Technical Aspects of Class Specification in Java Byte Code[1]

Aleksy Schubert[2] Jacek Chrząszcz[3] Tomasz Batkiewicz[4]
Jarosław Paszek[5] Wojciech Wąs[6]

Institute of Informatics
University of Warsaw
ul. Banacha 2
02–097 Warsaw
Poland

Abstract

Bytecode Modeling Language (BML) is a recent specification language designed to support specification and verification of Java byte code files. We present an editor called *Umbra* for byte code files which supports the editing of BML specifications. This editor is accompanied by a library *BMLlib* which not only parses textual representations of BML specifications and prints the specifications in a textual form, but also writes specifications into Java class files and reads them from that format. The whole tool set allows to insert BML specification into class files or inspect class files with BML specifications and edit them, e.g., for debugging purposes.

Keywords: Java, byte code, specification, BML, GUI

1 Introduction

One of the main achievements of high level programming languages (e.g., Algol, Pascal, C) is the common adoption of abstraction as the main programming paradigm. That means that executable code can be divided into smaller pieces which can be, to a large extent, developed independently. The common problem here is that the independence is usually vaguely determined which often causes misbehaviour of developed software. One of the ways to prevent this is to write precise descriptions of what is required by and what is made available to a particular piece of code, i.e.,

[1] This work was partly supported by Polish government grant 177/6.PR UE/2006/7 and Information Society Technologies programme of the European Commission FET project IST-2005-015905 MOBIUS. This paper reflects only authors' views and the Community is not liable for any use that may be made of the information contained therein.

[2] Email: alx@mimuw.edu.pl

[3] Email: chrzaszcz@mimuw.edu.pl

[4] Email: twb011@wp.pl

[5] Email: jp209217@students.mimuw.edu.pl

[6] Email: wojtekwas@poczta.onet.pl

to provide precise implementation contracts. Moreover, the size of current software packages is so big that companies often outsource at least part of the software development process to external companies. A precise and automatically verifiable format of software contracts could reduce the risk that the resulting code is useless.

Specification languages provide formats to precisely describe software requirements and the environment in which the software is going to be executed. Moreover, these languages offer another way of abstraction — they allow not only to divide the programming effort into smaller pieces, but they also support writing solely about *what* should be done without all the details on *how* it is done. As they are designed to be easily understood by programmers, they provide means to document the source code. Providing documentation in a specification language also has the advantage that it is possible to automatically verify that the source code really implements the documented features. Furthermore, the presence of specifications allows to locate more precisely the actual reasons for program misbehaviour.

In case of resource restricted platforms (such as mobile phones, washing machines, car engines etc.), at least part of the software development process is done in a low level language. Software development requires usually more effort in this case, but there are fewer methods to enable fine-grained abstraction. Of course, many assembly language editors allow to divide the code into subroutines or macros, but there are virtually no formalisms to describe the work of a low level program in terms of *what* is to be done even though the development effort is even bigger than in the case of programming at the source code level.

Another important scenario in which verification of a specification may be useful is the case when the specification describes a required security policy which is ensured by the verification process. In this case, the software developers are not the only party who are interested in checking the property. The owner of the infrastructure in which the program is run will also be interested by the specification. In this case, however, the artifact which should be analysed is most often the executable, low level code instead of the source code. That is why it is important to have a way to describe properties of the code on the low, executable level. Moreover, it is also important to have a format of specification which can be understood by humans, in particular developers, as sometimes the specification and verification effort may only be possible at the byte code level (the code producer does not exist and we have to use its legacy code or the code producer is not willing to supply the source code or the code was written in a low level language). In this way a specification language for the low level language can be a desirable part of a proof-carrying code (PCC [22]) infrastructure and this is part of the infrastructure to be built within the MOBIUS project [21].

Bytecode Modeling Language (BML) was proposed by Burdy et al [5] as a specification language for the low level Java byte code language. This formalism is based on Java Modeling Language (JML) [17,18]. Both languages allow to write specifications according to *design-by-contract* principles. In particular one can specify the preconditions and postconditions of methods, object invariants, loop invariants, include asserts in the code etc. As the specification language is developed within the MOBIUS project and the main target of the project are Java-enabled mobile devices such as mobile phones, the current version of BML assumes some simplifications of the Java byte code which are present in the J2ME platform — the Java platform for mobile devices with restricted resources.

The Java Modeling Language has rich tool support (see [4] for an overview). In particular, there are tools which check JML specifications at runtime [7], in extended static checking fashion [13,10], and allow to perform software certification [23,20,1]. Moreover, there are also tools which support the generation of JML annotations [12,9]. Unfortunately, the tool support for BML is as of now far from being close to this rich functionality.

The successful adoption of new formalisms is highly dependent on how useful they are for the programmers and software designers. They can only be useful when tool support is provided that allows editing and manipulating the expressions of the formalism and then to obtain helpful feedback based on the supplied annotations. In this paper we present the first step in providing such tool support for the byte code specification: *Umbra*, an Eclipse [11] plugin to visually edit BML annotated byte code files and its backend library *BMLlib*, an independent library to manipulate BML specifications in byte code. This provides the most basic support for the BML formalism and forms a basis on which other tools can be built. Even this very basic support can serve as a way of documenting the class files which were developed directly without the source code files. Our editor also allows to merge changes done in the source code editor with those done in the byte code editor. Thanks to that one can optimize some methods in the source code and some in the byte code while keeping the possibility to repeatedly improve on both ends. In this process, the BML annotations, in particular assertions, may be used to make sure that the introduced optimisations do not break the assumed functionality. The programmers may work on the annotated files which contain the specifications and strip them with the use of an obfuscator (e.g., *ProGuard* [16]) when the code is shipped to the users.

The obtained editor is not the only existing editor of class files. The editor differs from the competitors such as *CafeBaBe* [6] or Java Bytecode Editor (*JBE* [15]) in that it does not provide the view of the whole tree-like structure of a class file. Instead, the editor focuses on the actual program. In a sense, it provides the user with the functionality similar to *Jasmin* [14], which allows to transform a textual representation of a class file into the actual class file. However, *Jasmin* resembles more a compiler than an editor in that it does not allow to modify existing classes. The additional functionality which is absent from the other solutions, but which is available in our *Umbra* tool, is the ability to edit both the byte code instructions and BML specifications. The *JACK* tool [2] has also the ability to produce byte code instructions and BML specifications, but it is impossible to edit the specifications. Besides the tool is not maintained any more and it handles an obsolete version of BML.

This paper is organised as follows. Section 2 shows an example specification which can be viewed and edited by the BML tools. Section 3 presents an outline of the BML language. Section 4 gives an overview of the tool functionality and presents our experience with its use. Section 5 relates the architecture of the editor and the BML library. We conclude in Section 6.

2 An example of using BML tools

In order to convey an intuitive feel on how the tools work and how the BML specifications are presented, we consider an example.

Source code

Consider the class presented in Figure 1. This is an excerpt from a class which implements a sequence of objects. The sequence is implemented in an array (`list`). For brevity, our example contains only one method `replace` that takes two references to objects `obj1` and `obj2` and replaces the first occurrence in our sequence with the second one.

```
 1  public  class List {

    private Object[]  list;

 5  ...

    /*@ requires list != null;
      @ ensures \result == (\exists int i;
 9    @                      0 <= i && i < list.length &&
      @                      \old(list[i]) == obj1 && list[i] == obj2);
      @*/
    public boolean replace(Object obj1,Object obj2) {
13    int i;
      /*@
        @ loop_modifies list[*], i;
        @ loop_invariant i <= list.length && i >=0 &&
17      @                (\forall int k;0 <= k && k < i ==>
        @                                list[k] != obj1);
        @*/
      for (i = 0;  i < list.length;  i++ ){
21      if (list[i] == obj1) {
          list[i] = obj2;
          return true;
        }
25    }
      return false;
    }
    }
```

Figure 1. An example class `List.java` which contains a single method `replace`

Apart from the Java instructions, the listing in Figure 1 contains specifications in JML. In line 7 the precondition of the method requires the callers of `replace` to ensure that the private field `list` of the `List` object is not `null` before the method is called. Besides the precondition, there is also a postcondition which must be fulfilled by the method when it returns. This postcondition states that the result of the method is a Boolean value which is true when some element in the list has the value `obj1` before the call of the method (`\old(list[i]) == obj1`) and `obj2` after the call (`list[i] == obj2`). Note that these specifications are not complete as they say nothing about the rest of the representation array; for example a legitimate implementation of this specification is a code which just fills in the array with `obj2` up to the first occurrence of `obj1`.

In addition to the specifications which describe the input-output behaviour of this method, there is also a specification of the loop that implements the `replace` method. The loop_modifies clause describes which variables can be modified during the execution of the loop (all the entries in the list array, list[*]; and the local variable i). The loop_invariant clause contains the invariant, i.e., a formula which should hold right in front of the loop body at each iteration of the loop. In this case, the invariant states that all the elements of the array list before the current value of the index i are different than the parameter obj2.

Byte code

We now consider translating the source code in Figure 1 to byte code. Since the binary byte code files are not human readable, we rely on a textual representation of the byte code which is similar to the output of the javap utility (actually, its mnemonics are generated by BCEL, Byte Code Engineering Library [3]). The general layout of such a textual representation is presented in Figure 2. The first line shows the package declaration for the given class. For uniformity, we decided to explicitly state a fixed package name ([default]) in case the class has no package specified. Then the name of the class and its content follow. The content consists of a certain number of class-level specifications (see next section for details), then the constructors are listed together with their specifications and instructions and finally the methods are listed also with specifications and instructions.

```
package [default]

public class List

/*@ invariants, static invariants, constraints etc. @*/

/*@
  @ specification of the constructor with no parameters
  @*/
public void <init>()
0:      ...
  instructions and specifications

other constructors


/*@
  @ specification of the method replace
  @*/
public boolean replace(Object obj1, Object obj2)
0:      ...
  instructions and specifications

other methods
```

Figure 2. The layout of the textual representation of the class file for the List class

The actual mnemonics of the compiled version of our example method replace are presented in Figure 3. The labels correspond to instruction positions in the binary representation of the code and depend on the length of particular instructions. This method is accompanied with BML specifications corresponding to the JML specifications from Figure 1. First, there is the input-output specification of the replace method. Then its header follows and at last there is the sequence of byte code mnemonics with the loop specification among them. Let us concentrate for a while on the byte code program. The instructions labelled 0–3 perform the assignment i=0 from line 20 of the source code. The following goto instruction jumps to the place where the loop condition (`i <= list.length`) is checked. This is done in instructions 27–33 of the byte code. Line 33 contains a conditional jump which, in case the loop guard is true, jumps inside the loop. The source code instructions in lines 21–24, that constitute the body of the loop are translated into lines 5–23 of the byte code. Finally, the increment instruction from line 20 is translated into the `iinc` instruction in line 24 of the byte code.

```
/*@
  @ requires this.list != null
  @  {|
  @   precondition true
  @   modifies \everything
  @   ensures \result ==
  @      (\exists int var_0;
  @         0 <= var_0 &&
  @         var_0 < this.list.length &&
  @         old_this.list[var_0] == obj1 &&
  @         this.list[var_0] == obj2)
  @   exsures  Ljava/lang/Exception;:   false
  @  |}
  @*/
public boolean replace(Object obj1, Object obj2)
0:     iconst_0
1:     istore_3
2:     goto              #27
5:     aload_0
6:     getfield          List.list  [Ljava/lang/Object; (18)
9:     iload_3
10:    aaload
11:    aload_1
12:    if_acmpne         #24
15:    aload_0
16:    getfield          List.list  [Ljava/lang/Object; (18)
19:    iload_3
20:    aload_2
21:    aastore
22:    iconst_1
23:    ireturn
24:    iinc              %3      1
/*@
  @ loop_specification
  @   modifies this.list[*], i
  @   loop_inv i <= this.list.length &&
  @     i >= 0 &&
  @     (\forall int var_0;
  @       0 <= var_0 && var_0 < i ==>
  @       this.list[var_0] != obj1)
  @   decreases this.list.length - lv[3]
  @*/
27:    iload_3
28:    aload_0
29:    getfield          List.list  [Ljava/lang/Object; (18)
32:    arraylength
33:    if_icmplt         #5
36:    iconst_0
37:    ireturn
```

Figure 3. The method `replace` in the `List` class

Apart from the byte code mnemonics, the listing in Figure 3 contains the specifications of the method input-output behaviour and the specifications for the loop. The `requires-ensures` pair which is present in the source code in the lines 7–11 is represented by the annotation before the byte code version of the `replace` method. In the byte code, we have certain flexibility in the placement of the `requires` clause in case only one such clause occurs in the source code. We can present the code in the `requires` clause at the beginning of the specification or in one of the alternative `precondition` clauses which are associated with postconditions (the intent is that whenever a particular alternative precondition holds at the entry to the method the corresponding postcondition must hold at the exit from the method). The `precondition` clause is accompanied not only by an `ensures` clause, but also by the `modifies` and `exsures` clauses. The latter are implicit in the source code and they must be made explicit at the byte code level.

In our case the `requires` clause from the source code is represented by the main `requires` clause in the specification. The `ensures` clause from the source code is translated into the `ensures` clause in the byte code preceded by `true` precondition. It is possible to formulate the condition in this way as the method can only be called provided that the main precondition (after `requires`) is fulfilled. The content of the `ensures` clause is indeed what we expect as it compares the `\result` with an existential expression that has exactly the same structure as the original one, but the name we quantified over is `var_0` instead of `i` and the references to the field `list` are prefixed with the references to `this` or `old_this`.

Besides the specifications that come directly from the byte code, there are also specifications which are implicit in the source code. The `modifies` clause expresses which variables can be modified in the course of the method execution. The default value `\everything` means that everything can be modified. The `exsures` clause expresses the postcondition in case an exception is thrown. Here it says that if an exception of a subclass of `Exception` is thrown from this method the `false` postcondition must be satisfied, so, in other words, no exception of a subclass of `Exception` can be thrown.

The body of the `replace` method contains the specification pertinent to the loop. Again, the clauses here correspond to the clauses in the source code (`modifies` and `loop_inv`) and are accompanied by a clause `decreases` which is absent from the source code. The presence of this clause shows that we are able not only to derive the byte code level specifications from the source code ones, but we can also introduce new specificational elements at the byte code level.

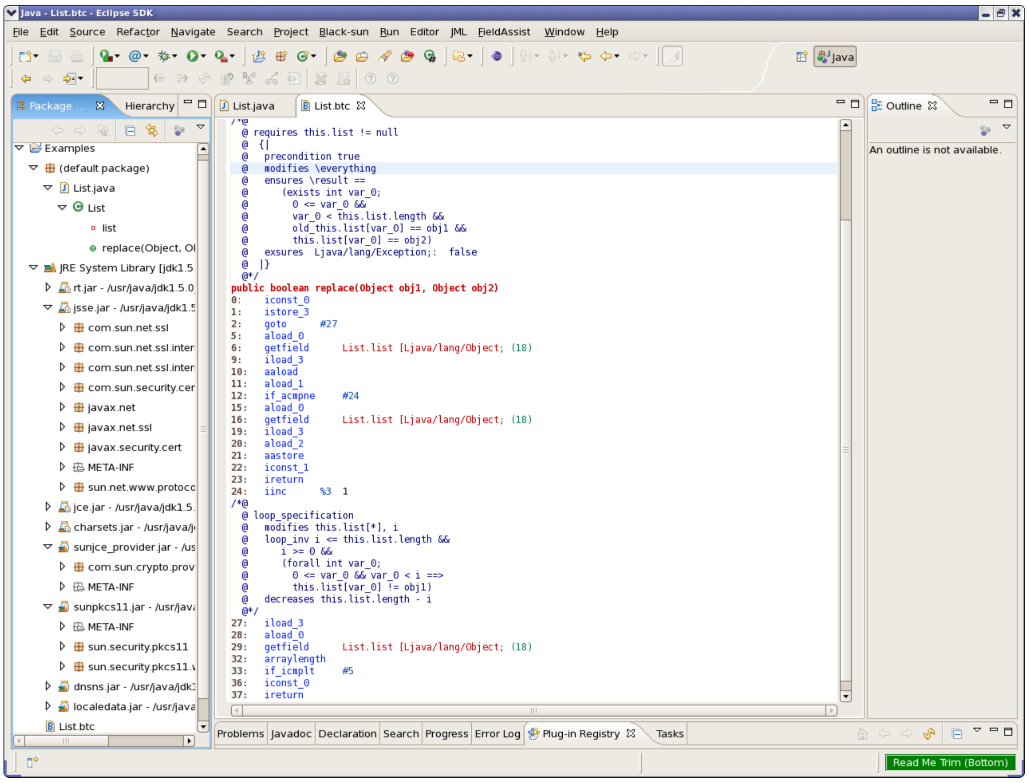
Figure 4 shows how the listing form Figure 3 looks like within the Umbra editor in the Eclipse integrated development environment.

3 Annotation language

The BML annotation language [5] is a direct descendant of JML [17,18] and the structure of specifications in both languages are very similar. Generally, the annotations can be divided into two groups: class annotations and method annotations.

Class annotations

Class annotations specify the behaviour of a class as a whole or of objects of that class. They also declare other elements (such as *ghost fields*, *model fields*, *data groups*, see below) that will be helpful in other specifications in that class or in other classes. The annotations refer either to all objects of that class — these are `instance` annotations — or to the class itself — these are `static` annotations. For


Java - List.btc - Eclipse SDK
File Edit Source Refactor Navigate Search Project Black-sun Run Editor JML FieldAssist Window Help
Java
Package ... Hierarchy
List.java List.btc
Outline
An outline is not available.
Examples
(default package)
List.java
List
list
replace(Object, Ol
JRE System Library [jdk1.5
rt.jar - /usr/java/jdk1.5.0
jsse.jar - /usr/java/jdk1.5
com.sun.net.ssl
com.sun.net.ssl.inter
com.sun.net.ssl.inter
com.sun.security.cer
javax.net
javax.net.ssl
javax.security.cert
META-INF
sun.net.www.protoco
jce.jar - /usr/java/jdk1.5.
charsets.jar - /usr/java/j
sunjce_provider.jar - /us
com.sun.crypto.prov
META-INF
sunpkcs11.jar - /usr/java
META-INF
sun.security.pkcs11
sun.security.pkcs11.
dnsns.jar - /usr/java/jdk1
localedata.jar - /usr/java
List.btc
Problems Javadoc Declaration Search Progress Error Log Plug-in Registry Tasks
Read Me Trim (Bottom)

```
/*@
  @ requires this.list != null
  @  {|
  @   precondition true
  @   modifies \everything
  @   ensures \result ==
  @      (exists int var_0;
  @        0 <= var_0 &&
  @        var_0 < this.list.length &&
  @        old_this.list[var_0] == obj1 &&
  @        this.list[var_0] == obj2)
  @   exsures  Ljava/lang/Exception;:  false
  @  |}
  @*/
public boolean replace(Object obj1, Object obj2)
0:    iconst_0
1:    istore_3
2:    goto      #27
5:    aload_0
6:    getfield      List.list [Ljava/lang/Object; (18)
9:    iload_3
10:   aaload
11:   aload_1
12:   if_acmpne     #24
15:   aload_0
16:   getfield      List.list [Ljava/lang/Object; (18)
19:   iload_3
20:   aload_2
21:   aastore
22:   iconst_1
23:   ireturn
24:   iinc      %3  1
/*@
  @ loop_specification
  @   modifies this.list[*], i
  @   loop_inv i <= this.list.length &&
  @      i >= 0 &&
  @      (forall int var_0;
  @        0 <= var_0 && var_0 < i ==>
  @        this.list[var_0] != obj1)
  @   decreases this.list.length - i
  @*/
27:   iload_3
28:   aload_0
29:   getfield      List.list [Ljava/lang/Object; (18)
32:   arraylength
33:   if_icmplt     #5
36:   iconst_0
37:   ireturn
```

Figure 4. The compiled `List` class inside the Eclipse environment. This picture presents most of the `replace` method together with all the relevant specifications in BML

the sake of brevity, we will describe only the instance version of all annotations here. The meaning of their static counterparts is analogous.

The most prominent example of a class annotation is *class invariant*. It specifies a condition that is supposed to hold for all objects of that class in all *visible* states, i.e., before and after every method call and after all constructors. In other words, the class invariant is another precondition and another postcondition of all methods. For example a class invariant for the `List` class described in the previous section could say that the elements of the list are not null. This invariant is not preserved by the `replace` method though.

Another class annotation is *history constraint*. It specifies how object field values can evolve with time. For example it can say that the length of the list can only grow.

The last interesting class annotation is *initially clause* which is a formula that is supposed to hold just after the object initialisation is complete. i.e., after execution of any constructor.

In order to increase expressibility of specifications one can declare on the class level a number of elements that can be used in specifications. These include:

- *Ghost fields* — fields existing only in the specification. These fields can be modified by explicit *set* annotations within method's code (see below). In the example of a list this could be e.g., the element most recently inserted in the list; the `replace` method should set this field to `obj2`.

- *Model fields* — these, together with *represents clauses* are a shorthand for a longer formula or expression, e.g., a field called `has_nulls` could be declared as a model field and the corresponding represents clause would contain a fomula stating that `has_nulls` is equivalent to the existance of null as an alement of the `list`. The separation of *represents clauses* from the *model field* declarations is useful to declare a model field in one class and the clause in a subclass or to specify the represents clause for a sub-field (i.e., for a model field of an object stored in a field).
- *Data groups* — these are named lists of fields and sub-fields that are referenced by `modifies` clauses of method specifications and loop specifications.

Method annotations

The most important kind of method annotations is called *method specification.* They describe the precondition, postcondition, and assignable clauses of the method. The precise description is given in Section 2.

Other method annotations are actually specification elements appearing in the code. They include:

- Declarations of *local ghost variables* — they are similar to *ghost fields* declared in a class but their lifetime and scope is limited to the execution of one part of a method.
- *Set* instructions — they correspond to Java assignments, but they operate on ghost fields and local ghost variables.
- *Loop specifications* — they introduce the invariant of the loop, the `modifies` clause, saying what can be modified by the loop body, and the `decreases` clause to prove the loop's termination.
- *Assert* instructions — they are similar to Java `assert` instructions, that is, they state facts about fields, variables etc. that are supposed to hold in a given point of the code execution.

Additional modifiers

BML introduces also new modifiers `non_null` and `nullable` to specify that a declared field, local variable, method parameter or a method result cannot (resp. can) have the null value. Note that this can also specified in a method specification or a class invariant using a normal formula, but using a modifier is much more concise.

Another important modifier is `pure`. It can accompany a method declaration, and it states that the method terminates and does not modify any existing objects; it can create new ones though. Pure methods can be used in specification formulae (see below).

BML formulae

The formulae used in invariants, assertions, and pre- and postconditions are Java Boolean expressions using only pure methods, augmented with a number of predicates such as `\old`, quantifiers such as `\forall`, `\exists`, and other similar operators such as `\sum`, `\num_of` etc.

Binary format of BML annotations

There is an ongoing effort to precisely document the binary format of BML annotations inside Java class files [8]. The class files representing byte code with annotations are regular Java class files, executable and usable by all Java tools, where annotations are stored within additional attributes (see Chapter 4.7 of [19]). The names of BML related attributes start with the prefix `org.bmlspecs` (e.g., as in `org.bmlspecs.ClassInvariant`, `org.bmlspecs.MethodSpecification`, etc.) and according to the specification of the Java Virtual Machine they are supposed to be silently ignored by the Machine, since their names are not part of the original JVM specification.

Of course, following the logical structure of class files, class specifications are stored as class attributes, method specifications as method attributes attached to a particular method and specifications inserted in the code are attributes of the JVM Code attribute of the given method.

4 Functionality and experience

Additional features in the Java editor

The *Umbra* plugin adds three new buttons to the toolbar of the Java editor: the first button generates the byte code mnemonics, the second one allows to move between a source code line and the corresponding sequence of byte code instructions, and the third one allows to merge changes done at the source code level into the (modified) byte code.

The button which generates the byte code mnemonics activates the byte code *disassembling* process and opens an Eclipse editor with the byte code representation of the current class. At this point, the user is able to edit the byte code mnemonics and introduce or edit BML specifications. We elaborate more on that later on.

The second button permits the user to see the byte code realisation of a particular line in the source code. The user points the source code editor cursor at a line of interest and then pressing the button *moves* the focus to the byte code editor of the same class in which the lines corresponding to the source code line are highlighted. This feature facilitates the process of understanding the byte code. In case the user cannot understand the structure of the byte code, she or he can go to the accompanying source code and point particular instructions to obtain a clear division of the instruction stream into more comprehensive chunks.

The third button provides the user with possibility to develop the code both at the source code level and at the byte code level. It allows to incorporate into the (modified) byte code version of the class the changes that were made at the source code level. It works so that whenever a method was modified at the source code level only, its new byte code representation can be safely *combined* with the (modified) byte code of other methods.

Features of the byte code editor

In the byte code editor, we have buttons that realise the features that correspond to the features in the Java editor: a button to *move* from byte code to the corresponding line in the source code and a button to *combine* modifications at the source code level with the ones at the byte code level. Besides, we have a button that saves the current content of the byte code document and reformats it using the internal pretty printing mechanism. Other buttons are auxiliary ones, they handle the history of changes, colouring mode, and display help information.

We decided not to associate the byte code editor with the `.class` files, but to make it operate on textual `.btc` (short for *ByTe Code*) files. The format of these textual files is similar to the one generated by `javap` utility from the standard Java Software Development Kit. This design choice was motivated by two reasons. First, it is useful to circulate the text files with the byte code in case one wants to demonstrate some issue (e.g., a useful byte code that is not well-formed according to the Java Virtual Machine). Another reason is that the textual format is more convenient in case additional information at the byte code level should be written in the file. The textual representation must always be defined in this case to enable the presentation in the editor. However, one can decide not to define the class file binary representation when the editor is associated with a textual file.

The *Umbra* editor allows to change the byte code mnemonics, i.e., to add new instructions, to change the existing ones, and to delete them. It checks the syntactic correctness of the edited code so that the user knows if his byte code script can be transformed into a class file. The code editor functionality is limited in the sense that it does not allow to add or to remove methods. These actions must be performed at the source code level — i.e., one must add a method on the source code level and commit the change into the class file.

Except from the possibility to edit the instructions of the program the editor allows to edit the BML specifications. The user can add every specification she or he wants and the editor informs the user if the specification is syntactically correct.

Experience

The *Umbra* editor has already proved to be useful. We used it successfully to generate various experimental byte code files that illustrate the intricacy of the byte code verification process (e.g., files that are rejected by the standard JVM verifier even though they do have the property for the lack of which they are rejected) and inconsistencies between the semantics of Java and the semantics of the byte code (e.g., that final fields can be modified from the byte code level). The editor proved to be very useful for this task as it is quite easy to generate these class files compared to generation by hand.

We also used *Umbra* to do a small study on how easy it is to write specificataions to methods at the byte code level. It turned out that the editor needs additional features to make the process convenient. First of all, the current mechanism to handle comments added by the user is not sufficient: the comments should be stored in a permanent way. Additionally, the editor could display the names of local variables instead of their numeric representation. Otherwise the user is forced to continually perform fatiguing deciphering of their meaning.

5 Architecture

In this section we give the overview of architecture of the *BMLlib* library and the *Umbra* plugin.

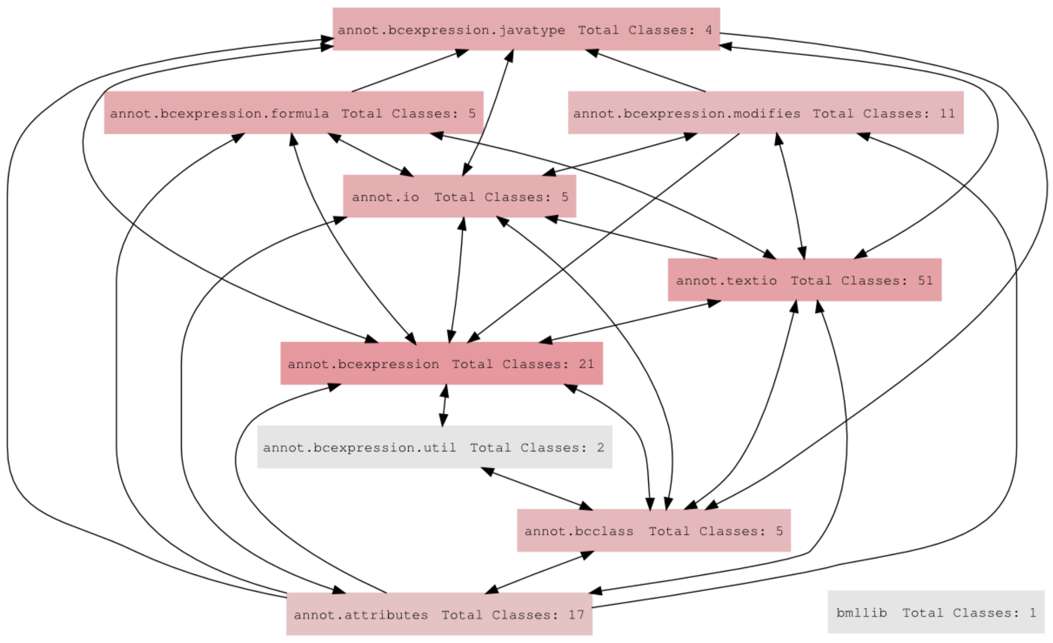

annot.bcexpression.javatype Total Classes: 4
annot.bcexpression.formula Total Classes: 5
annot.bcexpression.modifies Total Classes: 11
annot.io Total Classes: 5
annot.textio Total Classes: 51
annot.bcexpression Total Classes: 21
annot.bcexpression.util Total Classes: 2
annot.bcclass Total Classes: 5
annot.attributes Total Classes: 17
bmllib Total Classes: 1

Figure 5. The dependency graph of the *BMLlib* internal packages

BMLlib

The *BMLlib* uses the Byte Code Engineering Library (BCEL [3]) to read and write class files, and to process standard Java class components. The *BMLlib* contains 10 modules which handle different tasks associated with the library. The overall interdependencies between the packages are presented in Figure 5.

The `annot.bcclass` contains the representation of the basic building blocks of a BML enriched class file. This includes abstract representation of classes, methods and the constant pool. The classes in this package contain the methods which trigger interpreting and saving of BML annotated classes.

The module `annot.attributes` contains the abstract representation of BML attributes and handles the task of writing them to a class file.

The module `annot.bcexpression` together with its submodules which are located in packages `annot.bcexpression.formula`, `annot.bcexpression.javatype` and `annot.bcexpression.modifies` contain the abstract syntax tree representing the BML clauses. The syntax tree can be converted into binary form and embedded into a class file. Finally, the module `annot.bcexpression.util` contains walker classes that can be used to process the abstract syntax tree of BML clauses.

The module `annot.io` contains an interface for reading and writing BML attributes and expressions into class files and `annot.textio` contains an interface for reading and writing the textual representation of BML clauses.

The module `bmllib` contains only the class which handles the Eclipse plugin activation protocol. All other modules are completely independent from Eclipse and thus can be used in standalone tools like compilers, verifiers or annotation generators.

Umbra

The *Umbra* editor is tightly integrated with Eclipse. Apart from the *BMLlib* library, it also directly depends on the BCEL. *Umbra* is divided into several modules which are grouped into Java packages. Each of the modules handles a specific task associated with the functionality of the editor. The overall dependency graph of the packages in the *Umbra* editor is presented in Figure 6.

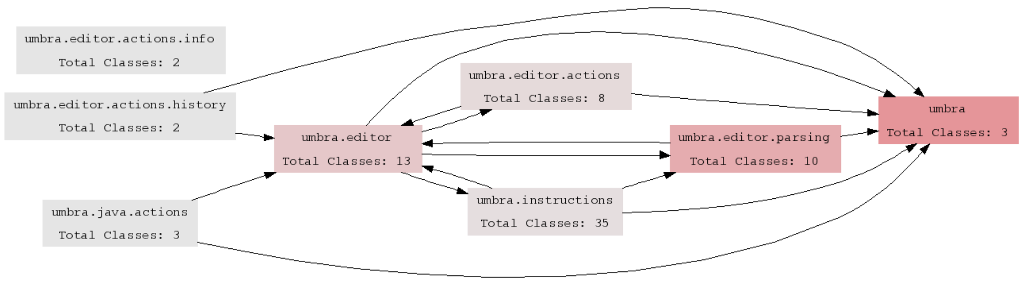


Figure 6. The dependency graph of the *Umbra* editor internal packages

The interface with the Eclipse plugin framework is handled in the `umbra` package. This concerns, however, only the activation interface. The contributions of the plugin to the GUI of Eclipse are spread over other modules (i.e., `umbra.java.actions`, `umbra.editor.actions` and its submodules, and `umbra.editor`). The `umbra` package contains also some classes which can in principle be used in any other module of the plugin.

The `umbra.java.actions` module contains the implementation of the actions that are triggered from within the Java source code editor (i.e., disassemble, synchronise, and combine). In particular, it triggers the creation of the byte code editor.

The byte code editor itself is implemented in the module `umbra.editor` and its submodules. This module contains an interface with *BMLlib* and with an internal abstract syntax tree for byte code mnemonics. The GUI actions which are available from within the byte code editor are implemented in the module `umbra.editor.actions` and its submodules.

The submodule `umbra.editor.parsing` contains a little parser which parses the general structure of the byte code file in the textual representation. This enables the possibility to present the byte code program in a colourful notation that eases the understanding of the program internal structure.

At last, the submodule `umbra.instructions` defines an abstract syntax tree which enables the syntax checking of the edited instructions. This is a crucial part which allows to state that a particular text is a syntactically well formed representation of a byte code program. This module does not perform any byte code verification though.

6 Conclusions

The constantly growing use of various programming languages that compile to the Java Virtual Machine platform may easily result in a situation in which the only common ground for understanding programs is the byte code level. We believe that the inclusion of specifications into the byte code can considerably ease the process of making the byte code programs understandable. Additionally, our development effort, the result of which is *Umbra* and *BMLlib*, can help to build a program verification platform that works at the byte code level. Moreover, as the BML specifications are embedded into class files the presented tools can be used at the code producer end in proof-carrying code scenarios to prepare parts of the certificates or at the policy provider side as a desirable code policies editor.

There are several trade-offs in the design of a byte code editor with specification support. The first issue is the representation of the edited files. We believe that there will be situations which will require to add additional information to the byte code which should be absent from the class files (e.g., comments that explain or document certain features of the code). Similarly, there are several trade-offs for the way BML specifications are presented to the user. We believe that in order to make the technology successful the specifications must be easy to understand by humans. We believe that a big advantage of BML is that it is similar to JML, which should make the formalism acceptable to programmers. Moreover we expect that with a little additional computational overhead one can make the specifications as comprehensive as the specifications at the source code level. It is also important that the specifications should not incur huge overhead on the class files as that would limit the applicability of the technology and impair their acceptance on the end user side. That is why the BML clauses are represented in a concise binary format inside class files.

We are aware that the development of specifications at the byte code level is not easy. Moreover, the current tool support makes it only one small step easier. However, the presented tools can serve to inspect the specifications generated by other tools which should enable easier debugging. Additionally, the separate *BMLlib* library allows to develop new tools (e.g., compiler, verifier, annotation generator, etc.) that can work with BML annotated class files.

We also believe that the existence of such a tool can encourage people to search for new methods to pretty-print the byte code instructions, to group them, or to provide hint systems to show the dependencies between them.

The *Umbra* editor together with accompanying libraries can be downloaded at `http://www.mimuw.edu.pl/~alx/umbra/`.

References

[1] Ahrendt, W., T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager and P. H. Schmitt, *The KeY tool*, Software and System Modeling **4** (2005), pp. 32–54.

[2] Barthe, G., L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova and A. Requet, *JACK: a tool for validation of security and behaviour of Java applications*, in: *FMCO: Proceedings of 5th International Symposium on Formal Methods for Components and Objects*, LNCS **4709** (2007).

[3] *BCEL — Byte Code Engineering Library*, available at http://jakarta.apache.org/bcel/.

[4] Burdy, L., Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino and E. Poll, *An overview of JML tools and applications*, in: T. Arts and W. Fokkink, editors, *Workshop on Formal Methods for Industrial Critical Systems*, Electronic Notes in Theoretical Computer Science **80** (2003), pp. 73–89, preprint University of Nijmegen (TR NIII-R0309).

[5] Burdy, L., M. Huisman and M. Pavlova, *Preliminary design of BML: A behavioral interface specification language for Java bytecode*, in: *Fundamental Approaches to Software Engineering (FASE 2007)*, LNCS **4422** (2007), pp. 215–229.

[6] *CafeBaBe — bytecode browser*, available at http://www.geocities.com/CapeCanaveral/Hall/2334/Programs/cafebabe.html.

[7] Cheon, Y., "A Runtime Assertion Checker for the Java Modeling Language," Ph.D. thesis, Iowa State University (2003).

[8] Chrząszcz, J., M. Huisman, J. Kiniry, M. Pavlova, E. Poll and A. Schubert, *BML Reference Manual*, available at http://www-sop.inria.fr/everest/BML.

[9] Cielecki, M., J. Fulara, K. Jakubczyk, Ł. Jancewicz, J. Chrząszcz, A. Schubert and Ł. Kamiński, *Propagation of JML non-null annotations in Java programs*, in: *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java* (2006), pp. 135–140.

[10] Cok, D. and J. R. Kiniry, *ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system*, in: G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet and T. Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, LNCS **3362**, 2005, pp. 108–128.

[11] *Eclipse — an open development platform*, available at http://www.eclipse.org.

[12] Ernst, M. D., J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz and C. Xiao, *The Daikon system for dynamic detection of likely invariants*, Science of Computer Programming **69** (2007), pp. 35–45.

[13] Flanagan, C., K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata, *Extended static checking for Java*, Programming Languages Design and Implementation **37**, 2002, pp. 234–245.

[14] *Jasmin — an assembler for the Java Virtual Machine*, available at http://jasmin.sourceforge.net.

[15] *JBE — Java Bytecode Editor*, available at http://cs.ioc.ee/~ando/jbe.

[16] Lafortune, E., *Proguard*, manual available at http://proguard.sourceforge.net/.

[17] Leavens, G., A. Baker and C. Ruby, *Preliminary design of JML: A behavioral interface specification language for Java*, Technical Report TR 98-06y, Iowa State University (1998), (revised since then 2004).

[18] Leavens, G., E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok and J. Kiniry, "JML Reference Manual," (2005), in Progress. Department of Computer Science, Iowa State University. Available from http://www.jmlspecs.org.

[19] Lindholm, T. and F. Yellin, "The Java (TM) Virtual Machine Specification (Second Edition)," Prentice Hall, 1999.

[20] Marché, C., C. Paulin-Mohring and X. Urbain, *The Krakatoa tool for certification of Java/JavaCard programs annotated with JML annotations*, Journal of Logic and Algebraic Programming **58** (2004), pp. 89–106.

[21] *Mobius — Mobility, Ubiquity and Security. Enabling proof-carrying code for Java on mobile devices*, http://mobius.inria.fr, information Society Technologies programme of the European Commission FET project IST-2005-015905.

[22] Necula, G., *Proof-carrying code*, in: *Proceedings of POPL'97* (1997), pp. 106–119.

[23] van den Berg, J. and B. Jacobs, *The LOOP Compiler for Java and JML*, in: *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS **2031** (2001), pp. 299–312.