Verification and certification of Java classes using BML tools[⋆]

Jacek Chrząszcz, Aleksy Schubert, and Tadeusz Sznuk

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

Abstract The Bytecode Modeling Language (BML) is a specification language for Java bytecode. 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. In this paper we present a case study which demonstrates the usability of the tools within the proof-carrying code scheme. We describe here all the stages of the certified bytecode generation. These include the generation of the bytecode class with BML specifications, the generation of the verification conditions to be prooved in Coq and then packaging of the verification conditions in a class file.

1 Introduction

Bytecode Modeling Language (BML) [4,6] is a specification language designed to allow the specification of properties in programs for which the bytecode form is available. BML is based on the principle of design-by-contract and it is strongly inspired by the Java Modeling Language (JML) [9,10,11]. JML is the *de facto* Java specification language, supported by a wide range of tools [3]. A set of tools [5] has been developed which allow one to manipulate class files with BML specifications.

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

[⋆] 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.

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.

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[1].

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;
- BMLVCgen, a generator of verification conditions in Coq for bytecode class files enhanced with 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* [6] while the description of the tools is available in [5]. The current paper gives a round trip that shows the way the tools can be used to obtain Java bytecode augmented with PCC certificates. Section 2 presents the way the verification conditions are generated. The way the generated conditions are proved is presented in Section 3. We conclude with the description of the certificate generation and checking in Section 4.

2 Generation of verification conditions

In order to generate the verification conditions to be proved, one has to provide the properties which are supposed to be checked against the code. The properties are in our case expressed in BML. Then the actual verifications conditions can be created. In our case these are Coq formulae which express the programs and their properties in terms of the Bicolano semantics. The translation procedure is presented in [8]. A more detailed presentation of the procedure is presented below.

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

2.1 Adding specifications to class files

The basic tool which enables the possibility to generate class files with bytecode and BML is the Umbra editor. It supports editing of BML specifications together with bytecode mnemonics. This editor can manipulate bytecode files in textual form in parallel with class files.

A separate part of the Umbra editor is a library called BMLLib. BMLLib parses fragments of BML specifications, prints them out in the aforementioned textual format, and reads and writes BML specifications in class files.

The Umbra editor is also equipped with an automatic tool to transform the JML specifications to BML ones called JML2BML. The compiler can also work as a standalone tool which takes source code with JML annotations, a class file and add the BML specifications to the class file. More details are presented in [7].

In order to understand how the bytecode specifications are generated, consider the class presented in Figure 1. This class provides an abstract implementation of a bill functionality and it calculates an aggregate cost for a series of investments based on an abstract implementation of a method that calculates the cost of a single round.

The translated version of the Bill class is presented in Figure 2.

2.2 The generation of the verification conditions in Coq

The bytecode subsystem is able to generate the verification conditions to be proved in Coq. The tool which does that, BMLVCgen, uses the mechanisms of the MOBIUS Direct Verification Generator so the resulting conditions are very similar to the ones obtained from that. This makes possible to conduct direct verification of bytecode programs when no source code is available. More details are mentioned in [5].

The MOBIUS DirectVCgen is a tool built upon bico, Bicolano, the ESC/Java2 parser, as well as PVE's prover backend. It is built around two verification condition generators, one over source code and one over bytecode. The BMLVCgen uses only the bytecode part of the DirectVCgen functionality.

The MOBIUS DirectVCgen is called "direct" because, contrary to the other verification condition generators of the PVE, it uses a simple weakest precondition calculi without any use of an intermediate language. The source VCgen is written in Java and the bytecode VCgen is written in Coq. It is part of the proof transforming compiler scheme, as explained in the deliverable 4.3 [13].

The background predicates for the bytecode verification condition generator are found in the theory of Bicolano. The tool bico is used to translate a program into the Coq formalisation. These mechanisms of the MOBIUS DirectVCgen are used by the bytecode subsystem to generate verification conditions based on the bytecode alone and BML specifications. The transition between bytecode combined with BML annotations and the DirectVCgen is done by a tool called BMLVCgen.

```
/**
 * 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).
 *
 * @author Hermann Lehner, Aleksy Schubert, Joseph Kiniry
 */

abstract class Bill {
  private /*@ spec_public @*/ int sum;
  /*@ public invariant 0 <= sum; @*/


  //@ modifies \nothing;
  //@ ensures \result <= x;
  abstract int round_cost(int x) throws Exception;

  /**
   * This method calculates the cost of the whole series of
   * investments.
   *
   * @return true when the calculation is successful and
   *     false when the calculation cannot be performed.
   */
  //@ requires 0 <= n && n < 1000000 && sum < 1000000;
  //@ ensures sum <= \old(sum) + n*(n+1)/2;
  public boolean produce_bill(final int n) {
    int i = 0;
    try {
      //@ loop_invariant i <= n + 1 && sum <= \old(sum)+(i+1)*i/2;
      for (i = 1; i <= n; i++) {
        sum = sum + round_cost(i);
      }
      return true;
    }
    catch (Exception e) {
      return false;
    }
  }
}
```

Figure1. The initial source code and JML specifications for class `Bill`

```
/*@
  @ requires 0 <= n && n < 1000000 && sum < 1000000
  @ modifies \everything
  @ ensures sum <= \old(sum) + n * (n + 1) / 2
  @ signals (java/lang/Exception) true
  @ signals_only \nothing
  @*/
public boolean produce_bill(int n)
0:    iconst_0
1:    istore_2
2:    iconst_1
3:    istore_2
4:    goto              #24
7:    aload_0
8:    dup
9:    getfield          Bill .sum I (23)
12:   aload_0
13:   iload_2
14:   invokevirtual     Bill .round_cost (I)I (25)
17:   iadd
18:   putfield          Bill .sum I (23)
21:   iinc              %2     1
/*@
  @ loop_specification
  @   loop_inv i <= n + 1 && sum <= \old(sum) + (i + 1) * i / 2
  @   decreases 1
  @*/
24:   iload_2
25:   iload_1
26:   if_icmple         #7
29:   iconst_1
30:   ireturn
31:   astore_3
32:   iconst_0
33:   ireturn
```

Figure2. Bytecodes and BML specifications for the method produce_bill in the Bill class

```
Module produce_billT_int.
Definition mk_pre :=
        fun  (heap: Heap.t) (this: value) (lv_1n: Int.t) =>
         (((((((Is_true (le_bool (Int.const (0)) lv_1n)) /\
               (Is_true (lt_bool lv_1n (Int.const (1000000))))) /\
               (Is_true (lt_bool (vInt (do_hget
                                           heap
                                           (Heap.DynamicField
                                           this
                                           BillSignature.sumFieldSignature)))
                                 (Int.const (1000000))))) /\
               (exists loc, this = Ref loc)) /\
               (isAlive heap this)) /\
               (assign_compatible p heap this
                  (ReferenceType (ClassType BillType.name)))) /\
               (forall (r6: Location) (x4:type),
                  ((((isAlive heap r6) /\
                     (assign_compatible p heap r6 x4)) /\
                     (x4 = (ReferenceType (ClassType BillType.name))))
                  -> (inv heap r6 x4)))).

   ...
End produce_billT_int.
```

Figure3. A fragment of the Coq file `Bill_annotations.v` which contains the definition of the precondition, postcondition and all asserts inside of the class methods.

2.3 The work with code

In the course of our case study we were able to generate verification conditions for a small application called Demonstrator. This application consists of 15 classes and was designed to be a simple, but typical application on mobile devices developed by TLS Technologies as a test bed for MOBIUS tools [14]. This application was subsequently annotated with JML specifications by the MOBIUS team from Radboud University in Nijmegen so that ESC/Java2 does not raise warnings. This code together with the JML annotations was used by our team. We successfully translated the JML specifications to BML ones with JML2BML compiler with only minor editing operations in Umbra which gave a little bit better structure of the specifications. These modifications were not essential from the point of view of the semantics. Based upon these specifications we generated verification conditions in Coq using BMLVCgen.

The task to prove all the verification obligations in Coq turned out to be very time consuming. Therefore, we were able to verify only one class within the given time resources. We decided to verify the already mentioned Bill class. A fragment of the conditions generated for the class is presented in Figure 3.

```
/**
 * 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).
 *
 * @author Hermann Lehner, Aleksy Schubert, Joseph Kiniry
 */

abstract class Bill {
  private /*@ spec_public @*/ int sum;
  /*@ public invariant 0 <= sum; @*/


  //@ modifies \nothing;
  //@ requires 0 < x;
  //@ 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 true when the calculation is successful and
   *    false when the calculation cannot be performed.
   */
  //@ requires 0 <= n && n < 10000 && sum < 10000;
  //@ ensures sum <= \old(sum) + n*(n+1)/2;
  public boolean produce_bill(final int n) {
    int i = 0;
    try {
      /*@ loop_invariant 0 < i && i <= n + 1 &&
        @                0 <= \old(sum) && \old(sum) <= 10000 &&
        @                0 <= sum && sum <= \old(sum) + (i-1)*i/2 &&
        @                0 <= n && n < 10000 && \old(n) == n;
        @*/
      for (i = 1; i <= n; i++) {
        sum = sum + round_cost(i);
      }
      return true;
    }
    catch (Exception e) {
      /* in fact this return case requires additional
       * signals (Exception) \old(sum) == sum;
       * in the specs of round_cost.
       */
      return false;
    }
  }
}
```

Figure4. The final source code and JML specifications for class `Bill`

Problems with the specification The work with the example revealed that the initial specifications, especially the expression of the loop invariant and the initial bounds on the parameter n in the Bill class are not correct and not complete to carry the proof out.

First of all the bound on the parameter n in the original requires clauses were too big. Indeed, the sum of all the natural numbers $1, \ldots, 1000000$ does not fit in the 32-bit integers and in case a big number is used the pattern in the ensures clause does not hold. Additionally, we changed the assumption about the size of the `sum` field at the entry of the method. This was done to speed up some of the proofs checked by Coq and does not affect neither the correctness of the method nor the proof.

As far as the loop invariant is concerned, we had to amend it in three major ways. First, we had to supply the information on the bounds of all the involved variables. Second, we had to add the information that the variable n is not changed within the loop ($\backslash old(n) == n$). Finally, the arithmetic pattern $(i + 1) * i/2$ which was initially used in the specifications as a bound on the value of the `sum` field was not correct. In particular, it was impossible to infer the postcondition with the final bound $n*(n+1)/2$ in case $n = 0$ using that invariant form. Therefore, we changed the invariant to contain $(i - 1) * i/2$. With these amendments the proof went smoothly.

The final specification of the Bill class is presented in Figure 4.

Problems with the verification conditions In the course of the verification, it turned out that the definition of the formula which should be assumed at the entry to the method was too weak. The formula should specify that certain condition must hold after the method provided that the method is called in an appropriate state. In particular, the assumptions in the formula should state that `this` variable points to an appropriate object on the heap. These assumptions were missing and were added by us by hand to make the verification possible. Figure 3 contains a version of the formula which contains the assumptions added by hand. These are

```
Module produce_billT_int.
Definition mk_pre :=
              ...
              (exists loc, this = Ref loc)) /\
              (isAlive heap this)) /\
              (assign_compatible p heap this
                 (ReferenceType (ClassType BillType.name)))) /\


  ...
End produce_billT_int.
```

Additionally, it turned out that the generated conditions required us to prove `False` in case of the abstract method and in case of the implicit constructor. This choice can be explained in case of the abstract method as the method is actually not called in any run of the program. However, the choice for the constructor is semantically wrong as the implicit constructor can actually be called in case the class is subclassed by a class with no explicit constructor or in case the constructor in the subclass calls explicitly the implicit one in the superclass.

Problems with the process of proving In the course of the case study, it turned out that it is very difficult to find out what kind of proof obligation one proves in the particular moment. The original tactics and proof facilitation provided inside Coq worked relatively well in case all the specifications were correct and located in places predicted by the authors. However, they were extremely difficult to use in case something was wrong or slightly different than the authors assumed.

In particular, the invariant originally generated by the BML tools was located right before the conditional jump that checks the loop condition. However, the Coq facilities assumed this is located before the whole calculation of the jump condition starts. The relocation of the invariant to the place predicted by the authors made the proof simpler.

Another problem that showed up in the course of the proof development was a wrong modelling of boolean type in Bicolano. The semantics of the Java Virtual Machine requires that the boolean values are modelled as integers whereas the Bicolano formalisation assumes that the boolean values are bytes.

We developed a special technique of reducing the goal to head normal form. This allowed us to learn the whole process of the verification condition reduction and find out where precisely we lack particular assumptions.

3 Proving verification conditions

The verification conditions generated by the VCGen developed to a number of different subgoals. In order to prove subgoals concerned with inequalities, a number of auxiliary tactics was developed. Since the Java Virtual Machine model uses machine integers (i.e. bounded integers with operations modulo), in order to translate JVM operations into Coq arithmetics, one must make sure there are no integer overflows. Moreover, in order to use the Coq arithmetic tactics (such as `omega`) one must first get rid of layers of abstraction introduced by the implementation of JVM in Coq. The tactics we developed faciliate these automatic steps.

The first tactic, called `zetujg` and presented in Fig. 5 translates the VCGen inequation appearing in the current goal into a `Z` equation, which can be treated by `omega`. For example, a goal which looks like this

```
Is_true (le_bool (Int.const 1) (Int.add n (Int.const 1)))
```

is translated into

```
1 <= Int.toZ n + 1
```

The tactic works as follows: if the current goal matches the general form of a VC-Gen inequation, the suitable inequality constant is unfolded, which uncovers the

`Int.toZ` injection. Then the tactic `arith` is called which applies homomorphism-like equations to both sides of the inequality in order to *push* `Int.toZ` down in the tree of the expression, transforming machine integer operations into `Z` ones. The application of these equations is often guarded by the condition that certain parts of expression are within the range machine integers. Therefore a number of auxiliary goals of the form `Int.range` t can be generated by `arith`. Some of them are hopefully trivial though.

Since there is no distinction between the main subgoal and the secondary ones, we must use the `try` construction for the rest of the tactic, but in fact it should never fail on the main subgoal. The rest of the tactic introduces the `Z` form of the inequality (the `assert` tactic) and proves the VCGen inequation from it. The `Z` form is left to the user.

```
Ltac zetujg :=
  match goal with
  | |- Is_true (le_bool _ _) =>
        unfold le_bool;
        arith; trivial; try (
        match goal with
        |- Is_true (Zle_bool ?x ?y) =>
                let Hi := fresh in
                assert (Zle x y) as Hi; [idtac |
                unfold Zle in Hi; (arith; trivial);
                try (unfold Zle_bool;
                destruct (x ?= y)%Z; simpl; tauto)
                ]
        end)
  | |- Is_true (lt_bool _ _) =>
        unfold lt_bool;
        arith; trivial; try (
        match goal with
        |- Is_true (Zlt_bool ?x ?y) =>
                let Hi := fresh in
                assert (Zlt x y) as Hi; [idtac |
                unfold Zlt in Hi; (arith; trivial);
                try (unfold Zlt_bool;
                destruct (x ?= y)%Z; simpl; (discriminate || tauto))
                ]
        end)
   | |- Int.range _ =>
        red;
        let x:=fresh in
        set (x:=Int.half_base); compute in x; subst x
   end.
```

Figure5. The tactic `zetujg`.

The last clause of the definition of the tactic is concerned with the goals of the form `Int.range` t. What it does is unfold certain constants in order to transform it to the form accepted by `omega`.

A very similar tactic, treating VCGen inequalities appearing in hypotheses is `zetujh` presented in Fig. 6. The way it works is of course very similar with one exception: since it is much more convenient to work on a goal than on a hypothesis, it starts with the `revert` tactic, and in order not to perform unwanted modifications on the original goal it is temporarily folded to a local definition which is unfolded at the end.

4 Generation of certificates

The bytecode tool set contains also a small tool CCT which enables the possibility to pack a certificate being e.g. a proof done in Coq to class files. This tool also makes possible to unpack and check that the code in the class file indeed has the property proven by the certificate.

This tool has been adapted to pack Coq proofs in a class level certificate as documented in [12]. The certificate is verified at the code consumer side by the following procedure:

- the generation of the verification conditions using BMLVCgen and the BML specifications in the class file,
- the extraction of the Coq proofs from the class file,
- the compilation by Coq of the proofs against the freshly generated verification conditions.

We consider the class file correctly verified in case the final compilation by Coq is successfull.

It is worth mentioning that this procedure assumes that BML specifications are in the trusted base. This need not be the case in the actual deployment environment. However, the mentioned above procedure can easily be supplemented by a step in which the BML specifications are generated or checked for consistency with actual security policy at the code consumer's side.

The Bill class with the certificate is available from the page `http://zls.mimuw.edu.pl/~alx/umbra/casestudy/`.

5 Conclusions

In the course of the case study we were able to generate BML specifications for a small application. The source code specifications were expressive enough to exclude the code warnings issued by ESC/Java2. Therefore, the BML tools are able to embed a reasonably expressive set of specifications into class files.

The process of the certificate generation requires a time consuming proof construction. The proof construction is the stage which requires a lot of additional work to be usable. In the course of the case study, we have made a small step

```
Ltac zetujh H :=
  match type of H with
  |  Is_true (le_bool _ _) =>
        unfold le_bool in H;
        match goal with
        | |- ?goal =>
                let g:=fresh "g" in
                set (g:=goal);
                revert H;
                arith; trivial; try (
                intro H;
                match type of H with
                | Is_true (Zle_bool ?x ?y) =>
                        let Hi := fresh "Hi" in
                        assert (Zle x y -> g) as Hi; [
                                clear H;
                                intro H;
                                subst g
                                |
                                apply Hi;
                                clear Hi;
                                unfold Zle;
                                unfold Zle_bool in H;
                                destruct (x ?= y)%Z; intros; (discriminate || tauto)
                        ]
                end)
        end
  |  Is_true (lt_bool _ _) =>
        unfold lt_bool in H;
        match goal with
        | |- ?goal =>
                let g:=fresh in
                set (g:=goal);
                revert H;
                arith; trivial; try (
                intro H;
                match type of H with
                | Is_true (Zlt_bool ?x ?y) =>
                        let Hi := fresh in
                        assert (Zlt x y -> g) as Hi; [
                                clear H;
                                intro H;
                                subst g
                                |
                                apply Hi;
                                clear Hi;
                                unfold Zlt;
                                unfold Zlt_bool in H;
                                destruct (x ?= y)%Z; simpl in H; tauto
                        ]
                end)
        end
   end.
```

Figure6. The tactic zetujh.

to make the process easier by constructing a tactic which transforms Bicolano formulae to arithmetical formulae.

A big obstacle in the course of the proof construction is the problem that the formulae the user sees are not comprehensive. In addition, the unfolding of many definitions causes a huge blow up of the formulae. This calls for a development of a set of intelligent tactics which facilitate a reasonable management of the proof development.

Moreover, the verification conditions that are generated are not always correct. A considerable review of the process of the verification condition generation is necessary. Although, one must say that the verification condition generation is a difficult task and it is not unusal to see some small design and development flaws in software of this size.

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. 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.
4. 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.
5. J. Chrząszcz, M. Huisman, and A. Schubert. BML and related tools. In *Formal Methods for Components and Objects*, Lecture Notes in Computer Science. Springer-Verlag, 2009. To appear.
6. J. Chrząszcz, M. Huisman, A. Schubert, J. Kiniry, M. Pavlova, and E. Poll. *BML Reference Manual*, December 2008. In Progress. Available from `http://bml.mimuw.edu.pl`.
7. J. Fulara, K. Jakubczyk, and A. Schubert. Supplementing java bytecode with specifications. In T. Hruška, L. Madeyski, and M. Ochodek, editors, *Software Engineering Techniques in Progress*, pages 215–228. Oficyna Wydawnicza Politechniki Wroclawskiej, 2008.
8. B. Grégoire and J.L. Sacchini. Combining a verification condition generator for a bytecode language with static analyses. In *Trustworthy Global Computing: Revised Selected Papers from the Third Symposium TGC 2007*, number 4912 in Lecture Notes in Computer Science, pages 23–40. Springer-Verlag, 2008.
9. 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.
10. 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).

11. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, July 2005. In Progress. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.
12. MOBIUS Consortium. Deliverable 4.2: Certificates, 2007. Available online from `http://mobius.inria.fr`.
13. MOBIUS Consortium. Deliverable 4.3: Intermediate report on proof-transforming compiler, 2007. Available online from `http://mobius.inria.fr`.
14. MOBIUS Consortium. Deliverable 5.1: Selection of case studies, 2007. Available online from `http://mobius.inria.fr`.