Byte Code Engineering

Markus Dahm

Freie Universität Berlin
dahm@inf.fu-berlin.de

Abstract. The term "Java" is used to denote two different concepts: the language itself and the related execution environment, the Java Virtual Machine (JVM), which executes *byte code* instructions. Several research projects deal with byte code-generating compilers or the implementation of new features via byte code transformations. Examples are code optimization, the implementation of parameterized types for Java, or the adaptation of run-time behavior through load-time transformations. Many programmers are doing this by implementing their own specialized byte code manipulation tools, which are, however, restricted in the range of their reusability. Therefore, we have developed a general purpose framework for the static analysis and dynamic creation or transformation of byte code. In this paper we present its main features and possible application areas.

1 Introduction

Many research projects deal with extensions of the Java language [13] or improvements of its run-time behavior. Implementing new features in the Java execution environment (Java Virtual Machine, JVM) is relatively easy compared to other languages, because Java is an interpreted language with a small and easy-to-understand set of instructions (the *byte code*).

The JAVACLASS API which we present in this paper is a framework for the static analysis and dynamic creation or transformation of Java class files.[1] It enables developers to deal with byte code on a high level of abstraction without handling all the internal details of the Java class file format. There are many possible application areas ranging from class browsers, profilers, byte code-optimizers, and compilers, to sophisticated run-time analysis tools and extensions to the Java language [1, 21, 3]. Other possibilities include the static analysis of byte code [22], automated delegation [8], or implementing concepts of "Aspect-Oriented Programming" [16]. We think that the most interesting application area for JAVACLASS is meta-level programming, i.e. load-time reflection [18], which will be discussed in detail in section 3.1.

Our approach provides a truly object-oriented view upon Java byte code. For example, code is modeled as a list of instructions objects. Within such a list one may add or delete instructions, change the control flow, or search for certain patterns of code using regular expressions.

We assume the reader to have some basic knowledge about the JVM and Java class files. A more detailed introduction to the API and the Virtual Machine can be found in

[1] The JAVACLASS distribution, including several code examples and javadoc manuals, is available at `http://www.inf.fu-berlin.de/~dahm/JavaClass/index.html`.

C. H. Cap (ed.), *JIT'99*

[9]. The paper is structured as follows: We first give a brief overview of related work and present some aspects and technical details of the framework in section 2. We then discuss concepts of byte code engineering and possible application areas in section 3 and conclude with section 4.

1.1 Related work

The JOIE [7] toolkit can be used to augment class loaders with dynamic behavior. Similarly, "Binary Component Adaptation" [15] allows classes to be adapted and evolved on-the-fly. Han Lee's "Byte-code Instrumenting Tool" [17] allows the user to insert calls to analysis methods anywhere in the byte code. The Jasmin assembler [20] can be used to compile pseudo-assembler code. Kawa, a Java-based Scheme system, contains the `gnu.bytecode` package [5] to generate byte code. The metaXa Virtual Machine [12] allows to dynamically *reify* meta level events, e.g. instance field access.

In contrast to these projects, JAVACLASS is intended to be a general purpose tool for "byte code engineering". It gives the developer full control on a high level of abstraction and is not restricted to any particular application area.

2 The JavaClass framework

The JAVACLASS framework consists of a "static" and a "generic" part. The former is not intended for byte code modifications. It may be used, e.g., to analyze Java classes without having the source files at hand. The latter supplies an abstraction level for creating or transforming class files dynamically. It makes the static constraints of Java class files, like hard-coded byte code addresses, mutable. Using the term "generic" here may be a bit misleading, we should perhaps rather speak of a "generating" API. UML diagrams – unfortunately too large for this paper – describing the class hierarchy of the framework can be found in [9].

2.1 Static API

All of the binary components and data structures declared in the JVM specification [19] are mapped to classes, where the top-level class is called `JavaClass`, giving the whole API its name. Instances of this class basically consist of a *constant pool*, fields, methods, symbolic references to the super class and to the implemented interfaces of the class. At run-time, these objects can be used as *meta objects* describing the contents of a class. This possibility will be discussed in detail in section 3.1.

The constant pool serves as a central repository of the class and contains, e.g., entries describing the type signature of methods and fields. It also contains String, Integer, and other constants. Indexes to the constant pool may be contained in byte code instructions as well as in other components of a class file and in constant pool entries themselves.