Jato: A compact binary file format for Java class

Sheng-De Wang and Yuhder Lin
Department of Electrical Engineering,
National Taiwan University
Taipei 106, TAIWAN
Email: sdwang@cc.ee.ntu.edu.tw

Abstract

Java has been a very important programming language, especially with the cross-platform characteristic. But the CLASS file format defined in the Java virtual machine specification contains many redundancies and replications of information. These redundancies most come from the "constant pool" of a CLASS file. We propose a compact binary file format and its associated archive format, called the Jato file format and Jatar, respectively, for the Java system. Using these two formats, many of the redundancies can be removed. We didn't utilize any text compression technique in the proposed formats, so they will not sacrifice the loading speed and thus are very suitable for use in the embedded environments. We've also implemented our class loader capable of loading the Jato files into a regular Java virtual machine. Using this approach, we show that the Jato file format is effective and promising while still keeping the cross-platform feature of Java.

1. Introduction

Since its debut in mid-1995, Java [1][2] has been an important language in the modern computer technology. The Java language is an object-oriented, robust, secure and high-performance programming language. With the help of Java Virtual Machine (JVM) [3][4] and the bytecode (class file) format, Java programs can achieve the "cross-platform" goal. Much research has been done for making Java a more suitable developing tool for embedded systems [5] [6] [7] [8]. Among them, the issues of real-time, compact bytecode formats and operating systems are considered. We are motivated by [9] and aimed to reduce the redundancies in bytecode files.

A Java environment is composed of Java compilers, Java Virtual Machine, and associated Java API libraries and Native Methods. After compilation, a Java source file with a `.java` file extension will become one or several binary files with a `.class` file extension. They are in the class file format specified in the Java virtual machine specification [3]. Java applications, applets or servlets are of the same format. This is the reason why Java can achieve "cross-platform" goal. In Java, each class will have its own CLASS file. To avoid ambiguity, in this paper we use all-capital "CLASS" when we are referring to the class file. When we say "class" in lower case, we just mean the class structure concept of the Java language.

The problems for porting Java to embedded systems come from two aspects. The first is the language feature itself that brings more and more complex implementations for JVMs. The second is the large size of the class libraries. To solve these problems, there are mainly two research topics: making subsets of standard Java, and code compression or conversion. Java suites, such as PersonalJava [10], Embedded Java [11], and Java 2 MicroEdition [12][13] of Sun, and Chai [14] of HP are light-weight or high-performance versions of the standard Java. They keep all the language semantics and features of Java, while making JVM less complex and reducing the size of the class libraries. The JavaCard [15] specification does more than those listed above. It no longer utilizes the JVM-bytecode architecture; instead, the original applications are converted by a converter with the class libraries into a single executable image for the JavaCard virtual machine, where some features of Java language has not been implemented.

As described by [16], the binary formats can be separated into two categories. One kind of binaries is for storage and network transmission; the main goal is to reduce the code size at the highest compression rate. This kind of binaries is not directly suitable for direct execution. Before execution, it needs to decompress the binaries first, then execute the decompressed codes. This format is called *wire-format*. Another kind of binaries is of *execution-format*. In this format, we want to reduce the size of code as much as possible, but on the other hand, we require that these binaries are suitable for being directly executed. For embedded devices, both the storage spaces and computing powers are limited. If the code is in *wire-format*, the decompression might be too expensive for these devices. We should keep the balance between code sizes and the computing complexities.

Most research for compressing CLASS files is in the *wire-format*, such as CLAZZ [17], JAZZ [18] or Pugh [19]. They all utilize some forms of compression techniques [20], like `ZIP` or `gzip` to achieve high compression rate. Some other techniques, like JAX [21] can be utilized to remove

0-7695-1153-8/01 $10.00 © 2001 IEEE

the "unused" code in the CLASS files [22]. But after this process, the resulting binaries will lose *release-to-release binary compatibility* [1] with the original CLASS files. And we cannot predict if a new application will need the removed codes. In this paper, we will introduce a compact binary format for Java, which is in the *execution-format*. Motivated by [9], we are aimed to develop an approach to reduce the code size of Java binaries, without utilizing any text compression techniques.

2. Design Issues of the Jato File Format

The proposed Jato file format is to provide a more compact and efficient format than the original CLASS file format, while keeping the release-to-release binary compatibility. The following codes show the layout of the CLASS file format.

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info
constant_pool[constant_pool_count-1]
;
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info
methods[methods_count];
    u2 attributes_count;
    attribute_info
attributes[attributes_count];
    }
```

The u4 and u2 represent 4-byte and 2-byte unsigned integer, respectively. The values recorded in the CLASS file are all big-endian. Each entry in the constant_pool will have the same form as:

```
   cp_info {
    u1 tag;
    u1 info[];
   }
```

According to the 1-byte tag, the info bytes will have different layout. Java defines 11 distinct kinds of cp_info entries, as shown in Table 1.

Table 1 The possible values for tag

Constant Type	Tag Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

Except for those four structures representing constant numbers (int, float, double, long), all structure will finally refer to a CONSTANT_Utf8_info structure. In fact, both the CLASS file and the JVM depend heavily on the CONSTANT_Utf8_info to achieve many functionalities, like dynamic loading/binding, object inheritance, etc. The CONSTANT_Utf8_info has the form:

```
  CONSTANT_Utf8_info {
   u1 tag;
   u2 length;
   u1 bytes[length];
  }
```

The 2-byte length indicates the size of the bytes representing the string information. In Java, all strings are in UNICODE, and coded using UTF8 format. The constant pool of a CLASS file plays a very important role in the Java system. It is the only way by which JVM can reference a field of other classes, invoke a method of other classes (in fact, if the JVM wants to invoke a method of the current class, it must also find its information such as its name and method descriptors in the constant pool), or dynamically load or link other classes. Figure 1 shows the relations between each kinds of cp_info. Notably, the CONSTANT_NameAndType_info and CONSTANT_Utf8_info can only be referred to by other constant pool entries or other parts of a CLASS file. They will never be touched directly in a Java method, that is, the representation of constant pool is self-contented.

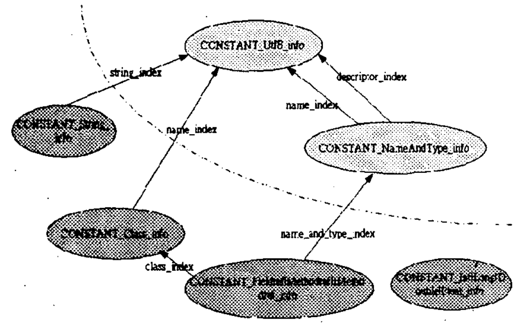

Figure 1 Relations between each kind of entries in the constant pool

Our main concern is to reduce the size of the constant pool. We use the concept of "*dividing fully-qualified name (FQN)*" in [9] and propose an extended "*division*" techniques. Nearly all the references in the constant pool of a CLASS file will finally point to a `CONSTANT_Utf8_info` structure. This is because the Java is a dynamic loading/linking system, and each class must have some ways to refer to the classes, fields, or methods that it will use in the other time. It seems that using the strings to represent these relations is the only possible solution. Since a string in its nature will take over more spaces than a primitive type or a reference offset, the constant pool will grow larger rapidly as the number of `CONSTANT_Utf8_info` structures increases. For a usual CLASS file, the constant pool may take 65%~70% [23] of the total files size. It is very large, comparing to the "symbol table" of other popular binary executable file formats, like ELF, COFF or PE.

3. Dividing the package names and the class names

We would like to remove the `tag` byte of each constant pool entry. The resolution is to divide the constant pool into several sections by their types. Then when we need a specific type of record, we'll just know where to get it without check its tag byte. The JVM instructions are in a similar situation. If an opcode needs an index into the constant pool, we can usually figure out the type of the targeted constant pool entry, not until checking the `tag` byte of it. But the `LDC` family are the exceptions. The `LDC`, `LDC_W`, `LDC2_W` may take a `CONSTANT_String_info`, `CONSTANT_Integer_info`, `CONSTANT_Float_info`, `CONSTANT_Long_info` or `CONSTANT_Double_info` as their operands. For dealing this situation, we add more instructions (only known by the Jato system) to existed instruction sets. These instructions are extended from `LDC`, `LDC_W` and `LDC2_W`, each consuming a specific type of operand, as shown in Table 2.

A class type in the CLASS file will be finally mapped to a string in the FQN form. We may reference many classes in some same packages, so the package name will be appeared as many times as we reference classes in it. For example, if we reference both the `String` class and the `System` class of the `java.lang` package, then we must have "*java/lang/String*" and "*java/lang/System*" in the constant pool, in which "*java/lang/*" are replicated.

Our idea is to divide each "**node**" of a FQN into a single record. For example, a `CONSTANT_Class_info` in which the `name_index` points to a string "*java/lang/String*" will be divided to three single "node_reocrd". Their `name_index`s will point to "*java*", "*lang*" and "*String*" respectively. The `node_record` structure is:

```
node_record{
    u2 name_index;
//pointing to a utf8 string
    u2 parent_index;
//pointing to another node_record
}
```

Table 2 Modified LDC instructions

Original opcode	Extended opcode	Descrption
`LDC(18)`	`LDCSTR(203)`	Load a String constant
	`LDCINT(204)`	Load an integer constant
	`LDCFLOAT(205)`	Load a float constant
`LDC_W(19)`	`LDCSTR_W(206)`	Using 2-byte index
	`LDCINT_W(207)`	Using 2-byte index
	`LDCFLOAT_W(208)`	Using 2-byte index
`LDC2_W(20)`	`LDC2LONG_W(209)`	Load a long constant, using 2-byte index
	`LDC2DOUBLE_W(210)`	Load a double constant, using 2-byte index

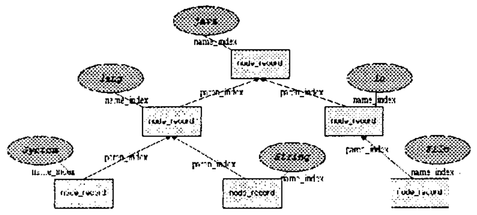

Figure 2 The proposed class naming system

When we need to refer to another classes in the java.lang package, we just need to add one node_reocord and its name to the constant pool, not the whole FQN. Thus, we can reduce the replications of the same path names that have been referenced before. Figure 2 gives a clear description.

We propose treating arrays as a specific type, not classes. The array record is:

```
array_record{
    u1 dimensions;
    u2 type_index;
}
```

The `dimensions` field just represents the dimensions of this array, and the `type_index` point to a `node_record`. We reserve the number 1-9 for primitive types. That is, if a `type_index` is not greater than 9, it represents a primitive type.

Following is the layout of the Jato file format:

```
JatoFile{
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u1 exist_region_flag;
    u1 exist_region_flag2;
    u2 field_ref_no;
    MCONSTANT_Fieldref_info ref_field_info[field_ref_no];
    u2 method_ref_no;
    MCONSTANT_Methodref_info ref_method_info[method_ref_no];
    u2 imethod_ref_no;
    MCONSTANT_Methodref_info ref_imethod_info[imethod_ref_no];
    u2 const_str_no;
    MCONSTANT_String_info ref_string_info[const_str_no];
    u2 const_int_no;
    MCONSTANT_Integer_info ref_int_info[const_int_no];
    u2 const_float_no;
    MCONSTANT_Float_info ref_float_info[const_float_no];
    u2 const_long_no;
    MCONSTANT_Long_info  ref_long_info[const_long_info];
    u2 const_double_no;
    MCONSTANT_Double_info ref_double_info[const_double_info];
    u2 class_ref_no;
    node_record ref_class_info[class_ref_no];
    u2 pknode_no;
    node_record pknode_info[pknode_no];
    u2 array_ref_no;
    array_record ref_array_info[array_ref_no];
    u2 md_no;
    MCONSTANT_MethodDescriptor_info ref_md_info[md_no];
    u2 utf8_no;
    MCONSTANT_UTF8_info ref_utf8_info[utf8_no];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    Mfield_info fields[fields_count];
```

```
        u2 methods_count;
        Mmethod_info methods[methods_count];
        u2 attributes_count;
        Mattribute_info attributes[attributes_count];
}
```

The parts in the rectangle is just the "**constant pool**". It is clear that we have divided the constant pool into several parts. We've removed the NameAndType record, added Array and MethodDescriptor records. And for convenience, we separate the package nodes from the classes records. So we have 13 divisions in our constant pool. But for a normal class, it will not usually have all kinds of the entries in its constant pool. So we use two flags bytes (totally 16-bit) to indicate if a division exists in this Jato file. Following the CLASS file format, the Jato file format is also big-endian.

The exist_region_flag and exist_region_flag2 tell the existences of the 13 divisions in the constant pool. Since we have 13 divisions, 2-byte (16-bit) flags are enough. If the n-th bit of exist_region_flag is a 0, then the n-th division will not exist. Each division is composed as the following form:

```
u2 item_counts;
RECORD_type
    RECORD_name[item_counts];
```

Following are the descriptions of each division.

• Division 1: ref_field_info

The ref_field_info is an array of MCONSTANT_Fieldref_info. Each original CONSTANT_Fieldref_info in the CLASS file will have a corresponding MCONSTANT_Fieldref_info in the Jato file.

• Division 2 and 3: ref_method_info and ref_imethod_info

The ref_method_info and ref_imethod_info are of the same structure, the MCONSTANT_Methodref_info. Each CONSTANT_Methodref_info in the CLASS file will have a corresponding MCONSTANT_Methodref_info in the ref_method_info array of the Jato file, and a CONSTANT_InterfaceMethodref_info will have one in the ref_imethod_info array.

• Division 4: ref_string_info

The MCONSTANT_String_info structure is:

```
MCONSTANT_String_info{
    u2 string_index;
}
```

where string_index points to a MCONSTANT_UTF8_info in ref_utf8_info. Each CONSTANT_String_info in the CLASS file will have a corresponfing MCONSTANT_String_info record in the Jato file.

• Divsion 5~8: ref_[int, float, long, double]_info

The MCONSTANT_Integer_info, MCONSTANT_Float_info, MCONSTANT_Long_info and MCONSTANT_Double_info are almost the same as CONSTANT_Integer_info, CONSTANT_Float_info, CONSTANT_Long_info and CONSTANT_Double_info in the CLASS file format, except for the tag byte. And each CONSTANT_[Integer, Float, Long, Double]_info in a CLASS file will have a corresponding MCONSTANT_[Integer, Float, Long, Double]_info in the Jato file.

• Division 9 and 10: ref_class_info, pknode_info

We've shown how a FQN is divided into several nodes. The leaf of a package tree just represents the class reference, and all type_index or class_index of other structures point to them (in some cases, ref_array_info). We put these leaves in ref_class_info array. The index of the ref_class_info array is counting from 10, because 0 is reserved for special use, and 1~9 represent the primitive types. All other nodes, except for the leaves, will be put in pknode_info array. The parent_index of a node_record structure is just the index into pknode_info, counting from 1. If a node_record has *zero* as its parent_index, we will know that it is at the root of a package tree.

• Division 11: ref_array_info

The array_record has no corresponding record in the original CLASS file format. When we recognize a class reference or a type is of an array type, we create one array_record for our Jato file. Then, how can we distinguish array refernces from class references when we see a type_index in other records, or in the index bytes of some JVM instructions? We just use *65535-x* to represent the real index *x* of the ref_array_info. That is, if we find a type_index is greater than class_ref_no, we just know that it is an array reference instead of a class reference. For example, if the class_ref_no is *100*, and we are encountering a type_index of which value is *65530*, we will refer to ref_array_info[5].

• Division 12: ref_md_info

The CONSTANT_MethodDescriptor_info is to describe a descriptor (signature) of a method. In the CLASS file, it is in the form of a CONSTANT_Utf8_info. We decompose it and build a MCONSTANT_MethodDescriptor_info in our Jato file.

• Division 13: ref_utf8_info

The MCONSTANT_UTF8_info is the same as CONSTANT_Utf8_info except for the tag byte. After conversion from CLASS to Jato, the number of these records should be reduced greatly because of the work we've done for class references, field refernces and method references. And we promise that there will not be two identical utf8 strings in the ref_utf8_info. Figure 3 shows the relations between each division.

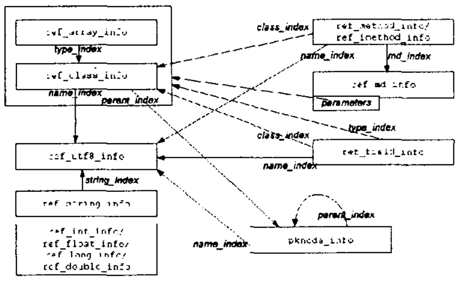

Figure 3 Relations between each constant pool regions in Jato file format

For generating the Jato files, we need a converter to convert the original CLASS files. Then, for executing the Jato files, we must have a *Class Loader* capable of loading Jato files into the JVM. For this approach, we don't need to make any modifications to the JVM. Of course, in a more direct manner, we may modify the JVM to give it the ability to execute the Jato files. Figure 4 gives a block diagram to describe our Jato system.

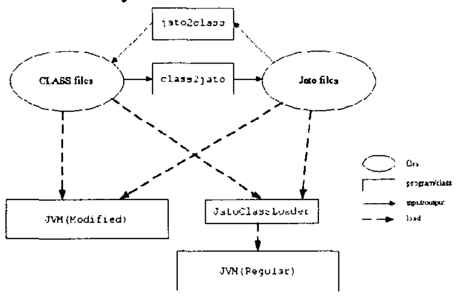

Figure 4 The Jato system

4. The Jatar Archive Format

The files in the Jatar archive can be categorized into 3 kinds: constant pool files, simplified Jato files, and files of other types. The constant pool files will be put in /00constant_pool directory of the archive, the remainings will go to /files. The merged constant pools are divided into 13 files, one file for one region. That is, we will have 13 files named ref_field_info, ref_method_info, ref_imethod_info, ref_string_info, ref_int_info, ref_float_info, ref_long_info, ref_double_info, ref_class_info, pknode_info, ref_array_info, ref_md_info and ref_utf8_info. We put the constant pool files in /00constant_pool directory of the archive. Since we have merged the constant pools of each Jato files, we don't need to keep a constant pool in each single Jato file. The layout of the simplified-Jato files is just the one of the Jato files removing the constant pool part. And these simplified-Jato files will be named with .sjato extension.

```
SimpJatoFile{
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    Mfield_info
fields[fields_count];
    u2 methods_count;
    Mmethod_info
methods[methods_count];
    u2 attributes_count;
    Mattribute_info
attributes[attributes_count];
}
```

The SimpJatoFile structure will contain only the informations of the class it represents. When it need a external reference, it will find that reference in proper constant pool file under /00constant_pool. For example, if this_class of a SimpJatoFile file is *103*, its information will be found at the *93*rd entry (as in the original Jato files, *zero* is for special use, and *1~9* is primitive types) of /00constant_pool/ref_class_info. The same as the Jar format, we can store some files other than simplified-Jato files in our archive, for example, audios, videos, or pictures. Figure 5 shows the general layout of a Jatar archive. The simplified-Jato files in /files will still obey the hierarchical relations of packages.

5. Evaluations

We take several famous software distributions in our evaluations for size reduction. They are listed in Table 3. Distributions of Java programs are usually stored in ZIP or JAR file format, which is a variation of ZIP(for most case, files are just merged together in the ZIP or JAR archives without compression). We extracted them into plain files, and did our experiments on these files. Since we are working on CLASS files, we will ignore files other than CLASS files. Table 4 shows the statistics for CLASS files of our selected distributions before conversion. And Table 5 shows the results after conversion (into Jato files).

Table 3 The tested software distributions

Distribution	File	Description
`runtime_122`	`rt.jar`	The run-time class library for jdk 1.2.2 of Sun
`jigsaw_20`	`jigsaw.zip`	The Jigsaw webserver ver 2.0 of W3C
`symclass_30`	`sym.zip`	The symantec classes for Visual Cafe 3.0 of Symantec
`j3d12b2`	`j3dcore.jar`	The Java3D API 1.2 beta 2 runtime API of Sun

Table 4 The statistics for the distributions

Distribution	files count	total size	total cp size	cp%
`runtime_122`	4317	9896767	5713895	57.73%
`jigsaw_20`	784	2069957	1280536	61.87%
`symclass_30`	1416	4084174	2799371	68.54%
`j3d12b2`	358	1807114	831826	46.03%

(*files count*: the number of CLASS files in this archive; *total size*: the total size of these CLASS files in bytes; *total cp size*: the total size of constant pools of these CLASS files in bytes; *cp%*: (*total cp size*)/(*total size*) x 100%)

Table 5 After `class2jato` conversion

Distribution	total size	total save%	total cp size	cp save%	cp%
`runtime_122`	7327856	25.97%	3201507	43.97%	43.69%
`jigsaw_20`	1439620	30.45%	659973	**48.46%**	45.84%
`symclass_30`	2809150	**31.22%**	1546518	41.76%	55.05%
`j3d12b2`	1419979	21.42%	455437	45.25%	32.07%

(*total size*: the total size of the result Jato files in bytes; *total save%*: the average file size saved in percentage; *total cp size*: the total size of constant pools of the Jato files in bytes; *cp save%*: the average size saved for constant pools in percentage; *cp%*: (*total cp size*)/(*total size*) x 100%)

Table 6 The Jatar archive sizes

Distribution	Jatar size	Jato files size	save% comapred to Jato files	CLASS files size	save% compared to CLASS files
runtime_122	4774029	7327856	34.85%	9896767	51.76%
jigsaw_20	912457	1439620	36.62%	2069957	55.92%
symclass_30	1613750	2809150	42.55%	4084174	60.49%
j3d12b2	1111838	1419979	21.70%	1807114	38.47%

Table 6 shows the archive size of each distribution and the comparisons between original Jato files and CLASS files. Except for `j3d12b2`, we had all other three distributions reduced their sizes for more than 50% through `class2jato` conversion and Jatar archiving.

6. Conclusions

In this paper, we have designed a compact binary format for Java, which is called the Jato file format. Using this format, we can remove many redundancies in the constant pools of original CLASS files. The experiment results show that we have done better work than existing approach. We believe that the Jato file format is promising and effective for embedded environement.

An archive format called Jatar, in which many Jato files are collected and share the same constant pool files, is also proposed. Merging many Jato files in this format, we can save more than 50% storage spaces for original CLASS files or JAR archives.

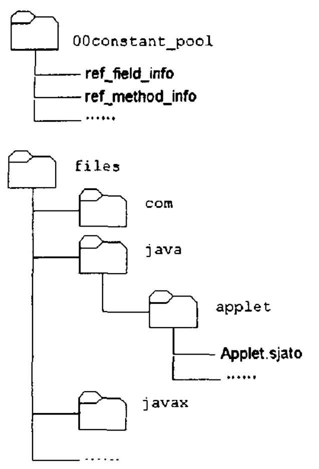

Figure 5 An Example of Jatar Archive

References

[1] J. Gosling, B. Joy and G. Steele. *The Java Language Specification*. Sun Microsystems, 1996

[2] J. R. Jackson and A. L. McClellan. *Java 1.2 By Example*. Sun Microsystems, 1999

[3] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification 2nd Edition*. Sun Microsystems, 1999

[4] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, 1997

[5] B. R. Montague. *JN: An Operating System for an Ebedded Java Network Computer*. Technical Report, Univ. of California, Santa Cruz, 1996

[6] A. Miyoshi and T. Kitayama. Implementation and Evaluation of Real-Time Java Threads. *Proceedings of the 18th IEEE Real-Time Systems Symposium*, 1997

[7] J. Tryggvesson, T. O. Mattsson and H. Heeb. JBED: Java for Real-Time Systems. Report, *Dr.Dobb's Journal*, November 1999

[8] J. S. Young, J. McDonald, etc. Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement. *DAC '98*, San Francisco, CA, 1998

[9] D. Rayside, E. Mamas and E. Hons. Compact Java Binaries for Embedded Systems. *Proceedings of the 9th NRC/IBM Centre for Advanced Studies Conference (CASCON'99)*, Toronto, November 1999

[10] Sun Microsystems. PersonalJava API Specification Ver. 1.2. Sun Microsystems, 2000, see http://java.sun.com/products/personaljava/

[11] Sun Microsystems. EmbeddedJava Application Environment Ver. 1.0.3. Sun Microsystems, 2000, see http://www.sun.com/software/embeddedjava/

[12] Sun Microsystems. Java 2 Platform, Micro Edition. Sun Microsystems, 2000, see http://java.sun.com/j2me/

[13] Sun Microsystem. *The K Virtual Machine (KVM)*. White Paper, 1999

[LIANG99] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Sun Microsystems, 1996

[14] Hewlett-Packard. Chai Technology. Hwelett-Packard, 2000, see http://www.chai.hp.com/

[15] Sun Microsystems. Java Card Techonlogy. Sun Microsystems, 2000, see http://java.sun.com/products/javacard/index.html

[16] J. Ernst, W. Evans, etc. Code Compression. *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, June 1997

[17] R. N. Horspool and J. Corless. Tailored Compression of Java Class Files. *Software – Practive and Experience, 28(12)*, October 1998

[18] Q. Bradley, R. N. Horspool and J. Vitek. JAZZ: An Efficient Compressed Format for Java Archive Files. *Proceedings of CASCON'98*, Toronto, Nov/Dec 1998

[19] W. Pugh. Compressing Java Class Files. *Proceedings of ACM/SIGPLAN Conference on Programming Language Design and Implementation (PLDI) '99*, May 1999

[20] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 1996

[21] F. Tip, C. Laffra and P. F. Sweeney. Practical Experience with an Application Extractor for Java. *OOPSLA '99*, November 1999

[22] D. Rayside and K. Kontogiannis. Extracting Java Library Subsets for Deployment on Embedded Systems. *Proceedings of the 3rd IEEE Conference on Software Maintenance and Re-engineering (CSMR'99)*, Amsterdam, March 1999

[23] D. N. Antonioli and M. Pilz. *Analysis of the Java Class File Format*. Technical Report, University of Zurich, France, 1998