Model Variables: Cleanly Supporting Abstraction in Design By Contract

Yoonsik Cheon, Gary T. Leavens,
Murali Sitaraman, and Stephen Edwards

TR #03-10
April 2003

Keywords: documentation, formal methods, model variables, programming by contract, runtime assertion checking, specification languages, tools, Java language, JML language.


Submitted for publication.

Copyright © 2003 by Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. All rights reserved.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA
Model Variables: Cleanly Supporting Abstraction in Design By Contract

Yoonsik Cheon, Gary T. Leavens
Dept. of Computer Science
Iowa State University
Ames, IA
{leavens,cheon}@cs.iastate.edu

Murali Sitaraman
Dept. of Computer Science
Clemson University
Clemson, SC
murali@cs.clemson.edu

Stephen Edwards
Dept. of Computer Science
Virginia Tech
Blacksburg, VA
edwards@cs.vt.edu

Abstract

In design by contract (DBC), assertions are typically written using program variables and query methods. This technique does not allow specifiers to write model-oriented specifications for interfaces in languages like Java. Moreover, the lack of separation between program code and assertions is confusing, because readers do not know what code is intended for use in the program and what code is only intended for specification purposes. This lack of separation also creates a potential runtime performance penalty, even when runtime assertion checks are disabled, due to both the increased memory footprint of the program and the execution of code maintaining that part of the program’s state intended for use in specifications. We present a new way of writing and checking DBC assertions without directly referring to concrete program states, using “model”, i.e., specification-only, variables and methods. The use of model variables and methods solves all of the problems mentioned above. In addition, these features allow one to write more easily assertions that are abstract, concise, and independent of representation details, and hence more readable and maintainable. We implemented these features in the runtime assertion checker for the Java Modeling Language (JML), but the approach could also be implemented in other DBC tools.

1 Introduction

It is well known that in design by contract (DBC) [28], one should not write contracts directly using private program fields, methods, and types, because doing so would couple the assertions to implementation details [29, 30, 31]. If this were done, and if these
public abstract class SortedIntListApprox {
    private /*@ spec_public @*/ int size = 0;

    //@ ensures size == \old(size + 1);
    public void add(int elem) { size++; /* ... */ }
}

Figure 1: The approximation approach. In the JML notation used in this and subsequent
examples, a method’s specification precedes its signature, and annotations are contained
in comments that start with at-signs (@). The annotation spec_public is necessary here,
because JML does not allow private fields to be used in public specifications. Making
a field spec_public says that the field is public for specification purposes. As in Eiffel,
\old(e) denotes the value of e in the pre-state.

program details were subsequently changed, then the assertions would have to be rewritten.
Furthermore, it would be difficult for clients to understand specifications if assertions were
written using such internal implementation details. The problem is that such assertions
would violate the principle of information hiding [36], thus they would cause problems for
maintenance.

To avoid such problems, Meyer and others [29, 30, 31] advocate writing assertions
more abstractly, using only public fields and methods. Therefore, to manipulate concrete
program states in assertions, programmers often have to introduce new public helper
fields, methods, or types whose sole purpose is to aid in writing assertions. Such helpers
are needed particularly for specifying the behaviors of mutation methods. There are three
conventional approaches for doing this in DBC [31]:

- **Approximation.** This approach approximates the behavior of a method by using
  helper variables. For example, in Figure 1 the add method of a sorted list type is
  specified using the approximation approach by introducing a helper field, size, that
  counts the number of elements in the list.

- **Query methods.** This approach uses query methods to help write assertions. A query
  method is a side-effect free method that observes the state of one or more objects.
  For example, Figure 2 shows how two query methods, contains and size, are used
  to specify the behavior of the method add.

- **Immutable types.** This approach uses immutable data to store the (abstract) state
  of an object; the behavior of a mutation method is then specified in terms of this
  state. Figure 3 shows how the immutable type JMLValueSequence can be used to
  specify the state of a sorted list object; the abstract state of an object is represented
  by JMLValueSequence as opposed to a concrete representation.

The approximation approach is a quick way to write assertions, but it often leads to
underspecification. In Figure 1, add is underspecified in that its specification does not
public abstract class SortedIntListQuery {
    /** Is the given element in this list? */
    public abstract /*@ pure @*/ boolean contains(int elem);

    /** Return the size of this list. */
    public abstract /*@ pure @*/ int size();

    //@ ensures contains(elem) && size() == \old(size() + 1);
    public void add(int elem) { /* ... */ }
}

Figure 2: The query method approach. Giving the JML annotation pure for a method means that method must have no side effects.

constrain the contents of the list in its post-state. For example, the specification does not say whether all the elements of the old list are still in the new list, or even whether the newly added element is there. It also does not say whether the elements are still sorted. Because of the underspecification in the approximation approach, the query method approach and the immutable type approach are better for applications of DBC such as unit testing [8] or safety-critical systems, where more complete specifications are necessary.\(^1\)

1.1 The Problem

All three approaches described above try to specify the behaviors of mutation methods abstractly by introducing, if necessary, helper fields, methods, or types. The approaches share similar shortcomings. The introduced helpers become regular program fields, methods, and types even though they are intended only for writing assertions. Such program fields, methods, and types clutter programs. This cluttering makes it difficult to read and understand programs because the purposes of fields, methods, and types become ambiguous. Are they only for representing and manipulating program states, only for specifying assertions, or both? There is no way to make this distinction in standard DBC notations.

This blurring of specifications and programs also incurs a runtime cost, even when runtime assertion checks are disabled, because helper fields are still maintained and manipulated. For example, it is easy to see in Figures 1 and 3 that the helper fields are manipulated in the bodies of methods such as add. However, even in Figure 2 the method add must update some storage to track the list’s size. These helpers may also require extra memory at runtime. Further, because there is no declaration of what public features are intended purely for specification purposes, these helpers are open to use in client code just like all other public fields, methods, and types. As a result, they are more difficult to

\(^1\)While the query method approach can be used to write complete specifications, one has to use some discipline, such as that described by Guttag and Horning [11], to make sure that all methods are completely specified in the sense that trivial implementations are not allowed. For example, in Figure 2, the formal specifications given allow the contains method to always return true.
import org.jmlspecs.models.*;
public abstract class SortedIntListImmutType {
    private JMLValueSequence theList = new JMLValueSequence();

    /** Return the elements of this list. */
    public /*@ pure @*/ JMLValueSequence listValue() {
        return theList;
    }

    @* public invariant listValue() != null
        @* 
        @* (\forall int i; 0 <= i && i < listValue().length();
    @* listValue().itemAt(i) instanceof JMLInteger)
    @* (\forall int i; 0 < i && i < listValue().length();
    @* ((JMLInteger)listValue().itemAt(i - 1))
    @* .compareTo(listValue().itemAt(i)) <= 0);
    @*

    @* ensures listValue().isInsertionInto(\old(listValue()),
    @* new JMLInteger(elem));
    @*
    public void add(int elem) {
        /* ... */
        theList = /* ... recompute theList’s value ... */;
    }
}

Figure 3: The immutable type approach. This example uses the type JMLValueSequence, whose objects are immutable. It also uses an invariant and quantifiers. Quantifiers in JML start with \forall or \exists and are followed by a declaration; after the declaration is an optional range predicate, which is followed by a body predicate. Some quantifiers with finite ranges, such as the ones in this example, can be executed by JML’s runtime assertion checker.
remove via conditional compilation or other approaches.

Why not just use comments or other informal conventions to indicate which fields, methods, and types are only for use in specifications? The problem is that then tools would not be able to distinguish which features are only for specification purposes. Thus, for example, the runtime costs associated with these additional specification-only features would still be present.

Another problem with the helper approach is that it is not possible to introduce new program fields into the “interfaces” found in languages like Java or C#. The problem is that in these languages interfaces cannot have constructors, method bodies, or mutable fields. Thus one has to write specifications for interfaces by first declaring any necessary helper methods, and then writing contracts for all methods in an algebraic style, by constraining each method in terms of other methods in the interface. In other words, in standard DBC tools one cannot write model-oriented specifications for interfaces. For example, consider converting Figure 3 from an abstract class to a Java interface. In an interface, the field theList could not be declared, and the methods given there could not have bodies. Thus, the model-oriented immutable type approach taken in Figure 3 would not work. Instead one would have to convert the specification to an algebraic style, using the query method approach. However, experience has shown that it is more difficult for most programmers to write specifications algebraically. Furthermore, because interfaces do not contain constructors, there is no way to describe the initial values of types in an interface. Thus, it is significantly more difficult for a programmer to specify an interface using standard DBC tools than it is to specify a concrete class. On the other hand, the ability to write specifications for interfaces is important in such languages. Interfaces provide a logical boundary between clients and implementors, and are useful for decoupling abstractions from implementations. Thus, interfaces are ideal places to attach contract specifications.

1.2 Outline
In the next section of this paper we describe our approach to solving the problems raised in this introduction. In Section 3 we describe the implementation of our approach. In Section 4 we describe related work. This is followed by a discussion and conclusions.

2 Our Approach
The approach we advocate is to extend the languages used in DBC to include explicit specification-only declarations. In this paper we limit ourselves to the simplest kinds of specification-only declarations—model variables and model methods. (The adjective “model” simply means that such declarations are specification-only declarations.) We also limit ourselves to languages like Java or C#, and thus the main kind of variable is a field; hence, the remainder of this paper discusses model fields. A model field is a specification-only field that describes the abstract state of some program fields [21, 22, 23]. Using such model fields, specifiers can write abstract assertions, by referring to model fields and manipulating their abstract values.
A model field can be accompanied by the specification of an abstraction function, which says how to map from values in program fields to the “abstract values” in model fields [13, 25]. An abstraction function allows tools to check assertions written with model fields at runtime. If convenient, abstraction functions can be specified using model methods.

Model fields and model methods are recognized as specification-only declarations by DBC tools; hence, they cause none of the above-mentioned problems associated with using only program fields and methods in specifications. In addition, assertions written with model fields are independent of representation details and thus can be more abstract. For this reason they can also be more concise. Thus, such assertions can be more readable and maintainable than low-level assertions. We will provide some evidence for these claims in the examples given in this paper, but the claims are also validated by our experiences in specifying larger and more realistic examples. We also believe that, because they are more abstract, such assertions also can be manipulated more easily during formal reasoning.

It is not necessary to abandon the traditional approaches of approximation, query methods, and immutable types to use model fields and model methods. Indeed, model fields can be used to elegantly implement the approximation approach, and they can be combined with the immutable type approach to yield all the benefits of that approach without the corresponding clutter and ambiguity. Furthermore, following the DBC paradigm, one can still allow program elements to be used in specifications. This is the idea behind the spec.public and pure modifiers in the Java Modeling Language (JML). These modifiers allow both private fields and query methods to be used in assertions. These annotations make it clear that such program elements are playing two roles, whereas model elements can only be used in assertions. On the other hand, private fields in JML cannot be used in public specifications; furthermore, program methods that are not pure cannot be used in assertions. Hence, JML allows one to clearly state the purpose(s) of a field or method.

We have implemented model fields and methods in JML’s runtime assertion checker [7, 5]. JML is a formal interface specification language for sequential Java; it has sophisticated features for writing abstract, precise, and complete behavioral descriptions of Java classes and interfaces [18, 19]. However, it can also be used as a simple DBC language. The JML runtime assertion checker generates Java bytecode from Java classes and interfaces with JML specifications. As is usual in DBC tools, unless an assertion is violated, assertion checking is transparent. Except for performance measures (time and space), the behavior of original program is unchanged.

Note that, while we use JML to explain our approach, this strategy could also be implemented in other interface specification languages or DBC tools. The contribution of this paper is a description of a language-independent idea and its implementation: namely, specification-only declarations.

As an example, the specification of a Java interface, SortedIntListType, is shown in Figure 4. This interface is written in JML following our approach. As an interface, it can be implemented by several different classes, for example by classes that use arrays or binary search trees for their data structures. This specification starts with an annotation that does a model import of the types in org.jmlspecs.models; these are immutable types, like JMLValueSequence, that JML provides for describing abstract values in the immutable types approach. Since this is a model import, and since it is in an annotation, the import is not part of the program as seen by a normal Java compiler. Like all “model” features in JML, the model import is only for purposes of specification.
The next annotation declares a model field, `theList`, which contains the abstract value of a sorted list. Model fields allow JML to support the immutable type approach even for interfaces. Again, the modifier `model` says that `theList` is not a regular program field and can only be used in specifications. As such, it cannot be used in the body of a normal Java method, such as `add`. The modifier `instance` in this declaration says that `theList` is thought of as a non-static field, i.e., as part of all instances of classes that implement this interface. The modifier `non_null` is a shorthand way of saying that `theList` is never null; it is equivalent to the invariant “`theList != null`”. The `initially` clause at the end of the model field declaration specifies constraints on the initial value of `theList` [32, 35]. The `initially` clause allows one to reason about `theList` by using data type induction [14], since it provides a base case for such an induction. By constraining the initial value for the model field in this way, the `initially` clause eliminates the need for a constructor that assigns the field a value, and thus supports reasoning about abstract field initialization in interfaces.

The model of a sorted integer list is a sequence of integer values, with the values in ascending order. The invariant specifies the details of this model. It says that, for all objects in the sequence, they are all instances of the type `JMLInteger` (and thus not null), and that the sequence is sorted in ascending order. JML enforces information hiding by ensuring that public invariants can only mention publicly visible names. (In JML, the modifier `instance` means that the invariant applies to all the instance methods.)

The specifications of the methods `size`, `get`, and `contains` show how this specification is model-oriented. These specifications all refer to the model field `theList`; in the query method approach they would have no postconditions. All of these methods are declared to be `pure`, which says that they cannot have side effects. This allows them to be used in specifications; in JML only pure methods are allowed to be called in assertions [20]. The specification of the `get` method demonstrates a precondition, which is given by a `requires` clause in JML.

As in the immutable type approach, the postcondition of the method `add` is written by referring to a helper field, but in this case it is a model field, `theList`. The postcondition does not mention that the resulting list is in sorted order, because that has already been stated in the invariant. In the specification of `add` there is also a redundant postcondition specified by the `ensures_redundantly` clause. This clause is used to highlight some properties that follow from the postcondition for the benefit of the specification’s reader [17]. In this case, the redundant postcondition highlights the specification that would have been given in the query method approach. (The JML checker also executes redundant postconditions as a way of debugging specifications.)

Another feature of our approach is that one explicitly specifies abstraction functions in JML. Figure 5 exemplifies this in the specification of the class `SortedIntList`, which implements the `SortedIntListType` interface in Figure 4. The implementation uses four private fields to form binary search trees. The `isEmpty` field tells whether the other fields are defined. As the invariant states, `isEmpty` is true if and only if the left and right subtrees are null. The `depends` clause says that whenever a method, such as `add`, specifies that `theList` can be modified, the fields that it depends on can also be modified [22]. Although `modifies` clauses are not discussed here, they allow one to write frame axioms that are needed in formal verification.

The abstraction function itself is specified in the `represents` clause of Figure 5. It says
//@ model import org.jmlspecs.models.*;
public interface SortedIntListType {
    //@ public model instance non_null JMLValueSequence theList
    //@ initially theList.isEmpty();
    //@ public instance invariant
    //@ (\forall int i; 0 <= i && i < theList.length();
    //@     theList.itemAt(i) instanceof JMLInteger)
    //@   \&\& (\forall int i; 0 < i && i < theList.length();
    //@     ((JMLInteger)theList.itemAt(i - 1))
    //@     .compareTo(theList.itemAt(i)) <= 0);
    //@
    //@ ensures \result == theList.length();
    //@ pure @*/
    int size();

    //@ requires 0 <= index && index < size();
    //@ ensures \result == ((JMLInteger)theList.itemAt(index)).intValue();
    //@ pure @*/
    int get(int index);

    //@ ensures \result == theList.has(new JMLInteger(elem));
    //@ pure @*/
    boolean contains(int elem);

    //@ ensures theList.isInsertionInto(\old(theList),
    //@ new JMLInteger(elem));
    //@ ensures_redundantly contains(elem)
    //@ \&\& size() == \old(size() + 1);
    //@
    void add(int elem);
}

Figure 4: An example interface specification in JML.
//@ model import org.jmlspecs.models.*;
public class SortedIntList implements SortedIntListType {

    private boolean isEmpty;
    private int val;
    private SortedIntList left, right;

    //@ private invariant isEmpty == (left == null && right == null);

    //@ private depends theList <- isEmpty, val, left, right,
    @ left.theList, right.theList;
    @ private represents theList <- abstractValue();
    @ private model pure JMLValueSequence abstractValue() {
        @ JMLValueSequence ret = new JMLValueSequence();
        @ if (!isEmpty) {
            @ ret = left.abstractValue()
            @ .insertBack(new JMLInteger(val))
            @ .concat(right.abstractValue());
            @ }
        @ return ret;
    @ }
    @

    //@ ensures theList != null && theList.isEmpty();
    public SortedIntList() { isEmpty = true; }

    //@ ... *
    public void add(int elem) {
        if (isEmpty) {
            isEmpty = false; val = elem;
            left = new SortedIntList(); right = new SortedIntList();
        } else {
            if (elem <= val) { left.add(elem); }
            else { right.add(elem); }
        }
    }
}

Figure 5: An example implementation with JML annotated Java code.
that the value of the model field theList is determined by the expression abstractValue(),
which appears on the right hand side of the represents clause. The method abstractValue
is a model method, which is specified immediately below the represents clause. As a model
method, it can only be called from within specifications. The code for this model method
can access types that were imported by model imports. Otherwise, its body is a regu-
lar Java method body that uses the types JMLValueSequence and JMLInteger. JML's
runtime assertion checker can execute model methods that have bodies. Thus, given an
object of type SortedIntList, the runtime assertion checker can compute the value of the
model field theList using this abstraction function. This, in turn, enables the execution
of the other assertions in SortedIntList.

Using model fields (and abstraction functions) to specify mutation methods fixes the
problems incurred by using public features for this purpose. Because a model field is not a
program field, it does not clutter up program code, or confuse a reader about its intended
use. The declaration itself clearly reveals the intended role of the field. A program field
is used to represent concrete implementation values; a model field denotes abstract values
and is used for writing assertions. Model fields cause no runtime penalty, in either time or
space, when runtime assertion checks are turned off; with checks disabled, a model field is
not compiled into bytecode by the JML compiler. The use of model fields also facilitates
program changes and refactoring. A change in a type’s private fields does not require
a change in its client-visible specification, because specifications are written in terms of
model fields (and public fields). The specification change is localized to its represents
claususes (i.e., the abstraction function). In JML, represents clauses must be private if they
refer to private fields (see Figure 5) [33], so such a change does not invalidate any prior
reasoning about the specification done by clients. Finally, using model fields allows us to
place behavioral specifications within Java interfaces in a model-oriented style.

3 Implementation of Model Features

In this section we discuss how the JML compiler [5] implements model fields (and their
represents clauses) and model methods.

The JML compiler compiles model methods (e.g., abstractValue in our example) into
Java method bytecode. This allows model methods to be used in checking assertions. The
main complication has to do with the JML compiler’s support for separate compilation; to
make this work the compiler currently uses Java’s reflection facility to invoke separately
compiled model methods dynamically [5, Chapter 7]. Otherwise the treatment of model
methods is relatively straightforward, and thus in the rest of this section we concentrate
on support for model fields.

The value of a model field is obtained by applying its abstraction function. Thus, the
JML runtime assertion checker can execute references to model fields if they are accompa-
nied by an abstraction function. As shown in the previous section, an abstraction function
is specified with a represents clause. In JML, a model field, x, is runtime checkable for a
type T if and only if there is a represents clause that functionally provides the value of x for
any object of type T. Note that, in the above definition, x does not have to be declared in
the same module as T; for example, the model variable theList is declared in the interface
shown in Figure 4, but the represents clause appears in the corresponding concrete class
SortedIntList (see Figure 5). Normally, one would expect this pattern in all interfaces, where model fields describing interface-level behavior are declared in the interface, and each implementing class provides its own unique abstraction function mapping concrete fields into the abstract model.

The JML compiler generates a model field access method for each declared model field. Each reference to a model field in an assertion is then translated into a call to the model field’s access method. If a model field has no functional represents clause, then the compiler generates a default implementation for the corresponding access method that throws a predefined exception to indicate an occurrence of a non-executable specification construct. Such an exception is interpreted as either true or false, depending on the context, by the smallest, enclosing boolean expression [5, Chapter 3] [7, 20]. An access method generated from a functional represents clause calculates the abstract value corresponding to the current program state. For example, the compiler generates the following access method for the represents clause of the model field theList shown in Figure 5.

```java
/** Generated by JML to access the model field theList. */
public org.jmlspecs.models.JMLValueSequence
   model$theList$SortedIntListType() {
   org.jmlspecs.models.JMLValueSequence rac$v0 = null;
   rac$v0 = this.abstractValue();
   return rac$v0;
}
```

The model field access method evaluates the expression part of the represents clause and returns it as the result. Note in the code above the call to the model method abstractValue, which is compiled into a normal method call. Thus, abstraction functions written in terms of model methods can be evaluated by the runtime assertion checker. In addition, an abstraction function for a model field can be defined in terms of other model fields. This allows one to write executable specifications in layers of abstractions.

3.1 Interface Model Fields

A model field declared in an interface (as in Figure 4) is called an interface model field. For an assertion that uses such a field to be checkable, the abstraction function for that field eventually must be defined in terms of program fields by a class that implements the interface.

Checking assertions becomes complicated for interfaces. The JML compiler turns top-level assertions such as pre- and postconditions into separate assertion checking pre- and postcondition methods. However, the compiler cannot add assertion methods directly to an interface because in Java all interface methods must be abstract. To work around this, the compiler generates a separate assertion checking class, called a surrogate class, as a static inner class of the interface. The surrogate class is responsible for checking all assertions specified in the interface. In particular, it hosts all assertion checking methods and code for handling model fields and represents clauses. Thus, in addition to the distribution of assertion checking responsibility, an object’s specification state is also distributed over the object itself and all the surrogate objects of the interfaces that the object’s class implements.
3.2 Inheritance

Model instance fields and instance represents clauses are inherited by subclasses and subinterfaces \[17, 19\]. Several things complicate the inheritance of model fields and represents clauses. The main complication is that an applicable abstraction function may be specified by a subtype (a subclass, a subinterface, or an implementing class). For example, an abstraction function for the interface model field \texttt{theList} in Figure 4 is specified by an implementing class \texttt{SortedIntList} in Figure 5. Under separate compilation, the applicable abstraction function cannot be determined statically at compile-time; e.g., when compiling the interface \texttt{SortedIntListType}, the JML compiler does not know whether an implementing class would provide an abstraction function for the model field \texttt{theList}. Another complication is that JML supports multiple inheritance of specifications; a class inherits specifications from its superclasses and the interfaces that it implements, and an interface inherits specifications from all its superinterfaces. Thus, the JML compiler cannot depend completely on Java’s inheritance mechanism to support JML’s inheritance, as Java allows only single inheritance. In sum, a key challenge is for a supertype to prepare, in the presence of multiple inheritance, to use abstraction functions that may be provided by its subtypes.

Our approach is to introduce two additional model field access methods, called a \textit{delegation method} and a \textit{dispatch method} respectively (see Figure 6) \[5, \text{Chapter 7}\]. A delegation method is generated from an interface model field and forwards all incoming model field access calls to the object being checked. An interface surrogate object has a reference to the object being checked, which knows how to interpret the model field. A dispatch method is generated by a top-level implementing class for each interface model and invokes an appropriate model field access method that is generated from an interface’s represents clause and added to the surrogate class of that interface. As the dispatch method is generated by an implementing class, it is possible to statically determine whether an applicable interface represents clause exists. If there is no such interface represents clause, then the dispatch method throws a pre-defined exception to indicate non-executability. The dispatch method may be overridden by a subclass, i.e., if the subclass specifies a represents
clause.

4 Related Work

The use of abstract values in specification is not new, as shown by the immutable type approach described in the introduction. Furthermore the use of abstract values has been a feature of several formal specification languages.

Anna [26], an annotation language for Ada, also makes a distinction between program code and specification-only declarations with its “virtual text” mechanism. Besides variables, one can also declare procedures and types in virtual text; JML allows similar declarations, but does not currently compile types declared in this way for runtime assertion checking. As in JML, variables declared as virtual text can be used only in annotations, e.g., in assertions. However, the main difference is that, in Anna, variables declared in virtual text are explicitly assigned to by statements in virtual text. JML has a similar feature, called “ghost variables” and “set statements” that appear only in JML annotations that manipulate them. However, Anna does not have the equivalent of JML’s model variables, which are specification-only variables whose values are implicitly given by using user-declared abstraction functions. Anna also does not have the equivalent of JML’s spec_public modifier.

In Larch family of interface specification languages [6, 12, 16, 39], specifications are written solely in terms of abstract values, specified algebraically, and no explicit mapping is specified between abstract and representation values. What distinguishes our approach from work on Larch, is the use of specification-purpose declarations to hold abstract values that can be used in runtime assertion checking. This is primarily due to the explicit specification of abstraction functions in JML.

The RESOLVE family [9, 35] is similar to the Larch family in that it also is a family of interface specification languages. Unlike most Larch-style languages, however, RESOLVE features a way to specify abstraction functions. In particular, the work by Edwards et al. [10] shares our objective of employing abstraction in assertion checking, using model-based RESOLVE specifications [38] and C++ implementations. Their assertion checking approach uses abstraction functions (and in general, abstraction relations) that are given in implementations, as we do in our approach. However, the focus of this previous work has been on automatically generating wrappers to check assertions. The code for checking the assertions must be supplied by programmers and the underlying tool merely incorporates them at appropriate places. We present a more comprehensive approach and a Java-based tool in this paper whereby much of the assertion checking is automated. Combining the idea of wrappers with the JML checker is among our plans for further research.

The Abstract State Machine Language (AsmL) [1] also has executable specifications written using abstract variables. However, in AsmL, the abstract variables are not usually directly associated with program variables through abstraction functions (although this can be done). Instead, in AsmL, one mostly writes specifications as abstract programs that directly manipulate abstract variables. Instead of constraining program code, as is done in DBC, AsmL specifications are usually kept separate from program code. Specifications are normally executed separately from programs when doing runtime assertion checking, and

\[\text{However, Larch/C++ also has some of the features found in JML [17].}\]
violations are detected by comparing the two outputs. This separation of specifications from programs is thus a paradigm shift from DBC.

In Eiffel [29, 30], there are no specification-only declarations. Thus Eiffel suffers from the clutter problems described in the introduction section. Because Eiffel has no specification-only declarations, it also has no way to directly specify abstract values. Eiffel's design has been very influential in the DBC community; for example, aside from JML, other design by contract tools for Java, such as iContract [15] and Jass [2], have no explicit support for abstract values. As a result, Eiffel-based approaches make it more difficult to write abstract specifications.

For specification-only fields, JML builds on the work of Leino and his co-authors [21, 22, 23, 24]. Leino's work clarified the semantics of specification-only variables, particularly with respect to frame axioms [3] (i.e., modifies clauses). Leino introduced specification-only variables to solve the problem of information hiding while still being able to specify and verify programs in a model-oriented style. However, Leino and his co-authors did not apply these ideas to DBC tools; that is, they did not use these ideas in runtime assertion checking.

The main focus of the work of Leino and his co-authors is how to connect specification-only variables to the program's concrete variables. They use two constructs: “depends” and “represents” clauses [22, 24]. When a frame axiom names specification-only variables, a depends clause says which concrete variables can be modified; all locations that the specification-only variable depends on may be modified. A represents clause gives an abstraction function that defines the value of a specification-only variable in terms of the concrete variables on which it depends. This semantics has been elaborated in a Java context by Müller (with Poetzsch-Heffter and Leavens) [33, 34]. The importance of this for the present paper is that JML’s semantics builds on this work, which allows JML to be used not only as a DBC tool, but also as a formal specification and verification language [4, 20, 27, 37].

5 Discussion

In addition to model fields and model methods, JML also has model classes and model interfaces. These are specification-only classes and interfaces. However, model classes and interfaces are not yet supported by the JML compiler, although we hope to implement these in future work. At present, the JML compiler treats any reference to a model type as non-executable.

In general, JML takes the approach of allowing a specifier to use an unrestricted syntax with some non-executable features, and its runtime assertion checker only executes a subset of the language, albeit a large subset. An example of such a non-executable construct is a reference to a model field for which there is no (functional) represents clause. Non-executable expressions are handled in a systematic way by JML’s contextual interpretation of expressions [5, Chapter 3] [7], which also deals with undefinedness resulting from exceptions that may occur during expression evaluation [20]. The contextual interpretation ensures that such non-executable constructs do not lead to spurious assertion violations.

Our approach to evaluating assertions written with model fields has a shortcoming in
terms of performance. Each reference to a model field results in the construction of a new abstract value for the model field. For model fields of class types, this may incur a runtime performance penalty in terms of memory and speed. For example, in our sorted list example (see Figure 5 in Section 2), each reference to the model field \texttt{theList} creates a new \texttt{JMLValueSequence} from the program fields \texttt{isEmpty}, \texttt{val}, \texttt{left}, and \texttt{right}. Frequent construction of abstract values from concrete representations may adversely affect the speed of runtime assertion checking, especially for large, container-style data structures.

Extra costs associated with recomputing abstract values could be reduced in principle by caching. These costs exists primarily as a result of choosing the simplest generation strategy for the current implementation. Model field values need only be generated once within a given assertion (or sequence of assertions checked adjacently), since, in JML, evaluating assertions cannot change the state of the underlying object [20] (and hence cannot change the state stored in model variables, once these values have been computed). As part of future work, we are exploring caching strategies that allow the runtime checker to avoid redundant construction of abstract values.

6 Conclusions

The use of abstract, specification-only features, such as JML's model fields and model methods, is an evolutionary advance over DBC approaches that use query methods and immutable types. In the query method approach, specifiers add side-effect free methods to be used in specifications, but there is no way to tell which methods are intended only for use in specifications. Thus programs can use such added query methods, and they may take up memory at runtime. Similarly, in the immutable type approach, the program maintains state for specification purposes, but there is no easy way to stop maintaining this state when assertion checking is disabled. These problems are solved in JML by distinguishing model fields and model methods from program fields and program methods.

A similar distinction between model and program entities could be made in other DBC notations and tools. What is needed are declaration forms that mark those features that can only be used in specifications. Tools can recognize these and thus keep the two kinds of declarations separate. Other languages could also adopt notations (like JML's \texttt{spec.public}) that mark declarations for use in both specification contexts and program contexts.

Having this distinction does not mean that one gives up on traditional DBC. Indeed, as we have shown, the ideas described in this paper support and extend DBC. Everything that can be specified in a language like Eiffel, including the approximation, query method, and immutable type approaches, can be written in JML. In return for marking the purposes of various declarations, the JML compiler can produce more efficient code when assertion checks are turned off, and there is less confusion between specification-only code and code intended for use by clients. Furthermore, it becomes easier to write more abstract specifications, since in production code there is no runtime cost associated with using the immutable type approach. Finally, one can use model-oriented specifications for interfaces, which is not possible with other approaches that lack model fields and methods.

The JML compiler implementing our approach is available with other JML tools [4] from the JML home page at \url{http://www.jmlspecs.org}.
Acknowledgments

The work of both Cheon and Leavens was supported in part by the US National Science Foundation under grants CCR-0097907. In addition, the work of all authors was supported in part by US NSF grant CCR-0113181. Thanks to David Cok and Curtis Clifton for comments on earlier drafts of this paper.

References


