Diplomarbeit

JustIce

A Free Class File Verifier for Java™

Enver Haase

<ehaase@inf.fu-berlin.de>

September 2001

Freie Universität Berlin
Institut für Informatik
Takustraße 9
D-14195 Berlin

Erklärung[1]
Hiermit versichere ich, die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfaßt zu haben. Es wurden nur die in der Bibliographie angegebenen Quellen benutzt.

Danksagung[2]
Während der Anfertigung dieser Diplomarbeit wurde ich von Prof. Dr. Elfriede Fehr und Dipl.-Inform. Markus Dahm betreut, wofür ich mich an dieser Stelle herzlich bedanke.

Desweiteren bedanke ich mich bei Keith Seymour, der mir eine Reihe sprachspezifischer Verbesserungsvorschläge sandte.

Autor[3]
Enver Haase
Gubener Straße 18
D-10243 Berlin
http://www.inf.fu-berlin.de/~ehaase

[1] I declare that I wrote this *Diplomarbeit* completely on my own and without the help of persons not listed. All sources of information are listed in the Bibliography section.

[2] The creation of this *Diplomarbeit* paper was supported and supervised by Prof. Dr. Elfriede Fehr and Dipl.-Inform. Markus Dahm. Keith Seymour suggested a lot of language-related improvements. Thank you.

[3] Author

Contents

Abstract

When Sun Microsystems developed their *Java Platform* in the early 1990s, it was originally designed for use in networked and embedded consumer-electronics applications. But when they introduced it around 1995, it quickly became used in World Wide Web browser software. This was a way to bring interactive content to demanding World Wide Web users. Sun took great care for the robustness of the platform: they planned to connect embedded devices and let them share data and code over a network. Defective devices transmitting bad data or unreliable network connections should not cause other devices to crash. This property made Java a good choice for the code-executing engine in World Wide Web browsers: defective server software or transmission errors would not cause the *Java Platform* to crash; this is also true for purposely malicious code hidden on the Web. The code-executing part of the *Java Platform* is called *The Java Virtual Machine* (the *JVM*, for short). This execution engine has to assure that the code to be executed is well-behaved; it has to *verify* the code. Therefore, the *verifier* is an integral part of every JVM, but JustIce implements a verifier that is not integrated in a JVM. It was implemented using a software library called the *Byte Code Engineering Library* (the *BCEL*, for short) by Markus Dahm [BCEL98, BCEL-WWW].

The BCEL is intended to give users a convenient mechanism to analyze, create and manipulate (binary) Java class files. It offers an object-oriented view of otherwise raw data, including program code. This library is, therefore, well-respected especially in the compiler-writer community whenever the JVM is chosen as the target machine of the compiler. Compiler back-ends use the BCEL to produce code for the JVM; and as new compilers may be faulty, they may produce bad code. Testing these compilers often is a difficult task. The generated code should not only be semantically correct, but it also has to pass the verifiers of all existing JVM implementations. Normally, a lot of human interaction is required to run test cases. If the code is rejected by a verifier, one often does not know why. Most verifiers emit error messages which do not identify the offending instruction.

JustIce presents an Application Programming Interface (API) that may be used to automate the procedure sketched above. The constraints imposed on class files are designed to be strict, therefore eleminating the need to run several verifiers on the generated code. If code passes the JustIce verifier, it should pass all other verifiers. JustIce was also designed to output human-understandable messages if the verification of some code fails.

The application range of JustIce is not limited to compiler back-ends, in the same sense as the BCEL is not only useful in this area. Transformations of

existing code and even generation of hand-crafted code fall into its scope, too. As a side effect, JustIce exports some data structures such as a control flow graph; so its API may also be used for applications targeting other problem areas such as static analyses of program code.

1 Introduction

1.1 Low Level Security as a Part of a Many-Tiered Strategy

The Java programming language is well-known for its inherent security facilities such as the lack of pointer arithmetic or the need for memory allocation and deallocation. Lesser known is that this is only the top of an iceberg; the *Java Platform* implements a many-tiered security strategy [Yellin-WWW]. It was designed to run even untrusted code – code that possibly was not produced by a compiler for the Java programming language, code that may be corrupt or code that may have malicious intent (such as stealing credit card number information from a hard disk drive). Three considerations were made:

- Untrusted code could damage hardware, software, or information on the host machine.
- It could pass unauthorized information to anyone.
- It could cause the host machine to become unusable through resource depletion.

While some security features such as type-safety or the already-mentioned lack of pointer arithmetic of the Java programming language are a convenient help for programmers, they can only help to reduce programming errors. Of course these features do not help targeting the above problems. At a lower level, however, the *Java Platform* implements a so-called sandbox: an area where code can be executed but that has well-defined boundaries shielding the rest of the system. This is achieved by means of a *Java Virtual Machine* (JVM) emulation; the host platform does not directly run untrusted code, but a *run-time system* which in turn runs the code, restricting its access to system resources.

A run-time system cannot safely assume that untrusted code is well-behaved. Code could cause stack overflows, stack underruns, or otherwise erroneous behaviour that may bring the run-time system into an undefined state – possibly allowing access to protected memory areas. One could protect the run-time system by letting it predict the effects of every single instruction just in time while actually executing it – but that would be too time-consuming to be applicable in practice.

Therefore, good behaviour of program code has to be enforced *before* it is actually executed – at least as far as this is possible. This is the lowest level of Java security; there has to be an integral component in every JVM

implementation doing so ([VMSPEC2], page 420). This part of the JVM is called the *class file verifier*, yet better known as the *bytecode verifier*. Technically speaking, bytecode verification is only a part of class file verification so *class file verifier* is a more embracing term. JustIce implements a whole class file verifier.

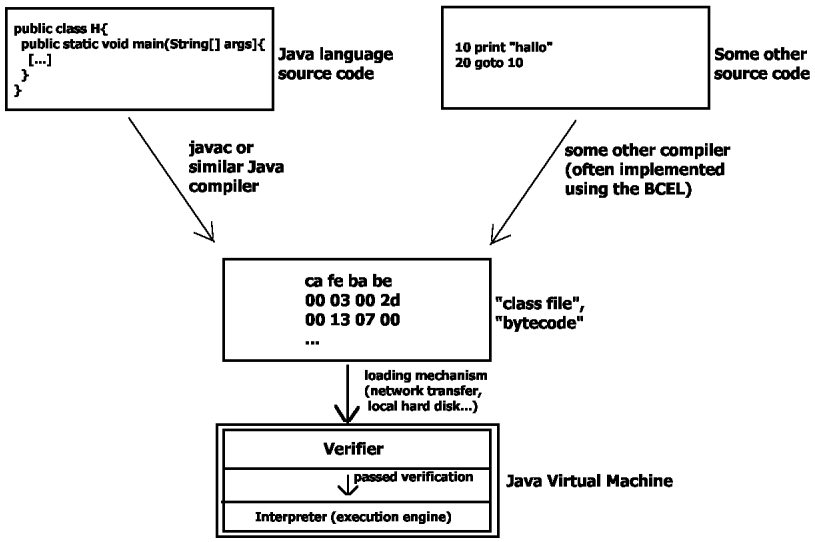


Figure 1.1: Concept of Class File Verification

1.2 Why Another Verifier?

As said before, every JVM implementation must contain a class file verifier, so it is reasonable to ask for the motivation behind creating just another class file verifier – especially one that is *not* part of a JVM implementation.

1.2.1 Bytecode Engineers Need JustIce

Shortly after the *Java Platform* was introduced, it was adopted with pleasure because of its inherent independence from operating systems and concrete hardware. Industry and educational institutions with heterogenous networked computers could now run the same software program on different host machines. Soon, many efforts were put into research and development of compilers for programming languages other than the Java programming language that use the JVM bytecode as target.

Nowadays, many other programming languages do have the JVM as its target platform; e.g. Fortran [f2j], Ada [AppMag-WWW], Scheme [KAWA-WWW] or modified Java language versions [GJ-WWW, PMG-WWW]. A vast collection of programming languages targeting the JVM can be found on the World Wide Web [PL4JVM].

All these compilers emit code for the JVM – and so all these compilers have to pass the JVM's verifier. Implementors of such compilers have to consider the security related constraints the JVM poses on the generated code. It is difficult to test if the emitted code works on all JVM implementations, passing all JVM verifier implementations. This is especially problematic if not all of the project's class files are loaded into the JVM during a test run, because then they will not be verified.

Having an opportunity to verify the transitive hull of referenced class files (starting with some main class file) would be of help; JustIce offers it.

The Bytecode Engineering Library by Markus Dahm is often used as a compiler back-end to emit code, but it is also used to hand-craft code or to implement bytecode transformations. Because JustIce works closely together with the BCEL, users of the BCEL do not even have to leave their development environment to run the JustIce verifier.

To our knowledge, JustIce is the only implementation of a Java class file verifier that was written in the Java programming language [langspec2] itself[1]. Because of its *Verification API*, it can be included in other software projects written in Java with more ease than any other verifier implementation in a different programming language could provide.

1.2.2 JustIce is Verbose

Usually, when classes pass the verifier, it is mute. JustIce, in contrast, distinguishes between verification results and messages. Messages are often warnings, but the reason for emitting such a warning instead of a negative verification result is because the class file does not pose a threat to the integrity of the JVM and thus does not have to be rejected.

When a verification error occurs and the class file is rejected, even the built-in verifiers usually produce some output saying so. As an example, consider the following verifier run:

```
ehaase@haneman:/home/ehaase > java Cc
Exception in thread "main" java.lang.VerifyError:
(class:  Cc, method:  ttt signature:  ()V)
Recursive call to jsr entry
```

One might ask *which* "jsr entry" (a branch target of a `jsr` or a `jsr_w` instruction) is called recursively and which instructions may be responsible for this. Compare this to JustIce's output:

```
[...]
   Pass 3b, method number 0 ['public static void ttt()']:
   VERIFIED_REJECTED
   Constraint violated in method 'public static void ttt()':
```

[1] In a personal communication, Robert Stärk told the author that there was a Java implementation of the verifier discussed in [JBook], written by Joachim Schmid using the BCEL. However, it is not released for public use yet.

```
   Subroutine with local variable '1', JSRs '[ 36:  jsr[168](3) -> astore_1,
8:  jsr[168](3) -> astore_1, 30:  jsr[168](3) -> astore_1, 23:  jsr[168](3)
-> astore_1]', RET ' 62:  ret[169](2) 1' is called by a subroutine which
uses the same local variable index as itself; maybe even a recursive
call?  JustIce's clean definition of a subroutine forbids both.
[...]
   Warnings:
   Pass 2:  Attribute 'LineNumber(0, 4), LineNumber(0, 5), LineNumber(15,
8), LineNumber(39, 11), LineNumber(47, 12), LineNumber(57, 13), LineNumber(64,
15)' as an attribute of Code attribute '<CODE>' (method 'public static
void ttt()') will effectively be ignored and is only useful for debuggers
and such.
   Pass 2:  Attribute 'LineNumber(0, 1), LineNumber(4, 1)' as an attribute
of Code attribute '<CODE>' (method 'public void <init>()') will effectively
be ignored and is only useful for debuggers and such.
   Pass 3a:  LineNumberTable attribute 'LineNumber(0, 4), LineNumber(0,
5), LineNumber(15, 8), LineNumber(39, 11), LineNumber(47, 12), LineNumber(57,
13), LineNumber(64, 15)' refers to the same code offset ('0') more than
once which is violating the semantics [but is sometimes produced by
IBM's 'jikes' compiler].
```

This output obviously has an answer to the above question; it shows the only `jsr` or `jsr_w` instructions possibly responsible for a recursive call (which is not allowed by the specification of the JVM). For the special –but clean– definition of subroutines JustIce uses, please see section 3.3.2.

Note also the warning messages. Class files that were not generated by Sun's *javac* compiler have a tendency to look a little different in some corner cases. IBM's *jikes* compiler, for instance, produces LineNumberTable attributes (see 2.1.1) which look different from those created by *javac*. Detecting such differences is desirable because future JVMs will have stricter verification checks[2] (which most old *javac*-compiled class files will probably still pass). JustIce guides bytecode engineers to create class files that are indistinguishable from those created by *javac* to retain compatibility with Sun's future JVM implementations. Figure 1.2 graphically shows the relationship between class files and the verifier[3].

1.2.3 JustIce is Free

Currently, there is no other free and complete open source verifier available known to the author. You may have a look at the JVM's source code by Sun Microsystems but you are not allowed to use the knowledge from that inspection for your own projects or even use their code. JustIce is a clean-room implemen-

[2] The Solaris port of Sun's JVM, version 1.3.0_01, already has (some of) the stricter checks built in. You may enable them using the command-line option '-Xfuture'. Nothing about this issue is mentioned in the specification [VMSPEC2].

[3] This is a simplicistic figure; unfortunately, there are class files produced by the *javac* compiler that do not pass the verifier. Please see section 7.2.2 for more details.

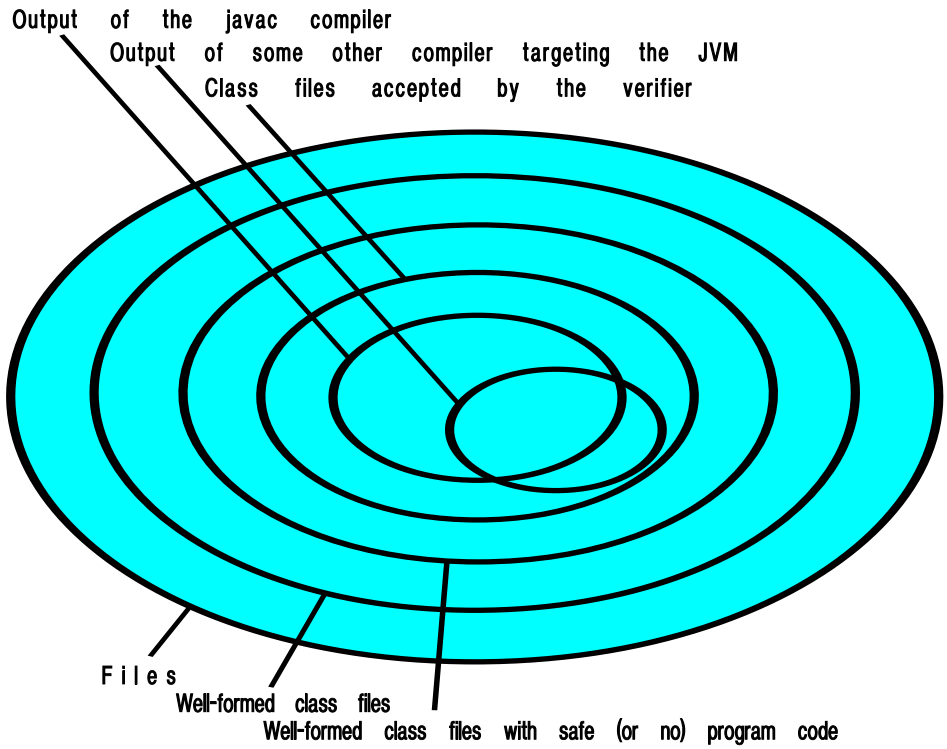


Figure 1.2: Venn diagram showing the operating domain of the Java verifier.

tation: the author wrote JustIce by only reading the Java™ Virtual Machine Specification, Second Edition [VMSPEC2] and comparing the behaviour of JustIce with the behaviour of commercial implementations of Sun Microsystems and IBM Corporation.

The open source JVM implementation *Kaffe* [Kaffe-WWW], for example, does not have a *complete* verifier built in (although mandated by the JVM specification).

Kissme [kissme-WWW], another open source JVM implementation, currently does not include any verifier at all.

The JVM implementations *SableVM* [SableVM-WWW] and Intel Corporation's *Open Runtime Platform* [ORP-WWW] are platforms to experiment with performance-enhancements. They are not intended to work as general-purpose JVMs so they do not need to implement verifiers.

Other open source projects that could make use of a free verifier include the Java compiler *gcj* which is part of the GNU compiler collection [GCC-WWW].

JustIce is covered by the well-known and respected software license *GNU General Public License* (GPL); see section 7.4. The author hopes other free software will benefit from it; from the JustIce software [JustIce] as well as from this paper describing some of the inner workings of JustIce.

2 The Java Virtual Machine

The Java Virtual Machine (JVM) is an abstract machine specified in [VMSPEC2]. It has no knowledge about the Java programming language; but only of a certain binary file format: the class file format. A class file contains machine instructions for the JVM (called *bytecodes*), a symbol table (called *constant pool*) and some other ancillary information.

On method invocation, a local stack frame is set up called the *execution frame*. It consists of an *operand stack* and *local variables* (which may be compared to registers of traditional machines).

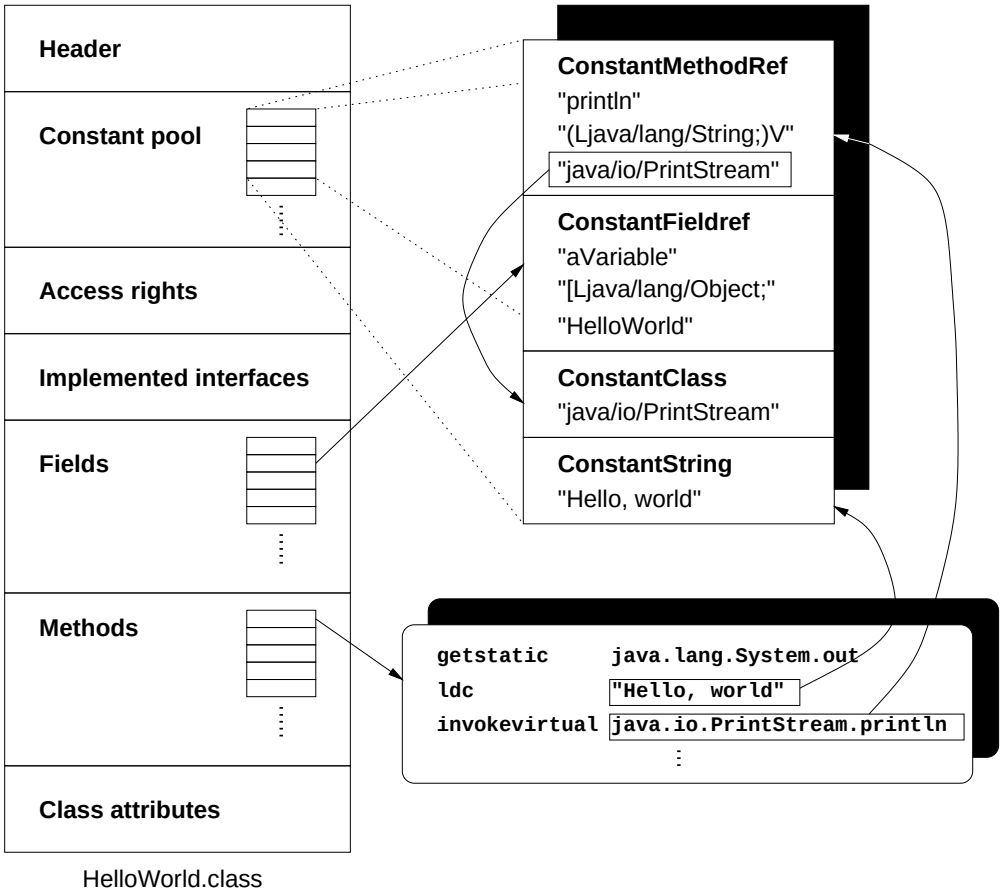
The instructions in the code arrays of class files are interpreted by the JVM. There are 212 legal instructions; they have read-access to the class file's constant pool and they can modify the operand stack and the local variables in their execution frame. An invoked method reads its arguments from the local variables. Certain instructions pass a return value to the invoking method.

2.1 The ClassFile Structure

Traditionally, the JVM loads its programs from files stored on file systems of host machines; these files have names that end with *".class"*. It is possible to store the files in various other ways; a so-called *class loader* is then used to transform the files internally to the desired, basic class file format. Therefore, it suffices to explain the structure of traditional class files. Every class file consists of a single `ClassFile` structure as defined below. It defines a single class as known from the Java Programming Language [langspec2]. The terms *class* and *class file* may therefore be used interchangeably.

As we will see, the `ClassFile` structure and its sub-structures are defined for upwards compatibility, i.e., new structure definitions can be added to the specification easily at a later time.

```
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;
```



A class file consists of constants, fields, methods, attributes and some ancillary information. This figure was taken from [BCEL98], used with permission of the author.

Figure 2.1: A Class File

```
  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];
}
```

You may read an 'u' as 'byte times'; e.g., 'u2' means 'two bytes in size'. We will not delve into too much detail here; the exact specification of the entries are published by Sun [VMSPEC2]. But one should note that besides some other information, a class file basically defines *attributes*, *constants*, *fields* and *methods*. Also, there are strong structural constraints imposed on class files. It is a verifier's task to validate them.

2.1.1 Attributes

The general format of an attribute is defined below.

```
attribute_info {
  u2 attribute_name_index;
  u4 attribute_length;
  u1 info[attribute_length];
}
```

An attribute is basically a typed data container; its type is determined by its name. Every JVM is required to be silent about attributes of types it does not know. On the other hand, newly defined attributes are required not to impose a semantical change on the class file. These attributes should be uniquely named; in fact, the pair (<attribute name>, <attribute length>) is required to be unique. This is guaranteed because attributes not defined by Sun Microsystems have to be named according to the package naming scheme of the Java Programming Language [langspec2]. Certain basic attributes are predefined. They are used in the `ClassFile` (see section 2.1), `field_info` (see section 2.1.3) and `method_info` (see section 2.1.4). Also, attributes may be nested: the `Code` attribute references other attributes.

Some examples for predefined attributes are listed below.

The ConstantValue attribute

The ConstantValue attribute has the following format:

```
ConstantValue_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 constantvalue_index;
```

```
}
```

The `ConstantValue` attribute represents the value of a constant field. It has a fixed length: it contains only a two-byte reference into the constant pool. Only `field_info` structures (see section 2.1.3) contain this type of attribute.

The Code Attribute

The `Code` attribute is used in the `method_info` (see section 2.1.4) structure. It represents the program code of a method and it is defined as follows:

```
Code_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 max_stack;
  u2 max_locals;
  u4 code_length;
  u1 code[code_length];
  u2 exception_table_length;
  {
    u2 start_pc;
    u2 end_pc;
    u2 handler_pc;
    u2 catch_type;
  } exception_table[exception_table_length];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

This is the most complex of all predefined attributes. Every method that has code (i.e., every non-native, non-abstract method) must have such an attribute. Note that the maximum stack depth and the number of local variables for a method invocation are defined here. This is important for the JVM when it creates an *execution frame* (see section 2.2.1) at the time the method is invoked.

Also, the exception handlers are defined here. Exception handlers prevent an executing method from an abrupt completion if an exceptional situation occurs. Code areas are said to be protected against a class of exceptional situations by an exception handler[1]. Algorithm 1 shows an example for the use of exception handlers. The exact meaning of the instruction opcodes is not important here, the most common instructions are explained later in this paper.

The most important item, however, is the `code` item. It defines the bytecode of this method; i.e., the JVM machine instructions.

[1] The JVM closely reflects the *exception* mechanism of the Java programming language [langspec2]. In the Java programming language, exceptions can be *thrown*, and they can be *caught* explicitly. If an internal JVM error occurs, the JVM also –implicitly– throws an exception.

Algorithm 1 Use of Exception Handlers

[Let `start_pc` and `end_pc` protect the area A to B, inclusive. Let the `catch_type` be "`java.lang.NullPointerException`". Let the `handler_pc` point to C.]

```
   aconst_null        ; push a NULL onto the operand stack.
A: nop                ; do nothing
B: getfield Foo::bar  ; dereference NULL, cause NullPointerExc.
   return             ; never executed
C: nop                ; this is executed:  we could handle
   nop                ; the NullPointerException
   return             ; leave method (complete normally)
```

The LineNumberTable Attribute

The `LineNumberTable` attribute is defined as follows:

```
LineNumberTable_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 line_number_table_length;
  {
    u2 start_pc;
    u2 line_number;
  } line_number_table[line_number_table_length];
}
```

This attribute describes the relation between source code line numbers and JVM instruction offsets in the `code` array of the `Code_attribute`; it can be used by debuggers to show the source code of currently executing JVM machine instructions. This attribute is usually a sub-attribute of a `Code_attribute`. Multiple `LineNumberTable` attributes may together represent a given line of a source code file.

2.1.2 Constants

All the constants together form the *constant pool*. The general `cp_info` structure is straightforward.

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

The 'tag' defines what 'info' follows it. Constants define either constant values or constant symbolic references, such as references to other classes. Currently, eleven constant types are defined: `Class`, `Fieldref`, `Methodref`, `Interface-Methodref`, `String`, `Integer`, `Float`, `Long`, `Double`, `NameAndType` and `Utf8`.

Most of the names are self-explanatory; the interested reader will find more information in the specification [VMSPEC2]. Constants can be nested; this is done by referring to the constant pool index of the enclosed constant.

See the following examples.

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

A CONSTANT_Utf8 represents a constant string. Such a string is e.g. used to describe names of methods, names of fields, names of attributes, types of methods or types of fields. This string is encoded in UTF-8 format, a variant of the unicode character set [Unicode]. The tag for this type of constant is simply the number 1, as defined in the Java Virtual Machine Specification, Second Edition [VMSPEC2].

```
CONSTANT_NameAndType_info {
  u1 tag;
  u2 name_index;
  u2 descriptor_index;
}
```

A Constant_NameAndType represents a name and a signature of a method, the tag is the number 12. Both `class_index` and `descriptor_index` refer to a `CONSTANT_Utf8`.

```
CONSTANT_InterfaceMethodref_info {
  u1 tag;
  u2 class_index;
  u2 name_and_type_index;
}
```

A `CONSTANT_InterfaceMethodref` describes a reference to a method defined in an interface class (see section [langspec2] for an explanation of interfaces), the tag is number 11. The interface class is referenced via a two-byte index into the constant pool. A `Constant_Class` is expected there describing a reference to some class file. Every method has a name, zero or more argument types and a return type; this is described in the `CONSTANT_NameAndType` that is also referenced via a two-byte constant pool index.

Note that there are implicit constraints on the integrity of a class file: for example, there must not be a `CONSTANT_Integer` where a `CONSTANT_Utf8` is expected for a certain entity. As another example, the names and the types of methods are encoded as strings in UTF-8 format [Unicode]. They have to be well-formed (according to the specification) to be valid.

2.1.3 Fields

Each field is described by a field_info structure as defined below.

```
field_info {
  u2 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

A field has to be unique in a class file with respect to its name and descriptor[2]. We see that fields reference constants in the constant pool via their constant pool indices (such as a `CONSTANT_Utf8` describing a field's name). An important attribute used by fields is the ConstantValue attribute (see section 2.1.1).

The `access_flags` entry is a bit vector that specifies the accessibility and other properties[3] of the field. E.g., a field with the `ACC_PRIVATE`[4] bit set is not accessible to other classes. A field with the `ACC_PUBLIC`[5] bit set is accessible to any other class. Any combination with both the `ACC_PRIVATE` and the `ACC_PUBLIC` bit set is not valid.

The `descriptor_index` refers to a `CONSTANT_Utf8` that symbolically encodes the type of the field.

2.1.4 Methods

Each method is described by a method_info structure as defined below.

```
method_info {
  u2 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

As we can easily see, this is exactly the same structure we already know as `field_info` (see section 2.1.3). The difference lies in the meaning of the enlisted entities. For example, an access flag saying a field was volatile (non-cacheable) would not make any sense if set in a `method_info` structure. Vice versa, an access flag saying the floating point instructions should work in "FP-strict" mode would be of no use if set in a `field_info` structure.

[2] The descriptor of a field describes its type. E.g., a descriptor of "[I" means "one-dimensional array of `int`".
[3] Often called *visibility*.
[4] Bit number 1.
[5] Bit number 0.

Methods use a different set of attributes than fields; for example, the `ConstantValue` attribute (see section 2.1.1) is of no use here. The `Code` and `Exceptions` attributes frequently used by methods are of no use for fields on the other hand.

2.2 The Execution Engine

Before a piece of code (the code of a "method") is executed, an *execution frame* is set up. It consists of a program counter (as known from traditional CPUs), a set of local variables (similar to registers known from traditional CPUs), and an operand stack. For each new invocation instance of a method, a new execution frame is set up; it is destroyed on method termination.

Because a method may invoke other methods or itself recursively, there is a global method invocation stack.

There also is a garbage-collected heap shared among the execution frames. This heap is used for object allocation (see section 2.2.2).

The number of local variables is not fixed. Every method defines how many local variables are used for its code (up to 65536).

Also note that there is no equivalent of a *Processor Status Word* (PSW) in the JVM. Traditionally, a PSW has flags that are set implicitly during execution of the instructions (such as an overflow or is-zero flag). This is often used for conditional branching. The JVM, however, uses the operand stack to store the result of a comparison instruction explicitly. This result is often read from the stack by the JVM's conditional branching instructions.

Should exceptional situations occur (such as an out-of-memory situation), the JVM does not lock up. Instead, an "exception is thrown"; the currently executing program is signalled. These signals can be processed ("exceptions can be caught"). If such a signal is not handled by the currently executing method, the JVM will search a handler through the invocation hierarchy and stop execution only if none was found.

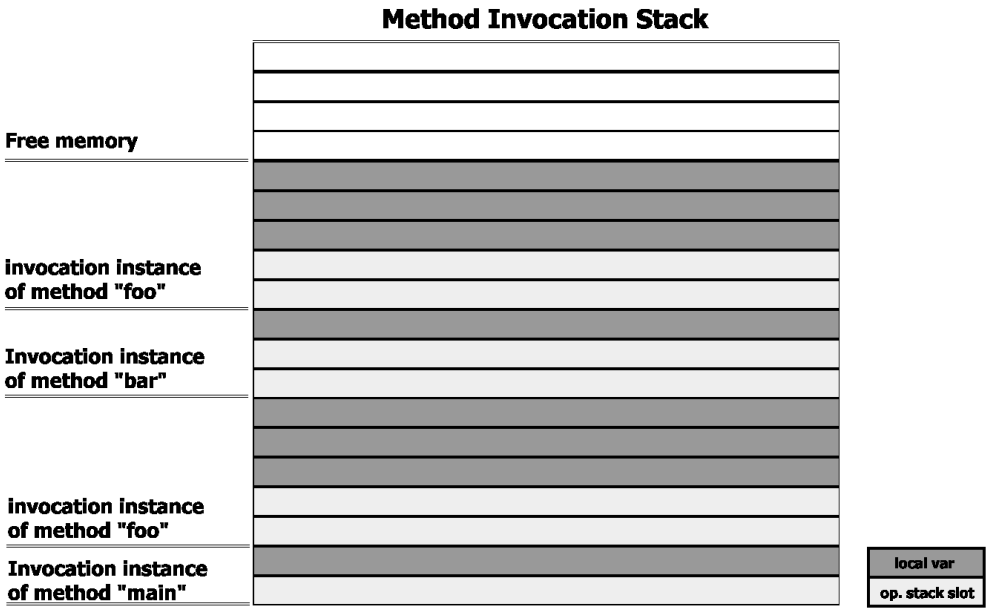
There is a thread mechanism in the JVM. Basically every thread creates an own method invocation stack (so there may be more than one active execution frame at a time), but this feature is not important for the rest of this text.

2.2.1 Local Variables and the Operand Stack

The method information in a class file defines how many local variables are used on this method's invocation. It also defines the maximum operand stack size. Together, the local variables array and the operand stack are called the *execution frame*.

A single stack slot has a width of 32 bits, which is also the width of a local variable. Therefore, values of types that occupy 64 bits (*double* and *long*) must be stored in two consecutive stack slots or local variables.

The verifier takes care that the stack cannot overflow and that it cannot underflow. Also, it takes care that instructions may only access local variables if they contain a value of a known, correct type (see section 3.3).



This figure shows a method invocation stack. Method `main` was invoked by the system, `main` invoked `foo`, `foo` invoked `bar`, and `bar` invoked `foo` recursively. This figure assumes `main` allocates one local variable and one operand stack slot, `foo` allocates three local variables and two operand stack slots and `bar` allocates one local variable and two operand stack slots.

Figure 2.2: Method Invocation Stack

2.2.2 Introduction to JVM Instructions

This section is derived from section 2.2 of [BCEL98], used with permission of the author.

The JVM's instruction set currently consists of 212 instructions, 44 opcodes are marked as reserved and may be used for future extensions or intermediate optimizations within the Virtual Machine. The instruction set can be roughly grouped as follows:

Stack operations: Constants can be pushed onto the stack either by loading them from the constant pool with the `ldc` instruction or with special "short-cut" instructions where the operand is encoded into the instructions, e.g., `iconst_0` or `bipush` (push byte value).

Arithmetic operations: The instruction set of the JVM distinguishes its operand types using different instructions to operate on values of specific type. Arithmetic operations starting with `i`, for example, denote an integer operation. E.g., `iadd` that adds two integers and pushes the result back on the operand stack. The Java types `boolean`, `byte`, `short`, and `char` are handled as integers by the JVM.

Control flow: There are branch instructions like `goto` and `if_icmpeq`, which compares two integers for equality. There is also a `jsr`[6] (jump into subroutine) and `ret` (return from subroutine) pair of instructions. Exceptions may be thrown with the `athrow` instruction. Branch targets are coded as offsets from the current byte code position, i.e., they are coded with an integer number.

Load and store operations for local variables like `iload` and `istore`. There are also array operations like `iastore` which stores an integer value into an array.

Field access: The value of an instance field may be retrieved with `getfield` and written with `putfield`. For static fields, there are `getstatic` and `putstatic` counterparts.

Method invocation: Methods may either be called via static references with `invokestatic` or be bound virtually with the `invokevirtual` instruction. Super class methods and private methods are invoked with `invokespecial`.

Object allocation: Class instances are allocated with the `new` instruction, arrays of basic type like `int[]` with `newarray`, arrays of references like `String[][]` with `anewarray` or `multianewarray`.

Conversion and type checking: For stack operands of basic type there exist casting operations like `f2i` which converts a float value into an integer. The validity of a type cast may be checked with `checkcast` and

[6] There is a "wide" version of `jsr` called `jsr_w`. The instructions `jsr`/`jsr_w` and `ret` play in important role in chapter 3.3.

the `instanceof` operator can be directly mapped to the equally named instruction.

Most instructions have a fixed length, but there are also some variable-length instructions: In particular, the `lookupswitch` and `tableswitch` instructions, which are often used by compilers to implement the Java language `switch()` statements. Since the number of `case` clauses may vary, these instructions contain a variable number of statements.

In a class file, the `code` item in the `Code` attributes (which in turn are attributes of `method_info` structures), is a byte array in which binary representations of JVM instructions are stored sequentially. This is also called *bytecode*.

The JVM is a stack-based machine. There are local variables which may be compared to registers, but most instructions work on the operand stack. E.g., the `iadd` instruction pops two integers from the operand stack and pushes the result of the add operation on top of the stack.

We will not list all of the instructions here, since these are explained in detail in the JVM specification. However, you will find the most common instructions in table 2.1, cited with slight corrections and modifications from chapter 4 of [JNS].

Table 2.1: Type Prefixes and the Most Common JVM Instructions

Prefix	Bytecode type
i	Integer
f	Floating point
l	Long
d	Double precision floating point
b	Byte
s	Short
c	Character
a	Object reference

Instruction	int	long	float	double	byte	char	short	object ref.	Function
?2c	X								Convert value of type <?> to character
?2d	X	X	X						Convert value of type <?> to double
?2i		X	X	X					Convert value of type <?> to integer
?2f	X	X		X					Convert value of type <?> to float
?2l	X		X	X					Convert value of type <?> to long
?2s	X								Convert value of type <?> to short

Instruction	int	long	float	double	byte	char	short	object ref.	Function
?add	X	X	X	X					Add two values of type <?>
?aload	X	X	X	X	X	X	X	X	Push an element of type <?> from an array onto the stack
?and	X	X							Perform logical AND on two values of type <?>
?astore	X	X	X	X	X	X	X	X	Pop an element of type <?> from the stack and store it in an array of type <?>
?cmp		X							Compare two long values. If they are equal push 0, if the first is greater push 1, else push -1
?cmpg			X	X					Compare two IEEE values of type <?> from the stack. If they are equal push 0, if the first is greater push 1, if the second is greater push -1. If either is NaN (not a number) push 1
?cmpl			X	X					Compare two IEEE values of type <?> from the stack. If they are equal push 0, if the first is greater push 1, if the second is greater push -1. If either is NaN (not a number) push -1
?const	X	X	X	X				X	Push a constant value of type <?> onto the stack
?div	X	X	X	X					Perform a division using two values of type <?> and push the quotient onto the stack
?inc	X								Increment the top of the stack (possibly by a negative value)
?ipush					X		X		Push a sign extended byte or short value onto the stack

Instruction	int	long	float	double	byte	char	short	object ref.	Function
?load	X	X	X	X					Push a value of type <?> from a local variable onto the stack
?mul	X	X	X	X					Perform multiplication of two values of type <?>
?neg	X	X	X	X					Negate a value of type <?>
?newarray								X	Create a new array of object references
?or	X	X							Perform logical OR on two values of type <?>
?rem	X	X	X	X					Perform a division using two values of type <?> and push the remainder onto the stack
?return	X	X	X	X				X	Return a value of type <?> to the invoking method
?shl	X	X							Perform arithmetic shift left on a value of type <?>
?shr	X	X							Perform arithmetic shift right on a value of type <?>
?store	X	X	X	X				X	Pop a value of type <?> and store it into a local variable
?sub	X	X	X	X					Perform a subtraction using two values of type <?>

The opcode names are mostly self-explanatory. In this paper, all bytecode is commented to support the intuitive understanding. Algorithms 2 and 3 show an example bytecode taken from [BCEL98]. It implements the well-known faculty function. To understand this example, it is important to know that method arguments are stored into the local variables of a newly created execution frame upon method invocation.

Algorithm 2 Methed *fac* in a class *Faculty*, Java programming language version

```
public static final int fac(int n){
  return (n==0)?1:n*fac(n-1);
}
```

Algorithm 3 Method *fac* in a class *Faculty*, Java bytecode version

```
Faculty.fac (I)I
0:  iload_0                          ; load argument onto stack
1:  ifne #8                          ; non-zero?  Then branch to 8.
4:  iconst_1                         ; push constant 1 onto stack
5:  goto #16                         ; jump to 16
8:  iload_0                          ; load argument onto stack
9:  iload_0                          ; load argument onto stack
10: iconst_1                         ; push constant 1 onto stack
11: isub                             ; subtract the stack top from
                                     ; the stack next-to-top which becomes
                                     ; the new stack top
12: invokestatic Faculty.fac (I)I    ; call method fac recursively,
                                     ; the new invocation
                                     ; instance's argument is the stack top
15: imul                             ; multiply the return value with the
                                     ; argument given to the current
                                     ; invocation instance
16: ireturn                          ; return value on top of the
                                     ; stack to the invoking method
```

3 Specification of the Verification Passes

Sun describes a four-pass class file verifier in The Java Virtual Machine Specification, Second Edition [VMSPEC2]. It is not necessary to implement the verification algorithms literally; and it is not possible anyway (see section 3.3.2). However, implementing a verifier with a multiple-pass architecture makes sense. It is a good thing to stay close to the specification because it is well-known throughout the bytecode engineering community. Also, the boundaries between the passes are not arbitrary. They are drawn to improve the performance of the verifiers built into JVMs. For example, classes are not verified (completely) before they are actually used but they are loaded as soon as they are referenced in a certain way. Most verifiers use the traditional multiple-pass architecture, including Kimera [Kimera-WWW]. Work in other directions (for instance, the one-pass-architecture proposed by Fong [Fong-WWW]) did not yield lasting results.

Pass one is basically about loading a class file into the JVM in a sane way and pass two verifies that the loaded class file information is consistent. Pass three verifies that the program code is well-behaved; pass four verifies things that conceptually belong to pass three but are delayed to the run-time for performance reasons.

Sometimes implementation details are discussed in this chapter. Whenever the specification [VMSPEC2] was ambigous about some issue, the behaviour of Sun's JVM implementations was observed. The discussed details are part of the specification of the JustIce verifier.

3.1 Pass One

The first pass of the verifier is only vaguely specified. It is there to assure a class file "**has the basic format of a class file. The first four bytes must contain the right magic number. All recognized attributes must be of the proper length. The class file must not be truncated or have any extra bytes at the end. The constant pool must not contain any superficially unrecognizable information**" ([VMSPEC2], page 141).

The right magic number is 0xCAFEBABE ([VMSPEC2], page 94), which is easy to assure.

It is not clear what "superficially unrecognizable information" exactly is, however. If an attribute is not known to the JVM (or verifier) implementation, it has to be ignored – so this does not seem to be "superficially unrecognizable

information". Attributes that are not used cannot be detected in pass one. One would have to look at the bytecodes to decide whether an attribute is used or not (which is not the domain of pass one, but of pass three).

Observations show that most existing JVM verifiers[1] ignore "extra bytes at the end" instead of rejecting class files bearing them.

The other two statements specify verification of the class file structure (and the structure of the attributes therein). But this is also the domain of pass two! Only by inspecting the way the JVM *loads*, *resolves* and *prepares* classes one will understand the precise boundary between verification passes one and two [Fong-WWW].

'Being careful when loading a class file' is a good definition for pass one: the structure of the file to load is untrusted. Every implicit statement such as "this attribute has a length of 1234 bytes in total" is validated.

Resolution is the transformation of a symbolic reference to an actual reference – i.e., as long as there is only a symbolic reference to an entity, this entity cannot be verified at all because it has not been loaded yet. Passes two and three are performed during the *resolution* of a class file; while loading of the class file –pass one– must have been performed before. *Resolution* as such is meaningless to JustIce; the term is only used to draw the borders between the verification passes.

3.2 Pass Two

The checks performed in pass two enforce that the following constraints are satisfied.

- Ensuring that final classes are not subclassed and that final methods are not overridden.
- Checking that every class (except `java.lang.Object`) has a direct superclass.
- Ensuring that the constant pool satisfies the documented static constraints: for example, that each `CONSTANT_Class_info` structure in the constant pool contains in its `name_index` item a valid constant pool index for a `CONSTANT_Utf8_info` structure.
- Checking that all field references and method references in the constant pool have valid names, valid classes, and a valid type descriptor.

As Frank Yellin puts it [Yellin-WWW]: pass two "performs all verification that can be performed without looking at the bytecodes". Also, "this pass does not actually check to make sure that the given field or method really exists in the given class; nor does it check that the type signatures given refer to real classes." Note that again *resolution* plays an important role to create the

[1] An example of a verifier with this behaviour is the one implemented in Sun's Solaris port of the JVM, version 1.3.0_01.

boundary between two passes; here it is the boundary between pass two and pass three. Because linking-time verification enhances the performance of the JVM, checks that basically belong to pass two are delayed to pass three. This leads to the obvious contradiction in the sentences cited above.

This performance enhancement has an ugly side effect. Consider a reference to a method m contained in a class file C that does not exist. As long as this reference is not *used*, i.e., *resolved*, the absence of C cannot be detected. Such a reference should in the author's opinion regarded as "superficially unrecognizable information" (see section 3.1) and therefore be detected.

This pass has to verify the integrity of the clas file's data structures as explained in section 2.1. As an example, consider the LineNumberTable atribute. Sun did not specify there has to be exactly one `LineNumberTable` attribute (or none at all) per method, so possibly there is more than one attribute of that kind. This lax specification is not necessary due to the fact that you can put all information in a single `LineNumberTable_attribute`[2], but Sun did specify it this way ([VMSPEC2], page 129).

Verifiers are requested to reject class files with inconsistent information in their attributes. However, here it may be that only by looking at all `LineNumberTable_attribute`s of a method, an inconsistency can be detected. JustIce does so and rejects class files with inconsistent `LineNumberTable` information.

Furthermore, it issues warnings if such an attribute is detected at all to discourage its use (see section 4.2). This is done because of possible different interpretations of the specification.

It should be noted that the use of attributes raises a few more problems to class file verification. A simple case is the presence of an unknown attribute that may safely be ignored. It is explicitly stated that such a class file must not be rejected. On the other hand, how should a verifier react if –for example– a `field_info` (see section 2.1.3) structure encloses a `Code_attribute`? JustIce will issue a warning but not reject the class file.

3.3 Pass Three

Performing pass three basically means *verifying the bytecode*. There are so-called "static constraints" on both the instructions in the code array and their operands. There are also so-called "structural constraints". The structural constraints specify constraints on relationships between JVM instructions, so some people (including the author) regard "structural constraints" as a misnomer; they should be called "dynamic constraints".

Static constraints are easily enforced using very simple checks. Here is an example for such a check: let there be a `Code` (see section 2.1.1) attribute with a `max_locals` value of 2. Only local variables number 0 and 1 may be accessed by the bytecode in this `Code` attribute. For all instructions accessing local variables, make sure they do not access any other local variable.

[2] Any number of `line_number_table` array entries fits nicely in a single `LineNumberTable_attribute` attribute.

Structural constraints are enforced using an algorithm sketched by Sun; it implements a symbolic execution of a method's code, by means of data flow analysis including type inference ([VMSPEC2], pages 143-151). This algorithm is called the *data flow analyzer*. It is intuitively easy to understand, but it is hard to prove its correctness. The reason for that is the very weak specification of its subtleties; especially *subroutines*, *wide date types* and *object initialization* (see below). The general approach, however, is sound [BCV-Soundness]. Here is an example for a structural constraint enforced by this algorithm: during program execution, at any given point in the program the operand stack is always of the same height, no matter which code path was taken to reach that point.

Pass three is the core of the verifier. Note that we will split this pass up into two passes, namely a pass verifying the static constraints and a pass verifying the structural constraints of a method's code. We will call these passes "pass 3a" and "pass 3b". In a way, they resemble pass one and pass two: the former pass carefully parses an entity, while the latter pass performs additional verification.

By defining pass four, the specification [VMSPEC2] implicitly excludes "certain tests that could in principle be performed in Pass 3", because they are "delayed until the first time the code for the method is actually invoked". On the other hand, verifiers are allowed to perform pass four partially or completely as a part of pass three. JustIce performs the pass four checks in pass 3a.

3.3.1 Static Constraints: Pass 3a

Sun gives examples of what the verifier does before starting the data flow analyzer ([VMSPEC2], pages 143-144):

- **Branches must be within the bounds of the code array for the method.**
- **The targets of all control-flow instructions are each the start of an instruction. In the case of a `wide` instruction the `wide` opcode is considered the start of the instruction, and the opcode giving the operation modified by that `wide` instruction is not considered to start an instruction. Branches into the middle of an instruction are disallowed.**
- **No instruction can access or modify a local variable at an index greater than or equal to the number of local variables that its method indicates it allocates.**
- **All references to the constant pool must be an entry of the appropriate type. For example: the instruction `ldc` can be used only for data of type int or float or for instances of class String; the instruction `getfield` must reference a field.**
- **The code does not end in the middle of an instruction.**
- **Execution cannot fall off the end of the code.**
- **For each exception handler, the starting and ending point of the code protected by the handler must be at**

> **the beginning of an instruction or, in the case of the ending point, immediately past the end of the code. The starting point must be before the ending point. The exception handler code must start at a valid instruction, and it may not start at an opcode being modified by the `wide` instruction.**

Most of these constraints are either static constraints on instructions or on their operands. A full list of constraints can be found in the Java Virtual Machine Specification, Second Edition ([VMSPEC2], pages 133-137).

The check for execution falling off the end of the code is an exception: this is a structural constraint and should therefore be performed in pass 3b. Sun's verifiers, however, reject code that has an unreachable `nop` at the end of the code array. Obviously, they reject the code before performing data flow analysis. For the sake of compatibility, JustIce performs this check in pass 3a.

Note that the JVM's instructions differ in length. Some instructions occupy only one byte (such as `nop`), others occupy three bytes (such as `goto`). Branch instructions could therefore target operands of instructions. For example, line 1 of algorithm 3 reads "`1: ifne #8`". If it would read "`1: ifne #7`", this code was malformed. A special case is the instruction `wide`. This instruction takes another instruction *as its operand*, so one could be misguided into thinking this embedded instruction was a valid target for branches. It is not.

The checks Sun delays until pass four are performed in pass 3a by JustIce. These are checks to ensure allowed and possible access to a referenced type, listed below.

- Is the type (class or interface) currently under examination allowed to reference the type[3]?
- Does the referenced method or field exist in the given class?
- Does the referenced method or field have the indicated descriptor (signature)?
- Does the method currently under examination have access to the referenced method or field?

3.3.2 Structural Constraints: Pass 3b

The structural constraints of JVM instructions are enforced by a data flow analyzer. This algorithm ensures the following constraints ([VMSPEC2], page 142).

> - **The operand stack is always the same size and contains the same types of values.**
> - **No local variable is accessed unless it is known to contain a value of an appropriate type.**
> - **ethods are invoked with the appropriate arguments.**

[3]Interfaces may contain code, this is normally used for static initialization of `final` variables.

- **Fields are assigned only using values of appropriate types.**
- **All opcodes have appropriate type arguments on the operand stack and in the local variable array.**

A full list of structural constraints can be found in The Java Virtual Machine Specification, Second Edition ([VMSPEC2], pages 137-139).

Sun's Verification Algorithm

Sun specifies the data flow analyzer by giving an informal algorithm ([VMSPEC2], pages 144-146). This algorithm it cited here completely because it is the very core of the verifier. According to this algorithm, every bytecode instruction has a "changed" bit. Initially, only the "changed" bit of the first instruction is set.

1. **Select a virtual machine instruction whose "changed" bit is set. If no instruction remains whose "changed" bit is set, the method has successfully been verified. Otherwise, turn off the "changed" bit of the selected instruction.**
2. **odel the effect of the instruction on the operand stack and local variable array by doing the following:**
   • If the instruction uses values from the operand stack, ensure that there are a sufficient number of values on the stack and that the top values on the stack are of an appropriate type. Otherwise, verification fails.
   • If the instruction uses a local variable, ensure that the specified local variable contains a value of the appropriate type. Otherwise, verification fails.
   • If the instruction pushes values onto the operand stack, ensure that there is sufficient room on the operand stack for the new values. Add the indicated types to the top of the modeled operand stack.
   • If the instruction modifies a local variable, record that the local variable now contains the new type.
3. **Determine the instructions that can follow the current instruction. Successor instructions can be one of the following:**
   • The next instruction, if the current instruction is not an unconditional control transfer instruction (for instance goto, return, or athrow). Verification fails if it is possible to "fall off" the last instruction of the method.
   • The target(s) of a conditional or unconditional branch or switch.
   • Any exception handlers for this instruction.
4. **erge the state of the operand stack and local variable array at the end of the execution of the current**

instruction into each of the successor instructions. In the special case of control transfer to an exception handler, the operand stack is set to contain a single object of the exception type indicated by the exception handler information.
• If this is the first time the successor instruction has been visited, record that the operand stack and local variable values calculated in steps 2 and 3 are the state of the operand stack and local variable array prior to executing the successor instruction. Set the "changed" bit for the successor instruction.
• If the successor instruction has been seen before, merge the operand stack and local variable values calculated in steps 2 and 3 into the values already there. Set the "changed" bit if there is any modification to the values.

5. **Continue at step 1.**

To merge two operand stacks, the number of values on each stack must be identical. The types of values on the stacks must also be identical, except that differently typed reference values may appear at corresponding places on the two stacks. In this case, the merged operand stack contains a reference to an instance of the first common superclass of the two types. Such a reference type always exists because the type Object is a superclass of all class and interface types. If the operand stacks cannot be merged, verification of the method fails.
To merge two local variable array states, corresponding pairs of local variables are compared. If the two types are not identical, then unless both contain reference values, the verifier records that the local variable contains an unusable value. If both of the pair of local variables contain reference values, the merged state contains a reference to an instance of the first common superclass of the two types.

Certain instructions and data types complicate the data flow analyzer, most notably the instruction `ret` (see section 2.2.2). The algorithm above even uses a special definition of *merging* for the `ret` instruction (see [VMSPEC2], page 151). The `ret` instruction is parameterized with a value of type `returnaddress` which is read from a local variable and used as a branching target. The `ret` instruction is there to implement a (control flow) return from a *subroutine*.

Reachability of Instructions

For the data flow analysis algorithm, you need to know all the possible control flow successors of every instruction, i.e., you need to build a *control flow graph*

(see below). Without the instructions `jsr`[4], `jsr_w` and `ret` this calculation would be easy. But to calculate successors of a `ret` instruction, you need a complete control flow graph: you need to find out which `jsr` or `jsr_w` and `ret` pairs belong together. Therefore, a cycle of self-dependency is created that has to be broken somewhere. This is explained in detail below.

This was also an issue that led to the definition of the term *subroutine* that JustIce uses. This definition allows the prediction of a `ret` instruction's target without performing control flow analysis.

Subroutines

Subroutines make the verification algorithm extremely difficult. They are harshly underspecified. Although "the Java virtual machine has no guarantee that any file it is asked to load was generated by that compiler", the subroutine specification explains how *javac* transforms "`try/catch/finally`" clauses into subroutines [VMSPEC2]. Intuitively, one gets the idea that a subroutine starts with some jump target of a `jsr` or `jsr_w` instruction and ends with a `ret` instruction. But the specification fails to correctly specify what subroutines exactly are at machine instruction level. Consider algorithm 4.

Algorithm 4 Is This a Subroutine?

```
00 jsr 03    ; Jump to "subroutine" at offset 03; push return
             ; address 03 onto stack.
03 pop       ; Pop the return address off the stack.
04 nop       ; No operation.
```

What is this? Is the *NOP* instruction part of a subroutine or not? Algorithm 5 shows another example.

Do we deal with one subroutine (which is the case if you define subroutines to start with a `jsr` or `jsr_w`'s target) or are these two subroutines (which is the case if you count the `ret` instructions and believe that there must be exactly one `ret` per subroutine)?

Recursive calls to subroutines are forbidden by the specification; however, Sun's verifier implementations are not consequently deciding which recursive calls to reject[5]. This is a failure due to a missing definition of the term *subroutine*.

While the first example passes Sun's verifier, the second example is rejected. The exact definition of the term *subroutine* cannot be deducted from ther behaviour of Sun's verifier.

A new, clean specification had to be defined. Such a specification can of course not be compatible with the behaviour of Sun's verifier in all corner cases.

[4]Remember, a `jsr` or `jsr_w` instruction is an unconditional branch instruction that jumps into a *subroutine*. Usually a `ret` instruction leaves the *subroutine*.

[5]This was experimentally found by the author and also published in [JBook].

Algorithm 5 One or Two Subroutines?

```
00 iload_0  ; Load a numerical 0 onto the stack.
01 jsr 05   ; Jump to "subroutine" at offset 05; push return
            ; address 04 onto stack.
04 return   ; Leave the method.
05 dup      ; Duplicate the stack's top.
06 astore 0 ; Store the return address from the stack into
            ; local variable 0.
07 astore 1 ; Store the return address from the stack into
            ; local variable 1.
08 ifeq 12  ; If there is a 0 on top of the stack, jump to
            ; offset 12.
11 ret 0    ; Return to offset 4 (because this is in local
            ; variable 0 here).
12 nop      ; No operation.
13 ret 1    ; Return to offset 4 (because this is in local
            ; variable 1 here).
```

A Precise Definition of the Term *Subroutine*

Because Sun –inappropriately– describes how *javac* creates subroutines, the definition presented here is based on the observation of *javac*'s behaviour. This makes the definition compatible with a lot of existing code, but without violating the validity of far-reaching conclusions earned by exploiting a clean definition[6].

- Every instruction of a method is part of exactly one subroutine (or the top-level).
- The first instruction of a subroutine is an `astore N` instruction that stores the return address in local variable number N.
- There must be exactly one `ret` instruction per subroutine. This instruction must work on the local variable N; i.e., it is a `ret N` instruction.
- Subroutines are not protected by exception handlers.
- No instruction that is part of a subroutine is the target of an exception handler.
- Subroutines of a subroutine do not access local variable N. A subsubroutine of a subroutine is also considered a subroutine here, in a recursive sense.

As we can see, a subroutine can be characterized by its set of instructions, the most important instruction being the target of some `jsr` or `jsr_w` instruction that is not part of the subroutine itself. Another important property is the local variable N the `ret` instruction is working on.

[6] Unfortunately, in some rare cases, *javac* produces code that is incompatible with the constraints related to our definition of *subroutine*. However, *javac* also produces code which is incompatible with Sun's verifier (see section 7.2.2).

This way, we can make sure subroutines are properly nested, so that JustIce would reject both the example bytecodes in algorithms 4 and 5.

The `astore` instruction mentioned above is so important because there is no JVM instruction that can read values of a `returnaddress` type from local variables. After entering a subroutine, the `astore` instruction pops the return address off the operand stack and writes it into local variable number N. Therefore we can be sure it will not be duplicated or deleted as in algorithms 4 and 5.

The constraints concerning exception handlers are defined to make sure that we can observe the control flow statically. If an exception is thrown from within a subroutine, the method simply "*completes abruptly*" ([VMSPEC2], page 74). If we would allow subroutine instructions to be protected by exception handlers, it would not be clear if the handling instructions are part of the subroutine or not.

We can also derive subsubroutines of subroutines recursively by exploiting the properly-nested property explained above.

The Control Flow Graph

A control flow graph is a directed graph with edges that represent possible branches of control flow. Similarly, the nodes describe groups of physically adjacent instructions that have to be executed one after another – without any possible control flow branch to another instruction but the physical successor[7]. Figure 3.1 shows such a control flow graph for algorithm 3, the implementation of the faculty function discussed earlier.

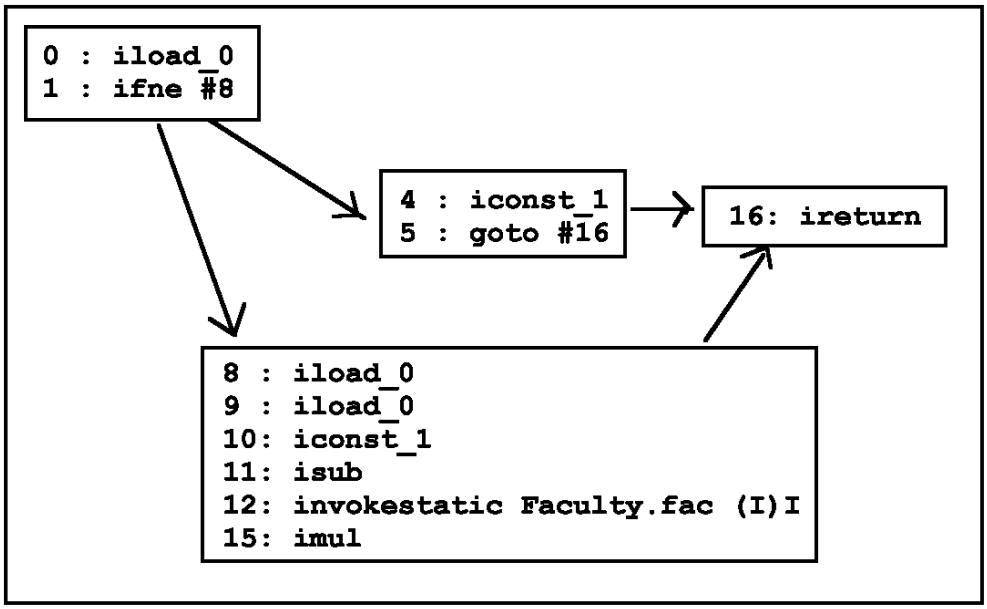


Figure 3.1: A Conventional Control Flow Graph

[7]More information about control flow graphs can be found in [DragonBook].

The JVM defines a sort of control flow orthogonal to the common execution of instructions, namely, the exception mechanism. Because every instruction could possibly throw an exception (say, a `java.lang.VirtualMachineError`) during its execution, the control flow graph calculated by JustIce always uses only one instruction per node. This also reflects the original verification algorithm given by Sun Microsystems. Figure 3.2 shows an example for such a control flow graph.

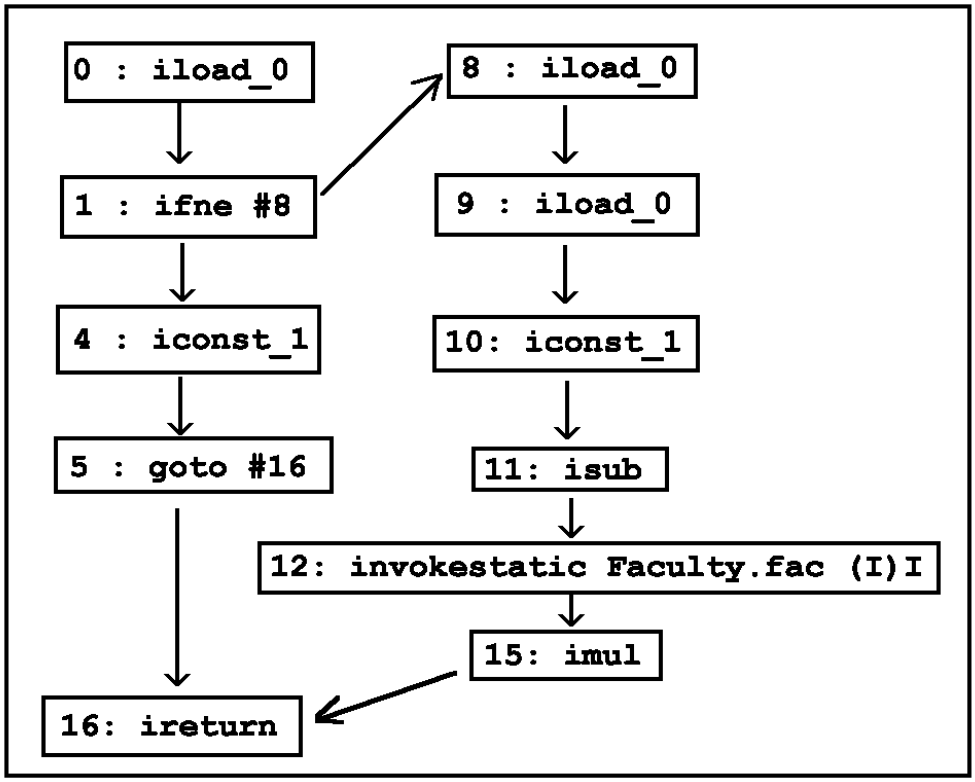


Figure 3.2: A Control Flow Graph as Used by JustIce

Instruction nodes are augmented with a data structure that represents the simulated operand stack and the simulated local variables array. When running the core verification algorithm, these nodes are put into a queue which is equivalent to tagging them with a *changed* bit as Sun describes[8].

Subroutines Revisited: Interplay With the Data Flow Analyzer

There is another problem concerning subroutines. Normally, when merging the type information of two simulated local variables, the common type is recorded as *unusable* if the types differ. This *unusable* value is then propagated to subsequent instructions to prevent read access.

[8] As explained later, JustIce uses a queue that allows duplicates: this is a slight semantical change.

This is not the case with the successors of the `ret` instruction. These successors are physical successors of some `jsr` or `jsr_w` instructions.

Subroutines are said to be *polymorphic* with respect to their local variables arrays. As an example, consider algorithm 6. This algorithm shows legal JVM code. In line 11, local variable 0 may contain a value of the `integer` or the `float` type; depending on the `jsr` instruction that entered the subroutine. Normally, this would cause the verifier to mark local variable 0 as *unusable* and propagate this information. The successors of the `ret` instruction are the instructions in lines 5 and 10. However, a correct verifier does *not* mark local variable 0 as *unusable* for them, because the local variable 0 was not accessed or modified in the subroutine.

Algorithm 6 Local Variables are Polymorphic in Subroutines

```
0 :  iconst_0    ; load integer constant 0 onto stack
1 :  istore 0    ; move it into local variable 0
2 :  jsr 11      ; enter subroutine
5 :  fconst 0.0  ; load float constant 0.0 onto stack
6 :  fstore 0    ; move it into local variable 0
7 :  jsr 11      ; enter subroutine again
10:  return      ; complete method
11:  astore 1    ; Subroutine entry:  move return address
                 ; into local variable 1
12:  nop         ; do nothing
13:  ret 1       ; return from subroutine
```

Basically, only the local variables accessed in the called subroutine (and the subroutines called from there, recursively) are merged with the corresponding successor of a `ret` instruction. This means that in this special case, three sources are used to construct the merged array of local variables type information (instead of only two): the `jsr`/`jsr_w` instruction, the `ret` instruction and the "old" type information of the `ret` instruction's target (which is the physical successor of the `jsr`/`jsr_w` instruction).

One possibility to deal with this situation is *inlining*. For instance, the verifier of the ElectricalFire JVM [EF] uses this approach: instruction nodes of subroutines are duplicated for every calling `jsr` or `jsr_w` instruction. This approach is equivalent to the one sketched by Sun (see [VMSPEC2], page 151).

JustIce uses a variant of this approach: instruction nodes are augmented with sets of local variables arrays. The local variables array used for merging a `ret`'s type information with the physical successor of some `jsr`/`jsr_w` instruction is keyed by that `jsr`/`jsr_w` instruction itself. This still implies a special merging mechanism for the `ret` instruction: only the physical successor of one `jsr`/`jsr_w` instruction can be merged with the `ret` at a time, because other `jsr`/`jsr_w` instructions have possibly not been symbolically executed yet and thus bear no type information at the time of merging. In this scenario, an instruction in a subroutine plays multiple roles; one for each occurence of a `jsr`/`jsr_w` that is calling the subroutine. The queue holding the instructions to symbolically execute is therefore required to allow duplicates.

Wide Data Types

The types `long` and `double` use two consecutive local variables if written to or read from a local variables array. Similarly, they use two operand stack slots. This makes type verification a bit more difficult because of subtle special cases. For example, when a method uses three local variables at maximum (local variables 0, 1 and 2), the code is not allowed to store a `double` value in local variable 2 (because local variable 3 would have to be occupied, too).

Instance Initialization and Newly Created Objects

It would be difficult to verify that a newly created instance is initialized exactly once, given all possible paths of execution flow in a method. Fortunately (from a verifier implementor's view), Sun puts constraints on object initialization that match the behaviour of the verifier — instead of putting sane constraints on object initialization and actually verifying them.

"A valid instruction sequence must not have an uninitialized object on the operand stack or in a local variable during a backwards branch [...]. Otherwise, a devious piece of code might fool the verifier into thinking it had initialized a class instance when it had, in fact, initialized a class instance created in a previous pass through a loop" ([VMSPEC2], page 148).

3.4 Pass Four

Pass four performs "certain tests that could in principle be performed in Pass 3" ([VMSPEC2], page 142). These tests are usually delayed by JVM implementations until run-time, because they possibly trigger the loading of referenced class file definitions. This is a performance enhancement. However, "A Java virtual machine implementation is allowed to perform any or all of the Pass 4 steps as part of Pass 3" ([VMSPEC2], page 143). The tests

- ensure that the referenced method or field exists in the given class
- check that the referenced method or field has the indicated descriptor (signature)
- check that the currently executing method has access to the referenced method or field.

JustIce has no run-time system and so the tests of pass four are performed in pass 3a.

There are tests that have to be performed at run-time: for example, if an object referenced by an object reference on top of the operand stack implements a certain interface or not [Fong2-WWW]. These are not considered part of the pass four verification.

4 Implementation of the Verification Passes

Occasionally, the behaviour of other verifier implementations was explained in section 3. This is not a mistake; the Java Virtual Machine Specification, Second Edition [VMSPEC2] is unfortunately not detailed enough to make a clean-room implementation of the JVM verifier possible. Having a close look at the behaviour of existing verifier implementations is sometimes necessary to interpret the specification correctly. For that reason, the behaviour of these implementations is part of the specification of JustIce whereever appropriate. Still, there are some minor differences in behaviour between JustIce and the traditional JVM built-in verifiers. These differences were observed by using the traditional verifiers, not by inspecting their source code.

JustIce is implemented in the Java programming language [langspec2] using the Byte Code Engineering Library [BCEL-WWW, BCEL98].

4.1 Pass One

The Byte Code Engineering Library (BCEL) presents an object oriented view of the class file structure. Therefore, an integral part of that library is parsing class files. JustIce uses the BCEL, so there was nothing left to do to load a class file in. Only minor changes were made to the BCEL to make it more verbose when exceptional situations occur; i.e., when a garbled class file is loaded in. The BCEL uses Java's exception mechanism to signal these situations; JustIce transforms this behaviour into the behaviour expected by users of the Verification API (see section 5).

Comparison to Sun's Implementation

There does not seem to be any difference in behaviour between JustIce and the traditional verifiers. Still, this conviction is a result of black box tests so it might not be true in corner cases.

Unknown attributes are ignored (though JustIce records a warning message, where the traditional verifiers don't).

Trailing bytes at the end of the class file are ignored in both versions, contradicting the specification. This was necessary because some Java run-time environments are broken concerning the handling of .JAR archive files. The mechanism of loading class files from these archives files using the Java Platform's API is used by BCEL and probably by Sun's JVM, too. It is possible that this is the reason why Sun's verifier itself does not enforce this constraint.

However, it does not really pose a threat to the integrity of any JVM known to the author. There is no entry in the `ClassFile` structure (see section 2.1) stating how long the class file is in its entirety, so a JVM implementor cannot possibly base a wrong decision on that.

4.2 Pass Two

JustIce does perform "all verification that can be performed without looking at the bytecodes" in pass two. For some reasons (like determining a valid ancestor hierarchy of a class), pass two of JustIce has to load referenced classes. Of course, this is done in a careful way: by pass-one-verifying them. If loading of a referenced class should fail (i.e., verification pass one fails on this class), the referencing class is rejected by JustIce's pass two. Pass two of JustIce does not pass-two-verify any referenced classes.

Also, JustIce's pass two emits a wealth of (warning) messages. Their target is to guide a bytecode engineer to create class files that are indistinguishable from those created by Sun's *javac* compiler with no debugging output. For example, the use of `LineNumberTable` attributes (see section 2.1.1) is discouraged, because these atributes are only useful for debugging purposes. Still, they can be the reason for a class file to be rejected – to be on the safe side, finished applications for the JVM should not be shipped with this debug information.

Most of the checks of pass two were implemented using the Visitor programming pattern [DesignPatterns] provided by the BCEL's *de.fub.bytecode.classfile* API. This made it possible to have all the verification split into several methods without having to define artificial boundaries. For instance, a `ConstantValue` attribute is verified in a method called *visitConstantValue(ConstantValue)*. This is a use of the object oriented view of class files the BCEL offers.

Comparison to Sun's Implementation

JustIce does not distinguish between run-time or link-time because it was not intended to implement a JVM. Therefore, the notion of *resolving* (see section 3.2) is useless for JustIce. The author believes that the specification of pass two given by Sun closely reflects their implementation (or the other way around)[1].

Sometimes, there are ambiguities in the specification. For instance, it is said that "If the constant pool of a class or interface refers to any class or interface that is not a member of a package, its `ClassFile` structure must have exactly one `InnerClasses` attribute in its `attributes` table". A class or interface that is "not member of a package" is better known as a *nested class* or *inner class* [InnerSpec], but this is something specific to the Java language. The *javac* compiler creates multiple, often funny-named[2] class files that are

[1] The Java Virtual Machine Specification, Second Edition, began as an internal project documentation ([VMSPEC2], page xiv). Unfortunately, this can still be felt sometimes.

[2] For anonymous classes defined in a class *X* the names are *X$1*, *X$2* and so on. For a named inner class *I* defined in class *C* the name is *C$I*. There is, however, no guarantee for that: this is only observed behaviour of javac. Please see section 7.2.1 for an example how this behaviour can lead to unexpected problems.

otherwise indistinguishable from normal class files.

Therefore, it is generally not possible to decide if such an attribute is missing; therefore Sun's implementation does not check this constraint. JustIce, in contrast, uses its warning mechanism if the name of a referenced class or interface could be a name of an inner class created by the *javac* compiler and the `InnerClass` attribute is missing.

The sets of accepted or rejected class files concerning pass two are equal using both Sun's implementation and JustIce, as exhaustive tests show. This can, however, not be proven because one would need to analyze Sun's source code for that (which is not intended: as already mentioned, JustIce is a clean-room implementation).

4.3 Pass Three

4.3.1 Pass 3a

One feature of the BCEL's *de.fub.bytecode.generic* package is parsing code attributes of methods and transforming them into so-called `InstructionList` objects. Consequently, this feature is used to implement pass 3a; a few additional checks have been implemented where BCEL is too "trustful" when parsing, i.e., where BCEL relies on the correctness of the class file.

Pass 3a consists of the checking of static constraints on instructions and static constraints on operands of these instructions. The successful creation an an `InstructionList` object already implies that the static constraints on instructions are satisfied. Similar to pass one, JustIce transforms the behaviour of BCEL's exception mechanism into the behaviour expected by users of the Verification API (see section 5).

The *de.fub.bytecode.generic* API provided by BCEL offers a Visitor design pattern similar to the one of the *de.fub.bytecode.classfile* API. The tests for the static constraints on operands of instructions are implemented by using it. For example, the constraints put on the operands of any `iload` instruction are verified using a *visitILOAD(ILOAD)* method defined in a Visitor class. This Visitor class implements all the checks for integrity of all instruction's operands. Algorithm 7 shows the impementation of the *visitILOAD(ILOAD)* method.

JustIce does not provide any run-time, so the tests of pass four (see section 3.4) are not delayed until run-time, but performed here.

Comparison to Sun's Implementation

Sun does not distinguish pass 3a and pass 3b. However, Sun's verifiers also have to ensure that the static constraints on instructions are satisfied before starting data flow analysis.

This is obvious because a data structure has to be built before the data flow analyzer can be run; and this data structure has to be built carefully[3] because passes one and two did not look at the bytecodes before.

[3]This actually means verifying the structural integrity of the bytecodes.

Algorithm 7 visitILOAD, Visitor ensuring static constraints on operands of instructions

```
 /** Checks if the constraints of operands of the said
instruction(s) are satisfied.  */
public void visitILOAD(ILOAD o){
  int idx = o.getIndex();
  if (idx < 0){
    constraintViolated(o, "Index '"+idx+"' must be
non-negative.");
  }
  else{
    int maxminus1 = max_locals()-1;
    if (idx > maxminus1){
      constraintViolated(o, "Index '"+idx+"' must not be greater
than max_locals-1 '"+maxminus1+"'.");
    }
  }
}
```

JustIce does implement pass four checks in pass 3a which Sun's verifiers do not. Because JustIce provides no run-time, the outcome of a verification failure is reported instantly. Traditional JVMs are required to silently delay the actions triggered by that knowledge until run-time.

4.3.2 Pass 3b

JustIce aims at implementing Sun's data flow analyzing algorithm as closely as possible. First, a control flow graph is built — which implies analyzing a method's subroutine calling structure first.

After that an implementation of the core algorithm sketched by Sun Microsystems is started. Verification failure is internally signalled by the Java exception handling mechanism which is then transformed to match the Verification API (see section 5).

Subroutines

Subroutines are modeled as instances of the `Subroutine` interface. They provide the following methods (note that an `InstructionHandle` is the BCEL's programming handle to instruction objects and that *X[]* is the common Java notation for *array of X*):

- *boolean contains(InstructionHandle)*
  Returns true if and only if the given `InstructionHandle` refers to an instruction that is part of this subroutine,

- *InstructionHandle[] getInstructions()*
  Returns all instructions that together form this subroutine,

- *int[] getAccessedLocalsIndices()*
  Returns an array containing the indices of the local variable slots accessed by this subroutine (read-accessed, write-accessed or both); local variables referenced by subroutines of this subroutine are not included,
- *int[] getRecursivelyAccessedLocalsIndices()*
  Returns an array containing the indices of the local variable slots accessed by this subroutine (read-accessed, write-accessed or both); local variables referenced by subroutines of this subroutine are included,
- *Subroutine[] subSubs()*
  Returns the subroutines that are directly called from this subroutine,
- *InstructionHandle[] getEnteringJsrInstructions()*
  Returns all the JsrInstructions that have the first instruction of this subroutine as their target,
- *InstructionHandle getLeavingRET()*
  Returns the one and only RET that leaves the subroutine.

Together with information from a simple analysis of the possible control flow transfer of all the other instructions but `ret` (see section 3.3), a control flow graph is built.

The Control Flow Graph

The control flow graph is a single instance with respect to a given method to verify. It is defined by providing access to a set of contexts of instructions. These are modeled as instances of the `InstructionContext` interface.

These instances enclose `InstructionHandle` objects (which represent an instruction in the bytecode), but they augment these objects with type information (a set of `Frames`, see below) as needed by the data flow analysis algorithm. Also, a method called *getSuccessors()* is provided that calculates the possible control flow successors of a given `InstructionContext` instance.

The most notable method defined in the `InstructionContext` interface is, however, the *execute(Frame, ArrayList, InstConstraintVisitor, ExecutionVisitor)* method. This method is used to symbolically execute a given instruction.

The `ArrayList` argument is there to record the subroutine calling chain. The properly-nested property of JustIce subroutines is exploited here: one can simply count `jsr`/`jsr_w` and `ret` instructions, similar to counting opened and closed braces in mathematical expressions.

A `Frame` is JustIce's model of an *execution frame*: a local variables array model together with an operand stack model. Every *InstructionContext* instance is augmented with such a frame (to be precise, a set of such frames as discussed in the specification of subroutines, see section 3.3).

When frames are merged, the *execute(Frame, ArrayList, InstConstraintVisitor, ExecutionVisitor)* method of some successor `InstructionContext` is called. The `Frame` argument represents is the current type information of the predecessing `InstructionContext`.

Visitors

As in pass 3a, the Visitor pattern of the BCEL *de.fub.bytecode.generic* API is also used in pass 3b. While it was used to verify the static constraints of pass three in pass 3a, it is now used to verify the structural constraints.

Before an instruction `X` is symbolically executed, the corresponding *visitX(X)* method is invoked on an `InstConstraintVisitor` instance. This instance is there to verify all the preconditions are met to safely execute the instruction `X`. The `InstConstraintVisitor` class therefore holds information about the preconditions of all 212 valid Java bytecode instructions. A simplified version of this Visitor's *visitILOAD(ILOAD)* method is listed in algorithm 8.

Similarly, the `ExecutionVisitor` class contains information about the behaviour of every bytecode instruction. An instance of this class is used to model the effect of the bytecode instructions on a *Frame* instance. Algorithm 9 shows the *visitILOAD(ILOAD)* method of this Visitor.

Algorithm 8 visitILOAD, Visitor ensuring the structural (dynamic) constraints of instructions

```
public void visitILOAD(ILOAD o){
     int produce = o.produceStack(cpg);
     if ( produce + stack().slotsUsed() > stack().maxStack() ){
       constraintViolated(o, "Cannot produce "+produce+" stack
slots:  only "+(stack().maxStack()-stack().slotsUsed())+" free
stack slot(s) left.\nStack:\n"+stack());
     }
[...]
  }
```

Algorithm 9 visitILOAD, Visitor symbolically executing instructions

```
/** Symbolically executes the corresponding Java Virtual Machine
instruction.  */
public void visitILOAD(ILOAD o){
  stack().push(Type.INT);
}
```

Comparison to Sun's Implementation

JustIce was originally aimed to be as compatible to Sun's implementation as possible. However, the unclear specification prevents clean room implementations (i.e., implementations whose programmers did not look into Sun's code) from perfect compatibility.

Fortunately, it JustIce closely matches Sun's implementation in its behaviour. As a test case, the author verified the transitive hull of the referenced class files starting with the *de.fub.bytecode.verifier.Verifier* class. This set includes most of the classes of the Java 2 API supplied by Sun Microsystems, i.e., a few hundreds of apparently correct classes. A very small number of class files was rejected by

Algorithm 10 Simplified Core Verification Algorithm of Pass 3b

```
public VerificationResult do_verify(Method m){
  ControlFlowGraph cfg;
  if (m.hasCode())
    cfg = new ControlFlowGraph(m)
  else
    return Good_VerificationResult;
  Frame f = new Frame(); // local variables and operand stack
  f.localVariables().initialize(m.signature()); // put formal param types into loc. vars
  InstConstraintVisitor icv = new InstConstraintVisitor();
  ExecutionVisitor ev = new ExecutionVisitor();
  try{
    circulationPump(cfg, f, icv, ev);
  }
  catch(VerificationFailure){
    return Bad_VerificationResult;
  }
  return Good_VerificationResult;
}

public void circulationPump(ControlflowGraph cfg, Frame startFrame, InstConstraintVisitor icv, ExecutionVisitor ev) throws VerificationFailure{
Instruction start = cfg.getFirstInstruction();
/*
Now merge the first frame (type info) into the first instruction.
Empty list -> no instructions have been executed before.
*/
start.execute(startFrame, EmptyInstructionList, icv, ev);
/*
Q is a Queue of pairs (Instruction, InstructionList).
*/
Queue Q = EmptyQueue;
/*
Put the first instruction into the queue. This is similar to initializing a breadth first search.
*/
Q.add (start, EmptyInstructionList);
/*
The main loop
*/
while (Q.isNotEmpty()){
  Instruction u = fst(Q.head());
  InstructionList ec = snd(Q.head());
  Q.removeHead();
  InstructionList oldchain = ec;
  InstructionList newchain = ec++[u];
  for (all successors v of u){
    /*
    execute returns true if type info has changed. It may throw VerificationFailures.
    */
    if (v.execute(u.getOutFrame(oldchain), newchain,icv,ev))
      Q.add((v, newchain));
  }
}
```

JustIce because of its different specification of subroutine constraints. No other rejects were encountered.

Most class files that are found to be rejected by Sun's verifier implementations are rejected by JustIce, too.

However, there are class file rejected by Sun's verifier implementations but not by JustIce. This should not occur, but JustIce does not mimic the programming errors of Sun's verifiers so far. Please see section 7.2.2 for a discussion on a selected incompatibility issue.

An automated testing suite could solidify the trust in JustIce's implementation which is not implemented yet. Please see section 6.3.1 for a discussion on that topic.

4.4 Pass Four

The tests Sun's verifiers perform during run-time but which in principle could be performed in pass three *are* performed in pass 3a by JustIce.

Comparison to Sun's Implementation

It sems natural that Sun's verifier implements the specification by Sun. Obviously, JustIce has no run-time so JustIce has no pass four. The checks Sun performs in pass four[4] are performed in pass 3a by JustIce.

[4]Some JVMs expose implementation mistakes concerning pass four verification. See section 7.2.2.

5 The Verification API

5.1 Introduction

The Application Programming Interface (API) of JustIce uses object oriented design patterns [DesignPatterns]. Readers not familiar with design patterns are encouraged to read at least about the *Visitor*, *Singleton*, *Observer* and *Factory* patterns.

JustIce currently consists of four packages: *de.fub.bytecode.verifier*, *de.fub.bytecode.verifier.exc*, *de.fub.bytecode.verifier.statics* and *de.fub.bytecode.verifier.structurals*. (We shall from now on omit the preceding *de.fub.bytecode*.) The most important of them is the *verifier* package. The class `VerifierFactory` can be found here; this is the place where all verification starts. The `VerifierFactory` creates `Verifier` instances; only the `VerifierFactory` can create these instances. A `Verifier` instance, in turn, has a one-to-one relationship with a class file to verify, "its class". You can instruct a `Verifier` instance to run a verification pass on its class yielding a `VerificationResult`.

All class files are fetched from the BCEL's class file repository, i.e., the class `Repository`. The class files stored there are either put there by the user or they are read from the file system. For a bytecode engineer who uses the BCEL this is convenient, because one does not have to save the dynamically created class file first in order to load it into JustIce.

Pass 1 and pass 2 are related to the `ClassFile` structure as such; passes 3a and 3b verify the bytecode of a method. If a class file was created using the BCEL, the BCEL user already knows how the `JavaClass` object looks like[1]. The number of methods is known and the order of the methods in the class file is known.

However, if this is not the case, one usually does not know the number of methods in a class file or the order of these methods. To carefully extract this information from an untrusted class file, one should first let a pass-2-verification run on this file. Afterwards, the information can be read from the `JavaClass` object the BCEL offers.

Finally, one is able to supply the "method index" needed by verification passes 3a and 3b.

Basically, after pass 2 has been run successfully on a class file, one can safely use the methods in the BCEL's *classfile* package on that class file. After pass 3a has been run successfully on a method, one can safely work on that method using the BCEL's *generic* package. After pass 3b has been run successfully on all methods in a class file, this class file will not be rejected by other verifiers.

[1] A `JavaClass` object represents a class file in the BCEL.

Often, the run of a verification pass implies recursively verifying other class files as well (because they are somehow referenced). Therefore, *Verifier* instances for these referenced classes are created transparently. To be notified when such an event occurs, one can implement the *VerifierFactoryObserver* interface and let the *VerifierFactory* register your implementation.

A Verifier creates instances of PassVerifiers. A PassVerifier instance in charge of performing some later verification pass transparently creates PassVerifier instances for the preceding passes. Therefore, users of the Verification API do not have to care about the order of verification passes; i.e., earlier passes are run always before later passes. All verification results are cached; this way an unsual order of calls to the *doPassX()* methods of the *Verifier* class does not even waste computing time.

5.2 Some Example Code

The code below shows an example of how to use the API provided by JustIce. It will verify the transitive hull of all referenced class files. Normally, while verifying a class, referenced classes are recursively verified performing *earlier* passes. Verifiers that are using pass 1 on their class will not load in any other classes (see section 3). Therefore, normally the transitive hull is *not* verified completely (it usually does not make sense to verify it, though – it's done here only to give an example of what can be done).

```
01 package de.fub.bytecode.verifier;
02 import de.fub.bytecode.verifier.*;
03 import de.fub.bytecode.classfile.*;
04 import de.fub.bytecode.*;
05 /**
06  * This class has a main method implementing a demonstration program
07  * of how to use the VerifierFactoryObserver.  It transitively verifies
08  * all class files encountered; this may take up a lot of time and,
09  * more notably, memory.
10  *
11  * @author <A HREF="http://www.inf.fu-berlin.de/~ehaase"/>Enver Haase</A>
12  */
13 public class TransitiveHull implements VerifierFactoryObserver{
14   /** Used for indentation.  */
15   private int indent = 0;
16   /** Not publicly instantiable.  */
17   private TransitiveHull(){ }
18
19   /* Implementing VerifierFactoryObserver.  */
20   public void update(String classname){
21     for (int i=0; i<indent; i++) {
22       System.out.print(" ");
23     }
24     System.out.println(classname);
25     indent += 1;
26     Verifier v = VerifierFactory.getVerifier(classname);
27     VerificationResult vr;
```

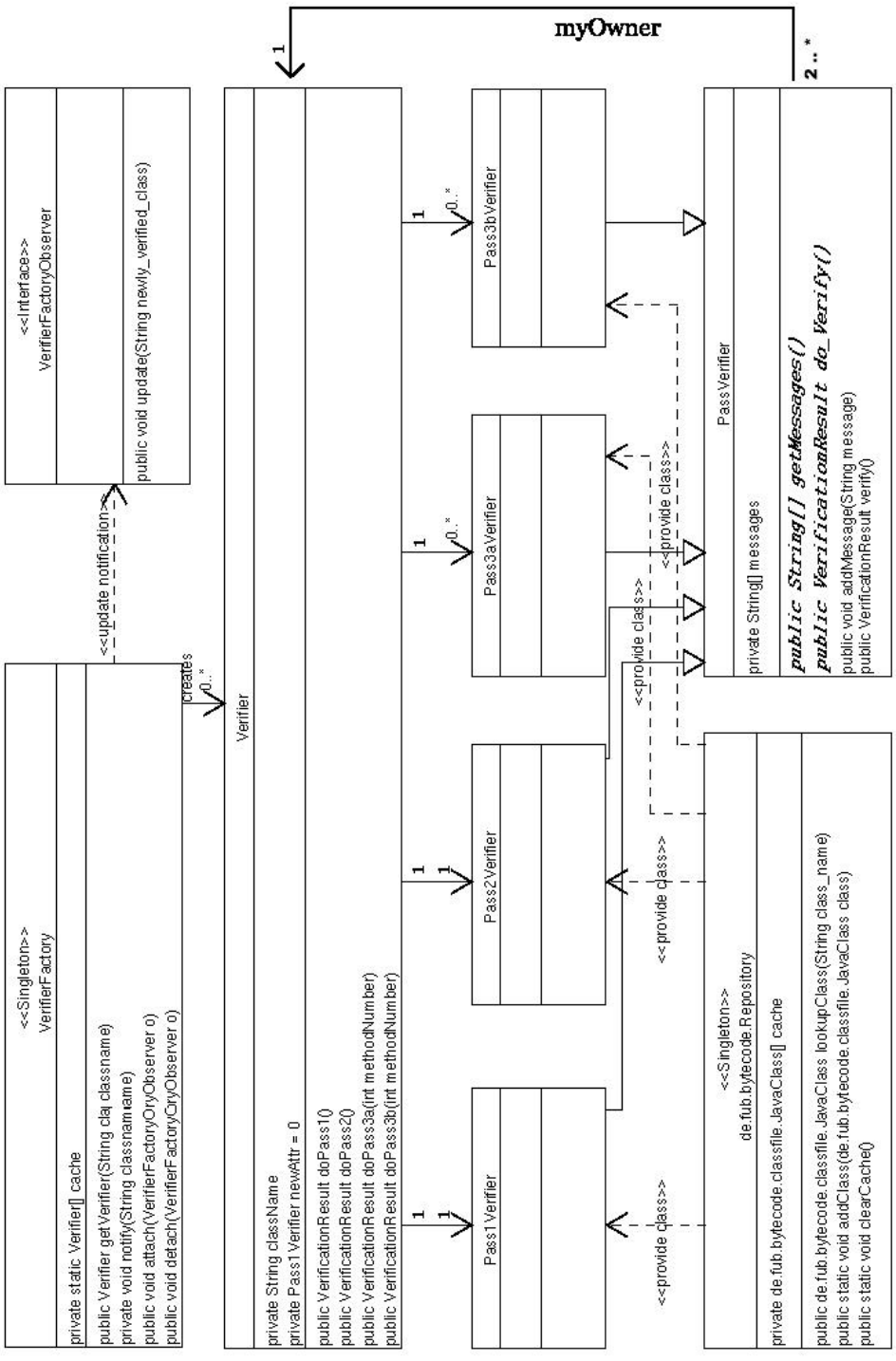


Figure 5.1: UML class diagram of the Verification API

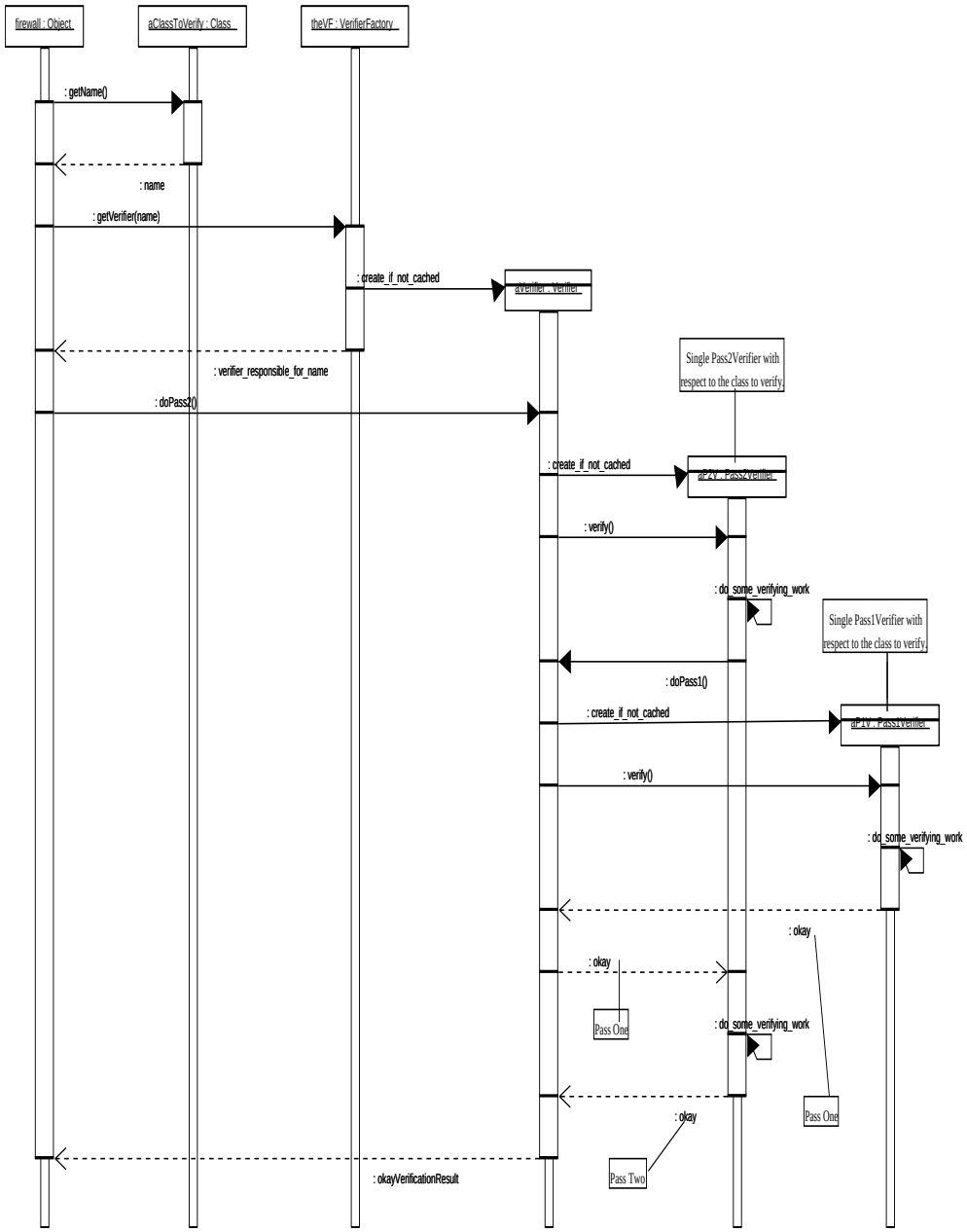


Figure 5.2: Informal UML sequence diagram showing the dependency of verification pass two on verification pass one.

```
28      vr = v.doPass1();
29      if (vr != VerificationResult.VR_OK)
30        System.out.println("Pass 1:\n"+vr);
31      vr = v.doPass2();
32      if (vr != VerificationResult.VR_OK)
33        System.out.println("Pass 2:\n"+vr);
34      if (vr == VerificationResult.VR_OK){
35        JavaClass jc = Repository.lookupClass(v.getClassName());
36        for (int i=0; i<jc.getMethods().length; i++){
37          vr = v.doPass3a(i);
38          if (vr != VerificationResult.VR_OK)
39            System.out.println(v.getClassName()+", Pass 3a, method "+
              i+" ['"+jc.getMethods()[i]+"']:\n"+vr);
40          vr = v.doPass3b(i);
41          if (vr != VerificationResult.VR_OK)
42            System.out.println(v.getClassName()+", Pass 3b, method "+
              i+" ['"+jc.getMethods()[i]+"']:\n"+vr);
43        }
44      }
45      indent -= 1;
46    }
47
48   /**
49    * This method implements a demonstration program
50    * of how to use the VerifierFactoryObserver.  It transitively
51    * verifies all class files encountered; this may take up a
52    * lot of time and, more notably, memory.
53    */
54    public static void main(String[] args){
55      if (args.length != 1){
56        System.out.println("Need exactly one argument:  The root class
          to verify.");
57        System.exit(1);
58      }
59      int dotclasspos = args[0].lastIndexOf(".class");
60      if (dotclasspos != -1)
61        args[0] = args[0].substring(0,dotclasspos); args[0] =
          args[0].replace('/', '.');
62      TransitiveHull th = new TransitiveHull();
63      VerifierFactory.attach(th);
64      VerifierFactory.getVerifier(args[0]); // the observer is called
        back and does the actual trick.
65      VerifierFactory.detach(th);
66    }
67  }
```

First, an instance of the *TransitiveHull* class is created in line 62. Note that this class implements the *VerifierFactoryObserver* interface.

A reference to the newly created instance is then passed to the *VerifierFactory* in line 63 by invoking its *attach(VerifierFactoryObserver)* method. After registering the new observer, the *VerifierFactory* will call the instance's *update(String)* method (defined in lines 20-46) whenever a new *Verifier* instance is created.

To trigger the verification, a first *Verifier* instance is fetched from the *VerifierFactory*. Because it is the very first *Verifier* instance that is fetched, we know that it has to be newly created. This is done in line 64. This instance is not used in the *main(String[])* method; but its creation leads to a invocation of the *update(String)* method which is defined in lines 20-46.

There, the name of the class to verify is printed (lines 21-25, line 45) and the four verification passes provided by JustIce are run. Note that one has to be careful not to try to verify a method that does not exist. JustIce would in this case throw an *InvalidMethodException*. Therefore, after successfully verifying that the structure of the class file to verify is well-formed (verification up to and including pass two, lines 26-31), the number of methods is fetched from the corresponding JavaClass object. (It is necessary to perform verification pass two on a class file to safely find out how many methods are defined in this class file.)

After determining the number of methods, these methods are verified performing passes 3a and 3b on them (lines 32-44).

By applying all verification passes on some class file C, all class files referenced by C are found. Therefore, new *Verifier* instances are created which are responsible for them. Because of that, the *update(String)* method described above is called for every referenced class. This is a recursive loop; the program terminates when there is no referenced class left to be verified.

The example above is simple yet powerful. Admittedly, it is of limited use to verify classes provided by the JVM vendor; therefore one would not normally verify all the transitive hull of referenced class files. However, a common use is verifying all classes of a project. Inserting a new line between line 20 and 21 like

```
if (!(classname.startsWith("de.fub.bytecode.verifier")) return;
```

would easily accomplish this goal if JustIce itself is the project to verify and all the project's class files are referenced by another class file in the project.

5.3 An Application Prototype

The API of JustIce is used to offer bytecode engineers an opportunity to create their own application programs. However, this dimension of configurability is often not needed.

JustIce comes with an application prototype which provides an easy-to-use user interface. Figures 5.3 and 5.4 show screen shots of this prototype built on the JustIce verifier. The boxes to the right contain verification information. From the top to the bottom the boxes represent the verification passes one, two, 3a and 3b and the warning messages, respectively.

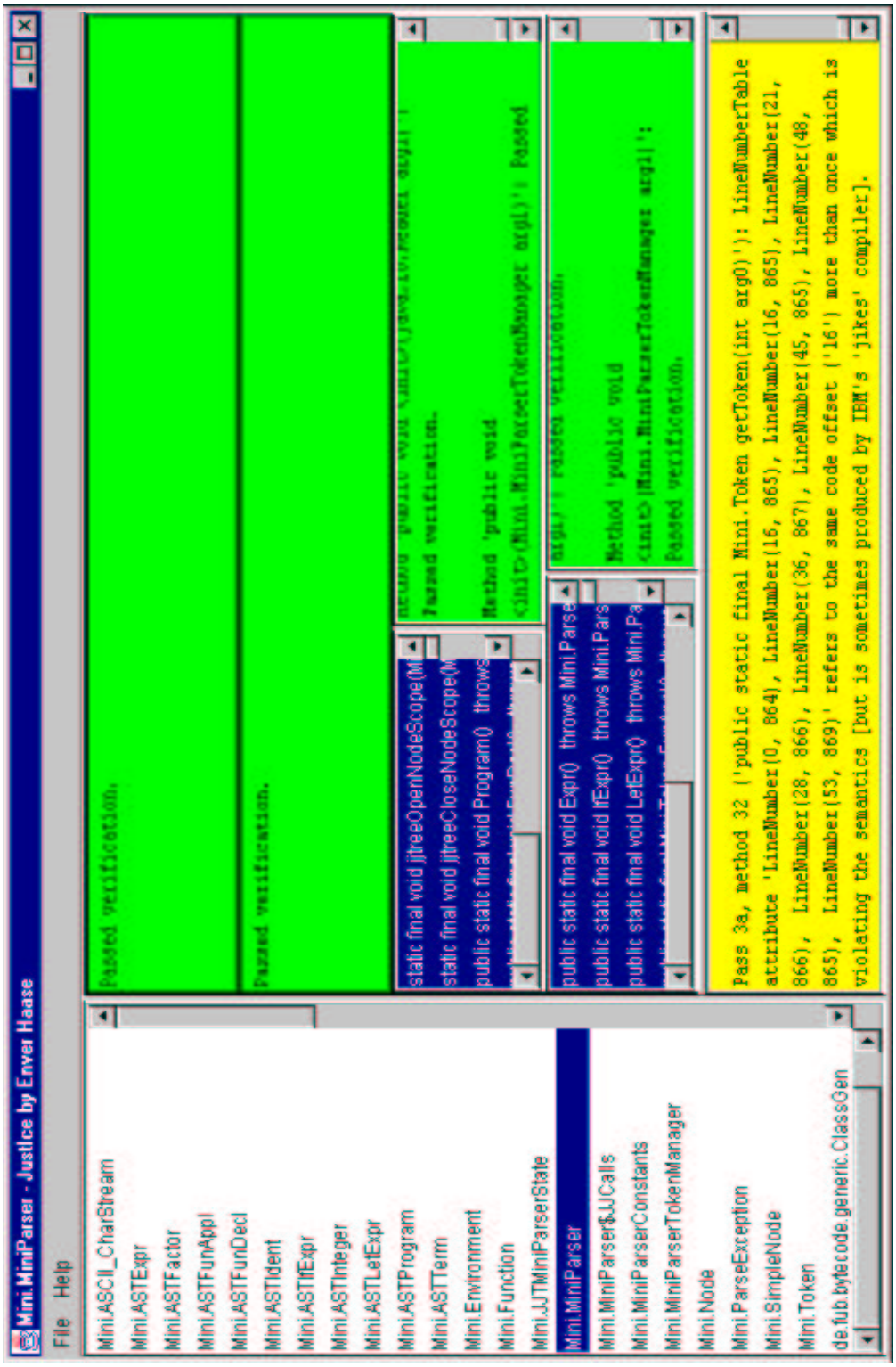


Figure 5.3: Verification of the Mini.MiniParser class file. Verification is passed, but JustIce suggests to remove unnecessary (debug information) attributes.

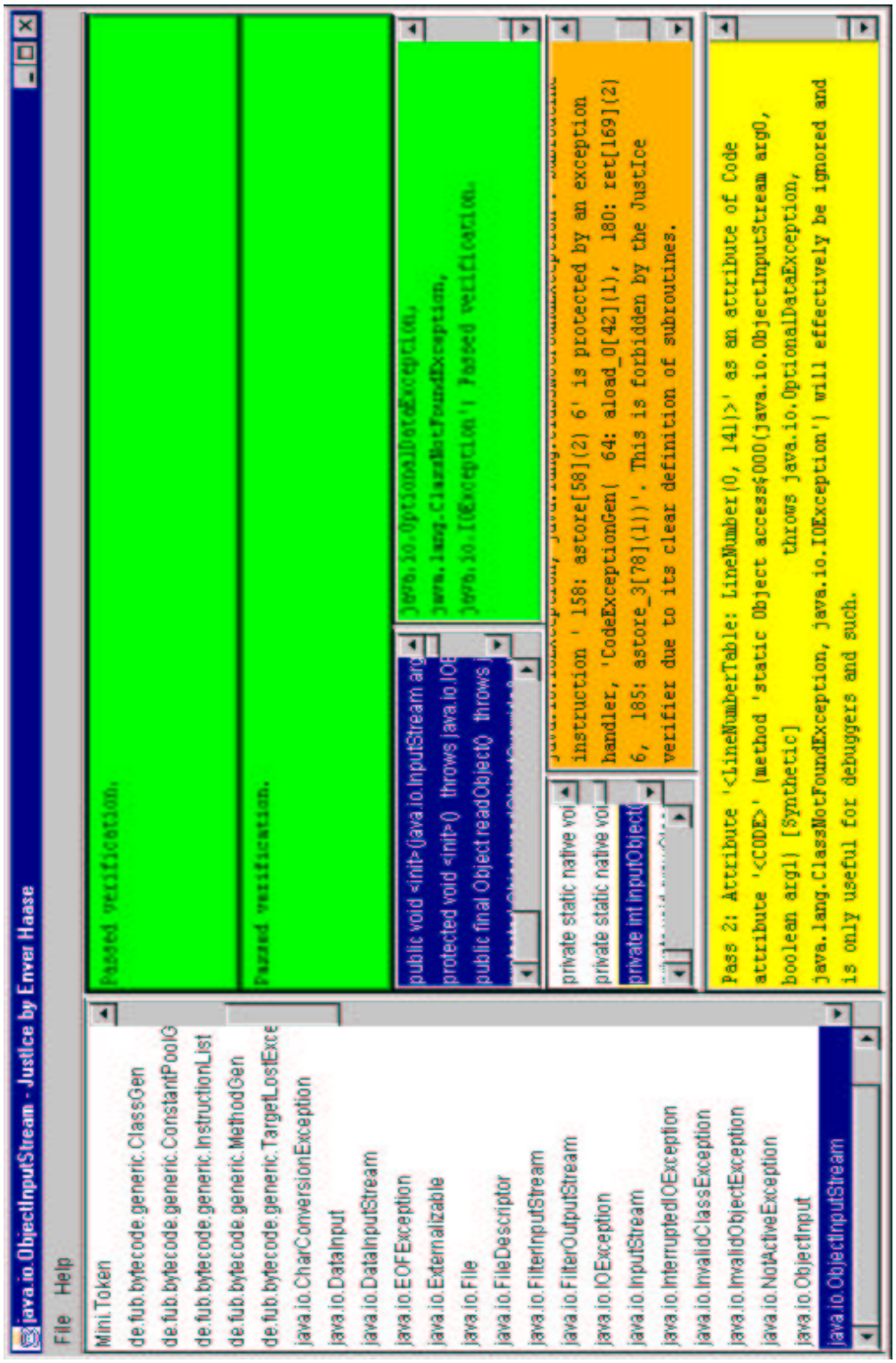


Figure 5.4: Verification of the java.io.ObjectInputStream class file. Verification is not passed because of an unsatisfied constraint related to subroutines.

6 Conclusion

6.1 What Was Achieved

About a third of the development time of JustIce was spent examining the various issues in connection with subroutines, i.e., issues concerning the bytecode instructions `jsr`, `jsr_w` and `ret`. This led to a new definition of the term *subroutine* (section 3.3.2)[1], a new implementation of this verification area (section 4.3.2) and a discussion on the arising incompatibilities (sections 4.3.2 and 7.2.2).

Only a few different verifier implementations exist at all, and most of them are incomplete. JustIce is a complete class file verifier implementation including a bytecode verifier.

The development of JustIce also led to improvements of the Byte Code Engineering Library [BCEL-WWW, BCEL98]. For instance, the `returnaddress` data type was introduced there. It was modeled as a parameterized type. Also, a programming error was repaired that led to inconsistent treatment of exception handlers in the BCEL.

The control flow graph used by JustIce can also be used in other projects; the Verification API provides access to this data structure[2]. Only because of the clarification of the subroutine issues could such a data structure be defined statically.

As an Open Source project, JustIce provides algorithms which may be reused in own projects. For example, every compiler targeting the JVM has to calculate the maximum amount of stack memory used by a method. This is also done by JustIce.

Finally, the need for a discussion on the meaning of *Java security* was identified (see section 6.3.4).

6.2 What Could Not Be Achieved

6.2.1 A Constraint Database

Efforts have been made to make JustIce verifier highly configurable. Unfortunately, this could not be accomplished by the author. For instance, it was planned to build a constraint database which would make it possible to turn on or off single checks during verification.

[1] A request for clarification of the subroutine issue, sent to the electronic mail address `jvm@java.sun.com` was not answered.

[2] A `ControlFlowGraph` instance can be created by invoking the *ControlFlowGraph(MethodGen)* constructor. A `MethodGen` is the BCEL's representation of a method.

While this might be possible in some cases, in general the constraints of the class file verifier are highly intertwined. For instance, without a well-formed constant pool one could not run the data flow analyzer in a sane way. As another example, if a user preferred not to care about stack underflow the verification algorithm would require complicated user interaction; i.e., the user would have to decide what type to put onto the simulated operand stack just before it is read.

One could model the interdependencies of the various constraints and allow only groups of checks to be turned on or off together. However, the author doubts this could be done in a way that is not prone to errors and that can be validated easily.

This is also the reason why only one error is reported if verification fails. Trying to continue verification and find more constraint violations leads only to consequential verification errors.

JustIce implements caching of verification results. If a bytecode engineer works on a class file and needs to run JustIce several times against it, JustIce will cache the verification results of the recursively referenced class files. Because of this, JustIce will be fast every subsequent time it is used to verify the class. This minimizes the impact of the above shortcomings.

6.2.2 A Perfect Verifier

JustIce does not implement a perfect verifier. Some class files with code that is safe to execute are rejected. Unfortunately, there has to be some degree of uncertainty concerning which class files to reject.

The JVM performs *initialization* of class files after loading and verifying them without error. This includes running the code in the special class initialization method called *<clinit>* if it exists (see [VMSPEC2], page 53). For the correct operation of the JVM it is important that this method does not contain an infinite loop. Verifying if this constraint is true is similar to the Halting Problem and therefore not generally computable [Unknowable]. A verifier has to omit the check and pass potentially unsafe class files.

For another example, consider algorithm 11 below.

Algorithm 11 Rejected class

```
public static int always_true()
Code(max_stack = 1, max_locals = 1, code_length = 2)
0:  iconst_1  ; push constant 1 onto stack
1:  ireturn   ; return constant 1 ("true")

public static void good_method()
0:  invokestatic NewClass0.always_true ()I (18)
                     ; Push "true" on stack
3:  ifne #10          ; If "true" is on stack jump to 10
6:  pop               ; Pop a value off the stack
7:  goto #6           ; jump to 6
10:return            ; complete method
```

This code is harmless, because lines 6 and 7 can never be executed (it would underflow the operand stack in an infinite loop). A class file with this code is rejected by JustIce and other verifiers, because the endless loop seems to be a malicious threat to the integrity of the JVM.

We conclude that there cannot be a perfect verifier. All that could be done is reduce the degree of uncertainty. For practical purposes, i.e., to be compatible with Sun's implementation, one should not even do that.

There is also a simple proof showing a perfect verifier does not exist in [JNS], chapter 6. It uses a diagonalization argument.

6.3 Future Work

Class file verification is an integral component of Java security; and application programs running on the Java Virtual Machine are often used in security critical areas. Several security holes and flaws have been found both in implementations and the specification of the Java class file verifier since it was introduced.

Recently, the area has experienced a leap as a theoretically founded, sound and complete Java environment was defined in [JBook]. Possibly Sun's engineers will use this work to improve Java and the Java verifier. JustIce will have to change to always keep close to the industry standard.

But JustIce itself can also be improved concerning practicability, and new software can be developed on top of the Verification API.

6.3.1 Improvements to JustIce

Introduction of Unique Identifers for Verification Results and Warning Messages

Currently, warning messages and verification results are conceptually text-based. Only *VerificationResult* objects include a numeric value which programs can use to decide if some class verification failed or not. A program like the prototype introduced in section 5.3 can currently not hide specific messages from the user without parsing text. This limitation should be removed in the future by using unique message numbers. This would also make translation of the messages into other languages easier.

A New Verification Strategy

The core verification algorithm cited in section 3.3.2 works by generalizing the knowledge about an object type along the inheritance hierarchy.

For instance, let there be an object of type `java.util.AbstractList` on the simulated stack of some modeled instruction. Let there be a loop so that the algorithm has to visit that same instruction again, this time with an object of type `java.util.AbstractSet` in that same stack slot. The verifier will compute the meet of the two types and record that there is some object of type `java.util.AbstractCollection` in that stack slot.

Remember that the instruction will be marked with a *changed* bit until no such re-typing change occurs any more (JustIce will actually put it into a queue).

This approach does not work very well when it comes to interface types instead of class files. For example, the meet of a `java.lang.Integer` and a `java.lang.Double` is a `java.lang.Number` because `java.lang.Number` is the first common super class. Both classes also implement the `java.lang.Comparable` interface, but `java.lang.Number` does not. This information is lost when replacing the type information. However, current verifiers do not reject the class files but make additional run-time checks necessary.

Fong noticed that this could be the reason for the `invokeinterface` opcode to be underspecified [Fong2-WWW] (also see section 7.2.1).

Stärk et al. suggest the use of *sets* of reference types instead ([JBook], pages 229-231). This could also be implemented in JustIce.

Keeping up with Specification Clarifications

As a clean-room implementation, JustIce depends on the clearness of the specification. Ambiguities could lead to programming errors.

Here we give one example: methods can be inherited in Java (for example, the method *clone()* is declared in the `java.lang.Object` class and therefore inherited by every other class).

Let a class `A` be a subclass of `java.lang.Object` and let class `B` be a subclass of `A`. Also, let class `B` override the definition of *clone()* with an own implementation.

If *javac* compiles a Java program that invokes this method, it is either referenced as *java.lang.Object::clone()* or as *B::clone()*. However, because `A` inherits this method, the reference *A::clone()* is legal, too.

In The Java Virtual Machine Specification, Second Edition ([VMSPEC2], page 291) it is said that the reference must be a "symbolic reference to the class in which the method is to be found". Statically, the method *clone()* can of course not be found in class `A`. One could therefore think the reference *A::clone()* was not legal.

In the meanwhile, Sun's engineer Gilad Bracha clarified this issue: "Of course. This is discussed in JVMS 5.4.3.4, which describes interface method resolution. I don't see the text on page 280 as contradicting that. The symbolic reference does give an interface in which the required method can be found, albeit as an inherited member. We could try and reword it in a more precise way, to eliminate any misunderstandings."

Keeping up with clarifications like this is an inevitable and on-going part of the development of JustIce.

Keeping up with Java Extensions

Recently, Sun Microsystems introduced a new attribute: the `StackMap` attribute which is an attribute local to the `Code` attribute (see section *2.1.1*). It was specified in [J2ME-CLDCS].

It is there to provide "limited devices" that perform a one-pass verification with type information that would normally have to be inferred by the verifier.

It is not used by the verification algorithm of JustIce now: it's currently an *unknown attribute* to JustIce.

Detecting Local Variable Accesses out of Scope

The `LocalVariableTable` attribute is a debug information attribute. Basically, it gives debuggers information about the original (source code) name and type of a given local variable.

JustIce builds data structures to warn if it detects contradicting and overlapping areas; e.g., if some local variable is anounced to carry an `int` value and a `float` value at the same time.

It could also be interesting to warn if a local variable is accessed for which no debug information exists. This is currently not implemented.

Extending the Verification API

JustIce can easily be extended to run certain analyses related to symbolic bytecode execution.

This includes the computation of the maximum number of used operand stack slots in a method or the computation of unused local variables in a method.

These analyses are normally costly to implement[3], but they are a waste product of the verifier's core algorithm.

A Verifier Validation Suite

The Kimera project [Kimera-WWW] was the first known project to implement a stand-alone Java verifier. The people behind the project had to test the behaviour of their verifier against the behaviour of the previous implementations. Tests have been run in order to validate the Kimera verifier. These tests range from simply introducing random one-byte errors into class files and automatically running Kimera against other verifiers to elaborate research work [Kimera-ProdGram, Kimera-TestingJVM].

Currently, JustIce comes only with a very limited possibility of running test cases against the native verifier of the host machine's JVM. The pioneering work of the Kimera project could be used to implement a validation suite for JustIce.

6.3.2 A Verifier Protecting an Intranet

Often, Java Virtual Machines are built into software used to browse the World Wide Web such as the KDE project's *Konqueror [KDE]* or Mozilla.org's *Mozilla* [Mozilla] products. Such Internet technology is also often used in corporate networks. Corporate networks based on internet technology are called *intranets*; these networks are normally protected from the Internet by a so-called *firewall* computer.

[3] Often, heuristics are used such as the method MethodGen.getMaxStack() in the BCEL [BCEL-WWW, BCEL98].

This computer's task is to provide access to the internet only to privileged employees and –even more important– it blocks access from unauthorized persons outside the intranet. The firewall machine is a single, bi-directional point of access.

However, normally web-browsing is considered harmless, so that the employees can unrestrictedly gather information, possibly visiting Java-enabled web sites. The JVMs built into the browser software run software downloaded from the World Wide Web; while the the built-in verifiers make sure that no dangerous code can be executed.

Let us assume someone discovered a security hole in the verifier implementation or implementations that are used on the corporate network's workstations; let us also assume a patch exists that would fix the problem.

A system administrator would have to spent a lot of time to repair every single verifier. A cheaper solution would be a verifier built into the firewall machine; such a verifier can easily be implemented using JustIce and its Verification API.

6.3.3 A Java Virtual Machine Implementation Using JustIce

The Java verifier is originally a part of the Java Virtual Machine. JustIce could also be part of a Java Virtual Machine. JustIce's class files (the program code JustIce consists of) could simply be integrated into the core Java class files. The execution engine would then run JustIce without actually verifying JustIce's class files themselves.

For scientific purposes one could also implement a JVM in the Java programming language. Such an implementation could, for example, serve as a debugger.

6.3.4 Drawing a Clear Line Between the Principle of Information Hiding and Security

The principle of information hiding has been (and still is!) a practice of experienced programmers for many years. It is there to reduce programming errors.

In the Modula-2 programming language [M2] this is achieved by explicitely dividing the program code in definition modules and implementation modules. In older programming languages, such as in the C programming language [C], this principle is implicitely used, too. Basically this is achieved by defining interfaces that only describe what the code of a program module does. These interface "headers" are included into user code instead of simply including the code itself.

In object-oriented programming languages such as in Delphi [D3], C++ [CPP-D, CPP-E] or Java [langspec2], this principle is refined to what is called object encapsulation. When a class is defined, certain key words such as `private`, `protected`, `friend`, `public`, `published` set the access rules for the members[4] of an object of the given class.

[4]The members of a class are its components: methods (program code) and fields (also called attributes or variables).

Still, this refined technique does not have anything to do with security. It is only there to aid programmers create a reasonable design. If every piece of code could manipulate every data structure, one would not know where to look for a programming error in the program source code. On the other hand, if some field is private in C++, one could (with some knowledge about the compiler used) still reference and modify this field by pointer manipulation. In addition to that, a second program like a debugger could watch even the data of private fields.

However, when a Java program is compiled into the language of the JVM, the information about the access rights of the fields and methods is included. This is where the principle of information hiding is exploited to provide security. For example, the verifier of the JVM has to make sure private fields are never accessed from a foreign piece of code. But there are many implementations of the JVM which have security flaws such as not honouring the access rights. There are debuggers for JVM bytecodes, too.

When one thinks about security, one has to think of some enemy who could try to harm the computer or information stored on that computer. From a JVM user's point of view, the JVM is relatively secure. Even running untrusted code cannot do much harm. Because the security flaws in different JVM implementations differ, they are probably not exploited most times.

From a Java programmer's point of view, the JVM is not secure. Untrusted users can do much harm. For example, an online banking application storing important data in Java fields (such as access information to the bank's database management system) is a threat to both the bank and its customers. This information could easily be extracted by a malicious user.

Another problem for Java programmers is the amount of symbolical information stored in class files. Today, it is easy to de-compile a Java class file back to Java language source code [JODE-WWW]. This source code can then be read and analyzed by the user. Facing this problem, the "only safe course of action is to assume that ALL Java code will at some point be decompiled" ([JNS], page 68).

We conclude that the principle of information hiding is not enough to provide a degree of security that both –users and programmers– could accept. Programmers should not believe a good design makes a program *secure*.

7 Appendix

7.1 History of JustIce

The author of JustIce once started to implement a class file decompiler like Jode [JODE-WWW]. It soon became clear that to successfully implement it, one should exploit the "well-behaved" property of class files (which essentially means that they pass a verifier, especially pass three) [Krakatoa-WWW].

JustIce was then developed to understand the "well-behaved" property of usual class files. It took much longer to complete than estimated because of the many inherent bugs and ambiguities in The Java Virtual Machine Specification, Second Edition [VMSPEC2].

Its name starts with a *J* like Java does, referring to the tradition of giving Java-related software such names. The second part of the name, *ICE*, was inspired by a novel by William Gibson [Neuromancer]. It is an acronym for *Intrusion Countermeasures Electronics*, something that is very much like today's firewall systems (see section 6.3.2). He credits the invention of *ICE* to Tom Maddox. The missing three letters were inserted to create a word that makes sense; in fact, choosing the three-letter combination *ust* resulted in the creation of a word with a double sense via bi-capitalization.

JustIce was written using and extending the excellent Byte Code Engineering Library [BCEL-WWW, BCEL98] by Markus Dahm. It really helped a lot and sped up development time.

It was also –last but not least– written to earn its author a German *Dipl.-Inform.* degree which one may compare to a *master* degree.

7.2 Flaws and Ambiguities Encountered

While designing, implementing and testing JustIce, a lot of interesting flaws and ambiguities were found in the specification [VMSPEC2], the Java compiler *javac* and the JVM *java*.

7.2.1 Flaws in the Java Virtual Machine Specification

The Java Virtual Machine Specification, Second Edition was derived from an in-house document describing the as-is implementation of Sun's genuine Java Virtual Machine ([VMSPEC2], page xiv). This sometimes leads to problems as there are still a few points left where Sun's engineers forgot to describe specification details to the public, in error assuming they would be implementation

details. Another source of mistakes are ambiguities, inherent to natural languages auch as English.

A Code Length Maximum of 65535 Bytes per Method

On page 152, The Java Virtual Machine Specification, Second Edition [VMSPEC2] says that code arrays may at most have a length of 65536 bytes because certain indices that point into the code are only 16 bits of width. Page 134 states the code must have „less than“ 65536 bytes. Therefore, the limitation stated on page 152 is not helpful, but only confusing.

Subroutines

The implementation of a provably correct verifier is not possible because of the ambiguities in the specification [VMSPEC2]. To reach this goal, various efforts have been made to describe the verifier and the JVM formally [Qian, StataAbadi, FreundMitchell, JBook, JPaper]. By restricting the code *javac* produces or by redefining the verifier's behaviour, however, they are never one-to-one with the behaviour of the existing JVMs.

Sun's specification does not define the term *subroutine* although it is used. Instead, it is explained what bytecode the Java *compiler* generates when a `finally` clause appears in the Java *language* source code – this definitely does not belong there, because a verifier must never assume the code it verifies was created by Sun's *javac* compiler.

Clarifying this issue could lead to an *official* formal specification.

The Specification Sometimes Satisfies the Verifier

Fong [Fong2-WWW] found in 1997 that the `invokeinterface` opcode was underspecified in the first edition of the Java Virtual Machine Specification. He managed to create a class file that did not implement a specific interface but nevertheless used `invokeinterface` to invoke a method. This class file passed the verifier (up to pass three), but the JVM found the problem during run-time (pass four). Fong concluded that the omission in the specification was done on purpose because the implementation of the data flow analyzer does not allow to check this constraint (please see section 6.3.1 for a description of how this limitation could be overcome). However, in The Java Virtual Machine Specification, Second Edition [VMSPEC2], the specification of `invokeinterface` is corrected.

Still, there is another case where one would suspect the specification describes the behaviour of the verifier: on pages 147 and 148 of the specification [VMSPEC2], verification of instance initialization methods and newly created objects is explained. “A valid instruction sequence must not have an uninitialized object on the operand stack or in a local variable during a backwards branch, or in a local variable in code protected by an exception handler or a `finally` clause”. Note that the Java language keyword `finally` does not really belong here (Sun should speak of *subroutines*), but more important is that this

specification is made to satisfy the verification algorithm: "Otherwise, a devious piece of code might fool the verifier".

The '$' Character as a Valid Part of a Java Name

Because the *javac* compiler may create class files with a '$' character in their names as a result of Java source files defining inner classes, this character should no longer be a valid part of a Java name to avoid problems. I.e., the method invocation *java.lang.Character.isJavaIdentifierPart('$');* should return the value `false`.

7.2.2 Flaws in the Implementation of the *Java Platform*

Sun's Verifier Rejects Code Produced by Sun's Compiler

Surprisingly, there are a number of examples in which such a thing happens.

Another Problem With Subroutines In [JPaper], Stärk and Schmid give a few code examples which are compiled correctly by the *javac* compiler but the resulting code is rejected by the traditional verifiers. Algorithms 12 and 13 show one of their examples given in the Java programming language and the resulting output of the *javac* compiler.

Algorithm 12 Stärk and Schmid's Rejected Class, Java Language Version

```
class Test1{
    int test(boolean b){
      int i;
      try{
        if (b) return 1;
        i=2;
      }
      finally {
        if (b) i = 3;
      }
      return i;
    }
}
```

If one tries to run this bytecode using a JVM by IBM Corporation, the code is rejected[1]:

```
ehaase@haneman:/home/ehaase > java Test1
  Exception in thread "main" java.lang.VerifyError:
  (class:  Test1, method:  test signature:  (Z)I)
  Localvariable 2 contains wrong type
```

In his lectures, Stärk explains that the problem lies in the polymorphic nature of JVM subroutines [JLectures]. Consider algorithm 13. In line 12, an `int`

[1] It is also rejected by Sun's JVMs and the Kimera verifier [Kimera-WWW].

Algorithm 13 Stärk and Schmid's Rejected Class, JVM Bytecode Version

```
int test(boolean arg1)
Code(max_stack = 1, max_locals = 6, code_length = 39)
0:  iload_1
1:  ifeq #11
4:  iconst_1
5:  istore_3
6:  jsr #27
9:  iload_3
10:  ireturn
11:  iconst_2
12:  istore_2
13:  jsr #27
16:  goto #37
19:  astore %4
21:  jsr #27
24:  aload %4
26:  athrow
27:  astore %5
29:  iload_1
30:  ifeq #35
33:  iconst_3
34:  istore_2
35:  ret %5
37:  iload_2
38:  ireturn
```

is put into local variable number 2. The subroutine starting at line 27 is then called from line number 13. Note that this subroutine accesses the local variable number 2. Finally, line 16 transfers control to line 37 where the verification problem occurs. An `int` should be read from local variable number 2, but this is marked `unusable`, because it was accessed in the subroutine.

However, the specification ([VMSPEC2], page 151) states:

- For any local variable that [...] has been accessed or modified by the subroutine, use the type of the local variable at the time of the `ret`.
- For any other local variables, use the type of the local variable before the `jsr` instruction.

As one can see, in the above example local variable number 2 holds an `int` data type in both cases; there is no need to mark it `unusable`. This is the reason why JustIce does not reject the above bytecode, thus being slightly incompatible with the behaviour of other verifiers.

The Maximum Method Length May Be Exceeded The *javac* compiler Sun included in the Java Development Kit version 1.3.0_01 does not check for the maximum method length of the `code` array in a `Code` attribute (see section 2.1.1). A test file containing 65000 lines like "`System.out.println("Test");`" was compiled, but the resulting class file was rejected by the verifier.

IBM Corporation's *jikes* compiler does not even generate code, but it locks up while compiling the test file.

A Compiler Issue Related to Inner Classes

The *javac* compiler has to name class files, even those of so-called anonymous classes [InnerSpec].

This can cause problems: an inner class *I* defined in a class *A* will be compiled into a class file called *A$I.class*. A Java class named *A$I* will also be compiled into a class file named *A$I.class* overwriting the former class file. Because Sun did not forbid the '*$*' character as a legal part of a Java identifier, the *javac* compiler should use a more sophisticated naming scheme.

Pass Four is Only Partially Implemented

Pass four defines run-time tests for constraints that could also be verified in pass three; it is only for performance reasons that these tests are delayed. Instead of having all the tests in one place, they are unnecessarily spread "making the validation of the verification algorithm itself extremely difficult" [Fong-WWW]. Risking security for better performance is often regarded as a bad decision. For instance, in the

```
java version "1.3.0_01"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.0_01)

Java HotSpot(TM) Client VM (build 1.3.0_01, mixed mode)
```

Java Virtual Machine, the pass four check for access rights was unintentionally omitted. Sadly, other vendors license Sun's code and base their own implementations on that code. Therefore, mistakes are often inherited throughout the JVM vendors. The

```
java version "1.3.0"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.0)
Classic VM (build 1.3.0, J2RE 1.3.0 IBM build cx130-20010626 (JIT
enabled: jitc))
```

Java Virtual Machine by IBM Corporation, for example, exposes the same mistake.

7.3 Related Work

7.3.1 The Kimera Project

It is a misfortune that the Kimera [Kimera-WWW] project closed the World Wide Web presence and that the source code of the Kimera verifier was never released – it would have been quite interesting to see how that respected verifier implementation deals with the problems arising concerning subroutine verification.

However, Kimera is the single other stand-alone verifier besides JustIce the author knows of. The people behind the project found important security breaches in JVM implementations of various World Wide Web browsers.

Also, they validated their verifier implementation and published several papers on JVM implementation verification [Kimera-ProdGram, Kimera-TestingJVM].

7.3.2 The Verifier by Stärk, Schmid and Börger

In [JBook], the authors define the Java programming language and the Java virtual machine formally using *Abstract State Machines* (ASM). This also includes the verifier; its specifications have also been implemented in the functional programming language AsmGofer [AsmGofer]. This implementation is included on the CD-ROM that accompanies the book.

The "*JBook verifier*" does not implement a complete class file verifier. It currently only implements the bytecode verification. Its input files are not class files itself, but a textual representation of class files in so-called Jasmin format [JVM]. Therefore, this implementation is merely of theoretical interest.

It does, however, implement a bytecode verifier that is founded on a *solid* theory. This theory could become the standard for the interpretation of the JVM specification [VMSPEC2]. It could even change the specification to remove its ambiguities.

There is also an unreleased version of this verifier implemented in the Java programming language using the BCEL. This implementation, if it should ever be released, promises a lot as it could combine usability and a solid theory.

7.4 The GNU General Public License

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights.

These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it. For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps:

(1) copyright the software, and

(2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you". Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program. You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.) These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole

which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it. Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program. In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.) The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable. If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or

any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program. If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances. It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice. This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Glossary

Access modifiers In the Java programming language, the use of the keywords `private`, `protected`, `public` (or the use of no keyword) defines the access rights for data or program code (also called visibility). This information is also used by the JVM: it is part of the class files. The most important modifier is `private` which is used to globally deny access to a field or method.

Access rights Access rights are granted or denied by the use of ▷access modifiers.

API Applications Programming Interface. Such an interface is used to include functionality of foreign program modules (often Java ▷packages) into own programs.

Debugger A program used to investigate the behaviour of another program. Often used to find and remove programming errors, so-called bugs.

Descriptor A symbolic description of type information. In the JVM's class files, strings in UTF-8 format [Unicode] are used to describe type information.

Field A member of a Java object or class, also called variable or attribute.

Method A member of a Java object or class. Methods include program code or they are abstract representatives for program code. A method can be compared to a *function* in programming languages like C or Pascal.

Opcode Operation Code. This denotes an instruction in an assembly-like computer language; to some people it means its binary representation.

Package A package is an entity used in both the Java programming language and the Java Virtual Machine definition. It is used to group classes that in the eyes of the programmer belong together. Package definitions have impact on ▷access rights granted to other classes.

Signature A method has a (possibly empty) set of arguments it expects, and it has a return type (possibly the `void` type). The type information of the arguments and the return type together is called signature. A signature can be expressed in terms of a ▷descriptor.

Type A field or a method argument has a type such as `int` or `String`. In the JVM's context, all values are typed. Types can be expressed in terms of a ▷descriptor.

List of Figures

List of Algorithms

Bibliography

[AppMag-WWW] AverStar's AppletMagic(tm): Ada for the Java Virtual Machine.
http://www.appletmagic.com

[AsmGofer] Joachim Schmid: AsmGofer.
http://www.tydo.org

[BCEL98] Markus Dahm: Byte Code Engineering with the BCEL API. Freie Universität Berlin, Institut für Informatik. Technical Report B-17-98.

[BCEL-WWW] Markus Dahm: Byte Code Engineering Library.
http://bcel.sourceforge.net

[BCV-Soundness] Cornelia Pusch: Proving the Soundness of a Java Bytecode Verifier Specification in Isabelle/HOL. Technische Universität München, Institut für Informatik.
http://www.in.tum.de/~pusch/

[C] Brian W. Kerninghan, Dennis M. Ritchie: The C Programming Language, Second Edition, ANSI C. Prentice-Hall 1998, ISBN 0131103628.

[CPP-D] Bjarne Stroustrup: Die C++ Programmiersprache. Addison-Wesly-Longman, 1998, ISBN 3-8273-1296-5.

[CPP-E] Bjarne Stroustrup: The C++-Programming Language, Third Edition. Addison-Wesley 1997, ISBN 0-201-88954-4.

[D3] Guido Lang, Andreas Bohne: Delphi 3.0 lernen. Addison-Wesley-Longman 1997, ISBN 3-8273-1190-x.

[DesignPatterns] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley 1995, ISBN: 0201633612.

[DragonBook] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: Compilers: Principles, Techniques, and Tools. Addison-Wesley 1985, ISBN: 0201100886.

[EF] ElectricalFire.
http://www.mozilla.org/projects/ef/

[f2j] Keith Seymour: f2j - Fortran-to-Java Compiler.
http://cs.utk.edu/f2j/

[Fong-WWW] Philip W. L. Fong: The mysterious Pass One, first draft, September 2, 1997.
http://www.cs.sfu.ca/people/GradStudents/pwfong/personal/JVM/pass1/

[Fong2-WWW] Philip W. L. Fong: A Flaw with the Specification of the Invokeinterface Opcode.
http://www.cs.sfu.ca/people/GradStudents/pwfong/personal/JVM/invokeinterface/

[FreundMitchell] Stephen N. Freund, John Mitchell: A Formal Framework for the Java Bytecode Language and Verifier. Department of Computer Science, Stanford University. Stanford, CA 94305-9045. Appeared in OOPSLA '99.

[GCC-WWW] GCC, The GNU compiler collection.
http://gcc.gnu.org

[GJ-WWW] GJ. A Generic Java Language Extension.
http://www.cis.unisa.edu.au/~pizza/gj/

[InnerSpec] Sun Microsystems: Inner Classes Specification.
http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html

[J2ME-CLDCS] Sun Microsystems: J2ME™ Connected Limited Device Configuration Specification.
http://jcp.org/aboutJava/communityprocess/final/jsr030/

[JBook] Robert Stärk, Joachim Schmid, Egon Börger: Java™ and the Java™ Virtual Machine. Springer-Verlag 2001, ISBN 3-540-42088-6.
http://www.inf.ethz.ch/~jbook/

[JPaper] Robert F. Stärk, Joachim Schmid: Java bytecode verification is not possible. ETH Zürich, Department of Computer Science 2000.
http://www.inf.ethz.ch/~staerk/pdf/jbv00.pdf

[JLectures] Robert F. Stärk: Java and the JVM: Definition and Verification (37-474).
http://www.inf.ethz.ch/~jbook/eth37474/
http://www.inf.ethz.ch/~jbook/eth37474/javaBV.pdf

[JNS] Robert Macgregor, Dave Durbin, John Owlett, Andrew Yeomans: JAVA™ Network Security. Prentice Hall 1998, ISBN 0137615299.

[JODE-WWW] JODE is a java package containing a decompiler and an optimizer for java.
http://jode.sourceforge.net

[JustIce] Enver Haase: JustIce. A Free Class File Verifier for Java™ .Freie Universität Berlin, Takustraße 9, D-14195 Berlin; September 2001.
http://bcel.sourceforge.net/
http://bcel.sourceforge.net/justice

[JVM] Jon Meyer, Troy Downing: JAVA Virtual Machine. O'Reilly 1997, ISBN 1-56592-194-1.

[Kaffe-WWW] Kaffe. Kaffe is a cleanroom, open source implementation of a Java virtual machine and class libraries.
http://www.kaffe.org

[KAWA-WWW] Kawa, the Java-based Scheme system.
http://http://www.gnu.org/software/kawa/

[KDE] KDE, the K desktop environment.
http://www.kde.org

[Kimera-WWW] The Kimera Verifier.
Currently off-line because of a World Wide Web presentation rework.
http://kimera.cs.washington.edu/verifier.html
http://www-kimera.cs.washington.edu

[Kimera-TestingJVM] Emin Gün Sirer: Testing Java Virtual Machines. An Experience Report on Automatically Testing Java Virtual Machines. University of Washington, Dept. of Computer Science and Engineering.
http://kimera.cs.washington.edu

[Kimera-ProdGram] Emin Gün Sirer, Brian N. Bershad: Using Production Grammars in Software Testing. University of Washington, Department of Computer Science.
http://kimera.cs.washington.edu

[kissme-WWW] kissme. A free Java Virtual Machine.
http://kissme.sourceforge.net

[Krakatoa-WWW] Todd A. Proebsting, Scott A. Watterson: Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?). The University of Arizona, Department of Computer Science.

http://www.cs.arizona.edu/people/saw/papers/Krakatoa-COOTS97.ps.Z

[langspec2] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: The Java Language Specification, Second Edition. Addison-Wesley 2000, ISBN 0201310082.

[M2] Niklaus Wirth: Programming in Modula-2, Fourth Edition. Springer-Verlag 1988, ISBN 3-540-50150-9.

[Mozilla] Mozilla.org (The Mozilla Origanization): Mozilla.
http://www.mozilla.org

[Neuromancer] William Gibson: Neuromancer. Ace Books 1994, ISBN 0441000681.

[ORP-WWW] Open Runtime Platform. A Platform For Bytecode System Research.
http://www.intel.com/research/mrl/orp/index.htm

[PL4JVM] Robert Tolksdorf: Programming Languages for the Java Virtual Machine.
http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html

[PMG-WWW] PMG. Poor Man's Genericity for Java.
http://www.inf.fu-berlin.de/~bokowski/pmgjava/index.html

[Qian] Zhenyu Qian: A Formal Specification of Java™ Virtual Machine Instructions for Objects, Methods and Subroutines. Bremen Institute for Safe Systems (BISS), FB3 Informatik, Universität Bremen, D-28334 Bremen, Germany.

[SableVM-WWW] SableVM. A Bytecode Interpreter.
http://www.sablevm.org

[StataAbadi] Raymie Stata and Martin Abadi: A Type System for Java Bytecode Subroutines. In: ACM Transactions on Programming Languages and Systems, Vol. 21, No. 1, January 1999, Pages 90-137.

[Unknowable] G.J. Chaitin: The Unknowable. Springer-Verlag 1999, ISBN 981-4021-72-5.
http://www.umcs.maine.edu/~chaitin/unknowable/

[Unicode] The Unicode Consortium: The Unicode Standard, Version 2.0. Niso Press 1996, ISBN 0-201-48345-9.
http://www.unicode.org

[Yellin-WWW] Frank Yellin: Low Level Security in Java.
http://java.sun.com/sfaq/verifier.html

[VMSPEC2] Tim Lindholm, Frank Yellin: The Java™ Virtual Machine Specification, Second Edition. Addison-Wesley 1999, ISBN 0-201-43294-4.