Verifying JML specifications with model fields

Cees-Bart Breunesse[⋆] and Erik Poll

Department of Computer Science, University of Nijmegen

Abstract. The specification language JML (Java Modeling Language) provides *model fields* as a means to abstract away from implementation details in specifications. This paper investigates how specifications with model fields can be translated to proof obligations. In order to do this, the semantics of model fields has to be made precise.

1 Introduction

The Java specification language JML (Java Modeling Language) [LBR01] provides *model fields* to allow specifications to abstract away from implementation details [CLSE03]. Model fields in JML play the role that abstract values play for traditional abstract data types. The idea is that clients of a class can use an abstract specification in terms of model fields, and that only when implementing the class we have to know how these abstract model fields are related to the actual concrete representation.

The ability to handle model fields is vital if our efforts at program verification with the LOOP tool [BJ01], or any other verification technology, are to scale up to bigger programs. This paper investigates how specifications with model fields can be translated to proof obligations as Hoare triples, or rather n-triples in the style of [JP01]. We only look at this issue from the point of view of someone implementing a class whose specification relies on model fields. The translation of specifications into these proof obligations is being incorporated in the LOOP tool, but we describe the translation at a relatively high level of abstraction here, so that it should be useful for other groups working on verification tools. More generally, it can be seen as an attempt at precisely defining the semantics of JML. To better understand model fields, we investigate several syntactic (de)sugarings of JML specifications with model fields. This paper does not try to give a definite answer on how to handle model fields. It is an attempt to clarify some issues, and to show possible solutions. Further investigations and case-studies must show which approach must be used in what setting.

The outline of the rest of the paper is as follows. Section 2 gives a brief introduction to the use of model fields in JML specifications. There are two ways in which an implementation can declare its concrete representation of a model field in JML. The first and simpler way is to give an explicit definition of this representation, by means of a *representation function*, using the operator `<-`. This is discussed in Section 3. The second, more general, way is to give a *representation relation* between the model field and its representation, using the keyword `\such_that`. This is discussed in Section 4. Dealing with these `\such_that` clauses poses the main challenge.

```
interface MyIntInterface {
    /*@ model int myint;
        invariant 0 <= myint && myint < 512 @*/

    /*@ requires   true;
        assignable myint;
        ensures    myint == (\old(myint) + 1) % 512; @*/
    void succ();
}
```

Fig. 1. Java/JML Interface: `MyIntInterface`

2 JML specifications with model fields

Our (extremely simple) running example of a JML specification with a model field is given in Fig. 1. It declares a model field `myint` that represents an integer value between 0 and 512. Such a model field is an imaginary field that can only be used for specification purposes. For instance, in Fig. 1, the model field `myint` is used in the specification of the method `succ`. This specification consists of several clauses:

– the `requires` clause specifies the pre-condition,
– the `assignable` clause lists fields or model fields which may be altered (in the case of a model field we means its dependees),
– the `ensures` clause specifies the post-condition that must be satisfied when `succ` terminates normally,
– the `invariant` clause specifies a class invariant, which is implicitly included in all pre- and postconditions.

For a client – i.e. someone using objects of type `MyIntInterface` – dealing with such a model field is quite different than for someone who is implementing a class that implements `MyIntInterface`:

– Essentially, clients can treat a model field as a Java field, and forget about the fact that it's not a real field, but that there is some representation hiding behind a model field. Unfortunately, this is only true if the representation is properly encapsulated, and if there are no unexpected ways in which one might alter the representation of the model field `myint`. Ensuring such proper encapsulation is an important and active research topic. A solution has been proposed for JML, using so-called universes, in [MPHL03]. We will completely ignore this issue in this paper, and we can, because we only look at model fields from the point of view of someone implementing a class that has model fields.
– Implementors will have to specify how the abstract model field `myint` is represented in the concrete implementation. In JML, this is done with so-called `represents` and `depends` clauses. These determine what concrete proof obligations follow from the abstract specification. This is the topic of this paper.

There are two kinds of represents clause:

– The first (using the `<-` notation) gives an explicit definition of the model field in terms of the real fields, by means of a representation *function*. For an example, see Fig. 2. This example, and the way to handle it, is discussed in Section 3. Dealing with such a represents clause is easy: we can basically treat the model field as a macro that is expanded to get the concrete proof obligations.

[⋆] Supported by the EU IST project VerifiCard (IST-2000-26238)

```
class MyIntImpl1 implements MyIntInterface {
    /*@ represents myint <- (f ? 256 : 0) + (b & 0xFF);
        depends    myint <- b,f; @*/

    byte b;
    boolean f;

    void succ() {
        f = (b == -2) ^ f;
        b = (byte)(b + 1);
    }
}
```

Fig. 2. An implementation using `<-`: `MyIntImpl1`

- The second (using the `\such_that` notation) more general kind only specifies a representation *relation* between the model field and the real fields. For an example, see Fig. 3. Dealing with these representation relations is more complicated, and is the main topic of this paper. This is discussed in Section 4.

3 Model fields with representation functions (using <-)

In JML an explicit definition of the model field in terms of the real fields can be given using a represents clause of the following form

```
  //@ represents myint <- ...
```

Here the expression to the right of the `<-`, in this case an integer expression, gives the *representation function* for the model field.

For example, in Fig. 2 a boolean and a byte are used to implement `myint`. Together, a boolean and a byte can represent a 9 bit numerical value, i.e. an integer between 0 and 512. The boolean `f` in Fig. 2 serves as the most significant bit. The represents clause in Fig. 2 gives an explicit definition of the model field `myint` in terms of the concrete representation in the field `b` and `f`. [1]

3.1 Syntactic desugaring: model fields as macros

To verify that the concrete implementation `MyIntImpl1` given in Fig. 2 meets the abstract specification `MyIntInterface` given in Fig. 1 is relatively straightforward. Basically, we can just treat the model field `myint` in the abstract specification `MyIntInterface` as a macro, and expand it to obtain the concrete proof obligation for `MyIntImpl1`. More specifically:

- we replace all occurrences of `myint` in `assignable` clauses by the `depends` clause specified in `MyIntImpl1`, i.e. by “`b,f`”.
- we replace all other occurrences of `myint` by the `represents` clause specified in `MyIntImpl1`, i.e. by “`(f ? 256 : 0) + (b & 0xFF)`”.

[1] This involves some Java technicalities: the Java expression `b & 0xFF` is the unsigned interpretation of the byte `b`, i.e. a value between 0 and 255.

```
class MyIntImpl2 implements MyIntInterface {
    /*@ represents myint \such_that (!f ==> myint == (b & 0xFF)) &&
                                   ( f ==> myint == (b & 0xFF) + 256);
        depends myint <- b,f; @*/

    byte b;
    boolean f;

    void succ() {
        ...
    }
}
```

Fig. 3. An implementation using `\such_that`: `MyIntImpl2`

Note that we then end up with a specification which no longer refers to a model field. We can deal with such specification already with the LOOP tool, so we can verify that `MyIntImpl1` meets this specification in the standard way.

This macro expansion in not done at the syntactical level by the LOOP tool, but is incorporated in the translation of JML to PVS. Basically, a model field is treated as a Java field, but instead of looking up its value from the heap, its value is defined using the representation function.

4 Model fields with representation relations (using `\such_that`)

Figure 3 shows class `MyIntImpl2`, which has the same implementation as class `MyIntImpl1`, but a different represents clause. The represents clause here uses a `\such_that` clause to specify a relation between the model field `x` and the real fields `f` and `b`. In this particular example, the relation is a total, one-to-one relation: for any given values of `b` and `f` there exists exactly one value of `myint` in the representation relation. In fact, the representation relation is simply the representation function specified in Figure 2, written in a different way.

However, in general a representation relation need not be a function. For example, the representation relation in Figure 4 is a partial relation, and it is many-to-one: given values of the concrete fields `j` and `k` there can be many values, or none, for the model field `x` that are in the representation relation.

Dealing with such representation relations, that are not total and one-to-one, is more difficult than dealing with functionals ones. We therefore distinguish three kinds of rep clauses:

1. "rep functions", written using the `<-` notation,
2. "functional rep relations", written using `\such_that`, but which specify a total, one-to-one function.
3. "proper rep relations", written using the `\such_that`, that are proper relations, i.e. not functions.

A reason for using the second kind of rep clause rather than the first can be convenience; some people may find it easier to read the rep clause in Fig. 3 than the one in Fig. 2.

```
class A {
    int j,k;
    //@ model int x;
    //@ represents x \such_that (j <= x && x <= k);

    /*@ requires true;
        ensures false; */
    void m() {
        j = 1;
        k = 0;
        // there is no x such that j <= x <= k !
    }

    void n() {
        j = 10;
        k = 12;
        //@ assert x == 10;
    }
}
```

Fig. 4. Example class `A`

For the remainder of this section, suppose we have a model field and represents clause as follows:

`//@ model` t `x;`
`//@ represents x \such_that` $R(\mathtt{x}, \overline{\mathtt{q}})$`;`

where
t is a type,
x is a model field,
$\overline{\mathtt{q}}$ is a vector of fields which make up the concrete representation of `x`,
R is the rep relation between x and $\overline{\mathtt{q}}$.

In class `A` of Fig. 4, the vector $\overline{\mathtt{q}}$ contains `j` and `k`.

4.1 Axiomatic approach

The approach used in [Lei00] is to consider the representation relation R as an axiom. This means one simply assumes that, at any point in the program, the model field `x` has some value satisfying the representation relation R. In other words, we treat $R(\mathtt{x}, \overline{\mathtt{q}})$ as an axiom that holds at every program point:

Approach 1 *At any point in the program we assume that the model field* `x` *has some value such that* $R(\mathtt{x}, \overline{\mathtt{q}})$ *holds.*

This approach is sound for rep functions and functional rep relations, but unsound for some proper rep relations. Consider for example the class `A` in Fig. 4. The, clearly incorrect, specification for method `m` can be proved correct using approach 1: if we assume that `(j <= x && x <= k)` holds at the end of the method `m()`, then we have a contradiction, and we can deduce the postcondition `false`.

Clearly, the problem with using R as an axiom arises in program states for which there is no x that satisfies R, i.e. if our representation relation is not total. Because [Lei00] considers only rep functions or functional rep relations, the axiomatic approach is sound in that setting.

4.2 Desugaring approach

Proper rep relations are tricky. There may be program points in which there is no value for the model field `x` that satisfies R. We therefore cannot assume that R holds everywhere, as the example above illustrates. When the model field `x` is mentioned in say a postcondition, we must make sure that there actually exists a value for x that satisfies R.

We can try to formalise this idea by desugaring every JML predicate (i.e. every JML boolean expression in a `requires`, `ensures`, `assert` or `invariant` clause) into an existential quantification which no longer has free occurrences of a model field `x`. Such syntactic desugarings are a convenient way to explain many details about the semantics of JML, e.g. see [RL00].

Approach 2 *Any JML predicate A(*`x`*) that possibly refers to the model field* `x` *is desugared to the JML predicate*

(`\exists` t `x;` R(`x`,$\overline{\texttt{q}}$) `&&` A(`x`))

where R is the representation relation.

Again, let us consider class `A` in Fig. 4 to observe whether approach 2 makes sense. We no longer have the problem with the specification for method `m` that occurred in the axiomatic approach, as clearly we cannot establish the postcondition

```
(\exists int x; j <= x && x <= k && false)
```

for method `m()`.

However, now we have a problem with method `n()`. Applying the desugaring given in approach 2 to the assert in method `n` results in the desugared assert:

```
//@ assert (\exists int x; j <= x && x <= k && x == 10);
```

This assertion is clearly true. However, the representation relation also allows 11 or 12 as possible values for the model field `x`, if `j == 10` and `k == 12`. So the assertion that `x == 10` does not necessarily hold at the end of `n()`, and should not be provable.

This illustrates that approach 2 is not what we want. The problem is caused by the fact that the representation relation is many-to-one. A more complicated desugaring, which we argue is sound, is to say that an assertion of the form

`//@ assert` A(`x`);

means that A holds for *every* possible value of `x` that satisfies the representation relation R, instead of just one, which is expressed approached 2. Moreover, there should be at *at least one* possible value for `x` satisfying R. This leads to our second desugaring approach:

Approach 3 *Any JML predicate A(*`x`*) that possibly refers to the model field* `x` *is desugared to the JML predicate*

```
(\forall t x; R(x,q̄) ==> A(x))  &&  (\exists t x; R(x,q̄))
```

where R is the representation relation.

Taking approach 3 the assert in method `n` is desugared to

```
//@ assert (\forall int x; (j <= x && x <= k) => x == 10) &&
//@        (\exists int x; (j <= x && x <= k));
```

for which the first part of the conjunction resolves to false, which is what we would expect. The strongest condition that we would expect to hold after execution of `n` is

```
//@ assert x == 10 || x == 11 || x == 12;
```

which covers the full range of possible values of `x`. Using approach 3, this is indeed the strongest postcondition that can be established.

Although desugaring approach 3 is what we need to tackle representation relations, the existential quantifications it involves can be quite a burden during proofs: during the verification of a method, at many times will witnesses for these existential quantifications have to be constructed by hand. The existential quantifier pops up every time a JML predicate referencing `x` is encountered. For fully annotated programs, these occurrences can be numerous.

To avoid giving a witness every time a model field is encountered, the person writing the specs might check whether the represents clause is a functional represents relation, which can directly be rewritten to a rep function using `<=`. If we are actually confronted with a proper represents relation, there is another way to avoid the problem with existential quantifiers, which we discuss in the next section.

4.3 Model fields as methods

An alternative desugaring of JML specifications that use model fields to JML specifications that does not use a model field, is to replace a model field by a method whose postcondition corresponds to the representation relation. (This method should maybe be a specification-only method, or, in JML-speak, a model method, but we ignore that issue here.)

For example, the model field `myint` used in Figure 3 could be desugared as shown in Figure 4.3. Note that the postcondition of the method `myint()` in Figure 4.3 is exactly the representation relation for the model field `myint` given in Figure 3.

The need for the keywords `pure` and `helper` is rather technical, and probably best ignored by people not familiar with or interested in the obscure details of JML[2]. It is of course simple to give an implementation of `myint()`, but we do not really have to do this, if we are satisfied with reasoning about `myint()` on the basis of its specification.

[2] The keyword `pure` means that the method `myint()` does not have any side-effects; this is required in order to be allowed to use the method in specifications, e.g. the postcondition of `succ()` and the invariant. The keyword `helper` means that the invariant is *not* implicitly included in the pre- and postcondition of `myint()`.

```
class MyIntImpl2 implements MyIntInterface {

    byte b;
    boolean f;

    /*@ requires true;
         ensures  (!f ==> \result == (b & 0xFF)) &&
                  ( f ==> \result == (b & 0xFF) + 256); @*/
    /*@ pure helper @*/ int myint() {
        ...
    }

    //@ invariant 0 <= myint() && myint() < 512;

    /*@ requires   true;
        assignable b,f;
        ensures    myint() == (\old(myint()) + 1) % 512;
      @*/
    void succ() {
        ...
    }
}
```

Fig. 5. Replacing model field by a method

In general, replacing a model field by a method can be more complicated that in the example above. If we have a partial representation relation, as was the case in the class `A` in Fig. 4, things are a bit more complicated. A first attempt at replacing the model field `x` by a method for class `A` in Fig. 4 might be:

```
class A{
    int j,k;

    /*@ requires true;
        ensures  j <= \result && \result <= k @*/
    /*@ pure helper @*/ int x() {
      ...
     }

    /*@ requires true;
        ensures false; @*/
    void m() {
       j = 1;
       k = 0;
       // there is no x such that j <= x <= k !
    }
  ...
```

However, here we run into the same problem as for approach 2, namely that we can prove the correctness of the – clearly inconsistent – spec for `m()`, if we assume the specification of `myint()` is correct. Again, the problem is that in the post-state of `m()` there exists no value for `x()` satisfying its spec.

The solution to this is that we should specify a suitable precondition for `x()` which ensures that its postcondition is satisfiable, e.g.

```
/*@ requires j <= k;
    ensures  j <= \result && \result <= k @*/
/*@ pure helper @*/ int x()
```

This specification for `x()` is satisfiable, something which could be established by proving

```
(j <= k) ==> (\exists int x; j <= x <= k)
```

or by providing an implementation for `x()` and proving that it meets the specification.

Let us make the approach above a bit more precise:

Approach 4 *We introduce a method* `x()` *to replace a model field* `x`*, and*

- *replace every occurrence of the model field* `x` *in a JML predicate by a call to the method* `x()`*,*
- *replace the declaration of model field* `x` *by a pure helper method* `x()`*, with a suitably chosen precondition P, and as a postcondition the representation relation, with* `\result` *substituted for* `x`*, i.e.*

  ```
  /*@ requires P;
      ensures  R(\result,q̄) */
  private model helper int x()
  ```

  for which one proves

  ```
  P ==> (\exists t; R(x,q̄) )
  ```

  possibly by giving an implementation for `x()` *and proving that it meets its specification.*

The main advantage of using approach 4 instead of approach 3 is that, provided P can be kept simple, it is less work to prove P than to construct a witness `x` that satisfies $R(x, \overline{\mathtt{q}})$, at every program point where the model field is used in a JML predicate.

5 Conclusions and Future Work

In this paper we looked only at model fields from the perspective of someone implementing a class that has model fields, not from the perspective of a client using such a class. This implementors' perspective was definitely more complex than we first anticipated, due to the possibility of representation *relations*, which may be partial or many-to-one, rather than using a representation *functions*.

Handling representation functions is relatively straightforward, but representation relations, when they are not total functions in disguise, are a big burden when it comes to verifying programs, as illustrated in Section 4.2. The alternative approach of looking at model fields as methods, discussed in Section 4.3, can save a lot of effort in verifying programs, at the cost of some extra work in specifying, namely coming up with a suitable precondition for the method replacing the model field.

Note that the approach discussed in Section 4.3, to introduce methods for model fields, can also be useful when doing runtime assertion checking. The JML runtime assertion checker [Che03,CL02] can perform runtime checks for assertions that involve a model field, but only if a representation function is specified using `<-`, not if a `\such_that` clause is given. For a model field `x` with a very complicated, but functional, rep relation, it usually is possible to define a method `x()` that computes this function and then replace the original `\such_that` clause by "`arepresent x <- x();`". Doing this would allow more assertions to be checked at runtime.

So far, we have only investigated relatively simple examples using model fields. For more complicated examples, that make use of the JML model classes which represent mathematical concepts such as sets and relations, we would of course like to express the proof obligations not in terms of Java implementations of sets and relations, but directly in terms of the underlying mathematical notions. (In our case, because we use PVS to prove our proof obligations, we would like to use PVS sets and relations.) This is still left as future work.

Acknowledgements
We would like to thank the anonymous referees for their comments, which were a great help in improving the paper.

References

[BJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *TACAS'01*, volume 2031 of *LNCS*, pages 299–312. Springer, Berlin, 2001.

[Che03] Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language.* PhD thesis, Iowa State University, 2003. Available as Technical Report TR 03-09, Department of Computer Science, Iowa State University.

[CL02] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *the International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328. CSREA Press, June 2002.

[CLSE03] Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. Technical Report 03-10, Department of Computer Science, Iowa State University, April 2003. Available from archives.cs.iastate.edu.

[JP01] Bart Jacobs and Erik Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *LNCS*, pages 284–299. Springer, Berlin, 2001.

[LBR01] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06p, Iowa State University, Department of Computer Science, August 2001. See `www.jmlspecs.org`.

[Lei00] Greg Leino, K. Rustan M. and Nelson. Data abstraction and information hiding. Technical Report SRC 160, Compaq Systems Research Center, 2000.

[MPHL03] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, 2003.

[RL00] Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical Report 00-03a, Iowa State University, Department of Computer Science, July 2000.