Extracting Source Models from Java Programs: Parse, Disassemble, or Profile?

Ivan T. Bowman, Michael W. Godfrey, and Richard C. Holt
{itbowman,migod,holt}@plg.uwaterloo.ca
University of Waterloo, Waterloo, Ontario, Canada

Abstract

Source models of software systems are often created during re-engineering to aid in performing tasks such as reachability analysis and software architecture recovery. It is therefore vital to be able to create source models that are both detailed and accurate. However, in practice the creation of these models is difficult and error prone: extraction tools often tell only part of the story.

We have been working on the automated extraction of source models for programs written in Java. Our approach considers Java programs from three points of view: parsing, disassembly, and profiling. We have found that these three techniques have advantages that are complementary. Parsing source code provides the most detailed information, but it is the most complex to implement. Disassembling Java byte code gives similar results to parsing, but is less complex technically. Profiling provides the least amount of detail, but does give important feedback on run-time behaviour, such as polymorphic function calls and reflective instantiation of objects by string input.

We have applied our tools to several systems, including Sun's javac Java compiler and the Jigsaw web server. We compare the source models extracted by each of our tools, and describe reasons for differences in the extracted models.

1 Introduction

Recent research has shown that automated tools can be used to help engineers understand software systems and evaluate them for various quality characteristics. Software engineering tools such as CIAO [5], Dali [9], PBS [7], Rigi [11], and RMTool [13] extract an entity-relational [2] model from a system implementation. Typically, these tools parse the system source code to extract a model of the system; however, profiling information and code instrumentation have also been used to derive this model. The form of the extracted source model varies from tool to tool, but for tools that deal with procedural languages such as the C programming language, a model that includes a function call graph seems natural. In addition to function calls, type usage and variable references have also been considered.

Submitted to the 1999 ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering Toulouse, France, September 6, 1999.

There are a variety of tools that extract a source model from programs written in the C language: Murphy *et al.* [12] compared nine tools that extract a function call graph from C source code. There are fewer extractors for the Java[1] programming language because of its relatively recent introduction. As existing software engineering tools are adapted to analyse Java code, new fact extractors will be needed to extract a source model from the Java system. Extraction techniques that are currently used may not be appropriate for Java programs. Also, there may be new extraction techniques that are particularly suited to the Java language.

One extraction approach that seems particularly promising for Java is based on disassembling the compiled source code. Unlike some other programming languages (such as C), compiled Java code (byte-code) contains type, method, and field names, and has a simple, standardized structure that can be easily disassembled. By examining the compiled source, a disassembling extractor can take advantage of the standard compiler to scan, parse, and analyse the source code. In addition to disassembly, source code parsing and profiling can be used to extract facts from a system implementation.

In order to explore the design space of possible extractors for Java systems, we implemented three fact extractors that use different sources of information: parsing, disassembly, and profiling. We compared the results of these extractors by applying them to two systems: javac (the Sun Java compiler) and Jigsaw (an implementation of a web server).

1.1 Organization

The remainder of this paper is organized as follows. Section 2 describes the source model that we extract. Section 3 describes the three extractors. Section 4 describes the results

[1] Java is a trademark of Sun Microsystems.

of comparing the extractors on the three systems. Finally, Section 5 summarizes our results and provides suggestions to developers of fact extractor.

2 Domain Model

In order to compare the results of different extractors, we must store the facts in a common format. Since we are interested in entity-relational model [2], the format is a schema consisting of *entities*, *relations* between these entities, and *attributes* that describe the entities and relations. This model can be readily stored in a relational database (as is done by Dali and CIAO) or in a textual representation (as is done by Rigi and PBS).

Figure 1 shows an overview of the schema we used for our extractors. We created this schema based on the models of Rayside *et al.* [15] and Korn *et al.* [10] and our observations of Java systems.

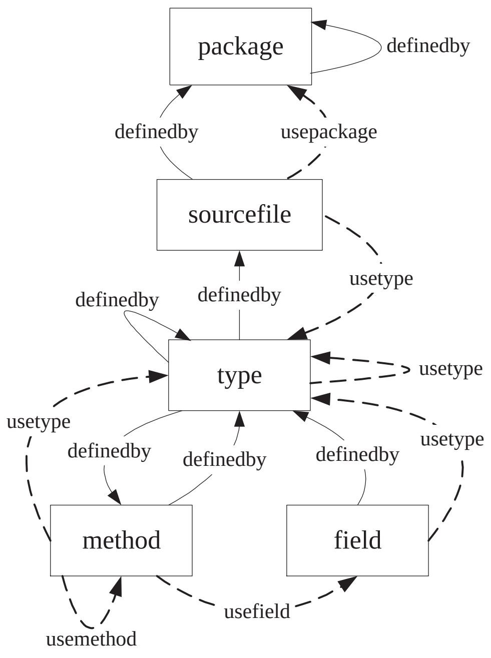

Figure 1: Schema for Extracted Facts

2.1 Entity Types

Our schema only represent facts for entities that are visible in some way outside of a compilation unit; for example, we do not include local variables as entities since these cannot be accessed outside of their defining method. However, we do represent classes defined within methods, since objects of these types can be returned and used outside of the defining method.

Our schema contains the following entity types:

- **package:** Packages are used by developers to group related classes and interfaces. Java packages are hierarchically defined.
- **sourcefile:** Each source file is a single compilation unit, and is associated with a single package. A source file defines zero or more classes or interfaces.
- **type:** Data types in Java can be primitive types (such as integers or characters), classes, interfaces, or arrays.
- **method:** A method is a Java function. All methods are defined in a class or interface.
- **field:** A field is a variable or constant that is part of a class or interface.

2.2 Relation Types

Our schema has two categories of relation: a *defined-by* relation links an entity to the entity that defines it, and a *uses* relation links an entity to the entities it refers to. All relation types represent a dependency, with the dependency direction being the direction of the relation; in this way, the extracted relations form a *dependency graph*. Within these two categories of relation, there are a number of more detailed relation types. For example, a method may *use* a field either by retrieving its value or modifying its value. In fact, there is a hierarchy of useful relation types. Table 1 shows the hierarchy of relation types that we use in our model.

The hierarchy of relation types allows tools to use a source model that is sufficiently detailed for the required analysis. For example, for reachability analysis, it may be sufficient to treat all relation types as the *dependson* type. Due to space limitations, this paper only discusses the following relations: *definedby*, *usepackage*, *usetype*, *usefield*, and *usemethod*. More specific relation types can be generalized to one of these relation types by using the hierarchy shown in Table 1. For example, the *getsfield* relation type generalizes to *usefield*, *uses*, or *dependson*.

2.3 Naming Entities

In order to meaningfully compare facts extracted from different extractors, we must have a way to consistently name entities so that an entity will be given the same name by all extractors. The Java specification [4] provides a unique naming scheme for packages, types, methods, and fields. However, for inner classes that are not visible outside of a compilation unit (such as anonymous classes defined within a method), the naming scheme is a only a recommendation for compilers, not a requirement [8]. In the cases where the specification does not provide a single correct name, we emulate the behavior of Sun's JDK 1.2 Java compiler (although this is not always compatible with other compiler naming schemes).

Java defines two formats for names: source-code names are based on the appearance of identifiers in the source-code, and byte-code names based on the appearance of names in

Relation	**From (A)**	**To (B)**	**bxjs**	**bxjb**	**bxjp**	**Description**
dependson	*any*	*any*	+	+	+	Entity **A** depends on entity **B**.
definedby	*any*	*any*	+	+	+	Entity **A** is defined by entity **B**.
packagedefby	package	package	+	+	+	Package **A** is a sub-package of **B**.
filedefby	sourcefile	package	+	+	+	Source file **A** contains a ‘package B;’ declaration.
typedefby	type	sourcefile	+	+	+	Type **A** is defined by source file **B**.
typedefby	type	type	+	+	-	Type **A** is defined within **B** (a class or interface).
typedefby	type	method	+	-	-	Type **A** is defined within method **B**.
methoddefby	method	type	+	+	+	Method **A** is defined by **B** (a class or interface).
fielddefby	field	type	+	+	+	Field **A** is defined by **B** (a class or interface).
uses	*any*	*any*	+	+	+	Entity **A** uses entity **B**.
usepackage	sourcefile	package	+	-	-	Entity **A** uses package **B**.
importondemand	sourcefile	package	+	-	-	Source file **A** imports the classes from package **B**.
usetype	*any*	type	+	+	+	Entity **A** uses type **B**.
caststo	method	type	+	+	-	Method **A** casts an expression to type **B**.
catches	method	type	+	+	-	Method **A** has a catch block for exception type **B**.
checksinstanceof	method	type	+	+	-	Method **A** uses an ‘instanceof’ expression for type **B**.
declaresthrows	method	type	+	+	-	Method **A** declares it may throw an exception of type **B**.
extends	type	type	+	+	-	Class **A** extends class **B**.
fieldtype	field	type	+	+	-	Field **A** is declared as type **B**.
implements	type	type	+	+	-	Class **A** implements interface **B**.
imports	sourcefile	type	+	-	-	Source file **A** imports **B** (a class or interface).
instantiates	method	type	+	+	+	Method **A** instantiates an object of type **B**.
parmoftype	method	type	+	+	-	Method **A** has a parameter of type **B**.
returntype	method	type	+	+	-	Method **A** returns a value of type **B**.
throws	method	type	+	+	-	Method **A** actually throws an exception of type **B**.
varoftype	method	type	+	+	-	Method **A** has a local variable of type **B**.
arraytype	type	type	+	+	-	Array **A** is an array of **B** elements.
usefield	method	field	+	+	-	Entity **A** uses field **B**.
getsfield	method	field	+	+	-	Method **A** retrieves field **B**’s value.
setsfield	method	field	+	+	-	Method **A** changes field **B**’s value.
usemethod	method	method	+	+	+	Entity **A** uses method **B**.
invokes	method	method	+	+	+	Method **A** calls method **B**.

Table 1: Hierarchy of Relation Types Extracted by Parsing (bxjs), Disassembly (bxjb), and Profiling (bxjp)

compiled code. We use byte-code names for our identifier naming scheme since not all entities have a source-code name (for example, anonymous inner classes have no source-code name but do have a byte-code name). We define the names of source files (which are not specified by the Java byte-code naming scheme) as the string ‘F-’, followed by the file’s package name, followed by the file name (without path information). By using identifiers with a standardized format, we can compare the facts extracted by different extractors.

3 Extractors

We implemented three different sources of facts that can be used to derive information about Java systems: *bxjs*, a parser that parses Java source code; *bxjb*, a disassembler that reads Java .class files; and *bxjp*, a profiler that monitors a running instance of a system. These three fact extractors emit entities, relations, and attributes in the TA format [6] conforming to the schema described in Section 2. We expect that each of these extractors will have certain strengths and weaknesses.

3.1 Parser

The bxjs source code parser has access to all of the information that can be computed statically: it can extract comments, and all references to types, fields, and methods that appear in the source code. However, the parser is the most complex of the three extractors since it not only has to scan and parse source code, but it also must read in Java .class files (for system classes without source code) and perform full data type analysis. The complexity of the source code parser makes it likely that there will be errors in its implementation. In particular, the parsing technique is vulnerable to incorrect interpretation of the Java specification, and also to changes in the specification that affect the syntax of Java programs.

The bxjs parser can extract all of the entity and relation types defined in Section 2. All source-code entities have associated line number information. All extracted *invokes* relations are to a target chosen by the compile-time type of the target object.

3.2 Disassembler

The bjxb disassembling extractor operates on Java .class files and has access to strictly less information than a parser (since a parser can reproduce the .class file, but the .class file cannot precisely reproduce the source code. Comments are not available in compiled code, and also some relations may not be present in compiled code due to optimizations performed by the Java compiler. Despite these weaknesses, a disassembling extractor is appealing because it is much simpler than a parser. The nature of compiled Java code makes it easier to reconstruct the source-code relations than for other languages [3]. The extractor need only read .class files; these are written in a simple, well-defined format. As future specifications of the Java language emerge, it is likely that the .class file specification will remain compatible with the current specification. Although less information is available compared to parsing, a disassembling extractor may be preferable because it leverages the significant development effort of creating a Java compiler.

For Java code compiled with full debugging information, the bxjb disassembler extracts all of the entity and relation types defined in Section 2 with the exception of the *importondemand* and *imports* relation types. A disassembler does not have access to source-code comments, but it can associate line numbers from the debugging information with extracted relations. However, a disassembler cannot associate line numbers with declarations of source entities.

3.3 Profiler

Unlike parsing and disassembly, the bxjp profiling extractor operates by monitoring the run-time performance of a system using the Java Virtual Machine Profiling Interface (JVMPI) [14]. Since run-time behavior is observed, the profiling extractor can determine what method override is chosen for each method invocation. This determination requires a knowledge of the run-time type of expressions, and thus cannot be performed statically by parsers or disassemblers. In addition to being able to determine the run-time method that is selected due to overriding, a profiling extractor can detect relations that are the result of run-time binding. For example, the Java reflection API allows Java programs to instantiate objects using a string that represents the class name. In some systems, configuration parameters determine what classes will be instantiated. Profiling can detect this instantiation of objects whereas parsing and disassembly approaches cannot.

Unfortunately, the profiling approach requires that the system be exercised fully. Since the quality of the results rely on test coverage, a profiling approach is unlikely to extract all of the possible relations. For example, it is often difficult to cause a system to enter error states that are not expected to occur. Relations arising from these states are unlikely to appear in facts extracted from profiling. In addition to missing relation due to incomplete coverage, the bxjp profiler extracts less of the domain model than bxjs or bxjp. The bxjp extractor extracts neither comments nor source locations.

4 Comparing the Extractors

In order to compare the results of the three extractors (bxjs, bjxb, and bxjp), we used them to extract facts from two systems: Jigsaw (a web server implementation), javac (Sun's Java compiler). For each of these systems, we extracted a source model using each of the extractor tools, then compared these extracted facts. Section 4.1 describes reasons we found for differences in the extracted source models, and Section 4.2 summarizes the differences between the extracted source models.

4.1 Reasons for Differences

We found the following reasons for differences in the extracted source models:

Constant fields. Java compilers do not emit field references if a field has a constant value at compile time; instead, they emit the constant value. Only the source code parser (bxjs) can extract relations to constant fields.

Constructors. The Java specification states that if a developer does not define a constructor for a class, the compiler will generate a default constructor. These generated constructors do not appear in the source code, so a parsing extractor might not extract these entities or the relations they cause. We chose to emulate a compiler with bxjs; it creates a default constructor or class initializer if necessary, and introduces a call to the superclass constructor if the programmer did not specify one.

Inner-class constructors. When compiling inner-classes, Java compilers add parameters to the constructor to pass the instance (*this*) of containing classes. In addition, classes defined within methods can refer to the method's parameters or local variables. These variables are passed as additional parameters to the constructor. The parsing extractor (bxjs) does not add these extra parameters; therefore, inner class constructors for bxjs have a different identifier than the same constructor extracted by either bxjb or bxjp.

Classes defined in methods. Java allows developers to define classes within method bodies. While parsing, it is easy to associate these classes with the method that defines them. However, this association is not stored in the compiled byte-code and so cannot be extracted by bxjb or bxjp; instead, these extractors associate these classes with the class containing the defining method.

Debugging statements. Many Java programs contain statements that only execute in a debugging version of a program. These statements are typically nested inside an '`if( debug )`' statement, where '`debug`' is a constant field that is set to true for development versions and false for released versions. If debugging statements can be determined to be unreachable at compile time, Java compilers do not emit byte-code for them (even if the compiler has optimizations disabled). If a disassembling or profiling extractor is used on a release version of a system, it will not be able to find these relations due to debugging statements.

String concatenation. Java compilers implement string concatenation by using a helper class, StringBuffer. Profiling and disassembling extractors will find relations from methods that use string concatenation to StringBuffer; these relations are not emitted from a parsing extractor.

Synthetic entities. In order to implement some language features, Java compilers create *synthetic* methods and fields that do not appear in the source code but are used in the byte-code. For example, synthetic methods and fields are used to store the instance (*this*) value for containing classes and to permit access to inner class fields that would be otherwise inaccessible. Synthetic methods and fields are not extracted by the bxjs, but they are extracted by bxjb and bxjp.

Unused local variables. When Java compilers emit debugging information, they emit the names and types of all used local variables. However, local variables that are not referenced (or are referenced only in unreachable code such as debugging statements) are not emitted. Only bxjs can determine the relation from methods to the data type of their unused local variables.

Optimizations. We performed our experiments on code that was compiled with full debugging information (including local variable types and names and line number information) and optimizations disabled. If, instead, optimized code is disassembled or profiled, then the extractor may miss entities and relations that are removed by optimization (for example, unused private methods may be eliminated). In addition, if a compiler inlines a method definition to improve performance, the disassembling or profiling extractor will attribute relations to the wrong method.

Reflection. The Java reflection API provides a mechanism for developers to instantiate objects based on a class name in a configuration file or input by a user. This mechanism is used extensively in Jigsaw to support configuration-time extensibility. In addition to instantiation, the reflection API allows programs to invoke methods and access fields using the textual names. Because these relations are determined at configuration time, the static extractors (bxjs and bxjb) cannot find these relations; however, the bxjp profiling extractor does report method invocation and object instantiation performed using the reflection API.

Native methods. Java allows developers to supplement their Java code with *native code* that is written in a system-level language such as C or C++. Native code can instantiate objects, invoke methods, access fields, and throw exceptions. Because native code is not implemented in Java, Java parsers and disassemblers cannot be used to extract facts from it. However, profiling can be used to monitor object instantiation from native code, and method invocation to or from native code.

Polymorphic method invocation. Java allows developers to override the implementation of superclass methods. At run time, the target of such a method invocation is chosen based on the run-time type of the associated object. Both static extractors (bxjs and bxjb) extract relations based on the compile-time type of an object, but bxjp extracts relations to the method that is selected at run-time.

4.2 Summary of Model Differences

Table 2 provides a summary of the entities extracted by each extractor (bxjs, bxjb, and bxjp) for the javac Java compiler and the Jigsaw web server systems.

	Jigsaw Web Server			javac Java Compiler		
	bxjs	bxjb	bxjp	bxjs	bxjb	bxjp
package	69	69	30	14	14	6
sourcefile	599	599	201	234	234	168
type	821	822	229	283	283	173
method	6049	5971	2196	2509	2508	1635
field	2784	2855	0	1456	1463	0
All	10322	10316	2656	4496	4502	1982

Table 2: Summary of Extracted Entities

The bxjs and bxjb extractors extracted a similar set of entities; they did not extract the same set due to the reasons presented in Section 4.1. The bxjp extractor only extracted entities that were loaded by the tests we used.

	Jigsaw Web Server			javac Java Compiler		
	bxjs	bxjb	bxjp	bxjs	bxjb	bxjp
uses	58151	60872	2254	30022	30891	3210
usepackage	1764	0	0	197	0	0
usetype	28263	29960	496	13553	15803	720
usefield	10047	10768	0	7612	5390	0
usemethod	18077	20144	1758	8660	9698	2490

Table 3: Summary of Extracted Relations

Table 3 summarizes the relations that the three extractors found in the two systems. Again, the bxjs and bxjb extractors extracted similar, but not identical, source models.

Due to space constraints, we do not compare the individual source models to show the precise nature of the differences. These results are available in an expanded version of this paper [1].

5 Results

We have defined a domain model for facts extracted from Java systems. By comparing the results extracted from three sources (parsing, disassembly, and profiling), we have identified characteristics to help developers of software engineering tools decide how to extract facts from Java systems.

	Extractor	Extraction Time	
	Size (LOC)	javac (54 KLOC)	Jigsaw (105 KLOC)
bxjs	14,453	71 s	135 s
bxjb	2,602	11 s	22 s
bxjp	265	1325 s	798 s

Table 4: Comparison of Extractor Complexity

Table 4 compares the complexity of each of the extractors, including the size of the extractor and the average runtime of the extractor for each system. The run-time for the bxjp extractor includes the time for running all of the tests to provide adequate coverage. All tests were performed on a Pentium II 450MHz with 256MB of RAM, running NT 4.0 and Sun JDK 1.2.

If static analysis is sufficient, then a disassembling extractor is probably the best choice. The bxjb extractor is simple and fast, and will likely work for future versions of the Java language. Although it is affected by optimizations performed by the compiler, bxjb can extract almost the entire domain model. In addition, a disassembling extractor does not need access to the system source code, and can extract facts using only part of a system implementation.

If an extractor needs very accurate results (for example, for reachability analysis including debugging statements), then a parsing extractor must be used to avoid missing relations that are omitted by compiler optimizations. A parsing extractor can extract the entire domain model, regardless of optimizations that would be performed by a compiler. In addition, a parser can extract comments and associate accurate line number information with all source-code entities. The accuracy of the parsing approach comes at a price in complexity; bxjs is the largest extractor, and is significantly slower than bxjb.

Finally, some systems (such as Jigsaw) use by-name instantiation. These systems may not be amenable to static analysis methods such as parsing and disassembly. Instead, a profiling approach can be used to find relations due to configuration-time binding: by-name instantiation, and method invocation based on run-time typing. Although it is fairly simple to implement a profiler using the JVMPI, profilers can only extract facts for a portion of our domain model. In addition, profiling results are highly dependent on the test coverage used during extraction.

Each of the extraction techniques we examined has unique qualities that make it suitable for reverse engineering efforts. By combining results from different extractors, tools may be able to get the benefits of all of these techniques without their drawbacks.

Acknowledgments

This work was supported in part by the CSER (Consortium for Software Engineering) as well as by CITO (Centre for Information Technology, Ontario). We would like to thank Derek Rayside and Kostas Kontogiannis for valuable discussions regarding our data model.

References

[1] Ivan Bowman. A Java domain model. Available at: http://plg.uwaterloo.ca/~itbowman/papers/java-erd.html, March 1999.

[2] P. P. Chen. The entitiy-relationship-model - towards a unified view of data. *ACM TODS*, 1(1):9–36, 1976.

[3] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, July 1994.

[4] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, August 1996.

[5] Judith Grass and Yih-Farn Chen. The C++ information abstractor. In *The Second USENIX C++ Conference*, April 1990.

[6] Ric Holt. An introduction to TA: The tuple-attribute language. Available at: http://www-turing.cs.toronto.edu/pbs/papers/ta.html, March 1997.

[7] Ric Holt. Software Bookshelf: Overview and construction. Available at: http://www-turing.cs.toronto.edu/pbs, March 1997.

[8] Inner classes specification. Available at: http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/.

[9] Rick Kazman and S. Jeromy Carrière. View extraction and view fusion in architectural understanding. In *Proceedings of ICSR5*, Toronto, Canada, June 1998.

[10] Jeffrey Korn, Yih-Farn Chen, and Eleftherios Koutsofios. Reverse engineering of Java applets. Available at http://www.research.att.com/~ciao/doc/chava.ps, December 1998.

[11] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilly, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[12] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S-C. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, 1998.

[13] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT'95*, pages 18–28, October 1995.

[14] The Java virtual machine profiler interface. Available at: http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/index.html, 1998.

[15] Derek Rayside, Scott Kerr, and Kostas Kontogiannis. Change and adaptive maintenance detection in Java software systems. In *5th IEEE Working Conference on Reverse Engineering*, October 1998.