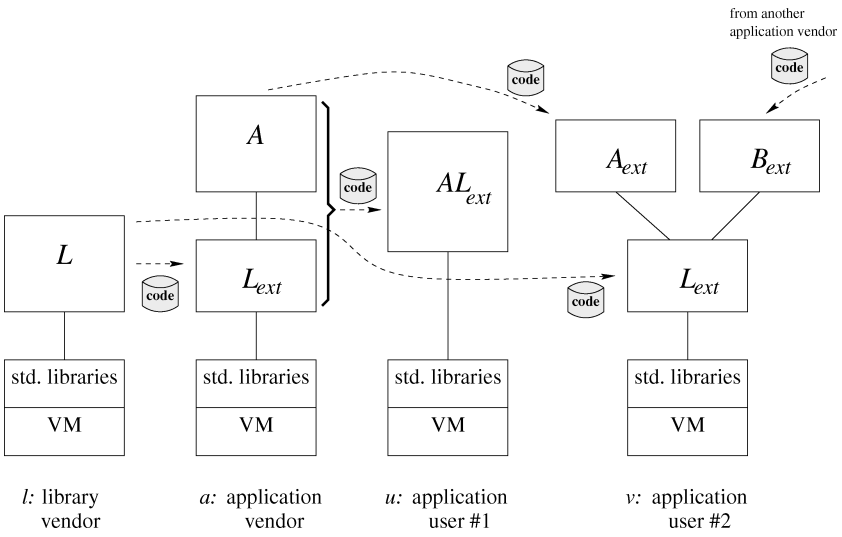

Fig. 7. Illustration of different distribution scenarios.

5. EXTRACTING OTHER KINDS OF SOFTWARE DISTRIBUTIONS

Thus far, we have only studied the extraction of applications. To illustrate the issues that arise when extracting other kinds of software distributions, Figure 7 depicts several distribution scenarios that frequently arise in practice. In this figure, a library vendor l is responsible for creating and distributing a class library L, an application vendor a is responsible for creating and distributing an L-based application A, and two users named u and v, use application A.

Scenario 1: Extract a library without assumptions about its clients. In general, library vendor l will want to make library L as small as possible. 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, since l cannot make assumptions about the way in which L is used by applications. In particular, it has to be assumed that any `public` and `protected` method[16] in L may be called by client applications. However, size-reducing transformations may still be applied to parts of L not exposed to users. In particular, `private` methods that cannot be called (directly or indirectly) by any `public` or `protected` method may be removed.

[16]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.

Another issue that arises when extracting libraries is whether or not the client should be able to create subclasses of the library classes. As we will see shortly, this will impact certain optimizations such as devirtualization and inlining.

Scenario 2: Extract a library in the context of a specific application. Application vendor u 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. For users such as u who do not have (the correct version of) L installed, 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 from L any method that is not transitively reachable from A's `main()` method. One setting where this scenario frequently arises is in the area of embedded system applications, where an application is packaged with the parts of the run-time libraries that it uses into a single executable (see Section 6).

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 a needs to ship A to a user who has (a *different implementation* of) L, who wants to *share* L between different L-based applications, or when L contains code that is platform-specific. This is illustrated in Figure 7 by the distribution A_{ext} that is being shipped by application vendor a to user v. In this case, determining the set of reachable methods in A requires that worst-case assumptions be made in cases where methods in A override methods defined in L. The experiments reported on in Section 4 are an instance of this scenario. Here, the standard Java libraries were not analyzed or extracted because we could assume these to be available on the client side, and to avoid making platform-specific assumptions.

For each of these distribution scenarios, different assumptions about the deployment environment must be made. The remainder of this section explores the issues that arise, discusses the additional information that needs to be supplied by the user to extract different kinds of software distributions, and presents a uniform solution in the form of MEL, a modular specification language for specifying extraction.

5.1 Requirements and Design Issues

In the subsequent discussion, we will use the term *software unit* to denote any collection of classes that constitutes a logical entity. Extracting software distributions other than complete applications requires information that cannot be obtained using static analysis alone, and that has to be provided by the user:

—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 an extractor to make different assumptions about the deployment environment. As Figure 7 illustrates, the same unit of software may even play different roles, depending

on the deployment scenario. Hence, the user needs to specify what role is played by a software unit.

—Modern object-oriented applications typically rely on independently developed class libraries. With the advent of virtual machine technology, libraries and applications are amenable to the same analyses, because the same representation is used. 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.

—Dynamic features such as *reflection* pose 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.

—Some interesting interactions between the above issues exist. For example, consider a situation where an application A is distributed together with an independently developed class library L that uses reflection. Then, the use of reflection in L may depend on the *features* in L that are used by A.

We will now investigate each of these issues in more detail.

5.1.1 *Roles of Software Units.* The same representation is used for library code and application code, and only the way in which software units are *used* and *composed* determines how they should be extracted. The term *role* will be used to refer to the way in which a software unit is used. We 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 extended by derivation after extraction.

—An *applet* is a software unit that is executed in the context of a browser. An applet extends class `java.applet.Applet` and its external interface consists of the methods of that class that it overrides. We assume that classes in applets are not extended by derivation after extraction.

—A *library* is an incomplete program that is used as a building block by other units. Classes in libraries may be extended by derivation. A library's external interface consists of any method and field with `public` or `protected` access rights.

—A *component* is another type of incomplete program. Unlike a library, classes in a component are assumed not to be extended after extraction. The external interface of a component contains every `public` method and field.

Other roles such as JavaBeans [Sun Microsystems 1997] and Servlets [Callaway 1999] can be modeled similarly.

5.1.2 *Specifying the Extraction Domain.* As the same representation is used for software units with different roles, it becomes 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.

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

(a)

```
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();
};
```

(b)

Fig. 8. **(a)** Example Java program that uses structural reflection. **(b)** Example Java program that uses dynamic loading.

It is important to realize that specifying the boundary between software units is not merely an issue of avoiding redundant work and shipping redundant code, but potentially also one of correctness. For example, if an application class contains a call to a method in a class that is not extracted, inlining that call on one platform may result in code that does not work on another platform.

5.1.3 *Dealing with Dynamic Features.* Java's reflection mechanism allows programs to do various forms of self-inspection. For example, in the program of Figure 8(a), the class that represents the type `T` of object `t` is retrieved by calling `java.lang.Object.getClass()`. Then, `java.lang.Class.getDeclaredMethods()` is called to obtain an array of objects representing the methods in `T`, and the name of each method is accessed by calling `java.lang.reflect.Method.getName()`. Hence, the program prints: "`foobar`". Clearly, program behavior depends on the *presence* and *name* of `T`'s methods, even though these methods are not invoked anywhere. Obviously, this use of reflection precludes program transformations such as the removal or renaming of methods in `T` that affect program behavior.

Dynamic Loading, another form of reflection, is a heavily-used[17] 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, a static analysis cannot

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

determine which classes are dynamically loaded.[18] The program of Figure 8(b) 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()`. The program then calls method `java.lang.Class.newInstance()` to create an 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 call graph construction algorithms that need to know which classes are instantiated, and which methods are invoked (see Section 3).

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 code is provided in an object file (e.g., a dynamically linked library) associated with the Java application. The native code may instantiate classes, invoke methods, and access fields, which 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[19] 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.

5.1.4 *Modeling Different Usage Contexts.* Consider Figure 9(a), which shows a small class library consisting of three classes `L`, `M` and `N`. Class `L` has two methods: `f()` and `g()`. A call to `f()` results in the dynamic loading of class `M`, and a call to `g()` results in the dynamic loading of class `N`. Hence, 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 stating that *any* client may access both `M` and `N` is clearly overly conservative. We therefore allow *conditional* specifications of the form "program construct X should be preserved if method m is reachable". This allows one to express how reflection

[18] 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.

[19] Determining where reflection is used in unfamiliar code can be difficult, especially if source is unavailable. To assist users with this task, the *Jax* distribution includes a tool that instruments calls to the reflection API. Running the instrumented application produces a file that lists the classes, methods, and fields that are accessed via reflection in that specific execution. Although there is no guarantee that all uses of reflection are exposed by the tool, in practice it usually suffices to exercise all menus, buttons and other GUI components.

```
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 { ··· };
```

(a)

```
import L;

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

(b)

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

(c)

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

(d)

Fig. 9. **(a)** Example class library that uses dynamic loading. **(b)** Example application that uses the library of **(a)**. **(c)** Specification *L.mel* for the class library of Figure 9(a) **(d)** Specification *A.mel* for the application of Figure 9(b).

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>

Fig. 10. BNF Grammar for the user-level information in MEL.

depends on the *usage* of a software unit's functionality, and enables the creation of a single, reusable configuration file that can be used to accurately extract a software unit in the context of any client.

5.2 A Specification Language

Figure 10 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. 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 that is not `included` is considered "external" in the sense that it will

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

Fig. 11. BNF grammar for the extractor-level information in MEL.

not be extracted, and that worst-case assumptions will be made about its behavior.

(2) A set of statements. *Role* statements designate the role of classes in the extraction domain as `application`, `applet`, `component`, or `library` (see Section 5.1.1). *Preserve* statements specify that classes, methods, or fields should be preserved because they are accessed using reflection or from within native methods, and that worst-case assumptions should be made about these constructs. Following the discussion of Section 5.1.4, program constructs can be conditionally preserved depending on the reachability of a specified method.

(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.

Figure 9(b) shows an example application `A` that uses the library of Figure 9(a). Observe that `A`'s `main()` routine creates an `L`-object and invokes `L`'s method `g()`. Figures 9(c) and (d) present MEL scripts *L.mel* and *A.mel* for `L` and `A`, 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.

5.3 Implementation Strategy

The specification language of Figure 10 was designed to make the specification of the extraction process convenient. However, the transformations applied by extraction tools typically require low-level information (e.g., potentially reached methods and instantiated classes). 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 assertions. An important benefit of this approach is that all roles and usage scenarios can be treated uniformly by the extractor.

Figure 11 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's name or signature may not be changed because it may be accessed from outside the extraction domain through reflection. There is also a conditional form of each of these constructs, to model the conditional preserve statements of Figure 10. The `extendible` and `overridable` assertions express that a class/method may be extended/overridden after extraction.

Table VI. Translation of Statements into Assertions

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` 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 VI shows how each type of MEL statement is translated into a set of assertions. The translation process for roles can be summarized as follows:

—For each role, the appropriate methods are assumed to be invoked from outside the extraction domain. For example, for classes that play a `library` role all `public` and `protected` methods are assumed to be invoked. Each such method is assumed to be `reached`, and its identity is preserved to account for the fact that external references to its name and signature may exist.

—For each `role`, the appropriate fields are assumed to be accessed from outside the extraction domain, and `accessed` and `preserveIdentity` assertions are generated. 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 started. We model this by asserting that each applet class is `instantiated`.

—For classes that play a `library` role, we assume that subclassing and method overriding may take place after extraction. To this end, we assert that the class is `extendible` and all of its `public` and `protected` methods are `overridable`.

A `preserveIdentity` assertion is generated for any program construct referenced in a `preserve` statement, because explicit references to the construct's name and signature may exist (e.g., using reflection). We conservatively assume that a `preserved` class is `instantiated` if it is not `abstract` or an `interface`, that a `preserved` method is `reached`, and that a `preserved` field is `accessed`. Translating conditional `preserve` statements involves carrying over the condition from the statement to the assertion, but is completely analogous otherwise.

It is hard to make completeness arguments about MEL. In designing MEL's statements, our goal has been to make specification of commonly occurring extraction scenarios convenient. MEL assertions are sufficient to ensure that a program construct will not be transformed in any way. In our implementation, we have given the user direct access to MEL assertions as a fall-back option for extraction scenarios that are not currently supported. One instance where this has already been useful is a situation where the main class of an application contained an unaccessed field called "copyright" containing a copyright message. Since this field was not accessed, an explicit `preserveField` assertion had to be supplied to preserve it.

5.4 Implementation

In our implementation of MEL, we added mechanisms for specifying the name of the generated zip file, and for selectively disabling optimizations and transformations. We will now describe how several extraction techniques introduced in Section 2 were adapted to accommodate MEL assertions. While we do not claim to be the first to adapt these optimizations to software distributions other than complete applications, we are not aware of any previous systematic treatment of the subject.

Call graph construction. In order to accommodate MEL assertions in our implementation of XTA,[20] the algorithm outline of Section 3.1 was adapted as follows. In step 1), the initial set of reachable methods is made to include any method m for which an assertion `reached` m exists. In addition, all sets S_f and S_m are initialized to contain each class C for which an assertion `instantiated` C exists. Then, in the iterative part of the algorithm, the following additional steps are performed:

—whenever a method m is added to N such that an assertion `instantiated` C `when reached` m exists, C is added to S_m if it does not occur in S_m, and
—whenever a method m is added to N such that an assertion `reached` m' `when reached` m exists, m' is added to N if it does not occur in N.

[20]RTA was adapted similarly.

Removal of dead methods and fields. Dead method removal relies solely on call graph information. No information is needed beyond what was discussed above. Dead field removal is adapted to handle MEL assertions by considering a field $C.f$ to be read-accessed if there exists an assertion `accessed` $C.f$. Conditional `accessed` assertions are treated similarly as conditional `reached` assertions.

Call devirtualization. A dynamically dispatched call to a method $C.m$ can be transformed into a direct call if only one method can be reached from the call site under consideration, and if 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 such as inlining [Scheifler 1977] can be adapted similarly. In the presence of MEL assertions, a call to a virtual method $C.m$ for which an assertion `overridable` $C.m$ exists cannot be inlined, because the method may be overridden after extraction, and we conservatively assume that the call site may resolve to these overriding method definitions.

Name compression. The presence of MEL assertions imposes additional constraints 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.

Method finalization and class finalization. MEL assertions are accommodated by not finalizing any class C for which an assertion `extendible` C exists, and by not finalizing any method m for an assertion `overridable` m exists.

Class hierarchy transformations. In order to accommodate MEL assertions, any class C for which there exists an assertion `preserveIdentity` C should not be removed, or merged into its base class.

5.5 A Case Study

We studied several extraction scenarios for *Cinderella*, an interactive geometry tool used for education and self-study in schools and universities. *Cinderella* consists of an application, used for constructing interactive geometry exercises, and an applet with which students can attempt to solve these exercises. Both are derived from a single code base, which is contained in a single zip file. Another fact of interest is that `Cinderella` relies on a class library called *Antlr* for parsing.

Table VII shows several distribution scenarios. The first two rows (`Application + Applet (unprocessed)` and `Antlr (unprocessed)`) show the original distributions of *Cinderella* and *Antlr*, respectively. The columns of the

Table. VII. Results of Multiple Distribution Scenarios for "Cinderella"

Cinderella	# classes	# methods	# fields	archive
Application + Applet (unprocessed)	337	3,057	2,391	658,397
Antlr (unprocessed)	130	1,392	684	225,324
Antlr (processed)	130	1,369	677	180,171
Applet (processed)	154	1,225	788	176,397
Application (processed)	265	2,363	1,732	390,833
Applet + Application (processed)	270	2,405	1,742	400,593
Applet + Antlr (processed)	171	1,325	831	179,702
Application + Antlr (processed)	280	2,456	1,773	385,900
Applet + Application + Antlr (processed)	292	2,508	1,786	411,328

table show the numbers of classes, methods, and fields, and the size of the zip file, respectively. The next row (`Antlr (processed)`) shows the result of extracting *Antlr* as a stand-alone library. The next three rows, (`Applet (processed)`, `Application (processed)` and `Applet + Application (processed)`) show the size of extracting the application, the applet, and their combination without *Antlr*. The bottom three rows show the application, applet, and their combination, extracted with the parts of *Antlr* that they use. The following observations can be made:

—The applet contains (roughly) a subset of the application's functionality, since the `Application + Applet (processed)` distribution (400,593 bytes) is only marginally bigger than the `Application (processed)` distribution (390,833 bytes).

—On the other hand, the size of the extracted applet (176,397 bytes) is significantly smaller than the combined distribution (400,593 bytes). Hence, users who only require the applet will prefer this distribution.

—The distributions that include *Antlr* are not much bigger than the distributions without *Antlr*. Hence, we can infer that *Cinderella* uses only a small subset of *Antlr*'s functionality. In fact, the distribution of *Cinderella* without *Antlr* (`Application (processed)`, 390,833 bytes) is *larger* than the distribution of *Cinderella* with *Antlr* (`Application + Antlr (processed)`, 385,900 bytes). This is due to the fact *not including Antlr* forces *Jax* to make conservative assumptions about the *Antlr* library that increase distribution size.

—Extracting *Antlr* by itself results in a nontrivial (about 20%) reduction of distribution size (due to the removal of several methods and fields and to the removal of redundant class file attributes). This confirms that extracting stand-alone class libraries is worthwhile.

5.6 Semantic Requirements and Challenges

The extraction techniques implemented in *Jax* have been designed with the preservation of program behavior in mind. Specifically, for a software unit U that is extracted by *Jax*, our goal has been to guarantee that:

(1) The behavior of the code in U itself is preserved.

(2) The behavior of a unit V that uses U is preserved if V's usage of U matches the assumptions made about U during extraction (e.g., if U was extracted

as a component, V's behavior is preserved only if it does not extend classes in U).

There is an interesting analogy with the concept of *binary compatibility*. The Java Language Specification (JLS) [Joy et al. 2000, Chapter 13] states that a number of changes may be made to a class C without requiring recompilation of pre-existing binaries compiled against C. For example, deleting a `private` field from a class does not necessitate recompilation of any other classes and is therefore a binary-compatible change. Extracting an *incomplete* application raises similar issues: We need to ensure that the interactions between the extracted software unit and other software units in its deployment environment are not affected by any transformations performed by *Jax*. Note that this is a *stronger* requirement than binary compatibility because being able to compile a software unit against an extracted version of U does not guarantee behavioral equivalence by itself.

Our approach to achieve goal (2) above is to rely on information about the deployment environment that the user has provided in MEL scripts, and to limit the transformations applied to U in such a way that software units that interact with U are not affected. We will now discuss the implications for the specific transformations performed by *Jax*.

Removal of classes, methods, and fields. Classes, methods, and fields that are accessed by other software units may not be removed because this would break binary compatibility [Joy et al. 2000, Section 13.4.11] (and therefore change program behavior). Moreover, a method $m()$ in U that overrides a method in another software unit can only be removed if it can be determined that method calls outside U will never dispatch to $m()$.

Renaming of classes, methods, and fields. The JLS [Joy et al. 2000, Section 13.4.12] states that the renaming of a method should be considered as a deletion and an addition of a method for the purposes of determining binary (in)compatibility. Following a similar argument as above, this implies that no class, method, or field can be renamed if it is referenced from outside U. Moreover, methods in U that override methods in other software units, or that may be overridden by clients of U cannot be renamed because this might cause behavioral changes. Additional restrictions on renaming due to reflection and native methods were already discussed in Section 5.1.3.

Finalization of classes and methods. The JLS [Joy et al. 2000, Section 13.4.2] states that making a class `final` will result in a `VerifyError` being thrown if a binary of a pre-existing subclass is loaded. *Jax* relies on the user to specify the classes that may be subclassed after extraction (such classes must be declared as *library* classes in MEL scripts), and assumes that any other class may be made `final`. The finalization of methods raises similar issues.

Method call inlining. Inlining a call site s amounts to changing the body of the method containing s, and does not break binary compatibility by itself [Joy et al. 2000, Section 13.4.20]. However, the method $m()$ that is invoked at s must be part of U. Otherwise, behavioral changes may occur if U is deployed

in an environment with a different implementation of m. A method in U for which all call sites have been inlined may only be removed if the conditions for removal (see above) have been met.

Class merging. Merging a class B into a class A has the effect of removing class B and adding B's methods to class A. This is only allowed if the conditions for removing class B have been met (see above). Moving a method from class B to class A does not affect binary compatibility [Joy et al. 2000, Section 13.4.11] or program behavior (even in the case where an `abstract` method is removed from A). Conditions related to class merging not specific to the extraction of incomplete applications were previously discussed in Section 2.

6. EXTRACTION OF EMBEDDED SYSTEMS APPLICATIONS

Embedded devices have a number of special characteristics that make the applications running on these devices excellent candidates for extraction:

—Embedded devices tend to have little RAM, and less powerful CPUs than desktop computers, because of cost, size, and power consumption considerations.
—Embedded devices tend to be diskless, and operating system and applications are stored in ROM or flash memory.
—Battery-powered devices often need to be turned off or hibernated when not in use, but long start-up times are generally not acceptable.

Several of the present authors have been involved in the development of a packaging tool called *SmartLinker*[tm], which is is incorporated into IBM's WebSphere® Studio Device Developer 4.0 (WSDD),[21] an environment for developing Java applications for embedded systems. SmartLinker incorporates most of the previously presented extraction techniques,[22] and shares a significant amount of code with *Jax*.

This section is concerned with issues specific to the extraction of embedded systems applications, and with additional extraction techniques and optimizations that are implemented in SmartLinker.

6.1 Additional Distribution Scenario: Partially Pre-Linked Libraries

In addition to the distribution scenarios discussed in Section 5, SmartLinker supports the extraction of *custom libraries*. A custom library contains a subset of a library's functionality, and address situations where several applications deployed on a single embedded device use a common subset of a library's functionality, but where each deployed application may use additional library features not in the common subset.

[21]See `www.ibm.com/embedded`.

[22]Name compression is currently not incorporated in SmartLinker, and only limited class hierarchy transformations are performed. SmartLinker optionally includes line number and local variable attributes in the output it generates for debugging purposes. In order to avoid the overhead of these attributes on the target device, debugging information is stored in a separate file which is read by a debug proxy in order to translate internal JVM addresses and offsets to symbolic names.

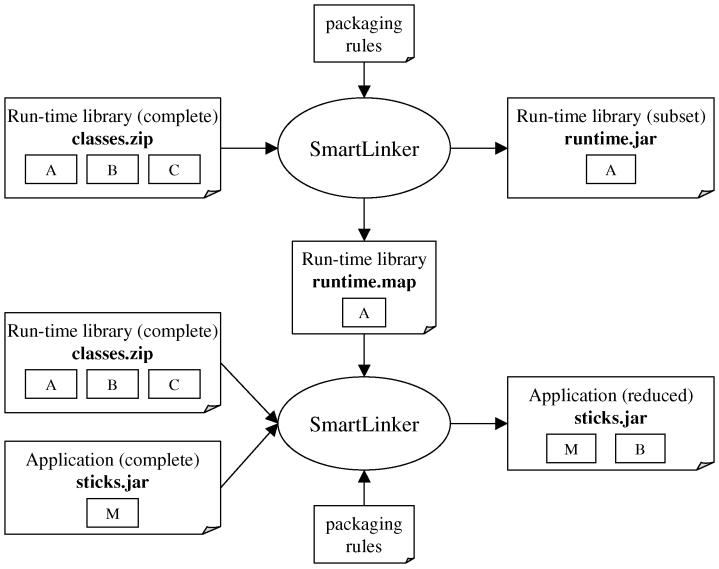

Fig. 12. Schematic view of SmartLinker's support for packaging partially pre-linked libraries. Commonly used library classes are pre-packaged in a shared JAR file, and other library classes are packaged in another JAR distribution along with the application.

The extraction of the custom library itself is essentially an instance of Scenario 1 of Section 5, but SmartLinker provides an additional mechanism to support the precise extraction of applications in the presence of custom libraries. When a custom library L_c is extracted from a complete library L, a *map file* is created that summarizes the dependences in L_c. This map file contains for every method in L_c: (i) internal references to other methods, fields and classes in L_c, and (ii) external references to methods, fields and classes not contained in L_c but contained in L.

In general, extracting an application A that depends on L_c requires: (i) the classes associated with A itself, (ii) the classes in L_c, and (iii) additional classes in L that are not in L_c, but that are used by A. However, only the classes of (i) and (iii) occur in the resulting distribution because the classes of (ii) are already available in the separately shipped custom library. The key issue is to avoid including too many of the classes in L with the extracted application A. This is accomplished by examining the map file of L_c during call graph construction. In particular, if a method `m` in L_c is found to be reached, the externally referenced classes, methods, and fields associated with `m` determine the additional parts of L that must be distributed with A, and the internal references are examined recursively to identify additional external references that must be taken into account.

Figure 12 depicts this process. The top part of the figure shows how a map file `runtime.map` is generated when extracting a custom library `runtime.jar` from a complete run-time library `classes.zip`. In the figure, the extracted subset of `classes.zip`'s functionality is shown as `A`. The bottom part of Figure 12 shows the extraction of an application "Sticks" (`sticks.jar`) that was developed using

Table. VIII. Results of applying SmartLinker to the PalmOS "Sticks" application. The JAR files in this figure contain uncompressed entries

benchmark	# classes	size (ZIP/JAR)	size (JXE)
Sticks	5	10,956	9,544
PalmOS utility classes	13	28,727	18,117
WCE PalmOS runtime libraries (complete)	344	1,068,560	617,481
WCE PalmOS runtime libraries (subset)	78	134,081	83,002
Sticks (packaged, no reduction/optimization)	63	1,108,243	240,748
Sticks (packaged, reduction, no optimization)	12	20,583	12,751
Sticks (packaged, reduction + optimization)	10	16,724	10,999

the complete library `classes.zip`. During the extraction of the Sticks application, the map file `runtime.map` is analyzed in conjunction with the complete run-time library `classes.zip` to determine the functionality in `classes.zip` that is required by `Sticks`, but which is not in `runtime.jar`. This additional functionality is shown as `B` in the figure.

6.2 Peephole Optimizations

SmartLinker performs a number of peephole optimizations that often have significant benefits in terms of application size and execution speed. These peephole optimizations include: (1) replacing consecutive identical load instructions with load, dup, (2) removing load, pop combinations, (3) removing jumps to the immediately following instruction, (4) replacing load, const, add, store instruction sequences with iinc, (5) removing no-ops, (6) removing store (load) when there are no more stores (loads) of the same local, and (7) removing store instructions that are followed by a return instruction.

6.3 Conversion to Other Representations

Embedded systems applications are typically not deployed using the standard Java class file representation, but in various device-specific formats. SmartLinker supports the Java Executable (JXE) format, a format specifically designed for packaging embedded applications in the context of WSDD. The JXE format has the property that it can be executed in-place by the JVM, which enables one to store applications in flash memory or ROM. Key differences between the JXE format and the standard Java class file representation are that the former uses a single, shared representation for strings and other constants (as opposed to using a copy per class), and that significantly less indirection is used. From the JXE representation, various device-specific representations can be produced, such as the PRC format in the case of PalmOS applications.

6.4 An Example: Packaging a PalmOS Application

Table VIII shows some results that were obtained for the "Sticks" PalmOS application, one of the examples shipped with WSDD. The top three rows of Table VIII show the inputs to the packaging process: the Sticks application itself (5 classes contained in a 10,956 byte JAR file), a small library containing several high-level PalmOS utility classes (13 classes in a 28,727 byte JAR file), and the WCE (WebSphere Custom Environment) run-time PalmOS libraries

(344 classes in a 1,068,560 byte JAR file), which provides a (small) subset of the functionality of the standard Java class library and a (large) subset of the PalmOS API. The rightmost column of Table VIII shows the size of the JXE representations of these three archives: 9,544 bytes, 18,117 bytes, and 617,481 bytes, respectively. This brings the total size of the original distribution to 362 classes and 645,142 bytes (in the JXE format). Because PalmOS devices typically only have a few megabytes of storage, it is clear that some reduction in footprint is needed.

The next row of the table (labeled "WCE PalmOS run-time libraries (subset)") refers to a custom library that has been independently extracted and contains a subset of the functionality of the complete run-time libraries. This custom library contains 78 classes (JXE size: 83,002 bytes) and packages the classes that are required by nearly every PalmOS application. When the Sticks application is extracted, none of the 78 classes of this custom library are included, as it is expected to be installed separately on the embedded device.

The last three rows of Table VIII illustrate the impact of some of the steps in the packaging process. The row labeled `Sticks (packaged, no reduction/optimization)` shows the extracted Sticks application before any classes, methods and fields have been removed. These 63 classes consist of: the 5 classes of the Sticks application itself, 1 utility class, and 57 classes from the complete libraries that do not occur in the custom library. The size of this distribution (JXE size: 240,748 bytes) is due to the fact that some of the classes in the complete library that Sticks refers to are very large (and includes, among other things, a 128KB class containing all 1200+ PalmOS function wrappers).

The row labeled `Sticks (packaged, reduction, no optimization)` shows the application after removal of unused classes (i.e., uninstantiated classes that do not contain live methods or fields), methods, and fields. Clearly, not much functionality in the 63 classes is actually used, because the application now comprises 12 classes, and a JXE size of 12,751 bytes.

The bottom row shows the application after inlining and peephole optimizations. This has further reduced the application to 10 classes (the 5 Sticks application classes, 1 utility class, and 4 run-time classes not in the custom library), and a JXE size of 10,999 bytes. The removal of two classes in this last step is due to the fact that all calls to the methods in these classes are inlined away.

6.5 Other Optimizations

In addition to size, speed is also an important issue for embedded systems applications. SmartLinker incorporates advanced optimizations such as: profile-directed inlining a user-specified percentage of the "hottest" call sites (i.e., the most frequently executed calls), and profile-directed ahead-of-time compilation of methods whose execution time is some user-specified percentage of the total measured execution time. We refer the interested reader to Eisma [2001] for details.

7. RELATED WORK

7.1 Historical Perspective

Several of the techniques incorporated into *Jax* borrow from previous work by some of the present authors. *Jax* implements the RTA [Bacon and Sweeney 1996; Bacon 1997] and XTA [Tip and Palsberg 2000] call graph construction algorithms. Accurate treatment of methods that override methods in external class libraries is very important for reducing archive size due to the importance of class libraries in Java. The detection of useless fields, including write-only fields, was previously studied by Sweeney and Tip [1998], and Tip et al. [1996] for C++. In this work, RTA was used to construct a call graph, and an average of 12.5% useless fields was measured for a set of C++ applications. In the context of *Jax*, we found that, on average, 38.5% of all fields *remain* after extraction. We conjecture that this difference is partly due to the pervasive use of class libraries in Java, and that these libraries tend to contain a lot of unused functionality. Furthermore, the larger percentage of unaccessed fields in Java applications could be due to the fact that Java lacks a macro facility and that Java programmers use static final fields to define constants. Java compilers propagate these constants so that no accesses to these fields remains, but the fields themselves are not removed.

The class hierarchy transformations used by *Jax* were originally proposed in the context of specializing class hierarchies [Tip and Sweeney 1997]. The goal of this work is to remove members from objects. Ignoring a number of details, the specialization algorithm constructs a new class hierarchy in which a new class is constructed for each variable and each member in the program. Inheritance relations between these classes reflect member access relationships between variables and class members, and subtype relationships between variables that must be retained to preserve program behavior. The class hierarchy transformations used in *Jax* were introduced in order to reduce the complexity of the resulting class hierarchy.

7.2 Application Extraction for Other Languages

The extraction of applications was pioneered in the Smalltalk community, where it is usually referred to as "packaging" [IBM Corporation 1995; Digitalk Inc. 1993; ParcPlace Systems 1992]. 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 [1994] and Agesen [1995] 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, Agesen [1995, page 146], 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 [1998] 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.

7.3 Compaction of Object Code

Optimizations aimed at reducing application size have also been applied at the object-code level, where such techniques are generally referred to as *code compaction*. Fraser et al. [1984] presented an algorithm based on suffix trees that detects repeated sequences of instructions, and replaces these with a single instance of the sequence and a set of branch/jump instructions. One can think of this *code factoring* as a form of procedural abstraction, although the algorithm goes beyond that by allowing *tail merging* (i.e., having the instruction sequences of procedures overlap). A number of improvements over this basic technique were later proposed by Cooper and McIntosh [1999] and by De Sutter et al. [2002] and De Sutter [2002]. Code factoring can be worthwhile at the object-code level, because the overhead of introducing additional procedures is relatively low. It is unclear how worthwhile code factoring would be in our setting, because of the overhead associated with introducing additional methods, and because syntactic constraints on byte codes may prevent the exploitation of factoring opportunities.

Debray et al. [2000] incorporated code factoring as well as a collection of traditional compiler-optimizations such as dead code elimination, strength reduction, and interprocedural constant propagation in *squeeze*, a code compactor that targets binaries for the Alpha processor. They report an average reduction in executable size of about 30%. In contrast to our work, where a program's call graph is the central data structure, the analyses in *squeeze* operate on an interprocedural control flow graph representation of the program. De Sutter et al. [2001] extended *squeeze* with techniques for compacting a binary program's data area. This is accomplished by using constant propagation to determine the values of addresses in code and data areas, and using the obtained information to determine code and data values that can be eliminated. De Sutter et al. report an average data size reduction of 24.3%, as well as an additional code size reduction of about 3% due to the removal of code that was only referred to from pointers stored in unused data (e.g., unused virtual function tables). *Squeeze* was further extended with improved code factoring capabilities that

specifically target code duplication that is due to templates and inheritance in C++ [De Sutter et al. 2002]. The factoring transformations in the resulting compaction tool (named *squeeze++*) identify similar but not identical procedures and merge these into a combined procedure that takes additional parameters. An average size reduction of 45% is reported on a set of C++ benchmarks.

7.4 Extraction and Obfuscation Tools for Java

There are several Java tools, DashO-Pro,[23] SourceGuard,[24] Jshrink,[25] Jmangle,[26] and JZipper[27] aimed at obfuscation (i.e., to make decompilation of class files into understandable source code more difficult) that perform some form of compression of class names, method names, field names, and package names. Of these tools, only DashO-Pro and SourceGuard go beyond simple name compression, and perform other transformations such as modifying an application's control flow. We are not aware of any published work on the algorithms used in any of these tools, nor on their internal architecture.

Rayside and Kontogiannis [1999] present a technique that uses an entity-relationship dependency graph for extracting embedded systems applications. A crucial difference with our work is that Rayside and Kontogiannis do not use call graph information as the basis for detecting unused program constructs, and only remove program constructs that are not referred to. The techniques by Rayside and Kontogiannis are much less accurate than ours, because they are incapable of removing anything that is referred to from dead code. Unlike *Jax*, this work does not explore the extraction of software distributions other than complete applications.

Thies [1999a, 1999b] presents a static analysis of class libraries where the goal is to compute "summary" information that concisely represents the side-effects of method invocations on actual parameters. This information can be used by a JIT at run-time to detect additional opportunities for optimizations such as common subexpression elimination, and loop invariant code motion. The analysis information that Thies computes is similar in spirit to the information captured in our MEL configuration files, although it is different in the sense that it is computed automatically. Thies does not discuss how he handles the use of reflection and dynamic loading in class libraries. It appears that Thies uses Class Hierarchy Analysis as the basis for computing the side-effects of methods that call other methods. It would be interesting to investigate how much precision can be gained by using a more sophisticated call graph construction algorithm.

Zaks et al. [2000] present an algorithm for devirtualizing call sites that exploits *sealing* of Java packages. Sealed packages were introduced in JDK 1.2.2 [Sun Microsystems 1999] for security reasons and to enforce consistency within a version of a JAR file. If a package is sealed within a JAR file, all classes defined

[23]See `www.preemptive.com`.
[24]See `www.4thpass.com`.
[25]See `www.e-t.com`.
[26]See `www.elegant-software.com/software/jmangle`.
[27]See `www.vegatech.net/jzipper`.

Table. IX. Results of converting the archives of Tables I and II to the JXE format (see Section 6), Pugh's packed representation, and to the Jazz representation. For each of the benchmarks, we show the size of the original archive, and the size the archive produced by *Jax* (both after conversion)

benchmark	JXE (original)	JXE (processed)	Pugh (original)	Pugh (processed)	Jazz (original)	Jazz (processed)
Hanoi	69,229	33,706	14,051	6,989	26,384	11,225
Jax	621,039	487,832	113,644	92,351	280,830	173,150
javac	411,631	337,944	77,233	64,813	164,441	115,896
bloat	497,995	375,564	88,354	70,642	252,187	129,617
mBird	2,921,538	410,381	505,585	85,184	467,219	160,055
Jinsight	526,474	437,191	99,804	83,875	207,027	184,537
JavaFig	481,361	343,658	92,819	67,853	220,183	144,543
CindyApplet	1,083,673	282,666	197,243	51,839	518,779	105,367
Cinderella	1,083,673	585,048	197,243	106,699	518,779	246,654
eSuite Sheet	1,950,291	519,192	570,522	114,513	644,734	214,344
eSuite Chart	2,349,267	869,856	654,357	187,728	811,585	410,801
Hyper/J	1,538,894	584,878	287,028	106,016	714,396	207,894
Res. System	4,101,554	2,338,994	735,801	430,188	2,581,288	1,443,430

in that package must be loaded from that JAR file. Zaks et al. found that a significant portion of the virtual call sites in the run-time library `rt.jar` could be devirtualized using this technique.

7.5 Alternative Representations for Java Class Files

Recently, Pugh [1999], Horspool and Corless [1998], and Bradley et al. [1998] proposed alternative, more space-efficient representations for Java class files. These representations rely on techniques such as the use of a global constant pool, efficiently representing names that share a common prefix, and separating different streams of information (e.g., opcodes and operands) and compressing the resulting streams separately. Similar "wire-format" representations that rely on creating streams of opcodes and operands that can be compressed separately were explored previously by Ernst et al. [1997] for compressing x86 machine code. Pugh reports archives that range between 17% and 49% of the size of the original representation for a representative set of benchmarks. Pugh also evaluates the Jazz representation by Horspool and Corless on his benchmarks, and measures archives ranging in size from 31% to 181% of the original representation. An important advantage of these representations is the enabling of information sharing between different class files. *Jax* can only introduce a limited amount of sharing by merging classes. On the other hand, application extractors can achieve significant size reductions by eliminating unused methods, classes, and fields. This is not addressed by compression techniques. Therefore, one would expect application extraction and more efficient class file representations to be largely orthogonal techniques for reducing application size.

In order to verify this conjecture, we converted the original class file archives of Table I and the archives produced by *Jax* as shown in Table II to the Jazz representation [Bradley et al. 1998], and to Pugh's Packed representation [Pugh 1999]. We also converted these archives to the JXE format used in SmartLinker (see Section 6). Table IX shows the archive sizes for the original application and the corresponding extracted application in ZIP files, JXE files, Pugh's packed representation, and the Jazz representation.

Figure 13 depicts the size of the extracted benchmarks in each of the four representations. In summary, we measured that the size of the extracted

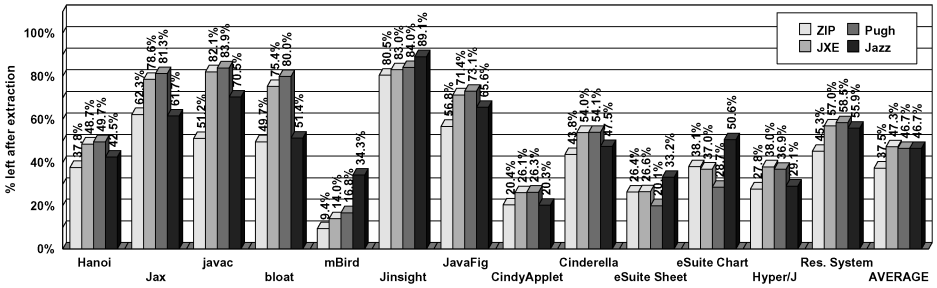

Fig. 13. Comparison of the archive size reductions obtained with *Jax* in each of the four class file representations under consideration. For each benchmark, the four bars indicate, from left to right, the size of the extracted benchmark using the standard class file representation, the size of the extracted benchmark using the JXE format, the size of the extracted benchmark using Pugh's representation, and the size of the extracted benchmark using the Jazz representation. All sizes of extracted benchmark are given as a percentage of the original size of that benchmark in the same representation.

applications ranges from 9.4% to 80.5% of the original size (average: 37.5%) when the standard zipped class file representation is used, from 14.0% to 83.0% of the original size (average: 47.3%) if the JXE representation is used, from 16.8% to 84.0% (average: 46.7%) when Pugh's packed representation is used, and from 20.3% to 89.1% (average: 46.7%) when the Jazz representation is used. These results demonstrate that application extraction clearly remains a highly useful size reduction technique when alternative representations are used, and that the benefits of application extraction are largely independent of the specific representation that is used.

A different approach is taken by Kistler and Franz [1997, 1999] who, instead of representing Java byte-codes more efficiently, propose a new intermediate representation called slim binaries as an alternative to Java byte-codes. This representation is based on a high-level abstract syntax tree and encoded using an adaptive compression scheme.

Rayside et al. [1999] proposed a new and smaller interpretable format for Java binaries instead of compressing the existing structure by focusing on the constant pool and code array. Unlike the work of Pugh and Horspool and Corless but like *Jax*, this representation does not require decompression before execution. Nevertheless, the new interpretable format requires either a slightly modified virtual machine or a customized class loader.

Clausen et al. [2000] explored an alternative approach to compression of Java byte codes that aims at reducing the memory footprint of low-end embedded systems applications. The approach by Clausen et al. replaces frequently recurring sequences of byte code instructions by new "macro" instructions, and requires minor modifications to a VM to recognize and expand these macros at run-time. Similar techniques were previously applied to x86 machine code by Ernst et al. [1997]. Clausen et al. report a memory footprint reduction of 15%, and a speed penalty that varies between 2% and 30%. It is important to realize that the work by Clausen et al. only addresses compression of instruction sequences, and not, for example, compression of constant pool entries.

Krintz et al. [1998] study an approach for reducing application start-up time by using a less strict execution model in which an application starts executing even as parts of it (classes and methods) are still being downloaded. The benefits of this work are likely to be orthogonal to those of application extraction.

8. CONCLUSIONS AND FUTURE WORK

8.1 Summary of Contributions

We have implemented a number of extraction techniques such as the removal of redundant methods and fields, inlining of method calls, class hierarchy transformations, and name compression in a practical extraction tool named *Jax*. We applied *Jax* to a set of large Java applications and measured that, on average, the class file archive for an application is reduced to 37.5% of its original size.

For a variety of reasons, extraction tools may need additional information that cannot be determined through static analysis. In particular, the use of dynamic language features such as reflection and dynamic loading, and the extraction of software distributions other than complete applications requires additional user input. We have defined a small, modular language for specifying the extraction of various types of software distributions, and described an approach in which all of these can be handled uniformly in an extraction tool. A small case study was presented in order to illustrate the practicality of the approach.

Several of the extraction techniques that were prototyped and validated in *Jax* have been applied successfully in IBM's WebSphere Studio Device Developer, an environment for programming embedded systems applications in Java. They make a crucial difference for scaling Java to embedded systems. We discussed a number of issues specific to the embedded systems domain, such as the support for partial (pre-)linking of libraries, and conversion to other representations.

8.2 Future Work: Optimization of Embedded Applications

Plans for future work in the area of optimization for embedded systems applications include various interprocedural optimizations of the generated code such as object inlining, interprocedural liveness analysis, and demand-driven context-sensitive analysis driven by profiling feedback. We have already conducted some initial experiments with an approach in which we continue to generate optimized bytecodes, using a translation to a register-based intermediate representation (IR) [Burke et al. 1999], followed by regeneration of bytecodes from this IR (similar to Pominville et al. [2001]). However, bytecode-level optimizations have their limitations. For example, there are many cases where a given `invokevirtual` call site always resolves to a specific method, but where the call cannot easily be devirtualized because the use of the `invokespecial` bytecode is restricted to certain situations [Lindholm and Yellin 1997]. An alternative approach would be to store analysis information as attributes in class files. A JIT could be adapted to recognize this information and exploit it by

performing more on-line optimizations, or by making existing optimizations more efficient.

8.3 Future Work: Interactive Program Development Tools

We plan to incorporate the analysis performed by *Jax* to visualize unnecessary program components in an interactive program development environment. One can easily imagine the benefits of a tool that could provide information about unreachable methods and unaccessed fields, enabling the user to determine if these components are simply redundant, or unreachable as the result of a bug. Such a tool could also provide hints to the user about classes and methods that can be declared final. Other software engineering applications that can benefit from analysis information such as call graphs include change impact analysis [Ryder and Tip 2001], refactoring [Fowler 1999], and regression test selection [Rothermel and Harrold 1997].

ACKNOWLEDGEMENTS

Lisa Martin and Sarvamangala Jagadeesh are gratefully acknowledged for their help with the development of *Jax*. The bug reports and comments from the many people who participated in the *Jax* discussion group on alphaWorks have been invaluable. We are grateful to Robert Berry, John Field, Harold Ossher, Ramalingam, Vivek Sarkar, Gregor Snelting, Alan Stevens, Robert Weir, and the anonymous TOPLAS referees for their constructive comments and feedback. We are also grateful to Bill Pugh, Quetzalcoatl Bradley, Nigel Horspool, and Jan Vitek for making their class file compression tools available.

REFERENCES

AGESEN, O. 1995. Concrete type inference: Delivering object-oriented applications. Ph.D. thesis, Stanford University. Appeared as Sun Microsystems Laboratories Tech. Rep. SMLI TR-96-52.

AGESEN, O. AND UNGAR, D. 1994. Sifting out the gold: Delivering compact applications from an exploratory object-oriented programming environment. In *Proceedings of the 9th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*. Portland, OR, 355–370. *ACM SIGPLAN Notices* 29(10).

AIGNER, G. AND HÖLZLE, U. 1996. Eliminating virtual function calls in C++ programs. In *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*. Lecture Notes in Computer Science, vol. 1098. Springer-Verlag, Linz, Austria, 142–166.

ARNOLD, K., GOSLING, J., AND HOLMES, D. 2000. *The Java Programming Language*, Third Edition Addison-Wesley.

BACON, D. F. 1997. Fast and effective optimization of statically typed object-oriented languages. Ph.D. thesis, Computer Science Division, University of California, Berkeley. Report No. UCB/CSD-98-1017.

BACON, D. F. AND SWEENEY, P. F. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*. San Jose, CA, 324–341. *ACM SIGPLAN Notices* 31(10).

BRADLEY, Q., HORSPOOL, R. N., AND VITEK, J. 1998. Jazz: An efficient compressed format for Java archive files. In *CASCON'98*. 294–302.

BURKE, M. G., CHOI, J.-D., FINK, S., GROVE, D., HIND, M., SARKAR, V., SERRANO, M. J., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. 1999. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proceedings of the ACM Java Grande Conference* San Francisco, CA.

CALDER, B. AND GRUNWALD, D. 1994. Reducing indirect function call overhead in C++ programs. *Proceedings of the 21st ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL'94)*, 397–408.

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

CHEN, Y.-F., GANSNER, E. R., AND KOUTSOFIOS, E. 1998. A C++ data model supporting reachability analysis and dead code detection. *IEEE Trans. Soft. Eng. 24*, 9 (Sept.), 682–694.

CLAUSEN, L. R., SCHULTZ, U. P., CONSEL, C., AND MULLER, G. 2000. Java bytecode compression for low-end embedded systems. *ACM Trans. Prog. Lang. Syst. 22*, 3 (May), 471–489.

COOPER, K. D. AND MCINTOSH, N. 1999. Enhanced code compression for embedded RISC processors. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'99)*. Atlanta, GA, 139–149. *ACM SIGPLAN Notices* 34(5).

DE SUTTER, B. 2002. Compactie van programma's na het linken. Ph.D. thesis, Gent University. In Dutch.

DE SUTTER, B., DE BUS, B., AND DE BOSSCHERE, K. 2002. Sifting out the mud: Low level C++ code reuse. Tech. rep., Ghent University. To appear in Proceedings OOPSLA'2002.

DE SUTTER, B., DE BUS, B., DEBRAY, S., AND DE BOSSCHERE, K. 2001. Combining global code and data compaction. In *Proceedings of the ACM SIGPLAN 2001 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'2001)*. Snowbird, UT.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, W. Olthoff, Ed. Springer-Verlag, Aarhus, Denmark, 77–101.

DEBRAY, S. K., EVANS, W., MUTH, R., AND DE SUTTER, B. 2000. Compiler techniques for code compaction. *ACM Trans. Prog. Lang. Syst. 22*, 2, 378–415.

Digitalk Inc. 1993. *Smalltalk/V for win32 Programming*. Digitalk Inc. Chapter 17: Object Libraries and Library Builder.

EISMA, A. 2001. Feedback directed ahead-of-time compilation for embedded Java applications. In *Java Optimization Strategies for Embedded Systems*, U. Assmann, Ed. Genova, Italy, 105–112. Workshop held at ETAPS'01.

ERNST, J., EVANS, W., FRASER, C., LUCCO, S., AND PROEBSTING, T. 1997. Code compression. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*. Las Vegas, NV, 358–365. *ACM SIGPLAN Notices* 32(5).

FOWLER, M. 1999. *Refactoring*. Addison-Wesley.

FRASER, C., MYERS, E. W., AND WENDT, A. L. 1984. Analyzing and compressing assembly code. In *Proceedings of the ACM Symposium on Compiler Construction*. 117–121. *ACM SIGPLAN Notices* 19(6).

GROVE, D., DEFOUW, G., DEAN, J., AND CHAMBERS, C. 1997. Call graph construction in object-oriented languages. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*. Atlanta, GA, 108–124. *ACM SIGPLAN Notices* 32(10).

HORSPOOL, R. N. AND CORLESS, J. 1998. Tailored compression of Java class files. *Software—Practice and Experience 28*, 12, 1253–1268.

IBM Corporation 1995. *IBM Smalltalk User's Guide*, Version 3.0 ed. IBM Corporation. Chapter 36: Introduction to Packaging, Chapter 37: Simple Packaging, Chapter 38: Advanced Packaging.

JOY, B., STEELE, G., GOSLING, J., AND BRACHA, G. 2000. *The Java Language Specification*, Second Edition Addison-Wesley.

KISTLER, T. AND FRANZ, M. 1997. Slim binaries. *Commun. ACM 40*, 12 (Dec.), 87–94.

KISTLER, T. AND FRANZ, M. 1999. A tree-based alternative to Java byte-codes. *Int. J. Parallel Prog. 27*, 1 (Feb.), 21–34.

KRINTZ, C., CALDER, B., LEE, H. B., AND ZORN, B. G. 1998. Overlapping execution with transfer using non-strict execution for mobile programs. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. San Jose, California, 159–169.

LINDHOLM, T. AND YELLIN, F. 1997. *The Java Virtual Machine Specification*. Addison-Wesley.

PANDE, H. D. AND RYDER, B. G. 1996. Data-flow-based virtual function resolution. In *Proceedings of the 3rd International Symposium on Static Analysis (SAS'96)*. 238–254. Springer-Verlag LNCS 1145.

ParcPlace Systems 1992. *ParcPlace Smalltalk*, ObjectWorks Release 4.1 ed. ParcPlace Systems. Section 16: Deploying an Application, Section 28: Binary Object Streaming Service.

POMINVILLE, P., QIAN, F., VALLÉE-RAI, R., HENDREN, L., AND VERBRUGGE, C. 2001. A framework for optimizing Java using attributes. In *Proceedings of the International Conference on Compiler Construction (CC'2001)*, R. Wilhelm, Ed. Eisenstadt, Austria, 334–354. Springer-Verlag LNCS 2027.

PUGH, W. 1999. Compressing Java class files. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*. Atlanta, GA, 247–258. *ACM SIGPLAN Notices* 34(5).

RAYSIDE, D. AND KONTOGIANNIS, K. 1999. Extracting Java library subsets for deployment on embedded systems. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'99)*. Amsterdam, 102–110.

RAYSIDE, D., MANAS, E., AND HONS, E. 1999. Compact Java binaries for embedded systems. In *Proceedings of the 9th NRC/IBM Centre for Advanced Studies Conference (CASCON'99)*. Toronto, CA, 1–14.

ROTHERMEL, G. AND HARROLD, M. J. 1997. A safe, efficient regression test selection technique. *ACM Trans. Soft. Eng. Method. 6*, 2 (April), 173–210.

RYDER, B. G. AND TIP, F. 2001. Change impact analysis for object-oriented programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*. Snowbird, UT.

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

SHAPIRO, M. AND HORWITZ, S. 1997. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL'97)*. Paris, France, 1–14.

SHIVERS, O. 1991. Control-flow analysis of higher-order languages. Ph.D. thesis, CMU. CMU–CS–91–145.

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN/SIGSOFT Symposium on Principles of Programming Languages (POPL'96)*. St. Petersburg, FL, 32–41.

Sun Microsystems 1997. *JavaBeans*, Version 1.01 ed. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043.

Sun Microsystems 1999. *Java 2 Software Development Kit*, Version 1.2.2 ed. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043. Available at http://java.sun.com/docs/books/tutorial/ext/security/sealing.html.

SUNDARESAN, V., HENDREN, L., RAZAFIMAHEFA, C., VALLÉE-RAI, R., LAM, P., GAGNON, E., AND GODIN, C. 2000. Practical virtual method call resolution for Java. In *Proceedings of the 15th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*. Minneapolis, MN), 264–280. *ACM SIGPLAN Notices* 35(10).

SWEENEY, P. F. AND TIP, F. 1998. A study of dead data members in C++ applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*. Montreal, Canada, 324–332. *ACM SIGPLAN Notices* 33(6).

SWEENEY, P. F. AND TIP, F. 2000. Extracting library-based object-oriented applications. In *Proceedings of the 8th International Symposium on the Foundations of Software Engineering (FSE-8)*. San Diego, CA, 98–107. ACM SIGSOFT Software Engineering Notes 25(6).

THIES, M. 1999a. A closer look at inter-library dependencies in Java-software. In *Java-Informations-Tage 1999 (JIT'99)*. Informatik Aktuell. Springer Verlag.

THIES, M. 1999b. Static compositional analysis of libraries in support of dynamic optimization. Technischer Bericht tr-ri-99-210, University of Paderborn. Aug.

TIP, F., CHOI, J.-D., FIELD, J., AND RAMALINGAM, G. 1996. Slicing class hierarchies in C++. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*. San Jose, CA, 179–197. *ACM SIGPLAN Notices* 31(10).

TIP, F., LAFFRA, C., SWEENEY, P. F., AND STREETER, D. 1999. Practical experience with an application extractor for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*. Denver, CO, 292–305. *ACM SIGPLAN Notices* 34(10).

TIP, F. AND PALSBERG, J. 2000. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*. Minneapolis, MN, 281–293. *ACM SIGPLAN Notices* 35(10).

TIP, F. AND SWEENEY, P. 1997. Class hierarchy specialization. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*. Atlanta, GA, 271–285. *ACM SIGPLAN Notices* 32(10).

TIP, F. AND SWEENEY, P. 2000. Class hierarchy specialization. *Acta Informatica 36*, 927–982.

ZAKS, A., FELDMAN, V., AND AIZIKOWITZ, N. 2000. Sealed calls in Java packages. In *Proceedings of the 15th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*. Minneapolis, MN), 83–92. *ACM SIGPLAN Notices* 35(10).

Received August 2001; revised May 2002; accepted July 2002