Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 141 (2005) 93–107
www.elsevier.com/locate/entcs

BAT$_2$XML: XML-based Java Bytecode Representation

Michael Eichberg[1]

Department of Computer Science
Darmstadt University of Technology
Darmstadt, Germany

Abstract

The creation, transformation and analysis of bytecode is widespread. Nevertheless, several problems related to the reusability and comprehensibility of the results and tools exist. In particular, the results of tools for bytecode analysis are usually represented in proprietary tool dependent ways, which makes it hard to build more sophisticated analysis on top of the results generated by tools for lower-level analysis.Furthermore, intermediate results, such as e.g., the results of basic control flow and dataflow analysis, are usually not explicitly represented at all; though, required by many more sophisticated analysis. This lack of a common format, for the well structured representation of the (intermediate) results of code analysis, makes the creation of new tools or the integration of the results generated by different tools costly and ineffective.

To solve the highlighted problems, we propose a higher-level XML-based representation of Java bytecode which is designed as a common platform for the creation and transformation of bytecode and explicitly enables the integration of arbitrary information generated by different tools for static code analysis.

Keywords: Java Bytecode, XML, Static Analysis

1 Introduction

Though, the creation, transformation and analysis of Java bytecode is widespread, it is still necessary to build custom tools and libraries to read in bytecode, to perform transformations, or for analyzing the code. Furthermore, for code analysis tools the results are usually represented in a tool dependent way,

[1] Email: eichberg@informatik.tu-darmstadt.de

doi:10.1016/j.entcs.2005.02.035

which makes reuse of the results hard. Additionally, the usage of intermediate results, which were created during the code analysis and would also be of interest to other tools, is not possible at all. For example, tools like FindBugs [16], which search for common bugs in the code, could be improved, if the results of sophisticated code analysis would be easily accessible.
The diversity of representations and tools for bytecode transformation and analysis makes the creation of new tools, or the merging of the results generated by multiple tools, costly and ineffective.

To solve these problems we propose a higher-level XML-based representation of Java bytecode. This representation is meant to be a generic basis for various applications, e.g., for detecting bad smells (i.e. finding violations of implementation restrictions and best practices and to detect bug patterns), to validate specifications against the implementation, or to have a good basis for exploring code.
Using XML has several advantages:

- First, sophisticated mature query engines for XML data sources exist, which makes the extraction of information easier when compared with the effort necessary to extract the information using a visitor [14] as in case of BCEL [2].
- Second, with XSLT a specification and corresponding tools exist that facilitate the transformation of XML documents into other XML documents or completely new artifacts.
- Third, since XML is also frequently used to represent other information, such as e.g., deployment descriptors, property files, build scripts and even code of other programming languages, reasoning about the dependencies between these artifacts is directly possible.
- Fourth, when using XML namespaces and schemas it is possible to consistently add information generated by different tools into the document structure. Hence, extending the representation with additional information does not break existing tools and queries.
- Fifth, manipulating an XML document which represents Java bytecode is easier than manipulating a corresponding object graph, because the XML document explicitly reifies the structure and is - to some extent - human readable.

In this paper we present BAT$_2$XML- a tool that provides a bidirectional mapping between Java bytecode and an XML representation on top of the Java bytecode toolkit BAT [7]. BAT$_2$XML abstracts from some details of Java bytecode to make creating, transforming and querying the XML representation

easier. [2] .

In the next section we discuss the design of BAT$_2$XML. After that, we discuss two applications of BAT$_2$XML: (1) using the XML based representation as a means to enable writing queries over Java bytecode. Queries can be used to detect bad smells or in the context of a code exploration tool. (2) As a basis for code generation and transformation. The paper ends with a discussion of related work, a summary and a short discussion of future work.

2 Implementation

BAT$_2$XML is designed to be a general purpose, higher-level representation of Java5 bytecode [19] that enables the creation of completely new classes, supports the manipulation of exiting classes and offers good querying possibilities. Before we discuss the design decisions behind BAT$_2$XML, we first give a short example. The bytecode sequence shown in listing 1, which simply prints "HelloWorld" to `System.out`, has the XML representation shown in listing 2.

```
1  public class HelloWorld extends java.lang.Object
2
3  public static void main(java.lang.String []);
4     0:   getstatic
5              //Field java/lang/System.out:Ljava/io/PrintStream;
6     3:   ldc
7              //String HelloWorld
8     5:   invokevirtual
9              //Method java/io/PrintStream.println:(Ljava/lang/String;)V
10    8:   return
11 }
```

Listing 1: Bytecode of "HelloWorld".

Please note that in the XML representation the bytecode offsets are omitted, the generic `ldc` (load constant) instruction (listing 1, line 6) is replaced by a `stringconst` instruction (listing 2, line 8), and the type information is represented in fully qualified binary form (e.g., listing 1, lines 4,7,11). Using this form for the XML representation supports comprehension and makes writing of queries easier, because most developers are not used to the bytecode's native representation of type information. However, for developers familiar with Java bytecode the generated file should be directly readable.

```
1  <class name="HelloWorld" sourcefile="HelloWorld.java" visibility="public" >
2    <inherits><class name="java.lang.Object" /></inherits>
3    <method name="main" visibility="public" static="true" >
4      <signature><parameter type="java.lang.String[]" /></signature>
5      <code>
6        <get declaringClassName="java.lang.System" fieldName="out"
```

[2] A download is made available at: http://www.st.informatik.tu-darmstadt.de/BAT

```
7             staticField ="true" type="java.io.PrintStream"/>
8         <stringconst><value>HelloWorld</value></stringconst>
9         <invoke declaringClassName="java.io.PrintStream"
10            methodName="println" >
11          <signature><parameter type="java.lang.String" /></signature>
12        </invoke>
13        <return />
14      </code>
15    </method>
16  </class>
```

Listing 2: XML representation of "HelloWorld".

After this short introduction to BAT$_\mathbf{2}$XML, we now discuss the design in relation to the Java bytecode:

- The number of instructions is minimized to make manipulation and querying easier, and to support comprehension of bytecode. For example, Java bytecode supports three different instructions to create new arrays: `newarray`, `anewarray`, `multianewarray`, and, to make the situation even worse, the `multianewarray` instruction can also be used to create one-dimensional arrays. Hence, the `(a)newarray` instruction exists for the sole purpose of optimizing the runtime execution. But, having special instructions to improve runtime performance makes the manipulation and querying harder. Therefore, in BAT$_\mathbf{2}$XML we abstract from these differences and provide only one parameterized instruction for sets of closely related instructions. Note that during the conversion of the XML back to Java bytecode the *most specific* instruction is automatically chosen. For example, to push the float value 0.0 onto the stack an `fconst_0` instruction is created and not an `ldc` instruction. Hence, if the original bytecode does not use the most specific instructions, the conversion: bytecode → XML → bytecode may generate a different class when compared with the original class, even if the intermediate XML file is not altered. Nevertheless, the execution semantics of both classes is identical, just the executed instructions are different. However, if the conversion starts with an XML document, that is, from XML → bytecode → XML, then the input and output documents are identical.
- Each instruction serves exactly one purpose. For example, instead of having one `ldc` instruction to put constant values with different types onto the stack, we use specialized instructions for each type. E.g., as seen in the introductory example to this section, we use an explicit `stringconst` instruction for putting a string value onto the stack. Furthermore, with respect to the operand stack, we do not distinguish between values occupying one or two stack values; at VM level double and long values occupy two stack values, but in BAT$_\mathbf{2}$XML every value, regardless of its type, occupies

one stack value.
As for the previous design decision, having instructions with exactly one purpose makes manipulation of existing classes and querying easier; the number of context information that is required to determine the (runtime) semantics of a bytecode instruction is minimized.

- As shown in the introductory example, all information is resolved, that is, the constant pool is completely hidden and all types are represented in binary form [5]. [3] Additionally, the target of intra-method jump instructions, such as e.g., `switch` and `if`, is specified by referring to the `id` of the target instruction and not by using bytecode addresses and offsets. E.g., an if-else structure is represented as shown below:

```
1 <if operator="ne" idref="m2i0" />
2  ...
3 <goto idref="m2i1" />
4 <get ... id="m2i0" />
5  ...
6 <return id="m2i1" />
```

In the above example, the target of the `if` instruction is either the `get` instruction, which has the `id ("m2i0")`, or the instruction immediately following the `if` instruction, if the condition is not satisfied. The target of the `goto` instruction is the `return` instruction. In short, an `id` attribute is used to mark a jump target and an `idref` attribute is used to reference it. For subroutines in Java bytecode, i.e. bytecode sequences where the `jsr` (jump to subroutine) and `ret` (return from subroutine) instructions are used, the jump target of the `ret` instruction is not directly available; the target instruction of the `ret` instruction is stored on the heap and is not a parameter of the instruction.

However, to make analysis of such sequences easier, BAT$_2$XML performs a control flow analysis to determine the jump target of the `ret` instruction in relation to the `jsr` instruction. E.g., in the following listing the `ret` instruction lists all jump targets in relation to the `jsr` instruction which called the subroutine. E.g., if the subroutine was called by the `jsr` instruction with the id `JSR1` (line 1) the target of the `ret` instruction (line 9) is the instruction with the id `i0` (line 2).

```
1 <jsr id="JSR1" idref="i3" />
2 <invoke ... id="i0" />
3  ...
4 <jsr id="JSR2" idref="i3" />
```

[3] Though, the Java Virtual Machine Specification permits classes with names (e.g., "`int`") that are not legal Java class types, such classes are never generated by Java compilers. However, if we detect a class with a name equal to a primitive type in Java an error is signaled.

```
<get ... id="i2" />
...
<store ... id="i3" />
...
<ret index="...">
  <opt:path>
    <opt:caller opt:idref="JSR1" />
    <opt:target opt:idref="i0" />
  </opt:path>
  <opt:path>
    <opt:caller opt:idref="JSR2" />
    <opt:target opt:idref="i2" />
  </opt:path>
</ret>
```

The elements below the ret element are in the special namespace `opt`, because this information is not used / required during the transformation from XML to Java bytecode; they are optional.
This representation of jump targets facilitates the easy creation and manipulation of instructions, because the number of necessary changes to other instructions or elements is minimized when instructions are added or removed.

- To make code analysis easier the control flow graph[4] of a method is explicitly represented. The control flow graph for the method `abs`, shown in listing 3, is depicted in figure 1. The XML representation of the method is given in listing 4.

```
public int abs(int value){
    if (value < 0)
      return -value;
    else
      return value;
  }
```

Listing 3: Sourcecode of a method which returns the absolute value.

```
<load index="1" fg:bb-idref="m2bb0" />
<if operator="ge" fg:bb-idref="m2bb0" idref="m2i0" />
<load index="1" fg:bb-idref="m2bb1" />
<neg fg:bb-idref="m2bb1" />
<return fg:bb-idref="m2bb1" />
<load index="1" id="m2i0" fg:bb-idref="m2bb2" />
<return fg:bb-idref="m2bb2" />
<fg:flow-graph>
  <fg:bb fg:id="m2bb2">
    <fg:pre fg:idref="m2bb0" />
  </fg:bb>
  <fg:bb fg:id="m2bb0">
```

[4] The control flow graph visualizations were built with a small XSLT which converted the XML representation of the graph in a *dot* [15] file. The *dot* file is then interpreted by the corresponding tool and a svg or eps file is generated. The XSLT script is part of the BAT_2XML distribution.

```
13      <fg:succ fg:idref="m2bb2" />
14      <fg:succ fg:idref="m2bb1" />
15    </fg:bb>
16    <fg:bb fg:id="m2bb1">
17      <fg:pre fg:idref="m2bb0" />
18    </fg:bb>
19  </fg:flow-graph>
```

Listing 4: BAT$_2$XML Representation

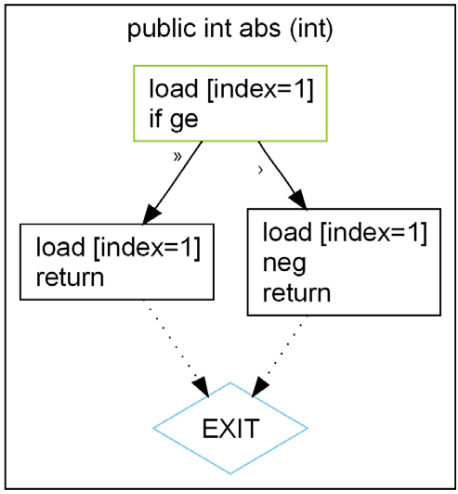

Fig. 1. Control-Flow Graph

The encoding of the control flow graph is bipartite. First, every instruction (listing 4, lines 1–7) is associated with the id (`fg:bb-idref`) of the basic block to which the instruction belongs. Second, the relationships between the basic blocks are encoded in its own structure (`fg:flow-graph`; listing 4, lines 8–19). For every basic block the predecessor (lines 10,17) and successor blocks (lines 13,14) are specified. Note that even though it would be sufficient to specify only one direction, e.g., only the successors of a block, we explicitly include both directions for convenience.

As in the case of the `ret` instruction, the control-flow graph represents information generated by static analysis of the bytecode and is not required during the conversion of XML to Java bytecode.

- In BAT$_2$XML, the try-catch information is also represented with respect to the control-flow graph, i.e. a try block lists all basic blocks for which exceptions are to be handled. For example, the code in listing 4 has the representation shown in figure 2. The graph representation is a direct representation of the control-flow graph and shows how each basic block, for which exceptions are to be handled, has a reference to the first basic block

of the catch handler.

```
1  public int test(int param){
2    try {
3      if (param == 0) return 0;
4      else return 1;
5    }
6    catch (Throwable t){ return 2; }
7     finally { System.out.println("Test"); }
8  }
```

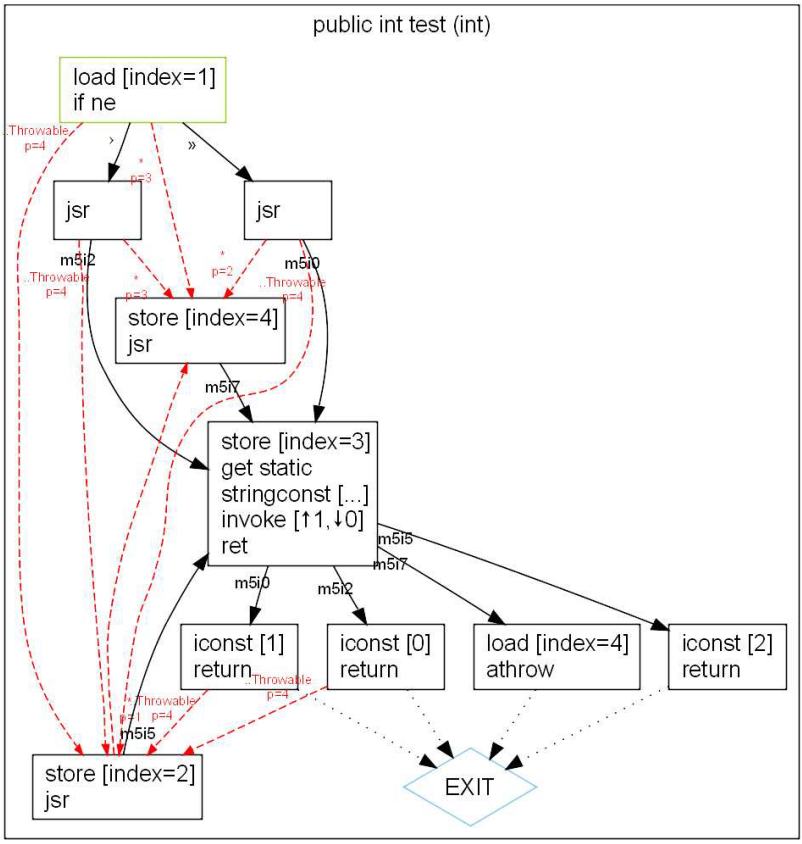

Fig. 2. Basic-block graph of the listing 4. (Compiler: Eclipse 3.1M4 with target Java 1.4)

Since the try-catch structure is orthogonal to the control flow graph, i.e. the start and the end instruction of a try block must not be the start and the end instruction of a basic block respectively, we split basic blocks at the borders of try blocks, if necessary.

- It should always be possible to convert the XML representation back to Java bytecode without the necessity to analyze further classes. This property makes the conversion very fast and minimizes the necessary effort for checking the validity of a generated class.

To sum up, BAT$_2$XML's representation is especially tailored to enable easy processing of classes by means of queries and transformations.

Additionally, due to the usage of XML the integration of information generated by other tools is possible and makes such a generic platform even more valuable. For example, to integrate the information generated by a tool for static type inference, e.g., [13], it is sufficient to put the information in the XML tree using a tool specific namespace. A tool specific namespace is necessary to avoid problems with existing queries and tools that operate on the representation. As an example, let's assume that the inference algorithm determined that a method with declared return type `java.lang.Object` actually always returns a value of type `java.lang.String`. The extended XML representation could then look like as in the following listing:

```
1 <bat:return>
2   <sti:inferred sti:type="java.lang.String">
3 </bat:return>
```

In this case `bat` refers to BAT$_2$XML's own namespace (omitted in the other examples for brevity) and `sti` is the namespace used to store the information determined by the inference algorithm. A query, which uses the additional information to select return instructions where the inferred type of the return value is `java.lang.String`, would be:

```
1 declare namespace bat = "http://BAT2−Java1.5(12/7/04)";
2 declare namespace sti = "http://www.codeinferencetool.com";
3
4 //bat:return[./ sti:inferred/@sti:type = "java.lang.String"]
```

3 Evaluation

Before we discuss major application areas of BAT$_2$XML, we first give an overview of the performance. To assess the performance, we converted all 12695 classes from Java5's rt.jar to XML. The overall process took ≈138.59 seconds on a 3Ghz Pentium IV and included the deserialization of the bytecode from the jar file, the generation of the XML structure, and finally writing the XML documents to the disk. In short, generation of the XML representation takes $\approx \frac{1}{100}$ seconds per class (including IO), which is reasonably fast for many applications and in particular for those discussed in the following.

3.1 Backend for Code Exploration

In Sextant [10], a tool for software exploration, BAT$_2$XML is primarily used to get an XML representation of Java bytecode which can conveniently be queried. The idea behind Sextant is to have a database in which all artifacts

of a project are stored in the same data format (XML). Storing all information in one data format makes it possible to write queries that cross artifact borders and put information spread over different kinds of artifacts into relation. For example, in Enterprise Java Bean (EJB) [6] projects the runtime behavior of an application is dependent on the deployment configuration. In fig. 3, for example, basically two different queries were repeatedly executed to explore the project. The exploration started with the `de.tud.CartBean` class. The first two queries returned the methods (`getText()`, `getValue()`) declared by the class along with the methods' transaction attributes. After that, queries to get all methods called by the first two methods were recursively executed.

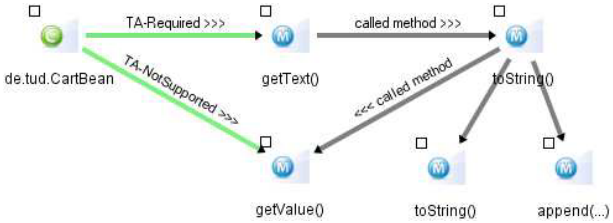

Fig. 3. *Visualization of an exploration of a small EJB project using Sextant.*

The query to get all methods called by a specific method is shown in listing 5. It nicely demonstrates how to use XQuery [4] to query the XML representation generated by BAT$_2$XML. The query starts (line 2) by searching all invoke instructions (`$method//invoke`) of the passed method (line 1, parameter: `$method`). After that, the method declarations of the invoked methods are searched (line 4–7). To get the method declarations the entire database (`$bat2xml:all`) is searched for method declarations where the declaring class (line 5), the method's name (line 6) and the method's signature (line 7) match the one specified by the invoke instruction. Matching the signatures is done by the predefined function `equalSignatures` (line 7). The function matches the parameter types and return type of two methods.

```
1 declare function bat2xml:callees($method as element()) as element()* {
2     for $invoke in $method//invoke
3     return
4         $bat2xml:all/(class | interface | enum | annotation)
5                         [@name=$invoke/@declaringClassName]
6                     /method[@name=$invoke/@methodName]
7                         [bat2xml:equalSignatures(./signature,$invoke/signature)]
8 };
```

Listing 5: *The function returns all methods called by the passed method.*

Note that the query shown in listing 5 does not take methods redefined in subclasses into account. If overriding methods should also be selected, the result of the `callees` function can be passed to the predefined method

`overridingMethods`.

Another application domain, where BAT$_2$XML is primarily used to generate a representation that can be easily queried, are frameworks for *bad smell* detection. That is, to detect violations of:

- **[best practices:]** Best practices are guidelines developers should follow to avoid problems in the future or to ease debugging. For example, a simple best practice, defined in Effective Java [3], is to always implement the `toString` method. A corresponding query is shown in listing 6. In the query, the "."s in line 1 and line 3 refer to one specific class at each moment. In the whole, the query is to be read as follows: "Select all classes (line 1) except (line 2) those that implement the `toString` method (line 3–5)."

```
1 .
2   except
3 ./method[@name="toString"
4             and empty(./signature/parameter)
5             and ./signature/returns/@type = "java.lang.String"]/..
```

Listing 6: *The query selects all classes that do not implement* `toString`.

- **[implementation restrictions:]** Implementation restrictions are restrictions defined by a framework that is to be used. For example, the Enterprise Java Beans (EJB) specification [6] defines a large number of restrictions, such as e.g., an Enterprise Java Bean should not define the finalize method.

and to detect:

- **[bug patterns:]** "Bug patterns are recurring correlations between signaled errors and underlying bugs in a program. [1]". A famous bug pattern is the result of copying and pasting code. Whenever a bug is detected it is usually only fixed in one location and not in all locations.

3.2 Backend for Code Generation / Transformation

Besides offering excellent querying capabilities, good tool support to generate / transform XML documents exists. Hence, a second application domain for an XML representation is its usage for the generation of new artifacts or the transformation of existing classes. E.g., with the support for meta data in Java5, information that was previously defined in deployment descriptors or xDoclet tags [24] is going to be defined as part of the Java code. But later on - often at deployment time - the meta information needs to be extracted to generate application specific documents and classes.

An example scenario is the generation of the WSDL file for a WebService. The JSR 181 [25] specifies the meta data to be defined along a Java class that implements a web service, and also specifies the mapping of the meta data to a WSDL file. A very simple web service is outlined in listing 7.

```
1 @WebService(name="PingService") public class PingService {
2
3   @WebMethod @Oneway  public void ping(){/*...*/}
4 }
```

Listing 7: A very simple WebService.

To generate the WSDL file, which is an XML file on its own, an XSLT stylesheet [17] could be used. An excerpt of such an XSLT stylesheet is shown in listing 8; the elements in bold already define the structure of the WSDL file.

```
1  <xsl:stylesheet>
2    <xsl:template match="/class[annotations//@type='javax.jws.WebService']">
3      <definitions>
4        <portType name="{//annotation[@type='javax.jws.WebService']
5                                    /member[@name='name']/value/text()}">
6          <xsl:apply-templates
7            select="./method[annotations//@type='javax.jws.WebMethod']" />
8        </portType>
9      </definitions>
10   </xsl:template>
11
12   <xsl:template match="//method">
13     <operation name="{@name}">...</operation>
14   </xsl:template>
15 </xsl:stylesheet>
```

Listing 8: XSLT to create a WSDL file from a class file.

Finally, in listing 9 the generated WSDL file for the `PingService` web service is shown.

```
1 <definitions xmlns="http://schemas.xmlsoap.org/wsdl">
2   <portType name="PingService">
3     <operation name="ping"> ... </operation>
4   </portType>
5 </definitions>
```

Listing 9: Generated WSDL file.

Besides using the XML representation to generate completely new artifacts, it is also possible to transform a class. For example, to implement a bytecode weaver for aspect-oriented programming [18] on top of it. In *Pointcuts as Functional Queries* [8] we propose to use XQuery as a pointcut language on top of BAT$_2$XML's representation. After evaluating the queries

(i.e. after locating the join point shadows [21]) the XML representation is transformed such that advice functionality is executed at runtime.

4 Related Work

The work most related to BAT$_2$XML are toolkits which also provide a higher-level representation of Java bytecode.
The NoUnit [23] tool, for example, also converts Java bytecode to XML, but cannot convert the XML back to Java bytecode. Anyway, the XML representation generated by NoUnit is especially tailored to create *Code Pictures* in a post-processing step and is not meant to be a generic application independent representation. Similar to NoUnit, the ByteML [20] framework also provides only a unidirectional mapping from bytecode to XML.

In contrast to the two previous frameworks, XJBC [12] defines a bidirectional mapping between XML and Java bytecode and also abstracts from details of the Java bytecode. Hence, XJBC is the most similar tool to BAT$_2$XML. However, BAT$_2$XML is Java5 compatible, defines explicit extension points, i.e. points where the representation can be extended to encode the results of further analysis, and provides a built-in representation of the control flow graph. Another difference is that, in BAT$_2$XML, we opted for generic parameterized instructions instead of instructions which more closely resemble the Java bytecode as done by XJBC, For example, in BAT$_2$XML all types of invoke instructions are represented by one invoke instruction with appropriate parameters; in XJBC each invoke instruction has its own XML representation. Based on our experience, the representation used by BAT$_2$XML makes queries more concise, because in most cases it is not desirable to distinguish the different types of invoke instructions.

5 Summary

BAT$_2$XML provides an XML representation of Java bytecode, which is directly usable to detect bad smells or to build a tool for code exploration on top of it. Further, it is possible to convert a transformed or generated XML representation back to Java bytecode, which is a convenient mechanism for generating / transforming Java bytecode. For executing queries it is possible to use the language XQuery [4] and for transformations to use XSLT [17]. Both have the advantage that they are mature, declarative languages and enable concise definitions of solutions for transformation or querying problems. In the evaluation section possible applications of an XML representation were discussed and two real applications were presented: Sextant [10] and XIRC

[9, 11], which both use BAT$_2$XML in their backends.

6 Future Work

Although, the current representation is already very useful and well supports the queries required to implement a code exploration tool, we made the experience that the implementation of queries to detect violations of implementation restrictions or best practices would be easier to write, if the XML representation would be richer. In particular, if the queries need to reason about the (intra-)method control and data flow corresponding information would be very useful. For example, to make sure that `this` (here: the reference of an object to itself) is not passed to another object. Hence, in the next version of BAT$_2$XML the XML representation is going to include the results of an intra-method data flow analysis. By carrying out the analysis once and making the results available we think the number of checkers that can be reasonably implemented at all will rise significantly. Further, we hope that we are able to make existing checkers more precise, that is, we want to reduce the number of false positives. Please note, though we are going to use techniques originally developed for decompiling Java bytecode back to Java source code [22], the representation will remain very different from Java source code. Having information like the declaring class and the type of an accessed field directly at hand is very valuable when developing queries. If the bytecode would be decompiled to Java source code it would be necessary to query this information, which is inconvenient and slow. However, the biggest problem we are going to tackle is to decide which information from which analysis should be included and how, i.e. how to make the additional information as easily accessible when writing queries as possible.

Acknowledgments

The author would like to thank Tobias Schuh and Barbara Wasilewski for the implementation of the converter from XML to Java bytecode. Further, the author would like to thank Christoph Bockisch, Cuma Ali Gencdal and Thorsten Schäfer for their contribution to the development of the underlying bytecode toolkit BAT.

References

[1] Allen, E. E., *Bug patterns: An introduction*, http://www-106.ibm.com/developerworks/library/j-diag1.html (2001).

[2] Apache Software Foundation, *Bytecode engineering library, BCEL*, http://jakarta.apache.org/bcel/ (2003).

[3] Bloch, J., "Effective Java Programming Language Guide," Addison-Wesley, 2001.

[4] Boag, S., D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie and J. Siméon, *XQuery 1.0: An XML query language*, Working Draft 12 November 2003, W3C.

[5] Bracha, G., *Java language specification*, Java Specification Request 901 (2004).

[6] DeMichiel, L. G., "Enterprise JavaBeans Specification, Version 2.1," SUN Microsystems, 2003.

[7] Eichberg, M. and C. Bockisch, *BAT*, http://www.st.informatik.tu-darmstadt.de/BAT (2004).

[8] Eichberg, M., M. Mezini and K. Ostermann, *Pointcuts as functional queries*, in: *Proceedings of the Second ASIAN Symposium on Programming Languages and Systems* (2004).

[9] Eichberg, M., M. Mezini, K. Ostermann and T. Schäfer, *Xirc: a kernel for cross-artifact information engineering in software development environments*, in: *Proceedings of the 11th IEEE Working Conference on Reverse Engineering* (2004).

[10] Eichberg, M., T. Schaefer, M. Vekic and S. Djukanovic, *Sextant*, http://www.st.informatik.tu-darmstadt.de/pages/projects/Sextant (2004).

[11] Eichberg, M., T. Schäfer and M. Mezini, *Using annotations to check structural properties of classes*, in: *Proceedings of Fundamental Approaches to Software Engineering* (2005), to appear.

[12] Engel, C., *Xjbc*, http://www.xjbc.org (2003).

[13] Gagnon, E., L. J. Hendren and G. Marceau, *Efficient inference of static types for java bytecode*, in: *Static Analysis Symposium*, 2000, pp. 199–219.

[14] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns," Addison-Wesley, 1995.

[15] Gansner, E., E. Koutsofios and S. North, *Drawing graphs with* dot, http://www.graphviz.org/Documentation/dotguide.pdf (2002).

[16] Hovemeyer, D. and W. Pugh, *Finding bugs is easy*, http://findbugs.sourceforge.net/docs/findbugsPaper.pdf (2003).

[17] Kay, M., *Xsl transformations (xslt) version 2.0*, Working Draft 12 November 2003, W3C.

[18] Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, *Aspect-oriented programming*, in: *Proceedings of the 11th European Conference on Object-Oriented Programming* (1997).

[19] Kramer, D., *Java virtual machine specification*, Java Specification Request 924 (2004).

[20] Kumar, A., *Byteml - java bytecode / xml interoperabilitiy framework*, http://byteml.sourceforge.net (2002).

[21] Masuhara, H., G. Kiczales and C. Dutchyn, *Compilation semantics of aspect-oriented programs*, in: *Foundations of Aspect-Oriented Languages Workshop at AOSD '02*, 2002.

[22] Miecznikowski, J. and L. J. Hendren, *Decompiling java bytecode: Problems, traps and pitfalls*, in: *CC '02: Proceedings of the 11th International Conference on Compiler Construction* (2002), pp. 111–127.

[23] NoUnit Team, *Nounit*, http://nounit.sourceforge.net (2004).

[24] XDoclet Team, *XDoclet: Attribute-Oriented Programming*, http://xdoclet.sourceforge.net/.

[25] Zotter, B., *Web services metadata for the java platform*, Java Specification Request 181 (2004).