Extracting Library-Based Object-Oriented Applications

Peter F. Sweeney

IBM

T.J. Watson Research Center

Hawthorne, NY

pfs@us.ibm.com

(Joint work with Frank Tip)

Background

Previous research: reducing C++ program size.

Apply previous research to Java + class file editor

= JAX [OOPSLA 99], an extractor that reduces software size!

JAX is available on www.alphaworks.ibm.com

The (static analysis) techniques that JAX uses:

- call graph construction
- dead method removal
- dead field elimination
- call devirtualization
- name compression
- class hierarchy transformation ...

Static analysis not enough!

- JAX adhoc approach to augment static analysis.

Goal of this Work

Unified approach to augment static analysis!

Road Map

- Java Extractor: background
- Why Static Analysis is not sufficient!
- Problem Statement
- The Solution: Modular Extraction Language (MEL)
- Case Study
- Future Work
- Related Work
- Conclusions

Why Static Analysis is not sufficient!

a. role
   - determines the methods that represent the external interface.

b. extraction domain
   - determines the software that is required for deployment. (Depends on the deployment environment!)

c. dynamic features
   - Introspection (structural inspection),
   - Explicit class loading at run-time, and
   - Native methods: black box components with call backs.

d. usage context
   - captures relationships between different program entities.

Roles

Application:

- external interface: main()
- can not be extended (no class derivation)

Applet:

- external interface: overridden methods of java.applet.Applet
- can be extended

Library:

- external interface: all public and protected members
- expected to be extended

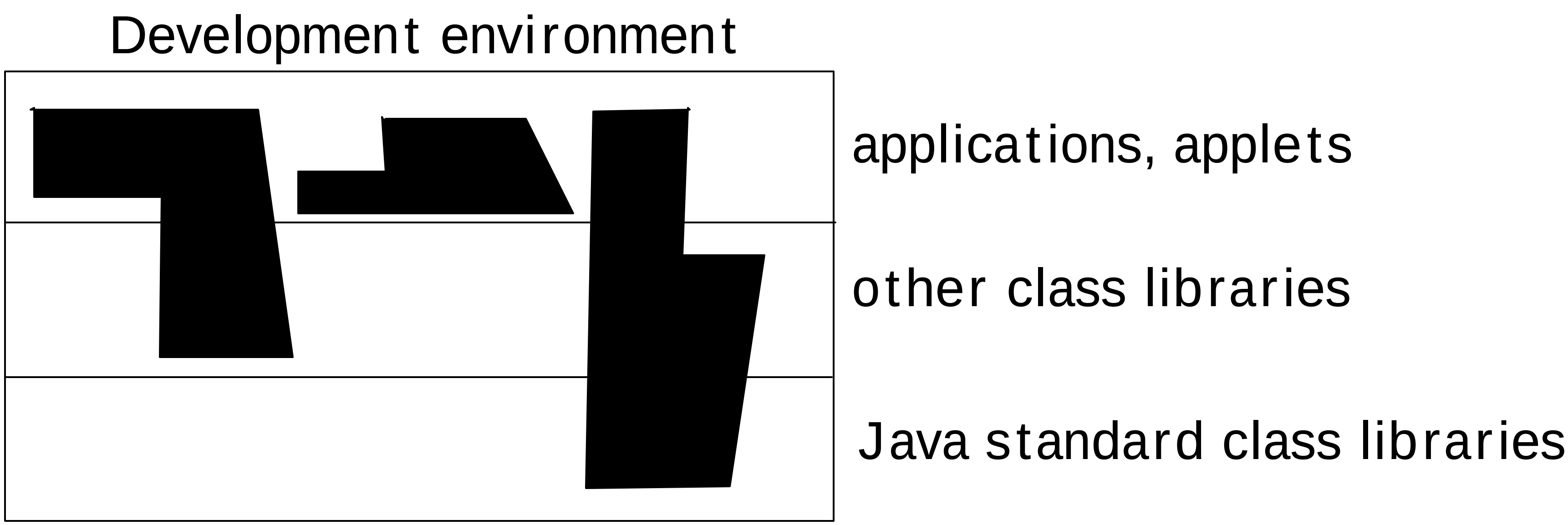
Component:

- external interface: all public members
- can not be extended

The same class can play multiple roles!

Extraction Domain

- The approach to software construction is layered.
- Must specify the boundaries between layers
  - depends on deployment environment
- Boundaries are an issue of "correctness".



Static analysis can't determine boundaries!

Dynamic Feature: Introspection

Example program	(code below)

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

public class Introspection {
   public static void main (String args[]) {
      try {
         Class c = Class.forName(args[0]);
         Method[] methods = c.getDeclaredMethods();
         for (int i=0; i < methods.length; i++) {
            Method m = methods[i];
            System.out.println(m.getName());
         } catch (Exception e) {System.out.println("..."); }
} } };
```

Static analysis can't determine which class has its methods accessed!

Dynamic Feature: Explicit Class Loading

<table>
<tr>
<td>Example program</td>
<td>

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

public class ExplicitClassLoad {
   public static void main (String args[]) {
      try {
         Class  c = Class.forName(args[0]);
         Object o = c.newInstance();
         ...
      } catch (Exception e) {System.out.println("Exception");
}  }; };
```

</td>
</tr>
</table>

Static analysis can't determine which class is instantiated and has its constructor is called!

Dynamic Feature: Native Methods

Java support for other languages.

Platform dependent implementations.

Black box that can:

- instantiate classes,
- invoke methods, and
- access fields in Java code.

Unlike bytecodes, the native code is difficult to analyze!

Usage Context

Example Library	`import java.lang.Class;`
	
	`public class L {`
	`   public static void f () {`
	`      ...   /* load class M */   ...`
	`   }`
	`   public static void g () {`
	`      ...   /* load class N */   ...`
	`   }`
	`};`

Static analysis can't capture the relationship that method f loads class M and method g loads class N!

Problem Statement

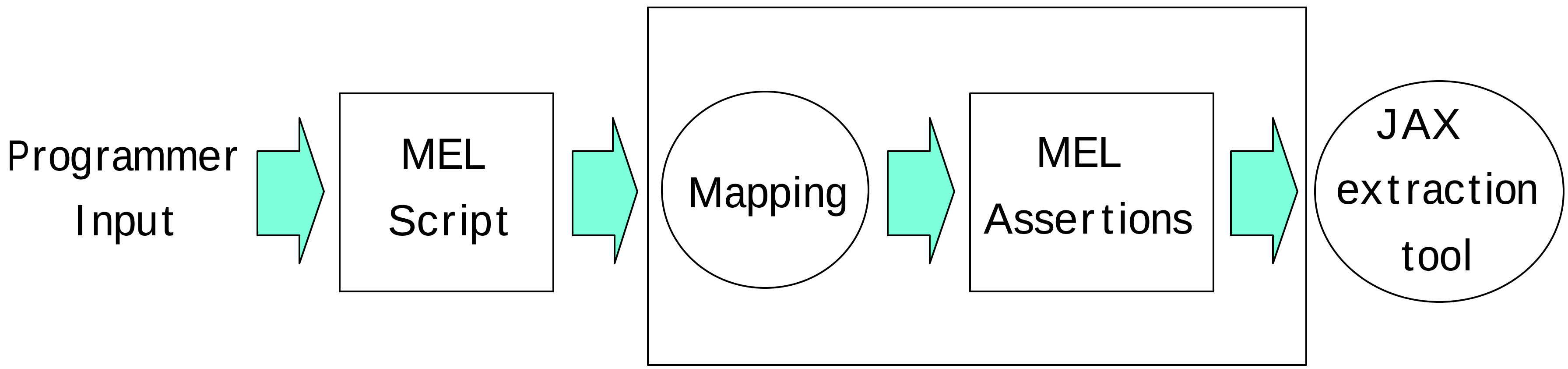
What is an "appropriate" interface for a user to specify software extraction?

What is an "appropriate" interface that software extraction is specified for an extractor?

MEL: Modular Extraction Language

Two level language:

- MEL Scripts: high level interface for user.
- MEL Assertions: low level interface to extraction tool.
- Mapping: scripts -> assertions.



Usage Context: special case

Example Library	import java.lang.Class;
	package L;
	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 { ... };

Example Application	import L;
	public class A {
	  public static void main (String args[]) {
	    ... L lib = new L(); lib.g(); ...
	  }
	};

Example MEL Scripts

MEL Script for Library L	path ... includepackage L; library L; preserve M when reached L.f() preserve N when reached L.g()
MEL Script for Application A	path ... includeclass A application A import L.mel

Classes A, L, and N are extracted, but M is not!

MEL Scripts: user interface

path <Directory>

path <ZipFile>

includeclass <Class>

includepackage <PackageName>

import <MEL Script>

application <Class>

applet <Class>

library <Class>

component <Class>

preserve <Class|Method|Field>

preserve <Class|Method|Field> **when reached** <Method>

MEL Assertions: extractor interface

extendible <Class>

overridable <Method>

instantiated <Class>

reached <Method>

accessed <Field>

preserveIdentity <Class|Method|Field>

preserveIdentity <Class|Method|Field> ***when reached*** <Method>

Map: roles to assertions

MEL Script	MEL Assertions
application C	***preserveIdentity*** C ***reached*** C.main(java.lang.String[]) ***preserveIdentity*** C.main(java.lang.String[])
library C	***preserveIdentity*** C ***extendible*** C ***reached*** C.m for every public or protected method C.m ***preserveIdentity*** C.m for every public or protected method C.m ***overridable*** C.m for every public or protected instance method C.m ***accessed*** C.f for every public or protected field C.f ***preserveIdentity*** C.f for every public or protected field C.f
...	**...**

Map: preserve to assertions

MEL Script	**MEL Assertions**
preserve C	***instantiated*** C when C is not an interface or an abstract class ***preserveIdentity*** C
preserve C **when reached** D.n	***instantiated*** C ***when reached*** D.n ***preserveIdentity*** C ***when reached*** D.n
...	***...***

Implementation

MEL assertions that different phases of JAX use.

- Call Graph Construction

  reached <Method>

  instantiated <Class>

- Call Devirtualization

  extendible <Class>

  overridible <Method>

- Dead Field Removal

  accessed <Field>

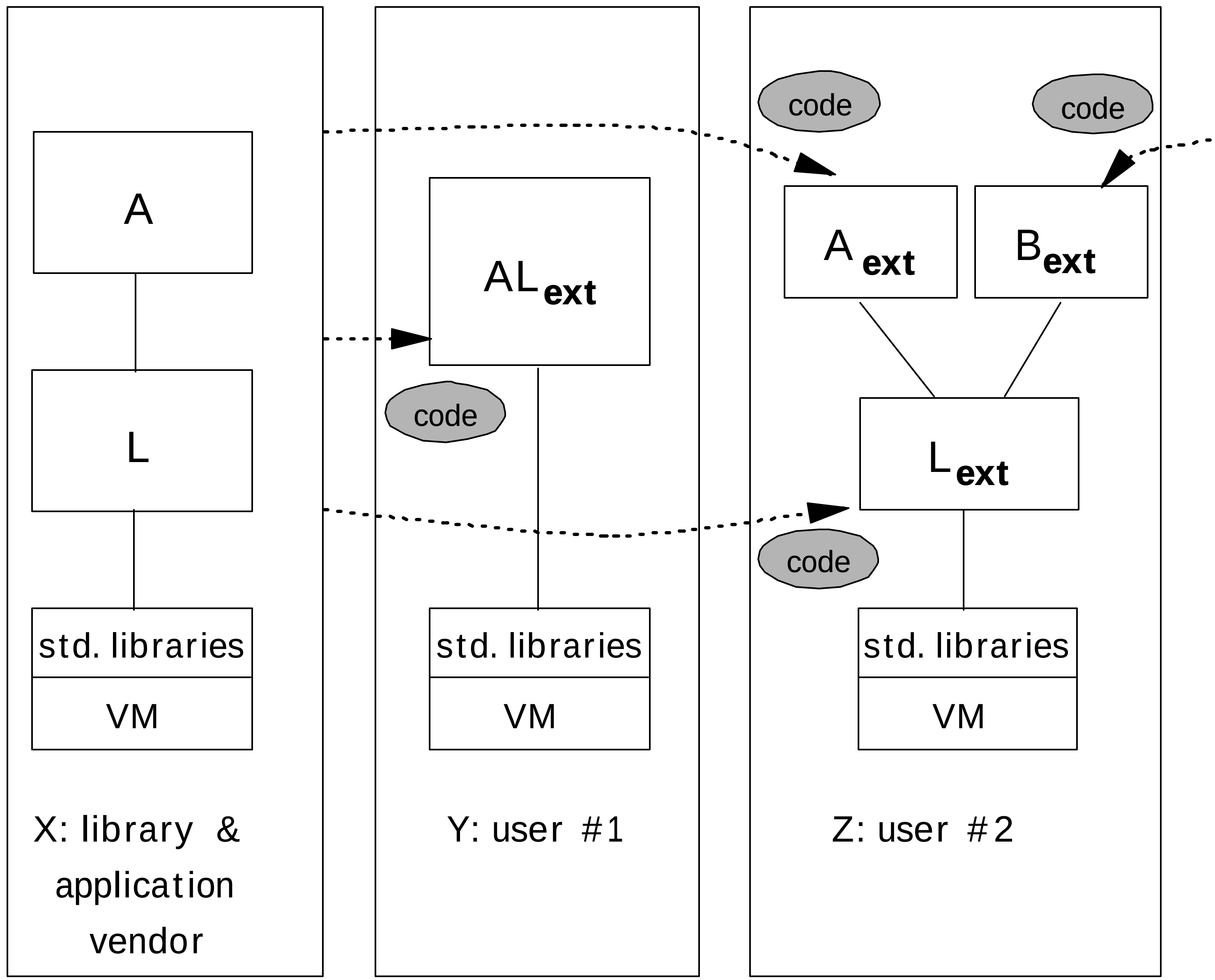
- Name Compression

  preserveIdentity <Class|Method|Field>

  extendible <Class>

  overridible <Method>

Distribution Scenarios



Case Study

Cinderella [www.cinderella.de] **interactive geometry tool.**

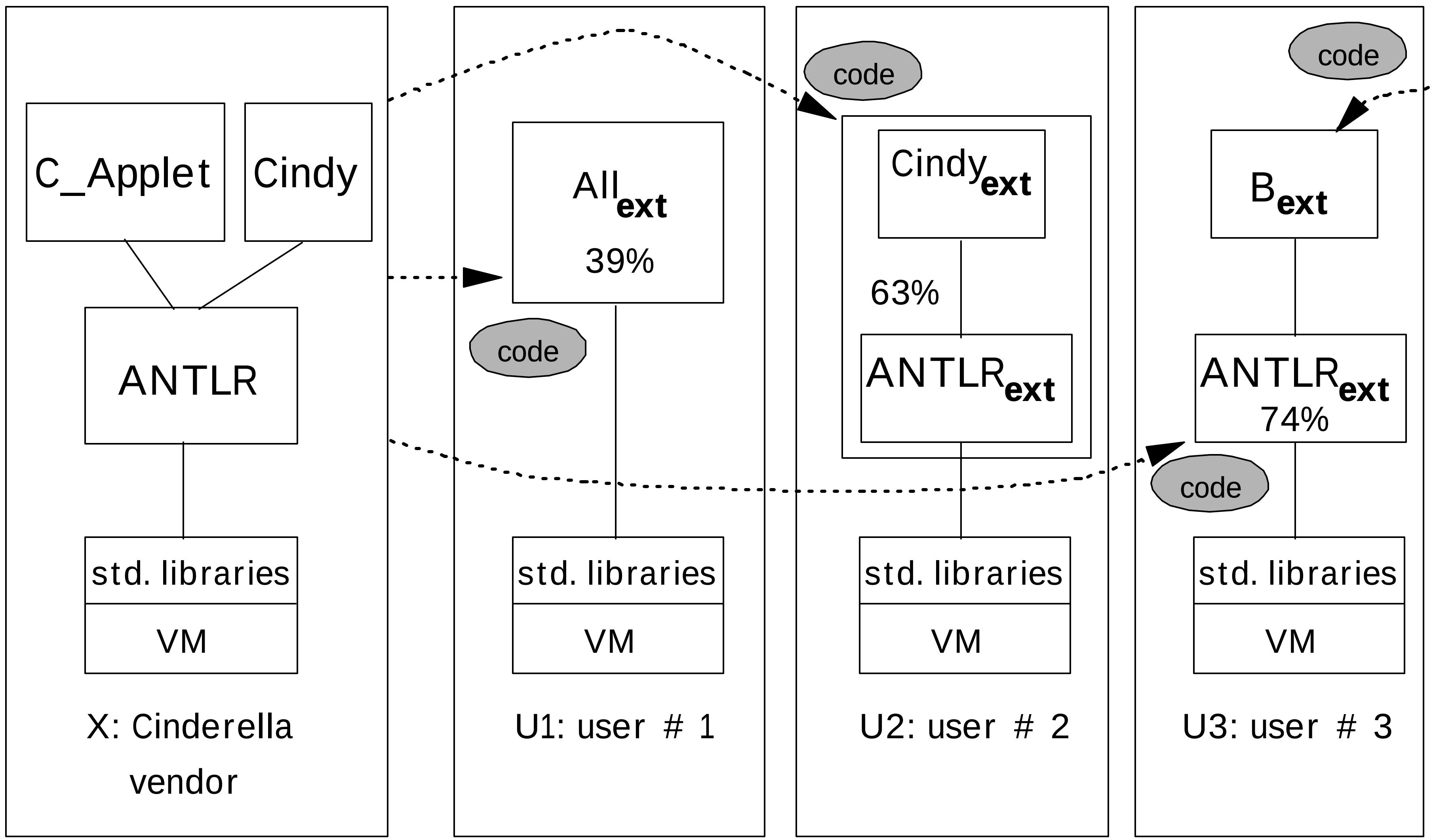
- application for constructing interactive geometry exercises, and
- applet for solving exercises.

Interesting observations:

- application and applet derived from same code base.
- Cinderella uses a library called ANTLR for parsing.

Original distribution of Cinderella contains application, applet and ANTLR.

Distribution Scenarios



Future Work

Embedded pervasive computing: new challenges.

Exploring alternative usage contexts.

Support additional roles:

- Java beans
- servlets

Related Work

Smalltalk community: extraction pioneered

Self language: Agesen and Ungar

Lisp community: tree shaking

C++: Acacia [Chen et. al.]

Commercial extraction tools:

DashO-Pro, Condensity

Conclusions

Software extraction can't be done by static analysis alone!

- roles
- dynamic features
- extraction domains
- usage context

Contributions

- small, modular specification language MEL
  - User interface
  - Extraction tool interface
- demonstrated MEL's feasibility
  - implemented on top of JAX
  - small case study