BML and related tools[⋆]

Jacek Chrząszcz[1], Marieke Huisman[2], and Aleksy Schubert[1]

[1] Institute of Informatics, University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland

[2] University of Twente, Faculty EEMCS, P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract The Bytecode Modeling Language (BML) is a specification language for Java bytecode, that provides a high level of abstraction, while not restricting the format of the bytecode. Notably, BML specifications can be stored in class files, so that they can be shipped together with the bytecode. This makes BML particularly suited as property specification language in a proof-carrying code framework. Moreover, BML is designed to be close to the source code level specification language JML, so that specifications (and proofs) developed at — the more intuitive — source code level can be compiled into bytecode level.

This paper describes the BML language and its binary representation. It also discusses the tool set that is available to support BML, containing BMLLib, a library to inspect and edit BML specifications; Umbra, a BML viewer and editor, integrated in Eclipse; JML2BML, a compiler from JML to BML specifications; BML2BPL, a translator from BML to BoogiePL, so that the BoogiePL verification condition generator can be used; and CCT, a tool to store proofs in class files.

1 Introduction

Typically, if formal methods are used in the process of software development, they are applied at source code level [18,24,6]. Modern programming languages introduce a strict structure on the code and provide a layer of abstraction that makes a program quite comprehensive for humans. The use of an appropriate specification language introduces another, even higher, level of abstraction into the software development process. An advantage of this abstraction is that it reduces the difficulty of program construction, in particular when it is supported by tools.

However, sometimes severe restrictions are made on program execution time or resource usage, and to satisfy these demands, code must be optimised. Because of the strict code structure imposed by high-level programming languages, it is

[⋆] 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.

often better to fine-tune programs at the lower level of executable code. But if one does this, one still needs to understand why and how the code works. Here a specification language can be useful as well, because it can reintroduce the abstraction that was eliminated by the compilation and optimisation process. Thus, a good specification language for executable code can provide a basis for the development of reliable, highly optimised programs in low-level form.

Moreover, since low-level languages can be the target platform for several different source code languages, a specification formalism for a low-level language can serve as a common ground for understanding software from different sources.

This led to the proposal of a program logic for Java bytecode [5] and, based on this, a specification language for bytecode — the Bytecode Modeling Language (BML) [8]. BML is based on the principle of design-by-contract and it is strongly inspired by the Java Modeling Language (JML) [14,17,18]. JML is the *de facto* Java specification language, supported by a wide range of tools [7].

One of the most promising applications of low-level specification languages such as BML is in the context of proof-carrying code (PCC). In this context, code that is shipped from the code producer to the code consumer comes together with a specification and a correctness proof. Since BML can specify executable code, it seems an appropriate specification language for foundational PCC [2,1], where a relatively small but expressive framework can capture the class of desirable properties of mobile code. Because of its expressiveness, BML specifications can give hints to the prover (e.g., one can supply loop invariants and suggest appropriate lemmas using assert statements), which can ease the automatic construction of proofs. To be able to ship BML specifications together with the code, a BML representation within Java class files is defined.

To be able to use BML in a PCC context, *and* as a specification language on its own, it is designed with the following two goals in mind: (*i*) it should be easy to transform specifications and proofs from the source code level to the bytecode level, and (*ii*) specifications should be comprehensive.

When BML is used in a PCC context, we expect it be used as an intermediate format. People will rather specify and verify their source code, and then translate these into properties and proofs of the executable code. Since Java is our privileged application language, we assume JML will be the source code specification language. Therefore, translation from JML specifications and proofs to BML should be as straightforward as possible. Realising a PCC platform for Java to support this use of BML is one of the goals of the MOBIUS project[3].

Since BML can also be used as a specification language on its own (for example, to ensure that a program optimisation is correct), the specifications should be intelligible. To achieve this, the language reuses many constructs from JML. Since JML is designed in such a way that it is intuitive and easily understandable for common Java programmers, we believe the same should apply to BML. Therefore, we developed a textual representation of bytecode classes augmented with BML that indicate clearly the relation between the specification and the different pieces of the program.

[3] See `http://mobius.inria.fr` for more information.

```
public class KeyPool {

    private int[] keyIds;
    //@ invariant keyIds != null;
    /*@ invariant (\forall int i,j; 0 <= i && i < j && j < keyIds.length;
      @                              keyIds[i] >= keyIds[j]); @*/

    //@ ghost int lastPos;
    //@ invariant 0 <= lastPos && lastPos < keyIds.length;
    /*@ invariant (\forall int i; lastPos < i && i < keyIds.length;
      @                            keyIds[i] == 0); @*/

    // ...other methods...

    /*@ requires keyId > 0 && lastPos < keyIds.length − 1;
      @ ensures (\exists int i; 0 <= i && i < lastPos && keyIds[i] = keyId);
      @*/
    public void insert(int keyId) {
        int i;
        /*@ loop_invariant −1 <= i &&
          @     (\forall int k; i < k && k < keyIds.length; keyId > keyIds[k]);
          @*/
        for (i = keyIds.length − 2; i >= 0 && keyId > keyIds[i]; i−−) {
                keyIds[i+1] = keyIds[i];
        }
        keyIds[i+1] = keyId;
        //@ set lastPos = lastPos + 1;
    }
}
```

Figure 1. Source code and JML specifications for class `KeyPool`

A crucial element for the success of a specification formalism is tool support. Therefore, a set of prototype tools is developed for BML. This tool set contains the following tools:

– BMLLib, a library to represent and manipulate specifications;
– Umbra, a BML editor within Eclipse IDE;
– JML2BML, a compiler from JML specifications to BML;
– BML2BPL, a translator of bytecode enhanced with BML to BoogiePL, a language from which verification conditions can be generated easily; and
– CCT, a tool to store proofs in class files.

A precise description of the BML language is given in the *BML Reference Manual* [10]. The current paper gives a brief overview of BML (Sect. 2) and its two representations (Sect. 3). Then it discusses the tools in the BML tool set (Sect. 4). We conclude the paper in Sect. 5.

Throughout the paper, fragments of a class `KeyPool` are used as example. Figure 1 shows relevant parts of the Java source code and JML specifications of this class. We expect the reader to be able to grasp the intention of this specification.

2 Overview of BML

As motivated above, the design of BML is very similar to its source-code-level counterpart JML: each element of a class file can be annotated with specifications. This section illustrates this by showing how the specifications in Fig. 1 are translated. Figure 2 shows the translation of the Java code, without the specifications. The full definition of BML can be found in [10].

It is important to note that BML covers most of the so-called JML Level 0, i.e., the essential part of JML that is supposed to be supported by *all* JML tools [18, Sect. 2.9]. The missing features are informal descriptions and extended debug statements. Informal descriptions are in fact a special kind of comments, which are impossible to formalise. The JML debug statements can contain arbitrary Java expressions, while BML debug statements allow only variable names, the value of which is supposed to be printed out by tools that execute BML specifications (e.g., a run-time checker). In addition, BML allows one to use pure method calls in specifications (mandated by JML Level 1 — describing features to be supported by *most* tools). Also, the expression language contains a few BML-specific constructs, to denote the size and elements of the operand stack and the size of arrays. Constants and variables are also addressed differently in BML: in the binary representation fields are encoded as an index in the constant pool (to the location where the `FieldRef` structure is stored), while local variables are referenced by a number that denotes their position in the local variable table. This is the same as fields and local variables are addressed in bytecode. For the sake of readability, in the textual representation, those numerical references are shown as appropriate identifiers.

2.1 Class-level Specifications

Class-level specifications specify behaviour of all instances of a class. The most prominent example of class-level specifications are invariants. An (instance) invariant specifies a property that should hold for all instances of that class, after completion of the constructor and before and after the execution of all methods of the class. Figure 3 shows the BML specification of the invariants and other class-level specification constructs for the class `KeyPool`. Notice that compared to the JML specification, the specifications are more rigid in format: the constructs are given in a fixed order and the receiver object is always mentioned explicitly. In addition, a keyword **\length** is used to denote the length of an array.

Another class-level specification is the declaration of a so-called *ghost* field. These are fields that exist only at specification level. To change their value, BML has a special **set** instruction (see also Sect. 2.3). Ghost fields can be used

```
package [default]
// ... Constant pool and Second constant pool omitted ...
public class KeyPool extends java.lang.Object
// ... class-level specifications omitted ...

// ... other methods omitted ...

// ... method specification omitted ...
public void insert(int)
0:     aload_0
1:     getfield          KeyPool.keyIds [I (20)
4:     arraylength
5:     iconst_2
6:     isub
7:     istore_2
//@ loop_specification   ...  specification  omitted  ...
8:     goto              #28
11:    aload_0
12:    getfield          KeyPool.keyIds [I (20)
15:    iload_2
16:    iconst_1
17:    iadd
18:    aload_0
19:    getfield          KeyPool.keyIds [I (20)
22:    iload_2
23:    iaload
24:    iastore
25:    iinc              %2      −1
28:    iload_2
29:    iflt              #42
32:    iload_1
33:    aload_0
34:    getfield          KeyPool.keyIds [I (20)
37:    iload_2
38:    iaload
39:    if_icmpgt         #11
42:    aload_0
43:    getfield          KeyPool.keyIds [I (20)
46:    iload_2
47:    iconst_1
48:    iadd
49:    iload_1
50:    iastore
//@ set ...  specification  omitted ...
51:    return
```

Figure 2. Bytecode for class `KeyPool`

```
/*@ public ghost int lastPos @*/
/*@ invariant keyIds != null @*/
/*@ invariant 0 <= this.lastPos && this.lastPos < \length(this.keyIds) @*/
/*@ invariant \forall int i,j ;  0 <= i && i < j && j < \length(this.keyIds)
  @               ==> this.keyIds[i]  >= this.keyIds[j]
  @*/
/*@ invariant \forall int i; this.lastPos  < i && i < \length(this.keyIds)
  @               ==> this.keyIds[i]  == 0)
  @*/
```

Figure 3. BML class-level specifications for class `KeyPool`

to represent values that are implicit in the actual code, but must be mentioned explicitly in specifications.

In our example, the `lastPos` ghost field represents the position of the last key inserted in the table. The invariants constrain the possible values of the field `keyIds` and the ghost field `lastPos`: `keyIds` cannot be `null`, `lastPos` should be less than the length of the array, the values in the array should be sorted in decreasing order, and all entries of the array to the right of `lastPos` should be 0. The value of `lastPos` is updated by a **set** instruction, placed before instruction label 51 in the bytecode (see Fig. 5).

Apart from invariants and ghost variable declarations, BML class-level specifications can also be static invariants, i.e., invariant properties over static fields; history constraints, that express a relation between two states before and after method calls; and model field declarations to abstract complex expressions (e.g. the sum of all elements in a table).

2.2 Method-level Specifications

Method-level specifications describe the behaviour of a single method. The basic principle is the use of pre- and postconditions. Preconditions state what is expected about parameters and the state of objects upon method invocation, while postconditions state what the method guarantees upon termination. It is possible to refer to the prestate of the method in the postcondition, using the keyword **old**. In addition, BML method-level specifications contain assignable, signals and signals-only clauses. These specify which variables may be modified by a method, which exceptions may be thrown by a method, and under which conditions. Assignable clauses are necessary for sound modular verification. In JML specifications, these clauses are often left implicit, using an appropriate default clause, but in BML they have to be specified explicitly. In addition, BML allows one to flag a method as *pure*, meaning that it does not modify state, and therefore can be used in specifications.

Figure 4 shows the BML specification for method `insert`. The precondition (keyword **requires**) specifies that parameter `keyId` should be strictly positive and ghost variable `lastPos` should be less than the length of the table minus 1, i.e., there should be space for inserting another key. The postcondition (keyword

```
/*@ requires keyId > 0 && this.lastPos < \length(this.keyIds) − 1
  @ modifies \everything
  @ ensures (\exists int i; 0 <= i && i < lastPos && this.keyIds[i] == keyId)
  @ signals (java/lang/Exception) true
  @ signals_only \nothing
  @*/
```

Figure 4. BML method specification for `insert` in class `KeyPool`

ensures) specifies that after completion of the method, one of the elements of the `keyIds` table is the newly inserted `keyId`. Notice that the implicit assignable, signals and signals-only clauses from Fig. 1 are explicit in the BML specification.

2.3 Code-level Specifications

The last group of BML specifications are those that refer to specific points of the code inside a method body. Such specifications are typically there to help automatic verification procedures. A common code-level specification construct is a loop invariant, specifying a condition that is met every time control is at the beginning of the loop. Loop invariants are necessary to prove partial correctness of a loop.

Figure 5 shows the BML specification of the loop invariant of method `insert` in class `KeyPool`. The loop ranges from label 8 to 39. All instructions before label 8 initialise the loop; the first eight instructions of the loop (labels 8–19) check the loop condition; and the loop body is implemented by the instructions labelled 22–35. The invariant is specified just before the beginning of the loop, i.e., before instruction 8. It states that the loop variable `i` never is less than -1, and all keys that have been examined, i.e., between `i` and the length of `keyIds`, are less than `keyId`.

Apart from the loop invariant, a BML loop specification also contains a loop variant, i.e., a non-negative integer expression that is supposed to strictly

```
7:     istore_2
/*@ loop_specification
  @ loop_inv −1 <= i && (\forall int k; i < k && k < \length(this.keyIds)
  @                ==> keyId > this.keyIds[k])
  @ decreases 1
  @*/
8:     goto     #28
// ... code omitted ...
50:    iastore
/*@ set this.lastPos = this.lastPos + 1 @*/
51:    return
```

Figure 5. BML code-level specifications for class `KeyPool`

decrease for each iteration of the loop. The variant is used to prove termination of the loop. In our example it is a meaningless 1, since no variant is given at the source code level. Thus, with this specification, it will not be possible to prove termination of this method.

Other BML code-level specifications include **set** instructions, used to change the value of ghost variables (before label 51 in Fig. 5); **assert** and **assume** annotations, to assert/assume facts about the program; and **debug** annotations, to print out values of a variable in case the program is executed by a BML-aware execution environment.

2.4 Verification of BML Specifications

Work on BML and its semantics was initiated by Mariela Pavlova [25] within the context of JACK (Java Applet Correctness Kit) [4]. Pavlova's work is based on an operational semantics of Java bytecode which covers a representative set of 22 instructions. She gives a semantics for a representative subset of the specification language in the form of a weakest precondition calculus. An overview of the work is presented in [9,8].

Development of the formal underpinning of BML continued in the context of the MOBIUS project (see [22] for more details). The Bicolano specification [26], within the proof assistant Coq, formalises the operational semantics of a considerable subset of bytecode instructions. On top of Bicolano a bytecode logic, called the MOBIUS base logic, is developed and formalised in Coq, following the principles of [5]. A translation from BML specifications into the MOBIUS base logic is defined (where BML predicates are translated using an additional deep embedding layer for assertions in Isabelle).

To make verification of BML specifications more practical, a translation into BoogiePL is necessary. BoogiePL is an intermediate language for program verification [12]. It has procedures and only 5 instructions (including assume and assert). This makes it easy to define a correct verification condition generator for it. The strength of BoogiePL lies in its non-determinism and its use of guards to control the program flow (similar to Dijkstra's guarded commands [13]). A program and its specification are translated into a BoogiePL program; verification conditions are generated from this BoogiePL program.

Lehner and Müller present a translation from bytecode instructions to BoogiePL [19]. Properties are specified in first-order logic. Using a translation that is similar to the one presented by Darvas and Müller for JML0 [11], BML specifications can be translated into first-order logic. In addition, Mallo [21] presents a direct translation for a subset of BML into BoogiePL.

3 Representation of BML Specifications

The bytecode presented in Fig. 2 is of course only a textual representation of the actual binary code. Various tools exist to produce such textual representations from class files, e.g., `javap` and Umbra (see Sect. 4.3). BML also has two representations: (i) a binary form, using non-standard attributes stored inside class files, and (ii) a textual representation, as shown in the previous section. This textual representation is very similar to JML, but a bit more rigid, as it must be in correspondence with the binary form. This section discusses the two representations of BML.

3.1 Binary Representation of BML

Several possibilities exist to define a binary format for BML.

A simple approach would be to use standard Java serialisation to dump Java objects representing abstract syntax trees of specifications. However, this choice would force BML tool builders to use Java and our abstract syntax tree definition. Instead, we preferred to define a precise specification of a binary format, that tool builders can freely manipulate.

As said above, BML specifications are stored inside a class file. It would also be possible to store specifications in a separate file, but since specifications refer to elements of the class file, it is most natural to take advantage of the possibility to add information to the class file.

To store additional information in class files, there are currently two different possibilities: attributes and annotations [15]. Attributes are more low-level, they appear in the specification of the Java Virtual Machine since its first version and they are used by the compiler to store the different optional elements of classes, such as the method code, line number tables and local variable tables. Attributes are also used in the initial tools that support BML, in the JACK environment [4].

Java annotations are introduced in Java 5.0 as a mechanism to describe properties of Java methods and classes (metadata). They have become the official standard of annotating Java code with machine checkable information. Support to compile and store them in special attributes inside a class file is available, as well as an API to inspect them at runtime. However, the main benefit of using annotations is at source code level. Inserting annotations in an already compiled class file is a bit contrary to the idea of annotations. Besides, Java annotations cannot be placed inside code[4], specifications would be necessary at least for code-level specifications. In addition, Java annotations are not used in JML, and adopting annotations would practically preclude specifying code written in earlier versions of Java than 5.0.

Taking all these considerations into account, we decided to define a precise binary format for BML specifications, stored as non-standard JVM attributes inside the class file.

Encoding of BML Specifications. The structure of Java class files is quite flexible. Some elements are obligatory, e.g., the magic number CAFEBABE[5], header, constant pool, field table and method table, but most elements are optional.

[4] Even the ongoing development in JSR308 [16] to allow one to use annotations in more places still does not support annotation of instructions.

[5] See `http://www.artima.com/insidejvm/whyCAFEBABE.html` for more information.

```
Invariants_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 invariants_count;
  { u2 access_flags;
    formula_info invariant;
  } invariants[invariants_count];
}
```

Figure 6. Structure of the `org.bmlspecs.Invariants` attribute

All optional elements are stored in so-called *attributes*, which are just blocks of bytes with a name (to distinguish them). Attributes can be stored in different "places" in the class file structure: there are class attributes, field attributes, method attributes, and code attributes. So, for example, the bytecode of a method is stored in a method attribute named `Code`, and inside this `Code` attribute there is also space for code-level attributes, such as `LineNumberTable` and `LocalVariableTable`. Abstract methods simply do not have a `Code` attribute.

BML specifications are stored in appropriately placed JVM attributes: class-level specifications are stored in class attributes, method specifications in method attributes and code-level specifications in code attributes. The names of all BML-related attributes start with a common prefix `org.bmlspecs`. An important class-level attribute is the *second constant pool*. This structure is similar to the standard constant pool, but it is used to stored all constants that are part of the specification only.

All class invariants are stored together in a single class-level attribute named `org.bmlspecs.Invariants`, whose structure is given in Fig. 6. To specify the format of these attributes, we use a C-like structure notation, cf. [20], where each entry is preceded by a special identifier, e.g., `u1`, `u2` and `u4`, that describes the type of the corresponding value. The first two fields of the `org.bmlspecs.Invariants` attribute are `attribute_name_index` and `attribute_length`. These are obligatory for all attributes. They contain an index in the constant pool, where the attribute name (here: `org.bmlspecs.Invariants`) is stored, and the length in bytes of the whole attribute. The next two fields, `invariants_count` and `invariants`, describe the invariants table: the number of invariants and the table containing the invariants themselves. Each entry of the table contains information about an invariant, namely its access flags (`public`, `protected`, `private`, `static`) and its formula.

Formulae and expressions are stored as the prefix traversal sequence of the abstract syntax tree of a given expression, with binary representation of operands. The names of variables and fields are represented as indexes of appropriate string constants in the constant pool (either the original constant pool or the second specification-only constant pool).

As another example, Fig. 7 specifies the binary format for the code-level attribute describing loop specifications. The first two mandatory fields are as be-

```
LoopSpecificationTable_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 loops_count;
  { u2 point_pc;
    u2 order;
    formula_info invariant;
    formula_info variant;
  } loops[loops_count];
}
```

Figure 7. Structure of the `org.bmlspecs.LoopSpecificationTable` attribute.

fore; the attribute name is now `org.bmlspecs.LoopSpecificationTable`. The last two fields describe the length and contents of the loop specification table. Each loop specification in the table is represented by the following elements: `point_pc`, the label of the instruction to which the specification is attached; the `order` entry that specifies the respective order in which code-level specifications should be considered if they are attached to the same instruction; and the `invariant` and the `variant` formulae. The order field is necessary if for example several **set** annotations are related to the same bytecode instruction: it ensures that the assignments are properly ordered.

Other BML specification constructs are encoded in a similar way. More details can be found in the BML Reference Manual [10].

Representation of Certificates. To support proof-carrying code, besides the attributes that contain specifications, one also needs attributes that can store a proof that an implementation respects its specification. A flexible, generic format in which different kinds of PCC certificates can be encoded is proposed in [23, Sect. 2.5]. In this proposal, the certificates can be divided into two groups: class-level certificates and method-level certificates. The format of these certificates is presented in Fig. 8(a) and Fig. 8(b), respectively. This format allows one to store certificates concerning various properties of bytecode, produced by tools supporting different technologies (e.g., fixpoints for abstract interpretation or type derivations). Note that the actual certification technology may choose not to use both of the certificate levels and to store the complete certificate information in the class-level attribute only or in the method-level attributes only.

To use BML specifications in a PCC context, this generic certificate scheme is instantiated to certificates that encode Coq proofs of the properties expressed in BML. In this case, the type checking engine of Coq, combined with a tool to generate a Coq representation of the program and its BML specifications, is the final certificate checker.

At the client side, the class file, BML specifications and the proofs that are encoded in the certificates are expanded to Coq modules. At class-level these modules include:

```
PCCClassCert {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 cert_type;
  u1 major_version;
  u1 minor_version;
  u2 imported_certs_count;
  u2 imported_certs[imported_certs_count];
  u4 proofs_section_length;
  u1 proofs_section[proofs_section_length];
}
```

(a)

```
PCCMethodCert {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 cert_type;
  u1 cert_major_version;
  u1 cert_minor_version;
  u4 proofs_section_length;
  u1 proofs_section[proofs_section_length];
}
```

(b)

Figure 8. Format of PCC certificates in class files

– a Coq representation of the class structure (i.e., fields, methods, code of methods etc.) and the BML specifications;
– a representation of properties for each method that express that whenever the method is called in a state in which the method's precondition is satisfied then the method's postcondition holds after the method returns; and
– proofs of the properties above.

The method definitions are generated based on the class file method structures. The method properties combine the BML pre- and postconditions with invariants. The certificates contain the necessary proofs.

In order to conceptually separate proofs of a class's interface properties (like invariants, method specifications, etc.) from the proofs of implementation details (like loop specifications, asserts etc.), the latter are included in separate Coq modules that are constructed from method-level specifications and certificates. These method-level Coq modules contain the following:

– a theorem that states that if the method is called in a state in which the precondition holds then the postcondition holds after a return from the method;
– for each assert, a theorem that states that if the method is called in a state in which the precondition holds then the assert holds in a related program position;
– for each strong invariant (i.e., an invariant that must be maintained by all program steps) and method a theorem that states that if the method is called in a state in which the precondition holds then the invariant holds at each instruction of the method;
– proofs of the theorems above.

3.2 Textual Representation of BML

Since we expect that programmers will read and edit specifications at bytecode level (for example, if one wishes to develop correct code that is more optimal than compilers can generate), we also defined a textual representation of bytecode files augmented with specifications. This ensures that programmers have a file format that is easy to read, exchange, and edit by common textual editors.

There is no standard for textual bytecode representation, but some popular tools (e.g., Sun's javap, Apache's BCEL[6], and ObjectWeb's ASM[7]) print out class files in a textual form to facilitate debugging and understanding of the code. However, these tools do not support parsing of any textual representation of class files (and thus also no editing).

There are also tools such as Javaa[8] and, more popular, Jasmin[9] that allow one to write classes and methods using Java bytecode mnemonics. However, they are not tuned to program specification and verification, and they require the user to supply much information that is not relevant for specifications. In addition, their source code is only available under a non-standard license, which makes it difficult to integrate them in an open source project such as MOBIUS. Therefore, we decided to develop our own bytecode viewer and editor, described in the next section, that adheres to the textual representation standard of BML.

We assume that programmers work at the same time with class files and textual files. Therefore, we decided not to display certain values such as the bytecode file version number or the contents of foreign attributes. In this way, only the relevant information is presented to the user. Roughly, the format displays in sequence: the package name, the class header with the class name and information about its location in the type hierarchy, the constant pools – including the second constant pool, fields, class-level specifications (such as invariants and constraints) and methods augmented with their method-level specifications and code-level specifications. All other information (e.g., the Line Number Table) is stored in the class file and not shown to the user.

4 Overview of the BML tools

Just as for a programming language, if a specification language is to be used successfully, it needs good tool support to read, write, and manipulate specifications. Moreover, one needs tools to check that a program respects the properties stated in the specifications.

This section provides an overview of the tools that are developed to support BML. We start with a brief description of JACK, the historical predecessor of the currently existing tools. Then the tools which support the current version of BML are presented: (*i*) BMLLib, a library to represent and manipulate specifications; (*ii*) Umbra, an interactive BML editor integrated into Eclipse IDE; (*iii*) JML2BML, a compiler from JML specifications to BML; (*iv*) BML2BPL, a translator from BML and bytecode to BoogiePL; and (*v*) CCT, a tool to package proofs in class files. Figure 9 shows how the different tools connect with each other.

[6] Byte Code Engineering Library, available from `http://jakarta.apache.org/bcel/`.
[7] Available from `http://asm.objectweb.org/`.
[8] Available from `http://tinf2.vub.ac.be/~dvermeir/courses/compilers/javaa/jasm.html`.
[9] Available from `http://jasmin.sourceforge.net/`.

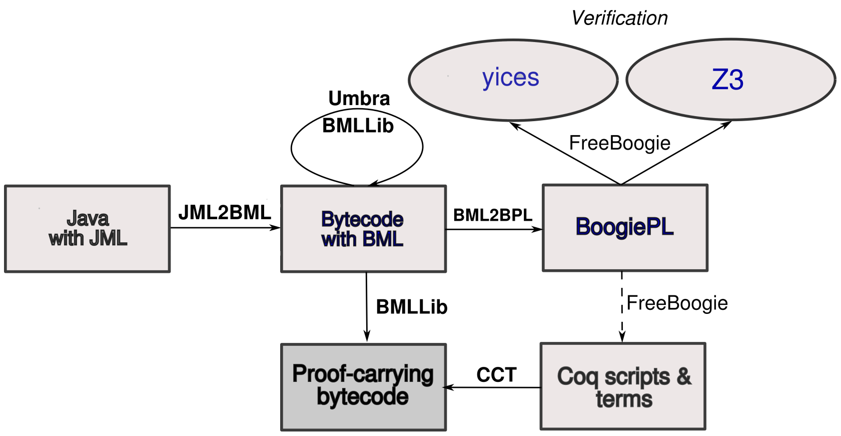


Figure 9. BML tool set.
The tools that are developed especially for BML are written in bold. A dashed line means that the tool is still under development

4.1 JACK

Overview and goals. JACK is a tool that integrates the verification machinery developed for JML with a programming environment, namely Eclipse. Programmers can manipulate Java source code and JML specifications, while the (textual) representation of bytecode specifications is hidden and can be viewed only on demand by expanding the structure of bytecode attributes [9].

Design of the tool. The JACK tool is an Eclipse plugin. It takes JML annotated source code and generates proof obligations expressed in an internal Java/JML Proof Obligation Language. The proof obligations are generated by means of a weakest precondition generator. Then one of the available provers (AtelierB, Simplify, Coq, PVS) can be used to discharge the generated proof obligations. In case the proof obligation cannot be discharged automatically, it can be viewed in the IDE and proved interactively.

Availability. The final release of JACK, both in binary and source form, is available from `http://www-sop.inria.fr/everest/soft/Jack/jack.html`.

4.2 BMLLib: a Library to Manipulate BML Specifications

Overview and goals. The most basic tool support that is needed for BML is parsing and pretty-printing of its textual representation as well as reading and storing of specifications in class files. This functionality is provided by the BMLLib library. In addition, this library provides a Java API to generate and manipulate BML specifications. Most of the tools discussed below depend on BMLLib.

Design of the tool. BMLLib is developed at the University of Warsaw. It uses the BCEL library as the basic library to manipulate class files. BCEL is known to be difficult and non-intuitive in use, but it has the advantage that it is maintained by the Jakarta project[10], which gives confidence in its future existence. BMLLib allows one to store and read BML specifications represented in class files. It defines an abstract syntax tree to represent BML specifications. The classes and methods augmented with specifications are implemented as delegate classes which can either return the specifications or the BCEL representation of the class or method, respectively. The parser of the specifications is written with the ANTLR parser generator[11], a highly reliable parser generator for Java. A detailed description of the library is presented in [27].

Additionally, BMLLib provides a translation from the BCEL representation into ASM representation used in BML2BPL (discussed below). This translation is necessary to enable the translation into BoogiePL and subsequent generation of proof obligations with FreeBoogie.

Availability. The alpha version of the library is available from `http://www.mimuw.edu.pl/~alx/umbra/`. It is written in Java and tested primarily under Linux and Windows.

4.3 Umbra: a BML Editor

Overview and goals. Most existing class file editors are developed as a series of windows that correspond to the layout of the attributes and other structures of the class file. This design leads to an environment which is not easy to navigate for a programmer. Instead, we developed Umbra as an Eclipse plugin that allows one to view, add, delete and edit BML specifications and bytecode in a textual representation. Moreover, if available, the textual representation is associated with the Java source code. This makes it possible to relate fragments of the source code with fragments of the byte code and the other way round [27]; in particular it allows one to see field and variable names, instead of indexes in the constant pool and local variable table.

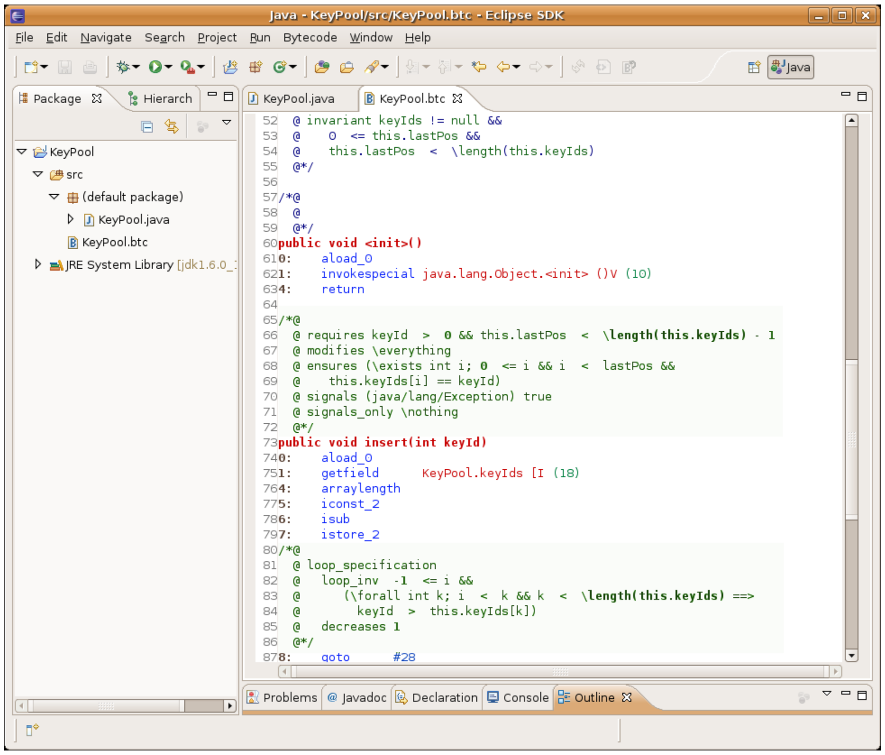
Furthermore, Umbra gives programmers the possibility to change not only the specifications, but also the bytecode instructions.

The Umbra plugin also provides a user interface for several of the tools presented below; in particular it has buttons to run the JML2BML compiler, the BML2BPL translator, and the FreeBoogie verification back-end.

Design of the tool. Umbra is developed as an Eclipse plugin that extends the Java editor plugin and adds its own functionality for editing class files. Umbra relies on the representation of class files provided by the BCEL library. The internal representation of BML specifications is provided by BMLLib. Fig. 10 shows the code from Fig. 2 being edited in Umbra.

[10] Available from `http://jakarta.apache.org/`

[11] Available from `http://www.antlr.org/`


Java - KeyPool/src/KeyPool.btc - Eclipse SDK
File Edit Navigate Search Project Run Bytecode Window Help
Package
Hierarch
KeyPool
src
(default package)
KeyPool.java
KeyPool.btc
JRE System Library [jdk1.6.0_
KeyPool.java
KeyPool.btc
Problems
Javadoc
Declaration
Console
Outline

```
 @ invariant keyIds != null &&
 @    0  <= this.lastPos &&
 @    this.lastPos  <  \length(this.keyIds)
 @*/

/*@
 @
 @*/
public void <init>()
0:    aload_0
1:    invokespecial java.lang.Object.<init> ()V (10)
4:    return

/*@
 @ requires keyId  >  0 && this.lastPos  <  \length(this.keyIds) - 1
 @ modifies \everything
 @ ensures (\exists int i; 0  <= i && i  <  lastPos &&
 @    this.keyIds[i] == keyId)
 @ signals (java/lang/Exception) true
 @ signals_only \nothing
 @*/
public void insert(int keyId)
0:    aload_0
1:    getfield      KeyPool.keyIds [I (18)
4:    arraylength
5:    iconst_2
6:    isub
7:    istore_2
/*@
 @ loop_specification
 @   loop_inv  -1  <= i &&
 @      (\forall int k; i  <  k && k  <  \length(this.keyIds) ==>
 @        keyId  >  this.keyIds[k])
 @   decreases 1
 @*/
8:    goto      #28
```

Figure 10. Bytecode for class `KeyPool` edited in Umbra

Availability. The alpha version of the editor is available from `http://www.mimuw.edu.pl/~alx/umbra/`. It is written in Java and tested primarily under Linux and Windows.

4.4 JML2BML: a Specification Compiler from JML to BML

Overview and goals. The JACK tool contains a compiler of JML annotations to BML. However, this compiler is highly integrated with the tool itself. Therefore, the need for a standalone JML to BML compiler arose.

The JML2BML compiler takes as input a Java source file with JML annotations, together with the corresponding class file and outputs the class file with proper BML annotations inserted. This allows the user to write the specifications at the more comprehensive source code level and then translate them into the bytecode level representation. At bytecode level these specifications can then be combined with specifications written by hand or with specifications coming from other tools. Note that JML2BML does not erase any specifications that are present in the class file, it only adds the specifications translated from the JML specifications.

Currently, the JML2BML compiler focuses on supporting JML Level 0, roughly corresponding to the subset of JML covered by the BML language.

Design of the tool. The compiler uses an enhanced Abstract Syntax Tree (AST) for the Java source code, taken from the OpenJML[12] compiler (a Java compiler with JML checker based upon OpenJDK). The result is stored in the class file, using the BMLLib library [27]. The compilation is described by a set of transformation rules that are one by one applied to the JML AST. This approach makes the compiler easily extensible. It is enough to just write a new translation rule to support additional features of the JML language. The JML2BML compiler is intergrated in the Umbra editor as a push-button, but it can also be used as a standalone tool.

Availability. The compiler is available from `http://www.mimuw.edu.pl/~alx/jml2bml/`. It is written in Java and tested primarily under Linux and Windows.

4.5 BML2BPL: a Translation from BML specifications to BoogiePL

Overview and goals. BoogiePL is an intermediate language designed to alleviate part of the burden of the transformation from the specified source code to proof obligations. The Boogie verifier (which is originally developed to reason about Spec# programs) has the ability to transform BoogiePL code into formulae for various proving back-ends including Simplify, Z3, and HOL/Isabelle [3]. There is also an open source alternative for the environment called FreeBoogie[13].

Lehner and Müller [19] presented a translation from bytecode to BoogiePL. On top of this, one of their students at ETH Zürich, Mallo, developed a tool that transforms BML-annotated bytecode into BoogiePL. This translation is only defined for a subset of the BML language as defined in the BML Reference Manual [10].

Design of the tool. The tool allows one to read class files with BML specifications, and outputs a BoogiePL encoding of the annotated classes. However, BML2BPL uses a non-standard way of representing the BML attributes in classes and it is based on the ASM bytecode library which is different from the one used in other BML-related tools. Therefore a suitable translation is implemented in BMLLib, that provides an interface between the standard representation and BML2BPL (see also Sect. 4.2).

Availability. The translator is available from `https://mobius.ucd.ie/trac/browser/src/BML_BPL_Translator`. It is written in Java and tested primarily under Linux and Windows.

[12] Available from `http://sourceforge.net/projects/jmlspecs`
[13] Available from `http://secure.ucd.ie/products/opensource/FreeBoogie/`.

4.6 CCT: a Tool for Packaging Certificates

Overview and goals. The Class Certificate Transformer (CCT) is a modular tool which is able to create and extract certificates from class files [29]. These certificates can for example be typing derivations, information inferred by abstract analysis, or proofs of BML specifications. In addition, CCT allows one to manipulate certificates by adding or removing data. Finally, it also allows one to add plugins which understand the internal structure of certificates and can generate the code which performs the actual verification. For example, one can add a module which retrieves typing information from a certificate and then runs a particular type checker on the program.

Design of the tool. CCT is built in a highly modular way. One can easily construct plugins for the tool to define the actual PCC certificate verification process. It also allows one to add different libraries that support manipulation of the class file structure so that one is not restricted to using BCEL or ASM for the verification tool.

Availability. The translator is available (in source code format) from `https://mobius.ucd.ie/trac/browser/src/CCT`. It is written in Java and tested primarily under Linux and Windows.

5 Conclusions and further work

This paper motivates the development of the specification language BML and its supporting tool set. BML is developed with the proof carrying code paradigm in mind. This motivates part of the design choices: in particular BML is designed to be closely related with the source code level specification language JML, and a binary representation to store BML in class files is defined. However, BML is also intended to be used as a specification language on its own, for example to reason directly about the correctness of low-level program optimisations. Therefore, BML specifications are also designed to be readable and understandable.

An important merit of BML is that it is largely supported by a tool set. The different tools are described in this paper. Currently, the main efforts are focused on filling in the remaining gaps to develop a complete platform for PCC. In particular, we concentrate on the following topics:

- extending the existing verification link through BoogiePL, since it is only defined for a subset of the BML language;
- using the planned extension of FreeBoogie to generate proof obligations in Coq; and
- development of the direct generation of proof obligations for Coq, using the methods of the verification condition generator described in [22, Sect. 5.1].

At the moment, the BML tool set has been tested on small examples only. In the near future, we plan to work on a more realistic case study, that demonstrates the usability of the tools for a non-trivial MIDP application. This case study should demonstrate that the PCC infrastructure works in the environment of the Java Virtual Machine.

In addition to these main goals, we work on a translation from an information-flow type system to BML, based upon the translation described in [28]. This should enable the BML-based verification system to incorporate a mechanism to ensure non-interference.

References

1. A. W. Appel. Foundational proof-carrying code. In J. Halpern, editor, *Logic in Computer Science*, page 247. IEEE Press, June 2001. Invited Talk.
2. A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Principles of Programming Languages.* ACM Press, 2000.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science.* Springer-Verlag, 2005.
4. G. Barthe, 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 *Formal Methods for Components and Objects: Revised Lectures from the 5th International Symposium FMCO 2006*, number 4709 in Lecture Notes in Computer Science, pages 152–174. Springer-Verlag, 2007.
5. L. Beringer and M. Hofmann. A bytecode logic for JML and types. In *Asian Programming Languages and Systems Symposium*, Lecture Notes in Computer Science 4279, pages 389–405. Springer-Verlag, 2006.
6. D. Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*, number 61 in LNCS, London, UK, 1978. Springer-Verlag.
7. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In *Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, 2003.
8. L. Burdy, M. Huisman, and M. Pavlova. Preliminary design of BML: A behavioral interface specification language for Java bytecode. In *Fundamental Approaches to Software Engineering*, volume 4422 of *LNCS*, pages 215–229. Springer-Verlag, 2007.
9. L. Burdy and M. Pavlova. Java bytecode specification and verification. In *Symposium on Applied Computing*, pages 1835–1839. ACM Press, 2006.
10. J. Chrząszcz, M. Huisman, A. Schubert, J. Kiniry, M. Pavlova, and E. Poll. *BML Reference Manual*, December 2008. In Progress. INRIA and University of Warsaw. Available from `http://bml.mimuw.edu.pl`.
11. Á. Darvas and P. Müller. Formal encoding of JML level 0 specifications in JIVE. Technical report, ETH Zurich, 2007. Annual Report of the Chair of Software Engineering.
12. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

13. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
14. B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, 2001.
15. JSR 175 Expert Group. A metadata facility for the Java programming language. Java Specification Request 175, Java Community Process, September 2004. Final release.
16. JSR 308 Expert Group. Annotations on Java types. Java Specification Request 308, Java Community Process, 2007. In progress.
17. G. T. Leavens, A. L. 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. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, P. Chalin, and D. Zimmerman. *JML Reference Manual*, February 2008. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.
19. H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. In M. Huisman and F. Spoto, editors, *Bytecode Semantics, Verification, Analysis and Transformation*, Electronic Notes in Theoretical Computer Science, 2007.
20. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.
21. O. J. Mallo. A translator from BML annotated Java bytecode to BoogiePL. Master's thesis, Software Component Technology Group, ETH Zürich, 2007.
22. MOBIUS Consortium. Deliverable 3.1: Bytecode specification language and program logic, 2006. Available online from `http://mobius.inria.fr`.
23. MOBIUS Consortium. Deliverable 4.2: Certificates, 2007. Available online from `http://mobius.inria.fr`.
24. Object Management Group. *Object Constraint Language. OMG Available Specification, Version 2.0*, May 2006.
25. M. Pavlova. *Java bytecode verification and its applications.* Thése de doctorat, spécialité informatique, Université Nice Sophia Antipolis, France, January 2007.
26. D. Pichardie. Bicolano – Byte Code Language in Coq. `http://mobius.inria.fr/bicolano`. Summary appears in [22], 2006.
27. A. Schubert, J. Chrząszcz, T. Batkiewicz, J. Paszek, and W. Wąs. Technical aspects of class specification in the byte code of Java language. In *Bytecode'08*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008. To appear.
28. A. Schubert and D. Walukiewicz-Chrząszcz. The non-interference protection in a bytecode program logic, 2009. Submitted.
29. T. Sznuk. Introduction of the proof-carrying code technique to Java class. Master's thesis, Institute of Informatics, The University of Warsaw, 2008. in Polish.