Replace this file with prentcsmacro.sty for your meeting,
or with entcsmacro.sty for your meeting. Both can be
found at the ENTCS Macro Home Page.

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

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

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

Abstract

Byte-code Modeling Language is a currently developed specification language designed to support specification and verification of Java byte code files. We present an editor *Umbra* of byte code files which supports 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 inspect class files with BML specifications and generate them which can be exploited both in order to develop specified versions of class files and to inspect existing class files for debugging purposes.

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

Version: 558M

1 Introduction

One of the main achievements of the high level programming languages (e.g. Algol, Pascal, C) is the common adoption of the abstraction as the main programming paradigm. That means the executable code is divided into smaller pieces that can be to much 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

[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

©2008 Published by Elsevier Science B. V.

what is required and what is made available to a particular piece of code. Moreover, the size of the currently built 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 the 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 also support writing purely about *what* should be done with all the details on *how* it is done omitted. As they are designed to be easily understood by programmers, they provide means to document the source code. The documentation done in a specification language has also the advantage that it can be verified automatically that the source code really implements the documented features.

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 the source code level programming. Furthermore, the presence of specifications allows to locate more precisely the actual reasons for program misbehaviour.

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 can also be affected by improper operation of the code. In this case, however, the artifact which should be analysed is 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 code producer does not exist and we have to use its legacy code, the code producer is not willing to supply the source code, the code was written in low level language) the specification and verification effort may be possible only at the byte code level. In this way a specification language for low level language can be a desirable part of a proof-carrying code (PCC) infrastructure and this is part of the infrastructure to be built within the MOBIUS project (see `http://mobius.inria.fr`).

Bytecode Modeling Language (BML) was proposed by Burdy et al [4] as a specification language for low level Java byte code language. This formalism is based on Java Modeling Language (JML) [12,13]. 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 the mobile devices such as mobile phones, the current version of BML assumes some simplifications of the Java byte code which are present in J2ME platform — the Java platform for mobile devices with restricted resources.

The Java Modeling Language has rich tool support (see [3] for an overview). In particular, there are tools which check the JML specifications at the runtime [5], in extended static checking fashion [10,8], and allow to perform actual software verification [15,14,1]. Moreover, there are also tools which support the generation of JML annotations [9,7]. The tool support for BML is now much undeveloped.

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 to edit and manipulate 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 the tool support for the byte code specification. The tools which are presented here allow to read and write BML specifications in class files and to edit them manually together with the byte code they describe. This provides the most basic support for the formalism and forms a base to which on can plug-in tools that are able to generate useful feedback. Even this very basic support can serve as a way of documenting the class files which were developed without help of the source code files. The existing functionality makes possible yet another scenario. The editor allows to merge changes from both source code editor and byte code editor. Due to that one can gradually replace the existing implementation in Java source code with manually optimised code that has the same functionality. The BML annotations, in particular asserts, 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 [11]) when the code is shipped to the users.

The obtained editor is not the only editor of the class files. The editor differs from the editors such as CafeBaBe (`http://www.geocities.com/CapeCanaveral/Hall/2334/Programs/cafebabe.html?200816`) or Java Bytecode Editor (`http://cs.ioc.ee/~ando/jbe/`) in that it does not provide the view of the whole tree-like structure of a class file. Instead, the editor concentrates the focus of its user on the actual program. In a sense, it provides the user with the functionality similar to Jasmin (`http://jasmin.sourceforge.net/`). It allows to edit a textual representation of a class file and then the textual representation is transformed into the actual class file. The additional functionality which is absent from the other solutions, but which is available in *Umbra* is the ability to edit both the byte code instructions and BML specifications.[7]

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

[7] Jack tool [2] has also the ability to edit both the byte code instructions and BML specifications, but it is not maintained any more and it handles an obsolete version of BML.

2 An example of using BML tools

One can get a good feeling on how the tool works and how the BML specifications look like by following an example.

Source code

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

```
public  class List {

private Object[] list;

...

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

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

Except from the Java commands the listing on Figure 1 contains specifications in JML. We have here the precondition of the method which requires all its callers to ensure that the private field `list` of the `List` object is not `null` before the method is called. Except from the precondition we have also a postcondition which must be ensured by the method when it returns. This postcondition says that the result of the method is a boolean value which is true when some element of the list representation 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 and that a legitimate implementation of this specification is e.g. 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 we have here a specification of the invariant of the loop that realises the `replace` method. The specification for the loop describes in `loop_modifies` clause which variables can be modified during its execution (all the entries in `list` array, `list[*]`; and the local variable `i`, `i`). Aside of that, it expresses in `loop_invariant` clause the invariant which should hold right before the content of the loop is executed. In this case, the invariant says that all the elements of the array `list` before the current value of the loop control variable `i` are different than the parameter `obj2`.

Byte code

The source code from Figure 1 can be translated to the byte code form. The binary byte codes are hardly human readable so we rely on a textual representation of the byte codes which is based on the output of `javap` utility. The general layout of such a textual representation is presented on Figure 2. First, we see 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 bunch of class-level specifications i.e. invariants, static invariants, constraints etc. Then the constructors are listed together with their specifications and instructions and finally the methods are listed with 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 on Figure 3. This method is accompanied with specifications corresponding to the JML specifications visible on Figure 1. First, we can see the input-output specification of the `replace` method. Then its header follows and at last we see the byte code mnemonics with the loop specification inlined in their sequence. Let us concentrate for a while on the byte code program. The instructions labelled 0–3 perform the assignment `i=0` from the line 19 of the source code. The following `goto` jumps to the place where the loop condition (`i <= list.length`) is checked. This is done in instructions 27–33 of the byte code. The line 33 contains a conditional jump which, in case the loop guard is true, jumps inside the loop. The instructions in the lines 20–23, that constitute the body of the loop are realised in the lines 5–23 of the byte code. Finally the increment instruction from the line 19 is realised by the `iinc` instruction in the line 24 of the byte code.

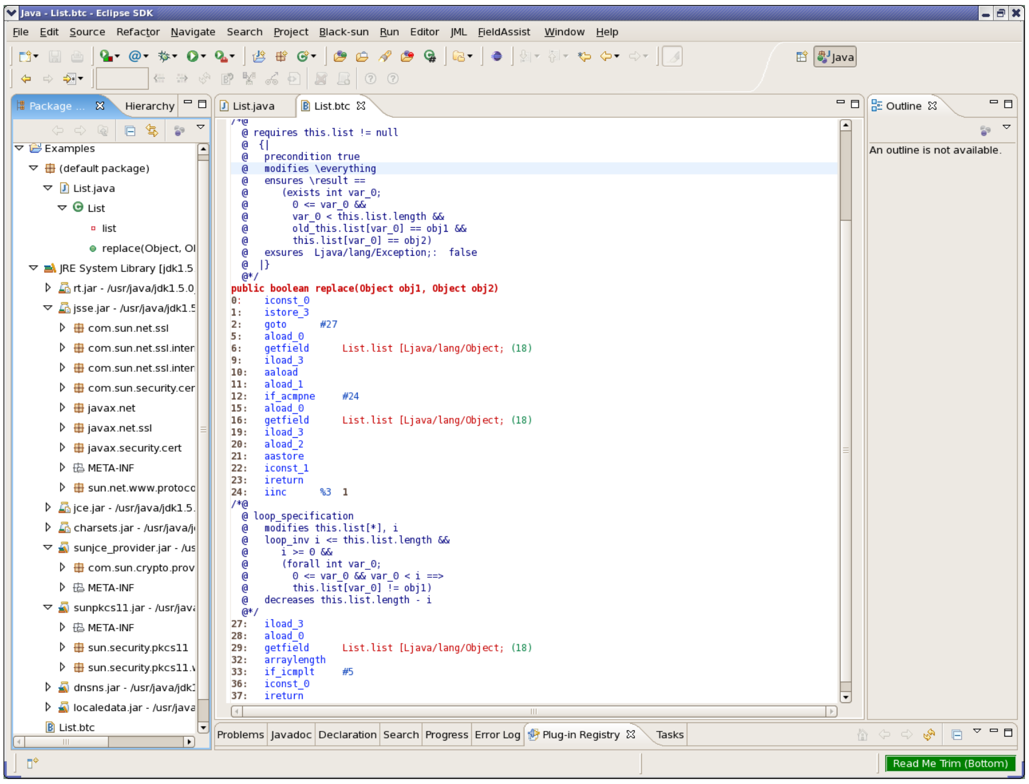
```
/*@
  @ requires this.list != null
  @  {|
  @   precondition true
  @   modifies \everything
  @   ensures \result ==
  @      (\exists int var_0;
  @         0 <= var_0 &&
  @         var_0 < this.list.length &&
  @         old_this.list[var_0] == obj1 &&
  @         this.list[var_0] == obj2)
  @   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

As we can see, the listing on Figure 3 contains except the byte code mnemonics 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 comment 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 associate 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 we also see the `modifies` and `exsures` ones. These clauses are implicit in the source code and they must be made explicit at the byte code level.

In our case we see that 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 realised by 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`.

Except from the specifications that come directly from the byte code we see also specifications which are implicit in the source code. The `modifies` clause expresses which variables are modified in the course of the method execution. The default value is that everything can be modified which is represented by the expression `\everything`. The `exsures` clause expresses the postcondition in case an exception is thrown and says that no subclass of `Exception` class can be thrown from this method as the condition after `Exception` is thrown must be `false`.



```
/*@
  @ 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
```

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)

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 body of the `replace` method contains the specification pertinent to the loop. Again, we can see here clauses which correspond to the clauses in the source code (`modifies` and `loop_inv`) 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.

One can see that the listing from Figure 3 can be also studied from within the Eclipse integrated development environment on Figure 4.

3 Annotation language

The BML annotation language [4] is a direct descendant of JML [12,13] 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 the sake of brevity, we will describe only instance version of all annotations here. The meaning of their static counterparts is straightforward.

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 guaranteed to hold 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 a class level a number of elements that can be used in specifications. These include:

- *Ghost fields* — fields which exist only in the specification, that are explicitly modified by *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. the length 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*. It describes the precondition, postcondition and assignable clauses of the method. It is described in previous section.

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

- *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 variables.
- *Loop specification* — they introduce the invariant of the loop together with the `modifies` clause, saying what can be modified by the loop body, and, optionally, the `decreases` clause to prove the loop's termination.
- *Assert* instructions — they are similar to Java `assert` instructions, i.e. facts about fields, variables etc. that are supposed to hold in a given point of the code execution.

Additional modifiers

Apart from the annotations described above there is also a very concise way to specify that a declared field, local variable, method parameter or a method result cannot have the null value. In spite of the fact that all most of this can be specified in the method specification or a class invariant, there is a possibility to add a `non_null` annotation to the field, local variable or the method declaration. This annotation acts like a modifier similar to the Java words like `static`, `public` or `private`.

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

As of today, these two annotations are not handled by *BMLlib* and *Umbra*.

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`, quantifier such as `\forall`, `\exists` and other similar operators like `\sum`, `\num_of` etc.

Binary format of BML annotations

There is an ongoing effort to precisely document the binary format of BML annotated Java classes [6]. These are regular Java classes, executable and usable by all Java tools, where annotations are stored within so called `attributes`, whose names start with the prefix `org.bmlspecs` (e.g. `org.bmlspecs.ClassInvariant`, `org.bmlspecs.MethodSpecification` etc). Of course 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 Code attribute of the given method.

4 Functionality and experience

Additional functionality in the Java editor

The *Umbra* plugin adds to the toolbar of the Java editor three new buttons: one which generates the byte code mnemonics, one which allows to move between a source code line and the corresponding sequence of byte code instructions, and one which allows to add to take into account in the byte code file the changes which were made at the source code level.

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 on that more 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 the pressing of the button moves him 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 the byte code understanding. 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 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 byte code representation can be safely *combined* with the byte code representation by replacing the old method with the new one.

Functionality in the byte code

In the byte code editor, we have buttons that realise the functionality that corresponds to the functionality 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. Except for that we have a button that saves the current content of the byte code document and reformats it using the internal pretty printing mechanism. These buttons, that directly process the byte code, are accompanied by auxiliary ones that handle the history of the 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. These textual files have format 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 verified by the Java Virtual Machine). Another reason is that the textual format is more convenient in case additional byte code level information should be associated with the file. The textual representation must be always defined in this case to enable the presentation in the editor. However, one can avoid the need to define the class file representation of this information when the editor is associated with textual file.

The *Umbra* editor allows to edit 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.

In contrast to other existing editors, our solution does not allow to edit the whole structure of the byte code file. Moreover, it presents the user with the actual code of the program instead of the tree-like structure which is usually the way the class file is presented (see for instance Java Bytecode Editor, CafeBaBe or Jack). This makes the solution close to tools such as Jasmine which allow to edit text files which are then compiled into class files. However, *Umbra* provides the user with the possibility to add BML specifications to her or his code.

Experience

The editor already in its current functionality 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 have the property ensured by the Java verification process, but are not accepted by the standard Java verifier) 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 did using *Umbra* a small study on how easy it is to specify files at the byte code level. It turned out that the editor needs additional features to make the process feasible. First of all, the current mechanism to handle comments added by the user is not sufficient. The user needs a format which allows to store her or his comments in a permanent way. Additionally, the editor could hint the names of variables as the numeric representation triggers many checks on which particular variable is meant.

5 Architecture

The overall tool set is divided into two components. The first one *Umbra* provides the user with the GUI to conveniently edit the byte code files together with the

BML specifications. It allows the user to change the byte code mnemonics and to modify textual representation of BML specifications. The second one *BMLlib* focuses solely on handling the BML specifications. It can parse their textual representation and store them as appropriate class file attributes in accordance with the BML specification [6].

5.1 BMLlib

The *BMLlib* contains 10 modules which handle different tasks associated with the library. The overall interdependencies between the packages are presented on Figure 5.

The `annot.bclass` contains the representation of the basic building blocks of the 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 the interpreting and saving of the BML annotated classes.

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

The module `annot.bcexpression` together with its submodules located in packages `annot.bcexpression.formula`, `annot.bcexpression.javatype`, and finally `annot.bcexpression.modifies` contain the abstract syntax tree of the BML clauses. The nodes of the tree handle also the writing of the particular expressions in the binary form which can be embedded into the class file. Finally, the module `annot.bcexpression.util` contains helper code which is used in many classes of `annot.bcexpression`.

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

The separate module `bmllib` contains only the class which handles the Eclipse plugin activation protocol.

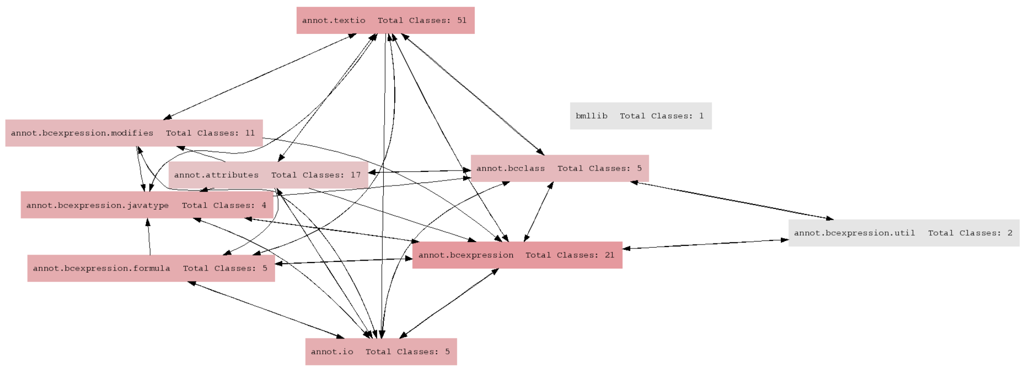

annot.textio Total Classes: 51
bmllib Total Classes: 1
annot.bcexpression.modifies Total Classes: 11
annot.attributes Total Classes: 17
annot.bclass Total Classes: 5
annot.bcexpression.javatype Total Classes: 4
annot.bcexpression.util Total Classes: 2
annot.bcexpression Total Classes: 21
annot.bcexpression.formula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

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

5.2 Umbra

The *Umbra* editor 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 on Figure 6.

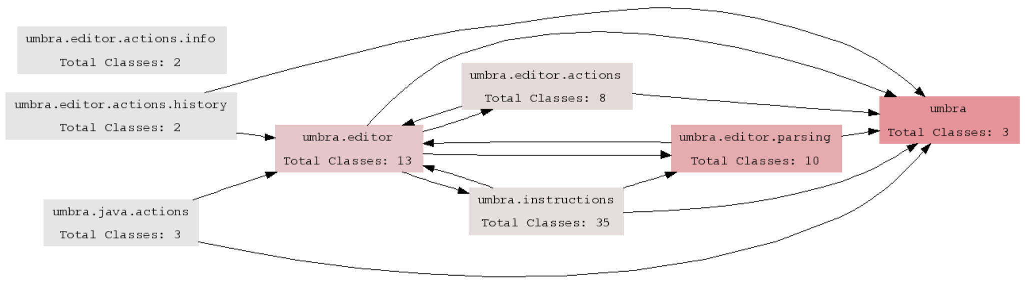


Figure 6. The dependency graph of the *Umbra* editor internal packages

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 tool.

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.eitor.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` contains 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 well formed representation of a byte code program (at least on the syntactical level, this module does not perform any byte code verification).

6 Conclusions

The constantly growing use of various programming languages that compile to 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*, brings closer the existence of a platform to verify programs that works at the byte code level. Moreover, as the BML specifications are embedded into the 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 desirable code policies.

There are several trade-offs in 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 files which should be absent from the class files (e.g. some 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. On the other hand, 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. We believe that a big advantage of BML is that it is similar to JML. This should make the formalism acceptable to programmers. On the other hand, we expect that with a little additional computational overhead one can make the specifications as comprehensive as the specifications at the source code level.

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 other tools (e.g. compiler) that can work with the BML enriched class files.

We also believe that the existence of such a tool can encourage people to research the possibilities to make their lives easier and to consider new methods to pretty-print the byte code instructions, to group them, or to provide hint systems to show the dependencies between them.

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 (2004), to appear.

[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 (2007).

[3] 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).

[4] 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.

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

[6] 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/index.html.

[7] Cielecki, M., J. Fulara, K. Jakubczyk and Łukasz Jancewicz, *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.

[8] 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.

[9] 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.

[10] 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. URL `citeseer.ist.psu.edu/flanagan02extended.html`

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

[12] 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).

[13] 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 Univer sity. Available from `http://www.jmlspecs.org`.

[14] 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.

[15] 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* (2001), pp. 299–312.