# Preliminary Design of BML: A Behavioral Interface Specification Language for Java bytecode*

Lilian Burdy, Marieke Huisman[1], and Mariela Pavlova[1]

INRIA Sophia Antipolis, France

**Abstract.** We present the Bytecode Modeling Language (BML), the Java bytecode cousin of JML. BML allows the application developer to specify the behaviour of an application in the form of annotations, *directly* at the level of the bytecode. An extension of the class file format is defined to store the specification directly with the bytecode. This is a first step towards the development of a platform for Proof Carrying Code, where applications come together with their specification and a proof of correctness. BML is designed to be closely related with JML. In particular, JML specifications can be compiled into BML specifications. We briefly discuss the tools that are currently being developed for BML, and that will result in a tool set where an application can be validated throughout its development, both at source code and at bytecode level.

## 1 Introduction

The use of formal methods to show conformance of an implementation *w.r.t.* a specification has become an accepted technique for the development of security-critical applications. Various tools exist that allow to specify and validate complex functional or security properties, using different techniques such as runtime assertion checking, testing and verification condition generation. However, often these techniques are restricted to source code level programs, while for many applications, and in particular for mobile code, one needs to be able to also specify and verify the executable (or interpreted) code.

Different possible reasons for this exist: the executable code may not be accompanied by its (specified) source, or one simply does not trust the compiler. And in an attempt to avoid *all* possible security threats, sometimes security-critical applications are directly developed at the executable level. Thus, it is essential to have the means to specify *and* to verify an application directly at this level, without the use of a compiler, and both specification and verification techniques should be tailored directly to the particularities of executable code. Moreover, in order to capture all relevant security requirements, the specification language used should be expressive enough for this.

* This work is partially funded by the IST FET programme of the European Commission, under the IST-2005-015905 MOBIUS project.

Proof Carrying Code (PCC) is a typical example where the need to specify and verify executable code directly is imperative, in particular when one wishes to capture complex security policies that cannot be checked with a typechecker. PCC is a possible solution to support the secure downloading of applications on a mobile device. The executable code of an application comes together with a specification, and the necessary evidence from which the code client can easily establish that the application respects its specification. In such a scenario, the code producer, who *has* to produce a correctness proof, will often prefer to do the verification at source code level, and then compile the specification and the proof into the level of executable code. Realising a platform to support this scenario is one of the goals of the MOBIUS project[1].

This paper describes the low-level specification language that we propose to specify the security requirements for mobile device applications. Since the most common execution framework for mobile devices is the J2ME platform, our low-level specification language is tailored to Java bytecode. Our language is designed to support the verification of *unstructured* code. And, as we want to be able to translate source code level specifications into bytecode level specifications, our specification language is also designed to be closely related to the Java Modeling Language (JML)[2].

Over the last few years, JML has become the *de facto* specification language for Java source code programs. Different tools exist that allow to validate, verify or generate JML specifications (see [9] for an overview). Several case studies have demonstrated that JML can be used to specify realistic industrial examples, and that the different tools allow to find errors in the implementations (see *e.g.* [8]). One of the reasons for its success is that JML uses a Java-like syntax. Specifications are written using preconditions, postcondition, class invariants and other annotations, where the different predicates are side-effect-free Java expressions, extended with specification-specific keywords (*e.g.* logical quantifiers and a keyword to refer to the return value of a method). Other important factors for the success of JML are its expressiveness and flexibility. JML is in particular suited to express many of the different security requirements that have been identified to be of interest for mobile device applications [13].

Therefore, we define a variation of JML especially tailored to bytecode, called BML, short for Bytecode Modeling Language. BML supports the most important features of JML. Thus, we can express functional properties of Java bytecode programs in the form of *e.g.* pre- and postconditions, class and object invariants, and assertions for particular program points like loop invariants. Because of the close connection with JML, JML source code level specifications can be compiled into BML bytecode level specifications without too much difficulty, basically by compiling the source code predicates into bytecode predicates. This allows to do development and verification at source code level, while still being able to ship bytecode level proofs. To the best of our knowledge, no other specification language with similar design goals exists for Java bytecode.

[1] See `http://mobius.inria.fr`.
[2] See `http://www.jmlspecs.org`.

Section 2 gives a quick summary of the relevant features of JML. Section 3 then gives a detailed account of BML, describing its syntax and semantics, while Section 4 proposes a format in which BML specifications can be stored in the class file (just as JML specifications can be written as special comments in the Java source code). Section 5 then discusses the compilation process from JML to BML, while Section 6 wraps up and discusses tool support and related and future work.

## 2 A short overview of JML

This section gives a short introduction to JML, by means of an example. Throughout the rest of this paper, we assume that the reader is familiar with JML, its syntax and its semantics. For a detailed overview of JML we refer to its reference manual [15]. Where necessary, we refer to the appropriate sections of this manual. A detailed overview of the tools which support JML can be found in [9].

To illustrate the different features of JML, Figure 1 shows an example class specification, defining the class `Bill`. It contains an abstract method `round_cost`, that computes the cost of a particular round. The method `produce_bill` is supposed to sum up the costs of the different rounds.

In order not to interfere with the standard Java compiler, JML specifications are written as special comments (tagged with `@`). Method specifications contain preconditions (keyword `requires`), postconditions (`ensures`) and frame conditions (`assignable`). The latter specify which variables *may* be modified by a method. In a method body, one can annotate all statements with an `assert` predicate and loops also with invariants (`loop_invariant`), variants (`decreases`) and loop frame conditions (`loop_modifies`). The latter is a non-standard extension of JML, introduced in [11], which we found useful to make program verification more practical. One can also specify class invariants, *i.e.* properties that should hold in all visible states of the execution, and constraints, describing a relation that holds between any two pairs of consecutive visible states (where visible states are the states in which a method is called or returned from).

The predicates in the different conditions are side-effect free Java boolean expressions, extended with specification-specific keywords, such as `\result`, denoting the return value of a non-void method, and `\old`, indicating that an expression should be evaluated in the pre-state of the method.

JML allows to declare special specification-only variables: logical variables (with keyword `model`) and so-called `ghost` variables, that can be assigned to in special `set` annotations.

In Figure 1, the specification for `round_cost` states that the result of the method should be positive, but less than the number of the round. The specification for `produce_bill` requires that we compute at least one round, and then ensures an upper-bound on the outcome of the method. We use a loop invariant and loop frame condition to prove the method body correct. Finally, the class invariant specifies that the `sum` field is always positive.

```
/* @author Hermann Lehner, Aleksy Schubert
 * The Bill class provides an abstract implementation of the bill
 * functionality. It calculates the aggregate cost for series of
 * investments based on the implementation of the method which gives
 * the cost of a single round (to be implemented in subclasses). */
abstract class Bill {

  private int sum;  //@ invariant sum>=0;

  /* This method gives a cost of a single round.
   * @param x is the number of the particular round
   * @return the cost of the investment in this round, it's not
   *         greater than <code>x</code> */
  //@ ensures 0 <= \result && \result <= x;
  abstract int round_cost(int x) throws Exception;

  /* This method calculates the cost of the whole series of investments.
   * @return <code>true</code> when the calculation is successful and
   *         <code>false</code> when the calculation cannot be performed */
  //@ requires n > 0;
  //@ ensures sum <= \old(sum)+n*(n+1)/2;
  public boolean produce_bill(int n){
    try{
      //@ loop_modifies sum, i;
      //@ loop_invariant 0 <= i && 0 <= sum && i <= n + 1 && sum <= \old(sum)+(i-1)*i/2;
      for (int i=1;i<=n;i++)
      { this.sum = this.sum + round_cost(i); }
      return true;
    } catch (Exception e){ return false; }
  }
}
```

**Fig. 1.** Class `Bill` with JML annotations

## 3 The Bytecode Modeling Language

### 3.1 Syntax of BML

Basically, BML has the same syntax as JML with two exceptions:

1. specifications are not written directly in the program code, they are added as special attributes to the bytecode; and
2. the grammar for expressions only allows bytecode expressions.

*Syntax for predicates and specification expressions* Figure 2 displays the most interesting part of this grammar for predicates and specification expressions, defining the syntax for primary expressions and primary suffixes (see Appendix A

```
predicate ::= ...

unary-expr-not-plus-minus ::= ...
        | primary-expr [primary-suffix]...
primary-suffix ::= . ident
        | ( [expression-list] )
        | [ expression ]
primary-expr ::= #natural                  % reference in the constant pool
        | lv[natural]                      % local variable
        | cntr                             % counter of the operand stack
        | st(additive-expr)                % stack expressions
        | length(expression)               % array length
        | constant | super
        | true | false | this | null
        | (expression)
        | jml-primary
```

**Fig. 2.** Fragment of grammar for BML predicates and specification expressions

for a short explanation of the syntax notation and the full grammar of predicates and specification expressions). Primary expressions, followed by zero or more primary suffixes, are the most basic form of expressions, formed by identifiers, bracketed expressions *etc.*

Since only bytecode expressions can be used, all field names, class names *etc.* are replaced by references to the constant pool (a number, preceded by the symbol `#`), while registers are used to refer to local variables and parameters. The register `lv[0]` of a non-static method always contains the implicit argument `this`, the other registers contain the parameters and the local variables declared inside a method body. Compilers often reuse local variable registers throughout the execution of a single method. Thus, when *e.g.* type checking an annotation containing a local variable, it has to be taken into account at which point in the code the annotation is evaluated (but notice that this is not more complicated than reusing the same local variable names in different block statements).

In addition, we can use the stack counter (`cntr`) and stack expressions (`st(`$e$`)`, where $e$ is some arithmetic expression) to describe intermediate states of a computation. These are only used to express predicates over states in the body of a method, they are not used in method specifications. We also add a special expression `length(`$a$`)`, denoting the length of array $a$. Since the source code expression $a$`.length` is compiled into a special bytecode instruction `arraylength`, we also need a special specification construct for this at bytecode level.

In Java source code, one can usually leave the receiver object `this` implicit. But compilation into bytecode makes this object explicit, *i.e.* instructions such as `putfield` *always* require that the receiver object is loaded on the operand stack. In analogy with this, BML specifications require that the receiver object is written explicitly in expressions (see Figure 3 below).

In JML, many special keywords are preceded by the symbol \, to ensure that they will not clash with variable names. For BML, we do not have to worry about

this: all variable names are replaced by references to the constant pool or local variable registers. Therefore, the new keywords are written without a special preceding symbol. However, for convenience, we keep the symbol for keywords that are also JML keywords.

At the moment, the use of pure methods is not part of the BML grammar, as there is still ongoing research on the exact semantics of method calls used in specifications. However, we believe that if the theoretical issues have been settled, eventually any tool supporting BML should also support this[3].

*Class and method specifications* BML contains equivalent constructs for all specification constructs of JML Level 0 (see [15, §2.9]), which defines the features that should be understood and checked by all JML tools. In addition, it contains several constructs from JML level 1, that we find important to be able to write meaningful specifications for the example applications studied in the MOBIUS project, namely static invariants; object and static constraints; and loop variants.

We choose to keep a notion of loop specification in BML, even though there is no high level loop construct in bytecode. However, to be able to prove termination, one needs to prove decrease of a loop variant, which makes the treatment of loops different from the treatment of other statements. Also, experiences with verification of realistic case studies have shown that it is beneficial to know which variables may be modified by the code block that corresponds to the loop. For this, we use the special clause `loop_modifies`. This allows to write concise specifications, and to efficiently generate proof obligations using a weakest precondition calculus. Moreover by keeping the notion of loop specification explicit in BML, we keep the correspondence with JML specifications more direct.

As mentioned above, specifications are stored as special attributes in the class file. This means that every class contains a table with invariant and constraint annotations, while each method has extra attributes containing its specifications. Finally, the code for the method body is annotated with local annotation tables for the assert annotations and the loop specifications. Section **??** defines the precise format of these attributes.

Since the bytecode and BML specifications are two separate entities, they should be parsed independently. Concretely this means that the grammar of BML is similar to the grammar of type specifications, method specifications and data groups of JML [15, §A.5, A.6, A.7], restricted to the constructs in JML level 0, plus the constructs of JML level 1 mentioned, but with the changes to the grammar for predicates and specification expressions, as mentioned above.

*An example BML specification* To show a typical BML specification, Figure 3 presents the BML version of the specification of method `produce_bill` of the JML example in Figure 1. Notice that the field `sum` has been assigned the number 24 in the constant pool, and that it is always explicitly qualified with `lv[0]`

[3] In fact, we think that both at source code and at bytecode level, specifications will benefit significantly from being allowed to use method calls in them.

```
{| requires lv[1] > 0
   ensures lv[0].#24 <= \old(lv[0].#24) + lv[1] * (lv[1] + 1) / 2 |}
 0 iconst_1
 1 istore_2
 2 goto 22
 5 aload_0
 6 aload_0
 7 getfield #24 <Bill.sum>
10 aload_0
11 iload_2
12 invokevirtual #26 <Bill.round_cost>
15 iadd
16 putfield #24 <Bill.sum>
19 iinc 2 by 1
loop_invariant 0 <= lv[2] && 0 <= lv[0].#24 && lv[2] <= lv[1] + 1 &&
               lv[0].#24 <= \old(lv[0].#24) + (lv[2] - 1) * lv[2]/2
entry loop:
22 iload_2
23 iload_1
24 if_icmple 5
27 iconst_1
28 ireturn
29 astore_3
30 iconst_0
31 ireturn
```

**Fig. 3.** Bytecode + BML specification for method `produce_bill` in class `Bill`

(denoting `this`). Further, `lv[1]` denotes the parameter `n`, while `lv[2]` denotes the local variable `i`.

The class invariant gives rise to the following BML specification (stored in the class file as a special user-specific attribute, as explained below):

```
invariant:  #24 >= 0
```

This expression is not qualified with `lv[0]`, as it is implicitly quantified over all objects that are an instance of a subclass of class `Bill` (*cf.* the JML semantics [15, §8.2]).

### 3.2 Well-formed BML specifications

A BML specification is said to be *well-formed* if it satisfies several structural and typing constraints, similar to the structural and typing constraints that the bytecode verifier imposes over the class file format.

Examples of typing constraints that a BML specification must respect are the following:

- field access expression $e.ident$ is well-typed only if $e$ is of a subtype of the class where the field described by the constant pool element at index *ident* is declared;
- array access expression $e_1[e_2]$ is well-typed only if $e_1$ is of array type and $e_2$ is of integer type; and
- predicate $e_1$`<:`$e_2$ is well-typed only if the expressions $e_1$ and $e_2$ are of type `java.lang.Class` (which is the same as the JML type `\TYPE` ).

Examples of structural constraints that a BML specification must respect are the following:

- all references to the constant pool must be to an entry of the appropriate type; for example, for field access expression $e.ident$, *ident* must reference a field in the constant pool; while for expression `\type(`*ident*`)`, *ident* must be a reference to a constant class in the constant pool;
- every *ident* in a BML specification must be a correct index in the constant pool; and
- if the expression `lv[`$i$`]` appears in a BML method specification, $i$ must be a valid index in the method's local variables table.

These well-formedness checks for BML are best implemented as an extension of the bytecode verifier.

### 3.3 Semantics of BML expressions

The semantics of BML specifications follows the semantics of JML specifications [15]. But, just as a JML specification can be mapped into a more fundamental Hoare triple specification, we can also define a semantics for BML in terms of a basic logic for Java bytecode, namely the so-called MOBIUS base logic. This logic will be the core of the PCC platform developed within the project. This logic (see [7] for an earlier version, without exceptions) has been proven sound in Coq *w.r.t.* a formalisation of the virtual machine. On top of this, a direct verification condition generator has been proven sound, also in Coq. And, as a first step towards efficient tool development, a translation of bytecode into guarded commands has been defined and proven correct, *w.r.t.* verification condition generation.

Defining the mapping of BML specifications into this MOBIUS base logic is defined in two steps. First the evaluation of predicates is defined over the program state (*i.e.* over the heap, store and operand stack), and second the complete BML specifications are translated into judgements of the MOBIUS base logic. Notice that this embedding allows to use the verification condition generator for the MOBIUS base logic also for BML specifications.

Judgements in the MOBIUS base logic are of the form $\mathsf{G}, \mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$, where $\mathsf{G}$ is a proof context, and $\mathsf{Q}$ the local annotation table, *i.e.* the table that associates assert annotations with particular instructions. Further, $A$ is a (local) precondition, relating the state at label $pc$ with the initial state, while $B$ is a (local) postcondition, relating the initial, current and final state, and $I$ is a

(local) invariant, *i.e* a predicate that is supposed to hold throughout execution of the current method.

Mapping class specifications (invariants and constraints) and method specifications into the MOBIUS base logic is straightforward. Since the MOBIUS base logic only has one postcondition, the normal and exceptional postconditions are combined into a single postcondition, specifying with a case distinction which conditions should hold if the state is normal or exceptional, respectively. Frame conditions are also added to the postconditions, specifying explicitly which variables are allowed to be changed. Since predicates in the MOBIUS base logic specify properties over the whole heap, this can be expressed directly: all locations that are not mentioned in the frame condition of the method (evaluated in the pre-state of the method) should be unchanged. Methods with multiple specifications are translated only after desugaring them into a single method specification *cf.* [20].

Assert and set statements are inserted directly in the local annotation table[4]. However, for loop specifications some manipulations are necessary to produce the appropriate assert annotations, due to the unstructured nature of bytecode. The loop invariants can be added directly to the local annotation table, but loop variants and loop frame conditions first are transformed into a sequence of assert and set annotations (after introducing appropriate ghost variables). This transformation is done at the level of BML, after which we can add the annotations to the local annotation table.

The transformation of the loop variant basically proceeds as follows. Let `variant` be the expression declared in the decreases clause. We declare ghost variables `loop_init` (initially set to true) and `loop_variant` (whose initialisation is not essential). If $l$ is the program point where we enter the loop, then at that point we add an assertion

```
//@ assert !loop_init ==> (0 <= variant && variant < loop_variant);
```

followed by:

```
//@ set loop_init = false;
//@ set loop_variant = variant;
```

This ensures that every time the loop entry point $l$ is reached again, the decrease of the loop variant is checked. Only a path that goes through the loop can set `loop_init` to false.

For transforming loop frame conditions, we use again that in the MOBIUS base logic we can express properties of the heap. We make a transformation into a sequence of assert and set statements, declaring ghost variables to remember the old heap and all locations mentioned in the loop frame condition, and a ghost variable `loop_init` as above. Then we assert at the entry point of the heap that if `loop_init` does not hold, any location that is not mentioned in the loop frame condition should remain unchanged. Notice that this assertion

[4] In fact, at the moment, the MOBIUS base logic does not support ghost variables; but these will be added in the near future.

cannot be directly expressed in BML, but it can be expressed in the MOBIUS base logic. Finally, in the MOBIUS base logic we add appropriate ghost variable updates to remember the old heap and the locations of the loop frame condition when the loop was first entered.

## 4 Encoding BML specifications in the class file format

To store BML specifications together with the bytecode it specifies, we encode them in the class file format. Recall that a class file contains all the information related to a single class or interface, *i.e.* its class name, interfaces implemented by the class, its super class and the methods and fields it declares. The Java Virtual Machine Specification [17] prescribes the mandatory elements of the class file: the constant pool, the field information and the method information. The constant pool is used to construct the runtime constant pool upon class or interface creation. This will serve for loading, linking and resolution of references used in the class. The JVM specification allows to add user-specific information to the class file ([17, §4.7.1]) as special user-specific attributes. When defining these attributes, we took the following considerations into account.

**Java compiler independence** The format of BML specifications in a class file must not depend on any particular (non-optimising) compiler.

**JVM compatibility** The class files augmented with the BML specification must be executable by any implementation of the JVM. The JVM specification does not allow insertion of any user-specific data in the list with bytecode instructions, therefore BML annotations must be stored separately from the method body. Where necessary the annotations contain information about the index of the instruction that they specify. For instance, the attribute that stores BML loop specifications contains also the index of the entry point instruction of the loop.

**Efficiency** The presence of BML annotations should not have any impact on the application's performance, *i.e.*, the augmented class file should not slow down loading or normal execution of the application. Notice that this requirement is important for mobile device applications, where one often has limited resources.

For each class, we add the following information to the class file:

- a second constant pool which contains constant references for the BML specification expressions;
- an attribute with the ghost fields used in the specification;
- an attribute with the model fields used in the specification;
- an attribute with the class invariants (both static and object); and
- an attribute with the constraints (both static and object).

Apart from the second constant pool, all extra class attributes basically contain the name of the attribute, the number of elements it contains, and a list with the actual elements.

```
Ghost_Field_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 fields_count;
    { u2 access_flags;
      u2 name_index;
      u2 descriptor_index;
    } fields[fields_count];
}
```

```
BMLMethod_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    formula requires_formula;
    u2 spec_count;
    { formula spec_requires_formula;
      u2 assignable_count;
      formula assignable[assignable_count];
      formula ensures_formula;
      u2 exsures_count;
      { u2 exception_index;
        formula exsures_formula;
      } exsures[exsures_count];
    } spec[spec_count];
}
```

**Fig. 4.** Attributes for ghost field declarations and method specifications

If a model or a ghost field is dereferenced in the specification, then a **constantFieldRef** is added to the second constant pool as the Java compiler would do for any dereferenced Java field in the original constant pool of the class. Note that this guarantees that the JVM performance will not be affected because of the encoding of the BML specification. In particular, if we would use the original constant pool for storing constants originating from specifications, the search time in the original constant pool might degrade significantly (especially for a large specification).

The left-hand column of Figure 4 shows the format of the ghost fields attribute. This should be understood as follows: the name of the attribute is given as an index into the constant pool. This constant pool entry will be representing a string `"Ghost_Field"`. Next we have the length of the attribute, which should be 2 + 6***fields_count** (the number of fields stored in the list). The **fields** table stores all ghost fields. For each field we store its access flags (*e.g.* `public` or `private`), and the name index and descriptor index, both referring to the constant pool. The first must be a string, representing the (unqualified) name of the variable, the latter is a field descriptor, containing *e.g.* type information. The tags **u2** and **u4** specifies the size of the attribute, 2 and 4 bytes, respectively. The model field attribute is similar.

In a similar way, we define attributes for class invariants and constraints, containing a list of invariants and constraints, respectively. These contain the predicate, a tag whether the invariant (constraint) is defined over instances or static, and appropriate visibility modifiers.

The JVM specification prescribes that the attribute with method information at least contains the code of each method. We add attributes for the method specification, set statements, assert statements, and loop specifications.

The attribute for lightweight behaviour specifications is shown in the right-hand column of Figure 4 (heavyweight behaviour specifications are handled similarly). The global requires formula is the disjunction of all preconditions in the different specification cases of the method. For each specification case, we then have a precondition (**spec_requires_formula**), a list of assignable expressions, a postcondition (**ensures_formula**) and a list of exceptional postconditions (stored in the **exsures** attribute). If a clause is not explicitly specified, its default value will be stored here.

The attributes for set and assert statements and loop specifications have the same format as *e.g.* the ghost variable attribute: a length entry and a list of elements. The elements storing set and assert statements do not only keep the appropriate predicate or expression, but also an index to the appropriate point in the bytecode. Similarly, the elements for loop specifications contain the loop specification (invariant, variant and frame conditions), plus an index to the bytecode instruction that corresponds to the entry of the loop. If the specification does not contain a loop variant, we indicate this, using a special tag for the **decreases** clause.

## 5 Compiling JML specifications into BML specifications

Since it is often easier and more intuitive to write specifications and do verifications at source code level, we have defined a compiler from JML to BML: JML2BML. BML is designed to be very close to JML, so that the correspondence between the original and the compiled specification remains relatively clear. Notice that in principle, the same can be done for the proofs, *i.e.* a source code level proof can be transformed into a bytecode level proof. It is future work to define this in full detail, but some work in this direction has already been done, see *e.g.* [6].

The compilation of the JML specification is separated from the compilation of the Java source code. In fact, JML2BML takes as input an annotated Java source file *and* the Java class file produced by a non-optimising compiler with the debug flag set. This debug information helps us to compile the annotations correctly.

From the debug information, we use in particular the **Line_Number_Table** and the **Local_Variable_Table** attributes. The presence of these attributes is optional [17], but almost all standard non-optimising compilers can generate these. The **Line_Number_Table** is computed as part of the compilation of a method; it links line numbers in the Java source code with the Java bytecode instructions. The **Local_Variable_Table** describes the local variables that appear in a method.

To be able to appropriately compile loop invariants, the control flow graph corresponding to the list of bytecode instructions resulting from the compilation of a method body must be a *reducible control flow graph*. This means basically that every cycle in the graph must have exactly one entry point, or in other words it is not possible to jump into the middle of a cycle from outside the cycle (see [1]

for the full definition of reducibility). Note that this is not a serious restriction; all non-optimising Java compilers produce reducible control flow graphs and in practice even most hand-written bytecode is reducible.

The compilation from JML specifications into BML compilations is defined in several steps. As mentioned above, we assume that the Java source code has been compiled with the debug flag set, and that we have access to the generated class file.

**Compilation of ghost and model field declarations** Ghost and model variables declared in the specification are compiled into the special class attributes that contains all ghost variable and model variable declarations.

**Linking and resolving of source data structures** The JML specification is transformed into an intermediate format, where the identifiers are resolved to their corresponding data structures in the class file. The Java and JML source identifiers are linked to their identifiers on bytecode level, *i.e.* the corresponding indexes either from the second constant pool or from the **Local_Variable_Table** attribute. This is similar to the linking and resolving stage of the Java source code compiler.

**Locating instructions for annotation statements** Annotation statements, like loop specifications and asserts are associated with the appropriate point in the bytecode program, using the **Line_Number_Table** attribute.
A problem is that a source line may correspond to more than one instruction in the **Line_Number_Table**. This makes it complicated to identify the exact loop entry instruction in the bytecode, and thus to know to which instruction the compiled loop specification should be associated.
To solve this, we use the following heuristics: if the control flow graph of the bytecode is reducible and we search from an index in the **Line_Number_Table** that corresponds to the first line of a source loop, then the first loop entry instruction found will be the loop entry corresponding to this source loop. We do not have a formal correctness proof for this algorithm, because it depends on the particular implementation of the compiler. However, our experiments show that the heuristic works successfully for Sun's non-optimising Java compiler.

**Compilation of JML predicates** JML predicates are Java boolean expressions. However, the JVM does not provide direct support for several integral types, such as byte, short, char, or for booleans. Instead, they are encoded as integers. Therefore, the compiler wraps up the boolean expressions in the JML specification by a conditional function, returning 1 if the predicate is true, 0 otherwise.

**Generation of user-specific class attributes** Finally, the complete specification is compiled into appropriate user-specific attributes, using the format defined in the previous section.

## 6 Conclusions and related work

This paper presents the Bytecode Modeling Language (BML). BML allows one to specify and verify an application directly at the level of bytecode. Its syntax

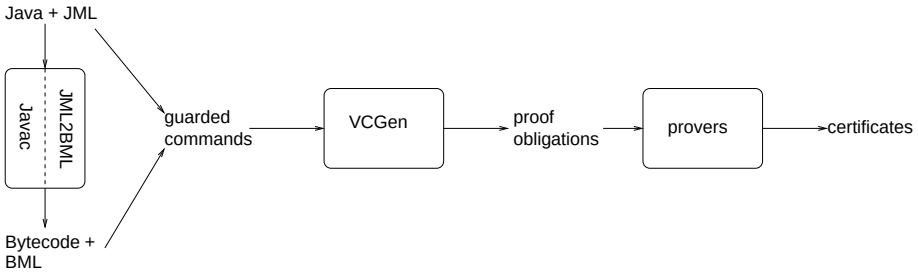


**Fig. 5.** Overview of MOBIUS tool set

and semantics are directly inspired by the source code level specification language JML. The possibility to reason direct at the level of bytecode, without relying on a compiler, is of major importance for guaranteeing the security of applications (for example in a context of mobile code, where some applications are written in bytecode directly, to avoid security problems related with compilation). However, to make such verifications tractable, it is important that the specification language is intuitive and provides a sufficient degree of abstraction, without the need to talk too much about the internal structure of the state (heap, store *etc.*). BML does exactly this: it is designed to be close to the source code level specification language JML and provides a high level of abstraction. It is designed for program verification, and its semantics supports the development of a verification condition generator for unstructured code. Moreover, because of its close connection with JML, it is not too complicated to compile source code level specification into bytecode level specifications.

The BML language as we have defined it now, corresponds roughly to JML level 0, *i.e.* that part of JML whose semantics is relatively well understood. However, more advanced constructs of JML can be easily added to BML, if required.

*Tool support* As part of the MOBIUS project, we plan to develop a program verification tool set that supports both JML and BML. Figure 5 outlines the general architecture of this tool set. Thus, both Java/JML and bytecode/BML can be used as input application. Annotated programs are translated into some guarded command format, for which an appropriate verification condition generator is used to generate proof obligations that can be discharged with a theorem prover (either automatic or interactive). To support the PCC platform, the provers will be instrumented to produce certificates. In addition, source code applications annotated with JML can be compiled into bytecode annotated with BML.

The development of the JML subcomponent of the tool set will be based on experiences with ESC/Java [12] and JACK [11]. Several tools and algorithms (notably the compiler and the verification condition generator) for BML have already been implemented, see [10, 19], but more work is needed to cover the whole language. Moreover, to make the tool set usable in practice, we will also need a tool to inspect and write BML specifications directly, and a run-time checker for

BML specifications. The latter can be implemented by a code transformation, inserting explicit run-time checks in the bytecode, or by extending the virtual machine to take the user-specific attributes with specifications into account. It is also important to have tool support for checking the well-formedness of BML specifications. Such a tool can be built as an extension the Java bytecode verifier.

Our initial experiments with compilation of specifications has shown that there exists indeed a correspondence between the proof obligations generated at source and at bytecode level, modulo differences in elimination of trivial goals, handling of boolean expressions, and the naming convention of generated variables [19]. Moreover, when the proofs are done with the Coq prover, different names are generated for hypotheses at source code and bytecode level. It is future work to clean up the compilation, so there is a one-to-one correspondence.

*Related work* The interest in specification and verification of bytecode applications is quite recent, and not too much work has been done in that direction. Several logics have been developed to reason about bytecode, *e.g.* by Bannwart & Müller [4] and within the MRG project [3]. However, in this work, no attention is given to how one can conveniently write understandable specifications for bytecode.

The development of BML is clearly inspired by the development of the JML specification language [15]. Both JML and BML follow the Design by Contract principle introduced first in Eiffel [18]. The Boogie project [5] introduces in similarly the Design by Contract principles into the C# programming language, both at source code level and for CIL, the .NET intermediate language. The possibility to check a property at run-time, using the `assert` construct, has been long adopted in the C programming language and recently also in Java (Java 1.5, see [14, §14.10]).

Finally, we should mention the Extended Virtual Platform project[5]. This project aims at developing a framework that allows to compile JML annotations, to allow run-time checking [2]. However, in contrast to our work, they do not intend to do static verification of bytecode programs. Moreover, their platform takes JML-annotated source code files as starting point, so it is not possible to annotate bytecode applications directly.

**Acknowledgements**

We thank Lennart Beringer and Olha Shkaravska for discussions about the semantics of BML.

## References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[5] See `http://www.cs.usm.maine.edu/~mroyer/xvp/`.

2. S. Alagić and M. Royer. Next generation of virtual platforms. Article in `odbms.org`, 2005. Available from `http://odbms.org/about_contributors_alagic.html`.
3. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs'04)*, volume 3223 of *LNCS*, pages 34–49. Springer, 2004.
4. F.Y. Bannwart and P. Müller. A logic for bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, volume 141 of *ENTCS*, pages 255–273. Elsevier, 2005.
5. 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 *FMCO*, LNCS. Springer, 2005.
6. G. Barthe, T. Rezk, and A. Saabas. Proof obligations preserving compilation. In R. Gorrieri, F. Martinelli, P. Ryan, and S. Schneider, editors, *Proceedings of FAST'05*, volume 3866 of *LNCS*, pages 112–126. Springer, 2005.
7. L. Beringer and M. Hofmann. A bytecode logic for JML and types. In *ASIAN Symposium on Programming Languages and Systems (APLAS 2006)*, 2006. To appear.
8. C. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: an experience report. *Science of Computer Programming*, 55:53–80, 2005.
9. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3), 2005.
10. L. Burdy and M. Pavlova. Java bytecode specification and verification. In L.M. Liebrock, editor, *proceedings of SAC'06*. ACM, 2006.
11. L. Burdy, A. Requet, and J.-L. Lanet. Java Applet Correctness: a Developer-Oriented Approach. In *Formal Methods (FME'03)*, number 2805 in LNCS, pages 422–439. Springer, 2003.
12. D. Cok and J.R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS*, volume 3362 of *LNCS*, pages 108–128. Springer, 2004.
13. Framework- and application-specific security requirements, 2006. Deliverable 1.2 for the Mobius project.
14. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Sun Microsystems, Inc., 2005.
15. G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. JML reference manual. Preliminary draft `http://www.cs.iastate.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html`, 2005.
16. H.F. Ledgard. A human engineered variant of BNF. *ACM SIGPLAN Notices*, 15:57–62, 1980.
17. T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification. Second Edition*. Sun Microsystems, Inc., 1999.
18. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd rev. edition, 1997.
19. M. Pavlova. *Specification and verification of Java bytecode*. PhD thesis, Université de Nice Sophia-Antipolis, 200x.
20. A.D. Raghavan and G.T. Leavens. Desugaring JML method specifications. Technical Report TR #00-03e, Department of Computer Science, Iowa State University, 2000. Current revision from May 2005.

# A Grammar for BML predicates and specification expressions

**NOTE: We intend to set up a webpage with the complete BML grammar before publication. This grammar is included for the reviewer's convenience, but will not be included in the final version of the paper.**

As in the JML Reference Manual [15], we use an extended Backus-Nauer Form (BNF) grammar to describe the syntax of JML. The extensions are as follows [16].

- Nonterminal symbols are written as follows: *nonterminal*.
- Terminal symbols are written as follows: `terminal`.
- Square brackets ([ and ]) surround optional text. Notice that `[` and `]` are terminal symbols.
- The notation ... means that the preceding nonterminal or group of optional text can be repeated zero or more times.

*predicate* ::= *spec-expression*
*spec-expression-list* ::= *spec-expression*
[ , *spec-expression* ] ...
*spec-expression* ::= *expression*
*expression-list* ::= *expression* [ , *expression* ] ...
*expression* ::= *conditional-expr*
*conditional-expr* ::= *equivalence-expr*
[ ? *conditional-expr* : *conditional-expr* ]
*equivalence-expr* ::= *implies-expr*
[ *equivalence-op implies-expr* ] ...
*equivalence-op* ::= `<==>` | `<=!=>`
*implies-expr* ::= *logical-or-expr*
[ `==>` *implies-non-backward-expr* ]
| *logical-or-expr* `<==` *logical-or-expr*
[ `<==` *logical-or-expr* ] ...
*implies-non-backward-expr* ::= *logical-or-expr*
[ `==>` *implies-non-backward-expr* ]
*logical-or-expr* ::= *logical-and-expr* [ '||' *logical-and-expr* ] ...
*logical-and-expr* ::= *inclusive-or-expr* [ `&&` *inclusive-or-expr* ] ...
*inclusive-or-expr* ::= *exclusive-or-expr* [ '|' *exclusive-or-expr* ] ...
*exclusive-or-expr* ::= *and-expr* [ `^` *and-expr* ] ...
*and-expr* ::= *equality-expr* [ `&` *equality-expr* ] ...
*equality-expr* ::= *relational-expr* [ `==` *relational-expr*] ...
| *relational-expr* [ `!=` *relational-expr*] ...
*relational-expr* ::= *shift-expr* < *shift-expr*
| *shift-expr* > *shift-expr*
| *shift-expr* <= *shift-expr*
| *shift-expr* >= *shift-expr*
| *shift-expr* <: *shift-expr*

| *shift-expr* [ `instanceof` *type-spec* ]
*shift-expr* ::= *additive-expr* [ *shift-op additive-expr* ] ...
*shift-op* ::= `<<` | `>>` | `>>>`
*additive-expr* ::= *mult-expr* [ *additive-op mult-expr* ] ...
*additive-op* ::= `+` | `-`
*mult-expr* ::= *unary-expr* [ *mult-op unary-expr* ] ...
*mult-op* ::= `*` | `/` | `%`
*unary-expr* ::= `(` *type-spec* `)` *unary-expr*
| `+` *unary-expr*
| `-` *unary-expr*
| *unary-expr-not-plus-minus*
*unary-expr-not-plus-minus* ::= `~` *unary-expr*
| `!` *unary-expr*
| `(` *built-in-type* `)` *unary-expr*
| `(` *reference-type* `)` *unary-expr-not-plus-minus*
| *primary-expr* [ *primary-suffix* ] ...
*primary-suffix* ::= `(` [ *expression-list* ] `)`
| '`[`' *expression* '`]`'
*primary-expr* ::= `\#`*natural* | `lv[` *natural* `]`
| *constant* | `super` | `true`
| `false` | `this` | `null`
| `(` *expression* `)`
| *bml-primary*
| *jml-primary*
*built-in-type* ::= `void` | `boolean` | `byte`
| `char` | `short` | `int`
| `long` | `float` | `double`
*constant* ::= *java-literal*
*bml-primary* ::= *array-length-expression* |
| *opstack-counter-expression*
| *stack-expresion*
*array-length-expression* ::= `length(` *expression* `)`
*opstack-counter-expression* ::= `cntr`
*stack-expression* ::= `st(` *additive-expr* `)`
*jml-primary* ::= *result-expression*
| *old-expression*
| *fresh-expression*
| *nonnullelements-expression*
| *typeof-expression*
| *elemtype-expression*
| *type-expression*
| *spec-quantified-expr*
*result-expression* ::= `\result`
*old-expression* ::= `\old (` *spec-expression* `)`
| `\pre (` *spec-expression* `)`

*fresh-expression* ::= `\fresh` `(` *spec-expression-list* `)`
*nonnullelements-expression* ::= `\nonnullelements` `(` *spec-expression* `)`
*typeof-expression* ::= `\typeof` `(` *spec-expression* `)`
*elemtype-expression* ::= `\elemtype` `(` *spec-expression* `)`
*type-expression* ::= `\type` `(` *type* `)`
*spec-quantified-expr* ::= `(` *quantifier quantified-var-decls* `;`
[ [ *predicate* ] `;` ]
*spec-expression* `)`
*quantifier* ::= `\forall` | `\exists`
*quantified-var-decls* ::= [ *bound-var-modifiers* ] *type-spec quantified-var-declarator*
[ `,` *quantified-var-declarator* ] ...
*quantified-var-declarator* ::= *ident* [ *dims* ]
*bound-var-modifiers* ::= `non_null` | `nullable`
*store-ref-list* ::= *store-ref* [ `,` *store-ref* ] ...
*store-ref* ::= *store-ref-expression*
| *store-ref-keyword*
*store-ref-expression* ::= *store-ref-name* [ *store-ref-name-suffix* ] ...
*store-ref-name* ::= `\#` *natural* | `super` | `this`
*store-ref-name-suffix* ::= `(` *store-ref-expression* `)`
| '`[`' *spec-array-ref-expr* '`]`'
*spec-array-ref-expr* ::= *spec-expression*
| *spec-expression* `..` *spec-expression*
| `*`
*store-ref-keyword* ::= `\nothing` | `\everything` | `\not_specified`