Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 264 (4) (2011) 35–50
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Encoding the Java Virtual Machine's Instruction Set

Michael Eichberg[1] Andreas Sewe[2]

Department of Computer Science
Technische Universität Darmstadt
Germany

Abstract

New toolkits that parse, analyze, and transform Java Bytecode are frequently developed from scratch to obtain a representation suitable for a particular purpose. But, while the functionality implemented by these toolkits to read in class files and do basic control- and data-flow analyses is comparable, it is implemented over and over again. Differences manifest themselves mainly in minor technical issues. To avoid the repetitive development of similar functionality, we have developed an XML-based language for specifying bytecode-based instruction sets. Using this language, we have encoded the instruction set of the Java Virtual Machine such that it can directly be used, e.g., to generate the skeleton of bytecode-based tools. The XML format hereby specifies both the format of the instructions and their effect on the stack and the local registers upon execution. This enables developers of static analyses to generate generic control- and data-flow analyses, e.g., an analysis that transforms Java Bytecode into static single assignment form. To assess the usefulness of our approach, we have used the encoding of the Java Virtual Machine's instruction set to develop a framework for the analysis and transformation of Java class files. The evaluation shows that using the specification significantly reduces the development effort when compared to manual development.

Keywords: Java Bytecode, Java Virtual Machine Specification, XML

1 Introduction

The development of programs that parse and analyze Java Bytecode [9] has a long history and new programs are still developed [2,3,4,7,13]. When developing such tools, however, a lot of effort is spent to develop a parser for the bytecode and for (re-)developing standard control- and data-flow analyses which calculate, e.g., the control-flow graph or the data-dependency graph.

To reduce these efforts, we have developed a specification language (OPAL SPL) for encoding the instructions of stack-based intermediate languages. The idea is that—once the instruction set is completely specified using OPAL SPL—generating both bytecode parsers and standard analyses is much easier than their manual

[1] E-mail: `eichberg@informatik.tu-darmstadt.de`
[2] E-mail: `sewe@st.informatik.tu-darmstadt.de`

doi:10.1016/j.entcs.2011.02.004

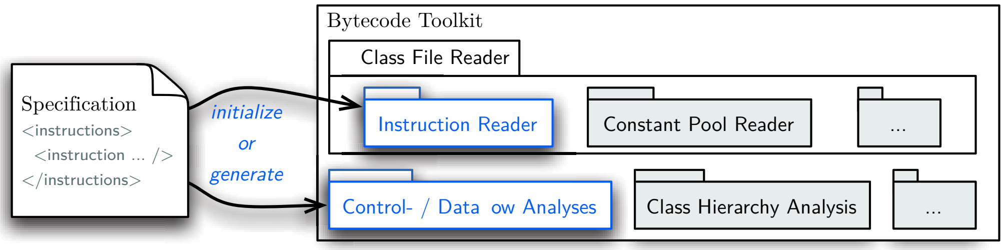

Fig. 1. Use Cases for OPAL SPL Specifications

development. To support this goal, OPAL SPL supports the specification of both the format of bytecode instructions and the effect on the stack and registers these instructions have when executed. An alternative use of an OPAL SPL specification is as input to a generic parser or to generic analyses as illustrated by Fig. 1.

Though the language was designed with Java Bytecode specifically in mind and is used to encode the complete instruction set of the Java Virtual Machine (JVM), [3] we have striven for a Java-independent specification language. In particular, OPAL SPL focuses on specifying the instruction set rather than the complete class file format, not only because the former's structure is much more regular than the latter's, but also because a specification of the instruction set promises to be most beneficial. Given the primary focus of OPAL SPL—generating parsers and facilitating basic analyses—we explicitly designed the language such that it is possible to group related instructions. This makes specifications more concise and allows analyses to treat similar instructions in nearly the same way. For example, the JVM's iload_5 instruction, which loads the integer value stored in register #5, is a special case of the generic iload instruction where the instruction's operand is 5. We also designed OPAL SPL in such a way that specifications do not prescribe how a framework represents or processes information; i.e., OPAL SPL is representation agnostic.

The next section describes the specification language. In Section 3 we reason about the language's design by discussing the specification of selected JVM instructions. In Section 4 the validation of specifications is discussed. The evaluation of the approach is presented in Section 5. The paper ends with a discussion of related work and a conclusion.

2 Specifying Bytecode Instructions

The language for specifying bytecode instructions (OPAL SPL) was primarily designed to enable a concise specification of the JVM's instruction set. OPAL SPL supports the specification of both an instruction's format and its effect on the stack and local variables (registers) when the instruction is executed. It is thus possible to specify which kind of values are popped from and pushed onto the stack as well as which local variables are read or written. Given a specification of the complete instruction set the information required by standard control- and data-flow analyses

[3] The complete specification is available at http://www.michael-eichberg.de/opal.

is then available.

However, OPAL SPL is not particularly tied to Java as it abstracts from the particularities of the JVM Specification. For example, the JVM's type system is part of an OPAL SPL specification rather than an integral part of the OPAL SPL language itself.

Next, we first give an overview of the language before we discuss its semantics.

2.1 Syntax

```
1. INSTRUCTIONS ::= instructions < TYPES EXCEPTIONS FUNCTIONS INSTRUCTION+ >
2. TYPES ::= types < TYPEDEF > // a common root type is required
3. TYPEDEF ::= type @name @pc? < TYPEDEF* >
4. EXCEPTIONS ::= exceptions < (exception @type)+ >
5. FUNCTIONS ::=
     functions (< function @name < signature < (param @type)* (returns @type) > > >)*
6. INSTRUCTION ::=
     instruction @mnemonic @deprecated? @transfers_control?
       < appinfo < /APPLICATIONSPECIFICCONTENT* >
         format < SEQUENCE+ >
         ( stack < (form < before < BEFOREOP+ > after < AFTEROP+ > >)+ > )?
         ( registers < LOAD? STORE? > )?
         ( exceptions < (exception @type)+ > )? >
7. SEQUENCE ::= sequence
     (  SEQELEM | padding_bytes @alignment | list @count @var < SEQELEM+ > |
        (implicit @var < /VALUEEXPRESSION >)  | (implicit_type @type < /TYPEEXPRESSION >) )+
8. SEQELEM ::= { u1 | u2 | u4 | i1 | i2 | i4 } @type? @var? < /EXPECTEDVALUE? >
9. BEFOREOP ::= ((operand @type @var?) | (list @loop_var? @count < BEFOREOP >) )* rest?
10.AFTEROP ::= (operand @type < /VALUEEXPRESSION? >) rest?
11.LOAD ::= load @type @index @var
12.STORE ::= store @type @index < /VALUEEXPRESSION >
```

Fig. 2. Grammar of the OPAL Specification Language (OPAL SPL)

The OPAL Specification Language (OPAL SPL) is an XML-based language. Its grammar is depicted in Fig. 2 using an EBNF-like format. Non-terminals are written in capital letters (INSTRUCTIONS, TYPES, etc.), the names of XML-elements are written in small letters (types, stack, etc.) and the names of XML-attributes start with "@" (@type, @var, etc.). We refer to the content of an XML-element using symbols that start with"/" (/VALUEEXPRESSION, /EXPECTEDVALUE, etc.). "<>" is used to specify nesting of elements. "(),?,+,*,{},|" have the usual semantics. For example, `exceptions` < `(exception @type)+` > specifies that the XML-element `exceptions` has one or more `exception` child elements that always have the attribute `type`.

2.2 Semantics

Format Specification

Each specification written in OPAL SPL consists of four major parts (line 1 in Fig. 2). The first part (types, lines 2–3) specifies the type system that is used by the underlying virtual machine. The second part (exceptions, line 4) declares the exceptions that may be thrown when instructions are executed. The third part (functions, line 5) declares the functions that are used in instruction specifications. The fourth part is the specification of the instructions themselves (lines 6–12), each of which may resort to the declared functions to access information not simply stored along with the instruction. For example, invoke instructions do not store the signature and declaring class of the called methods. Instead, a reference to an entry in the so-called constant pool is stored. Only this constant pool entry has all information about the method. To obtain, e.g., the return type of the called method, an abstract function TYPE methodref_return_type(methodref) is declared that takes a reference to the entry as input and returns the method's return type. Using abstract function declarations, we abstract—in the specification of the instructions—from the concrete representation of such information by the enclosing bytecode toolkit.

The specification of an instruction consists of up to four parts: the instruction's format (lines 7–8), a description of the effect the instruction has on the stack when executed (lines 9–10), a descriptions of the registers it affects upon execution (lines 11–12), and information about the exceptions that may be thrown during execution (end of line 6). An instruction's format is specified by sequences which describe how an instruction is stored. The u1, u2 and u4 elements (line 8) of each format sequence specify that the current value is an unsigned integer value with 1, 2 and 4 bytes, respectively. Similarly, the i1, i2 and i4 elements (line 8) are used to specify that the current value is a (1, 2 or 4 byte) signed integer value. The values can be bound to variables using the var attribute and can be given a second semantics using the type attribute. For example, <i2 type="short" var="value"/> is a two-byte signed integer value that is bound to the variable value and has type short with respect to the instruction set's type system. Additionally, it is possible to specify expected values (line 8). This enables the selection of the format sequence to be used for reading in the instruction. E.g., <sequence><u1 var="opcode">171</u1>... specifies that this sequence matches if the value of the first byte is 171. A sequence's list element is used to specify that a variable number of values need to be read. The concrete number of elements is determined by the count attribute. The attribute's value is an expression that can use values that were previously assigned to a variable. The sequence elements implicit and implicit_type are used to bind implicit value and type information to variables that can later on be used in type or value expressions (line 7, 10 and 11). To make it possible to aggregate related bytecode instructions to one logical instruction, several format sequences can be defined. The effect on the stack is determined by the number and type of stack operands that are popped (line 9) and pushed (line 10). If multiple stack layouts are specified, the effect on the stack is determined by the first *before-execution* stack layout that matches; i.e., to determine the effect on the stack a data-flow analysis is necessary.

Unique Prefix Rule

One constraint placed upon specifications written in OPAL SPL is that a format sequence can be identified unambiguously by only parsing a prefix of the instruction; no lookahead is necessary. In other words, if each format sequence is considered a production and each u1, u2, etc. is considered a terminal, then OPAL SPL requires the format sequences to constitute an LR(0) grammar.[4] This unique prefix rule is

[4] Note that OPAL SPL does not have a notion of non-terminal; thus, the grammars are actually weaker than LR(0). Also, the list element (cf. Sec. 3.8) is allowed only *following* a unique prefix.

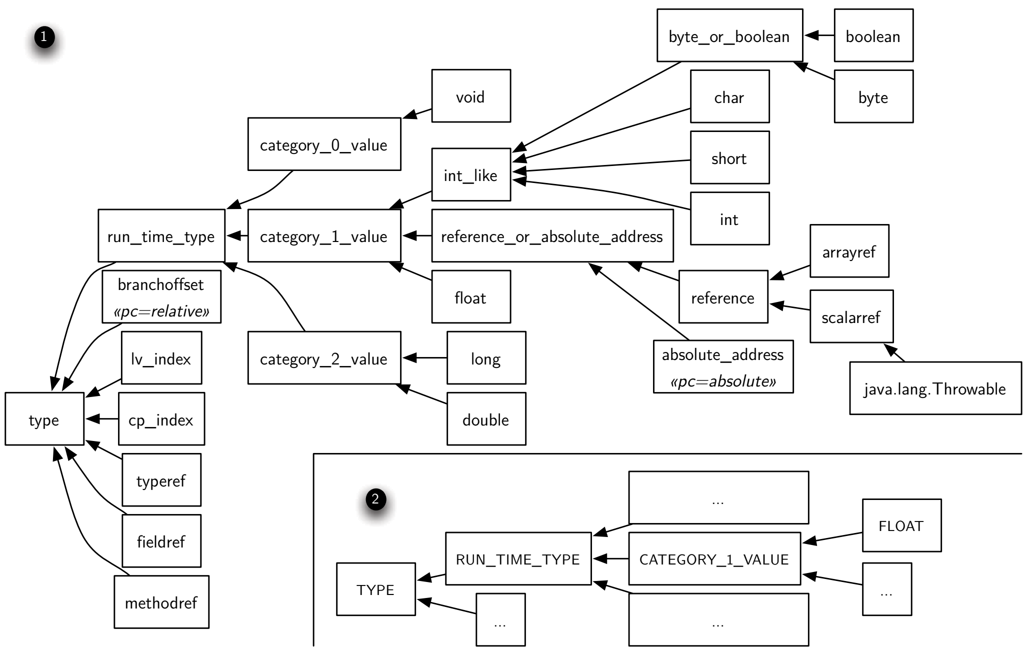

Fig. 3. OPAL SPL Type System

checked automatically (cf. Sec. 4); furthermore, this rule facilitates generating fast parsers from the specification, e.g., using nested **switch** statements.

Type System

OPAL SPL does not have a hard-coded type hierarchy. Instead, each specification written in SPL contains a description of the type system used by the bytecode language being described. The only restriction is that all types have to be arranged in a single, strict hierarchy.

The Java Virtual Machine Specification [9]'s type hierarchy is shown in Fig. 3 (1). It captures all runtime types known to the Java virtual machine, as well as those types that are used only at link- or compile-time, e.g., branchoffset, fieldref and methodref. The hierarchy is a result of the peculiarities of the JVM's instruction set. The byte_or_boolean type, e.g., is required to model the **baload** and **bastore** instructions, which operate on arrays of **byte** or **boolean** alike.

OPAL SPL's type system implicitly defines a second type hierarchy ((2) in Fig. 3). The declared hierarchy of types (1) is mirrored by a hierarchy of kinds (2); for every (lower-case) type there automatically exists an (upper-case) kind. This convention ensures their consistency and keeps the specification itself brief. The values of kind INT_LIKE are int, short, etc., just as the values of type int_like are 1, 2, etc. Kinds enable parameterizing logical instructions like areturn with types, thus making a concise specification of related instructions (e.g., **freturn**, **ireturn**, and **areturn**) possible (cf. Sec. 3.12).

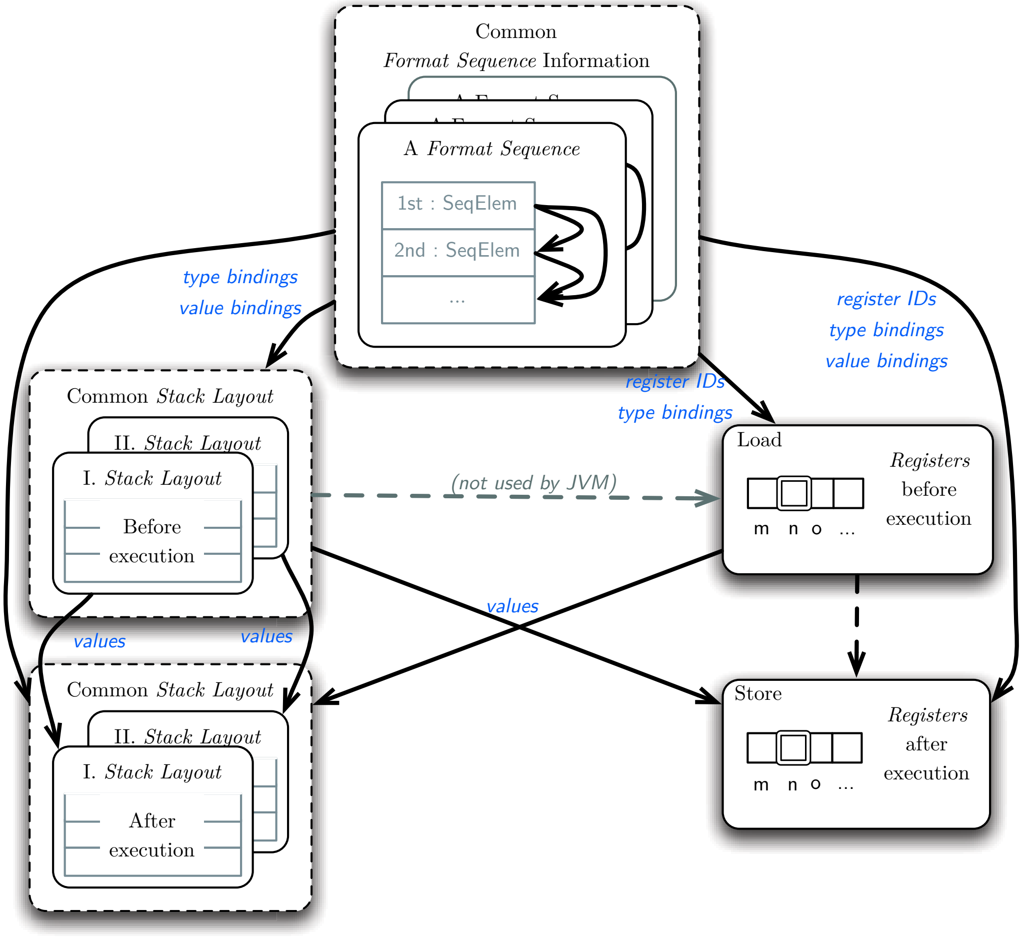

Fig. 4. Flow of information when parsing an instruction

Information Flow

In OPAL SPL, the flow of information (values, types, register IDs) is modeled by means of named variables and expressions using the variables. In general, the flow of information is subject to the constraints illustrated by Fig. 4. For example, variables defined within a specific format sequence can only be referred to by later elements within the same format sequence; a variable cannot be referred to across format sequences. If the same variable is bound by all format sequences, i.e., it is common to all format sequences, then the variable can be used to identify register IDs, the values pushed onto the stack, etc. Similarly, if an instruction defines multiple stack layouts, then a value can only flow from the i-th stack layout before execution to the i-th stack layout after execution and only information that is common to all stack layouts before execution may be stored in a register.

3 Design Discussion

The design of the OPAL specification language (OPAL SPL) is influenced by the peculiarities of the JVM's instruction set [9, Chapter 6]. In the following, we discuss those instructions that had a major influence on the design.

*3.1 Modeling the Stack Bottom (**athrow**)*

All JVM instructions—with the exception of athrow—specify only the number and types of operands popped from and pushed onto the stack; they do not determine the layout of the complete stack. In case of the **athrow** instruction, however, the stack layout after its execution is completely determined (Fig. 5, line 6); the single element on the stack is the thrown exception. This necessitates explicit modeling of the stack's contents beyond the operands that are pushed and popped by a particular instruction. The explicit modeling of the rest of the stack (line 5) hereby allows for the (implicit) modeling of stacks of a fixed size (line 6).

```
1 <instruction mnemonic="athrow" transfers_control="always">
2   ...
3   <stack> <form>
4       <before> <operand type="java.lang.Throwable" var="exception" />
5                <rest /> </before>
6       <after>  <operand type="java.lang.Throwable">exception</operand> </after>
7   </form> </stack>
8   ...
9 </instruction>
```

Fig. 5. OPAL SPL specification of the **athrow** instruction

*3.2 Pure Register Instructions (**iinc**)*

The flow of information for instructions that do not affect the stack—e.g., the JVM's **iinc** instruction—is depicted in Fig. 7 and adheres to the general scheme of information flow (cf. Fig. 4). After parsing the instruction according to the format sequence (Fig. 6, lines 3–5, the two variables lvIndex and increment are initialized.[5] The value of the former variable is then used to identify the register whose value is to be incremented. The register's value is thus bound to the variable value, which is incremented and stored back into the same register.

This encoding illustrates OPAL SPL's capability to model instruction sets of register-based VMs; their instructions simply do not affect the stack (lines 9–10), but only the registers (lines 13–14).

*3.3 Interpretation of Arithmetic Instructions (**iinc**, add, sub, etc.)*

The specification of **iinc** (Fig. 6) also illustrates OPAL SPL's ability to model computed values, e.g., add(value, increment). This information can subsequently be used, e.g., by static analyses to determine data dependencies or to perform abstract interpretations.

[5] Note that **iinc** also supports a second, **wide** format sequence which binds the same two values.

```
<instruction mnemonic="iinc">
  <format>
    <sequence> <u1 var="opcode">132</u1>
               <u1 type="lv_index" var="lvIndex"/>
               <i1 type="byte" var="increment"/> </sequence>
    ...
  </format>
  <stack> <form>
      <before> <rest/> </before>
      <after> <rest/> </after>
  </form> </stack>
  <registers>
    <load type="int" var="value" index="lvIndex"/>
    <store type="int" index="lvIndex">add(value, increment)</store>
  </registers>
</instruction>
```

Fig. 6. OPAL SPL specification of the **iinc** instruction

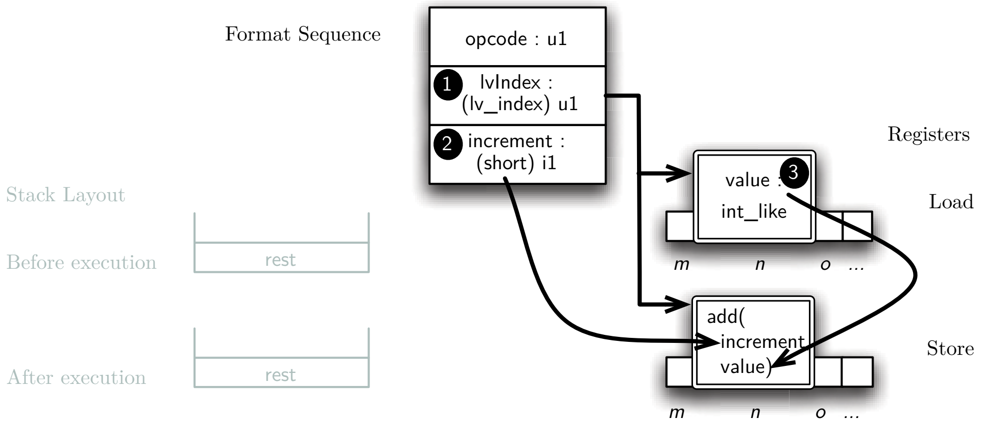

Fig. 7. Flow of information of the **iinc** instruction

*3.4 Constant Pool Handling (***ldc***)*

The Java class file format achieves its compactness in part through the use of a constant pool. Hereby, immediate operands of an instruction are replaced by an index into the (global) pool. For example, in case of the load constant intruction **ldc**, the operand needs to be programmatically retrieved from the constant pool (Fig. 8, line 5). To obtain the value's type, one uses the reflective type_of function that the enclosing toolkitx has to provide (line 14).[6]

3.5 Multiple Format Sequences, Single Logical Instruction

An instruction such as **ldc**, which may refer to an integer value in the constant pool, is conceptually similar to instructions such as **iconst_0** or **sipush**; all of them push a constant value onto the operand stack. The primary difference between the format sequences of **ldc** (Fig. 8, lines 3–5) and **iconst_0** (lines 6–7) is that the former's operand resides in the constant pool. In contrast, **sipush** encodes its

[6] In this case type_of could be supplanted by implicit_type (cf. Sec. 3.12) in conjunction with the constant_pool_type function. However, type_of allows for a clearer specification.

```
 1 <instruction mnemonic="push">
 2   <format>
 3     <sequence> <u1 var="opcode">18</u1> <!-- ldc -->
 4                <u1 type="cp_index" var="cp_index"/>
 5                <implicit var="value">constant_pool_value(cp_index)</implicit> </sequence>
 6     <sequence> <u1 var="opcode">3</u1> <!-- iconst_0 -->
 7                <implicit var="value">0</implicit> </sequence>
 8     <sequence> <u1 var="opcode">17</u1> <!-- sipush -->
 9                <i2 type="short" var="value"/> </sequence>
10     . . .
11   </format>
12   <stack> <form>
13       <before> <rest/> </before>
14       <after>  <operand type="type_of(value)">value</operand>
15                <rest/> </after>
16   </form> </stack>
17 </instruction>
```

Fig. 8. OPAL SPL specification of the **ldc**, **iconst_0**, and **sipush** instructions

operand explicitly in the bytecode stream as an immediate value (line 9).

To facilitate standard control- and data-flow analyses, OPAL SPL abstracts away from such details, so that similar instructions can be subsumed by more generic instructions using explicit or implicit type and value bindings. A generic push instruction (Fig. 8), e.g., subsumes all JVM instructions that just push a constant value onto the stack. In this case the pushed value is either a computed value (line 5), an implicit value (line 7), or an immediate operand (line 9).

*3.6 Variable Operand Counts (***invokevirtual***,* **invokespecial***, etc.)*

Some instructions pop a variable number of operands, e.g., the four invoke instructions **invokevirtual**, **invokespecial**, **invokeinterface**, and **invokestatic**. In their case the number of popped operands directly depends on the number of arguments of the method. To support instructions that pop a variable number of operands, OPAL SPL provides the list element (Fig. 9, line 8). Using the list element's count attribute, it is possible to specify a function that determines the number of operands actually popped from the stack. It is furthermore possible, by using the loop_var attribute, to specify a variable iterating over these operands. The loop variable (i) can then be used inside the list element to specify the expected operands (line 10). This enables specification of both the expected number and type of operands, i.e., of the method arguments (lines 8–10).

Using functions (methodref_arg_count, methodref_arg_type, ...) offloads the intricate handling of the constant pool to externally supplied code (cf. Sec. 3.4)—the enclosing toolkit; the OPAL specification language itself remains independent of how the framework or toolkit under development stores such information.

3.7 Exceptions

The specification of invokevirtual (Fig. 9) also makes explicit which exceptions the instruction may throw (line 16). This information is required by control-flow analyses and thus needs to be present in specifications. To identify the instructions which may handle the exception the function (caught_by) needs to be defined by

```
<instruction mnemonic="invokevirtual">
  <format>
    <sequence> <u1 var="opcode">182</u1>
               <u2 type="cp_index" var="cpIndex"/>
               <implicit var="methodRef">constant_pool_methodref(cpIndex)</implicit> </sequence>
  </format>
  <stack> <form>
      <before> <list loop_var="i" count="methodref_arg_count(methodref)">
                 <operand type="methodref_arg_type(i,methodref)"/>
               </list>
               <operand type="methodref_receiver_type(methodref)"/>
               <rest/> </before>
      <after>  <operand type="methodref_return_type(methodref)"/>
               <rest/> </after>
  </form> </stack>
  <exceptions> <exception type="java.lang.NullPointerException"/> ... </exceptions>
</instruction>
```

Fig. 9. OPAL SPL specification of the **invokevirtual** instruction

the toolkit. This functions computes, given both the instruction's address and the type of the exception, the addresses of all instructions in the same method that handle the exception. Similar to the handling of the constant pool, OPAL SPL thus offloads the handling of the exceptions attribute.

*3.8 Variable-length Instructions (***tableswitch***,* **lookupswitch***)*

The support for variable-length instructions (**tableswitch**, **lookupswitch**) is similar to the support for instructions with a variable stack size (cf. Sec. 3.6). In this case, an elements element can be used to specify how many times (Fig. 10, line 7) which kind of values (lines 8–9) need to be read. Hereby, the elements construct can accommodate multiple sequence elements (lines 7–10).

```
 1 <instruction mnemonic="lookupswitch" transfers_control="always">
 2   <format>
 3     <sequence> <u1 var="opcode">171</u1>
 4                <padding_bytes alignment="4"/>
 5                <i4 type="branchoffset" var="defaultOffset"/>
 6                <i4 type="int" var="npairsCount"/>
 7                <elements count="npairsCount">
 8                  <i4 type="int" var="matchValue"/>
 9                  <i4 type="branchoffset" var="branchoffset"/>
10                </elements> </sequence>
11   </format>
12   ...
13 </instruction>
```

Fig. 10. OPAL SPL specification of the **lookupswitch** instruction

The variable number of cases is, however, just one reason why **tableswitch** and **lookupswitch** are classified as variable-length instructions; the JVM Specification mandates that up to three padding bytes are inserted, to align the following format elements on a four-byte boundary (line 4).

*3.9 Single Instruction, Multiple Operand Stacks (***dup2***)*

The JVM specification defines several instructions that operate on the stack independent of their operands' types or—if we change the perspective—that behave differently depending on the type of the operands present on the stack prior to their execution. For example, the **dup2** instruction (Fig. 11) duplicates the contents of two one-word stack slots.

```
<instruction mnemonic="dup2">
  ...
  <stack>
    <form>
      <before> <operand type="category_2_value" var="value" />
               <rest /> </before>
      <after>  <operand type="category_2_value">value</operand>
               <operand type="category_2_value">value</operand>
               <rest /> </after>
    </form>
    <form>
      <before> <operand type="category_1_value" var="value1" />
               <operand type="category_1_value" var="value2" />
               <rest /> </before>
      <after>  <operand type="category_1_value">value1</operand>
               <operand type="category_1_value">value2</operand>
               <operand type="category_1_value">value1</operand>
               <operand type="category_1_value">value2</operand>
               <rest /> </after>
    </form>
  </stack>
</instruction>
```

Fig. 11. OPAL SPL specification of the **dup2** instruction

Instructions such as **dup2** and **dup2_x1** distinguish their operands by their computational type (category 1 or 2) rather than by their actual type (int, reference, etc.). This makes it possible to compactly encode instructions such as **dup2** and motivates the corresponding level in the type hierarchy (cf. Sec. 2.2). Additionally, this requires that OPAL SPL supports multiple stack layouts.

In OPAL SPL, the stack is modeled as a list of operands, not as a list of slots as discussed in the JVM specification. While the effect of an instruction such as **dup2** is more easily expressed in terms of stack slots, the vast majority of instructions naturally refers to operands. In particular, the decision to base the stack model on operands rather than slots avoids explicit modeling of the higher and lower halves of category-2-values, e.g., the high and low word of a 64 bit **long** operand.

*3.10 (Conditional) Control Transfer Instructions (**if, goto, jsr, ret**)*

To perform control-flow analyses it is necessary to identify those instructions that may transfer control, either by directly manipulating the program counter or terminating the current method. This information is specified using the instruction element's optional transfers_control attribute (Fig. 12, line 1). It specifies if control is transfered conditionally or always. The target instruction to which control is transferred is identified by the values of type branchoffset or absolute_address. For these two types the type system contains the meta-information (cf. Fig. 3) that the values have to be interpreted either as relative or absolute program counters.

```
<instruction mnemonic="ifgt" transfers_control="conditionally">
  <format>
    <sequence> <u1 var="opcode">157</u1>
               <u2 type="branchoffset" var="branchoffset"/> </sequence>
  </format>
  ...
</instruction>
```

Fig. 12. Specification of the **ifgt** instruction

```
 1 <instruction mnemonic="newarray">
 2   <format>
 3     <sequence> <u1 var="opcode">188</u1>
 4                <u1 var="atype">4</u1>
 5                <implicit_type var="T">boolean</implicit_type> </sequence>
 6     <sequence> <u1 var="opcode">188</u1>
 7                <u1 var="atype">5</u1>
 8                <implicit_type var="T">char</implicit_type> </sequence>
 9     ...
10   </format>
11   <stack> <form>
12       <before> <operand type="int"/>
13                <rest/> </before>
14       <after>  <operand type="array(1, T)"/>
15                <rest/> </after>
16   </form> </stack>
17   ...
18 </instruction>
```

Fig. 13. OPAL SPL specification of the **newarray** instruction

*3.11 Multibyte Opcodes and Modifiers (***wide** *instructions,* **newarray***)*

The JVM instruction set consists mostly of instructions whose opcode is a single byte, although a few instructions have longer opcode sequences. In most cases this is due to the **wide** modifier, a single byte prefix to the instruction. In case of the **newarray** instruction, however, a suffix is used to determine its precise effect. As can be seen in Fig. 13, the parser needs to examine two bytes to determine the correct format sequence.

3.12 Implicit Types and Type Constructors

The specification of **newarray** (Fig. 13) also illustrates the specification of implied types and type constructors. As the JVM instruction set is a typed assembly language, many instructions exist in a variety of formats, e.g., as **iadd**, **ladd**, **fadd**, and **dadd**. The implicit_type construct is designed to eliminate this kind of redundancy in the specification, resulting in a single, logical instruction: add. Similarily, **newarray** makes use of type bindings (lines 5, 8).

But, to precisely model the effect of **newarray** on the operand stack, an additional function that constructs a type is needed. Given a type and an integer, the function array constructs a new type; here, a one-dimensional array of the base type (line 14).

3.13 Extension Mechanism

OPAL SPL has been designed with extensibility in mind. The extension point for additional information is the instruction element's appinfo child, whose content can consist of arbitrary elements with a namespace other than OPAL SPL's own.

To illustrate the mechanism, suppose that we want to create a Prolog representation for Java Bytecode, in which information about operators is explicit, i.e., in which the ifgt instruction is an if instruction which compares two values using the greater than operator, as illustrated by Fig. 14.

```
instr (METHODID, PROGRAM_COUNTER, if(gt, Branchoffset)).
```

Fig. 14. Prolog representation of an if instruction

To support this feature, we designed a small XML language to encode information about operators. The additional information is specified using child elements of the appinfo element as exemplified in Fig. 15, lines 2–4.

```
1 <instruction mnemonic="ifgt" transfers_control="conditionally">
2   <appinfo> <cg:parameterized base="if">
3                <cg:operator name="gt"/>
4             </cg:parameterized> </appinfo>
5   ...
6 </instruction>
```

Fig. 15. Application specific information.

4 Validating Specifications

To validate an OPAL SPL specification, we have defined an XML Schema which ensures syntactic correctness of the specification and performs basic identity checking. It checks, for example, that each declared type and each instruction's mnemonic is unique. Additionally, we have developed a program which analyzes a specification and detects the following errors: (a) a format sequence does not have a unique prefix path, (b) multiple format sequences of a single instruction do not agree in the variables bound by them, (c) the number or type of function's arguments is wrong or its result is of the wrong type.

In addition to these errors, we warn about the following potential issues: (a) a declared type, function or exception is not used, (b) a format sequence defines no variable with the name opcode, (c) the same opcode value is used in sequences that belong to different instruction definitions[7], (d) an instruction mnemonic that contains "if", "goto", "ret", "jsr", "jump", "throw", or "switch" does not set the transfers_control attribute, (e) an instruction specifies more than one format sequence and more than one stack form. These additional checks have proven to be useful to detect and fix (subtle) errors early on.

5 Evaluation

Correctness of the Specification

We have used the specification of the JVM's instruction set [9] for the implementation of a highly flexible bytecode toolkit. The toolkit supports four representations

[7] The decision to enable multiple sequences that contain the same opcode value was necessary to model the **newarray** instruction.

of Java bytecode: a native representation, which is a one-to-one representation of the Java Bytecode; a higher-level representation, which abstracts away some details of Java bytecode—in particular from the constant pool; an XML representation which uses the higher-level representation; a Prolog-based representation of Java Bytecode, which is also based on the higher-level representation.

We have extensively tested the developed framework and were able to import all class files part of JDK 6, Tomcat 6.0.18, and Eclipse 3.5. Additionally, we have compiled Apache Ant 1.7.1 with different compilers (`javac` and Eclipse's built-in compiler) and different compiler settings and were also able to decode these class files. Hence, we are confident that the encoding of the JVM specification is correct.

Usefulness of the Approach

Based on the specification, we have developed two generators which are both implemented using XSLT. The first XSLT transformation generates the classes to represent all instructions and is 350 lines long. Each generated class represents an instruction as a Java object and offers the functionality to get an XML and a Prolog representation of the concrete instance of an instruction. The second XSLT transformation generates the parser for a code array which creates the instance of the instruction classes on the fly. This transformation is another 300 lines long. We compared this with the Bytecode Code Engineering Library (BCEL) [2] which uses a similar approach for representing and handling instructions. When compared to the instruction-related code of BCEL, the generator is between 15 and 20 times smaller.

Another advantage of the approach is that changes that affect all instructions are localized. For example, in case of the Prolog representation we tested several different representations which often affected all instructions. Nevertheless, in general less than 40 lines of code of the generator needed to be changed.

6 Related Work

Applying XML technologies to Java bytecode is not a new idea [5]. The XML serialization of class files, e.g., allows for their declarative transformation using XSLT. The XMLVM [11] project aims to support not only the JVM instruction set [9], but also the CLR instruction set [8]. This requires that at least the CLR's operand stack is transformed [12], as the JVM requires. The description of the effect that individual CLR instructions have on the operand stack is, however, not specified in an easily accessible format like OPAL SPL, but rather embedded within the XSL transformations.

The rules of Hoare-style program logic can also serve as a specification of the JVM instruction set [1]. While such a specification goes beyond OPAL SPL as far the instructions' effect on the VM's state (operand stack, registers, etc.) is concerned, it does not describe instruction formats and also groups instructions (cf. Sec. 3.4) only implicitly, through derivation rules, if at all.

The Project Maxwell assembler system [10] is able to describe instruction formats that are more complex than those commonly encountered in high-level intermediate languages, namely those of the IA32, PowerPC, and SPARC instruction set architectures. These descriptions are then used to generate assemblers and disassemblers as well as test cases for either. The system is unable, however, to describe the instructions' effect; only their format is described. Unlike OPAL SPL, these descriptions are not made available in a language-independent format like XML, but rather constructed programmatically, using a domain-specific language embedded into Java.

Vmgen [6] is a generator for efficient interpreters for stack-based intermediate languages. While it can also be used to generate code for register-based intermediate languages, it cannot describe such instructions declaratively, as can be done using the load and store elements in OPAL SPL. Its descriptions also do not cover the format of the bytecode itself; thus, it is not possible to generate a parser from vmgen's descriptions. One notable feature of vmgen is its (almost) uniform treatment of operand stack and instruction stream, which simplifies the description of instructions with immediate operands. OPAL SPL does not achieve the same degree of uniformity because it describes how instructions are stored in class files.

7 Conclusion and Future Work

In this paper, we have first discussed a language for the specification of both the format and the execution semantics of bytecode based instructions with respect to memory access. The language was used to encode the semantics of the JVM's instruction set. The resulting encoding of the JVM Specification was subsequently used for the development of a Java Bytecode framework that reads in class files and performs standard control- and data-flow analyses; e.g., to transform the stack-based bytecode representation into an SSA representation. Our evaluation shows that using the specification as the foundation for the development of bytecode toolkits significantly reduces the number of lines of code that need to be developed and also reduces the development time of such toolkits.

In future work, we will investigate the use of OPAL SPL for the encoding of other bytecode languages, such as the Common Intermediate Language. This would make it possible to develop (control- and dataflow-) analyses with respect to the OPAL SPL and to use the same analysis to analyze bytecode of different languages.

Acknowledgement

The authors would like to thank Lucas Satabin for implementing the type checker. This work was supported by www.cased.de.

References

[1] Fabian Bannwart and Peter Müller. A program logic for bytecode. *Electronic Notes in Theoretical Computer Science*, 141(1):255–273, 2005. Proceedings of the First Workshop on Bytecode Semantics,

Verification, Analysis and Transformation (BYTECODE 2005).

[2] The Bytecode Engineering Library (BCEL). http://jakarta.apache.org/bcel/manual.html, 2006.

[3] Eric Bruneton. ASM 3.0: A Java bytecode engineering library. http://download.forge.objectweb.org/asm/asm-guide.pdf, February 2007.

[4] Shigeru Chiba. Javassist - Java Programming Assistant 3.11.0.ga. http://www.csg.is.titech.ac.jp/~chiba/javassist/, 2009.

[5] Michael Eichberg. BAT2XML: XML-based java bytecode representation. *Electronic Notes in Theoretical Computer Science*, 141(1):93–107, 2005. Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2005).

[6] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen: a generator of efficient virtual machine interpreters. *Software Practice & Experience*, 32(3):265–294, 2002.

[7] IBM. The t. j. watson libraries for analysis. http://wala.sourceforge.net/, 2006.

[8] ISO/IEC, Geneva, Switzerland. *Information technology – Common Language Infrastructure (CLI) Partitions I to VI*, ISO/IEC 23271:2006(E) edition, 2006.

[9] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[10] Bernd Mathiske, Doug Simon, and Dave Ungar. The Project Maxwell assembler system. In *PPPJ '06: Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java*, pages 3–12, New York, NY, USA, 2006. ACM.

[11] Arno Puder. Byte code transformations using XSL stylesheets. In *SNPD '08: Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 563–568, Washington, DC, USA, 2008. IEEE Computer Society.

[12] Arno Puder and Jessica Lee. Towards an XML-based bytecode level transformation framework. *Electronic Notes in Theoretical Computer Science*, 253(5):97–111, 2009. Proceedings of the Fourth Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2009).

[13] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings of the 9th International Conference on Compiler Construction (CC)*, volume 1781, 2000.