Adapting JML to generic types and Java 1.6

David R. Cok
Eastman Kodak Company Research Laboratory
1999 Lake Avenue
Rochester, NY 14650 USA
david.cok@kodak.com

ABSTRACT

Despite the current effort to implement the Java Modeling Language for Java 1.5, and in particular for generic types, there has been no analysis of the effect of such a transition on JML itself, nor of what language changes should be implemented to take best advantage of the features of current Java. This paper analyzes the interactions between JML and the new features of Java 1.5 and 1.6, and it proposes appropriate changes to JML. Many implementation details for JML tools can be handled by choosing an existing Java 1.5+ compiler as a base; however, there are adjustments to the typing of JML expressions that would be appropriate, and there are issues needing careful attention arising from refinements, autoboxing, lock ordering operations, specification of enhanced for loops, type erasure, and the runtime execution of specifications involving type parameters. The features are implemented experimentally in OpenJML, an OpenJDK-based implementation of JML.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs

Keywords

specification, JML, Java, generics

1. INTRODUCTION

The Java Modeling Language (JML) [7, 8] has been a successful, widely used specification language for Java programs. Many tools [2] have been generated and many research groups[1] have used it as a basis for research and experimentation. The JML2 tool suite was written and maintained for Java 1.4. Java 1.5, introduced in 2004, brought significant changes to Java, but the work to evolve JML tools to work with Java 1.5 stalled for lack of resources. That omission is only now being addressed by the building of new versions of JML on top of Eclipse [4] and OpenJDK [12], although those projects are not yet ready with released tools.

[1]Publications from a variety of groups using JML are given at `http://www.jmlspecs.org`.

Not explicitly addressed in those tool-building activities, however, are the changes to JML itself that are needed to keep it aligned with the generic type and other capabilities of Java 1.5 and 1.6. This paper addresses that deficiency.

Versions of Java beginning with Java 1.5 introduced many new features into the language; we will consider the following here:

- generic types and methods
- enhanced for statement
- autoboxing and unboxing
- annotations
- varargs
- static import
- enum types
- `java.lang.SuppressWarnings`
- the Java compiler, AST and annotation processing APIs

This paper presents an assessment of JML with respect to these features of Java, recommending alterations and extensions to JML where that would be beneficial for the language. We also note areas where similar activities are underway in other groups and highlight aspects that would benefit from cooperation. The issues discussed are relevant to interface specification languages for other programming languages with generic types, but we consider them only in the concrete context of Java and JML.

2. ENHANCEMENTS TO JML

In the following subsections we discuss the impact on and propose alterations to JML to accommodate the evolution of Java. A JML tool that is built upon a Java compiler will be able to accommodate most language changes without difficulty. The implementation effort is reduced further if the Java compiler infrastructure is used for parsing and type-checking JML expressions as well. Nevertheless, there are several issues that must be attended to.

2.1 Generic methods and types

The most significant language addition in Java 1.5 was generic types and methods. Type names in a JML specification may now be parameterized with concrete types or type variables, and model methods may now have type parameters. This affects the JML grammar in ways corresponding to the changes in the Java grammar: alterations are needed in the definitions of *reference-type*, *class-definition*, *interface-definition*, *class-extends-clause*, *name-weakly-list*, *method-decl*, and *primary-expr* (for method calls).

For tools built on existing compilers (and maintained by others) the infrastructure needed to support generic types in JML comes with the Java compiler. Parameterized model methods and parameterized model types by themselves pose no new difficulties.

2.2 Type parameters

However, the body of a class or method now has some additional type names in scope—those of the type parameters. In the case of methods, the scope of these type parameters extends over the method specifications. Thus the name lookup procedure for method and type specifications must include the type parameters. Model methods also may now be generic, so signature matching and type resolution for model methods must be enhanced in the same way as it is for ordinary Java methods.

Type names are not common in method and type specifications since those are mostly expressions. However, they can occur in type literals, **`instanceof`** and cast expressions, in declarations in **`forall`** and **`old`** clauses in method specifications, and in quantified and set comprehension expressions.

Within type and method specifications, the properties of a particular type parameter must be ascertained. Java properties such as the methods defined for the type are determined just as in the program code. JML also needs to determine which specifications apply to a method of a type parameter. Those are defined as the union of the specifications from the types making up the upper bounds of the type parameter.

2.3 Refinement

The first significant departure from what a Java compiler provides is in resolving refinement. JML allows the specifications for a class to be in a separate file from the source itself. In fact, the source may not even be present, since we may be specifying a binary file. In addition, there may be more than one refinement file for a given class. Refinement resolution consists of attaching each specification to the correct Java construct. So, declarations in specification files must be matched to Java declarations or entities in binary files.

The first task is to match class declarations. This is straightforward since there is at most one class with a given fully qualified name. There remains to be sure that the type parameters of the class in the specification file match those defined in Java, both in number and in any bounds restrictions. In general the parameter names may not be the same among the various class declarations, so a mapping of names may be required. It would be prudent to require that type parameter names be consistent among Java source and any JML specification files, simply to aid comprehension.

The second task is to match the declarations within a class. Fields have unique names, so they can be matched by name and their types checked for equality, taking into account type parameters. Method names are not unique and must be matched by signature, taking into account type parameters of the class and any type parameters declared for the method itself. (Thus refinement resolution must occur after the type name portion of a compiler's symbol table is built; it cannot be a simple textual match.) This sort of generic signature mapping is not needed in Java and must be implemented by the JML tool itself. It is made simpler, and does not restrict JML expressibility, if corresponding type variables have the same names.

Proposal: *The names of type parameters of a parameterized class, interface or method must be the same in all JML specification files for that program construct and must match those used in the Java source file, if that is available. (This not strictly necessary, but is a convenience for implementation.)*

2.4 Type specifications

The type variables of a generic class are in scope in type specification clauses such as invariants and constraints. It is conceivable that, like generic methods, one may want to parameterize axioms, invariants, constraints or initially clauses as well. The syntax would be straightforward and would take this form:

`axiom <T> (` *predicate* **`);`**

Of the possibilities, parameterized axioms would appear to be the most useful. Here is an example:

`axiom <T> ( JMLObjectSet.<T>EMPTY().size() == 0 );`

Experimentation may uncover good use cases that cannot be naturally expressed without parameterization. The implications for the encoding of specifications into the logic of target theorem provers are unknown, however, so it is best to leave this potential feature as experimental.

Proposal: *Reserve the following syntax for parameterized axioms, on an experimental basis, pending good use cases and practical experience:*

axiom-clause := `axiom` (*predicate* |
  < *TypeParameter* [, *TypeParameter*]... > (*predicate*)) ;

where TypeParameter is a nonterminal defined in the Java Language Specification.

2.5 Method specifications

Method specifications are in the scope of any class or method type parameters, so type resolution needs to be applied just as it would be for the formal parameters or in body of the method. Although there are declarations in **`forall`** and **`old`** clauses, no additional parameterization of the specification appears to be useful.

There is one syntactic location where the JML grammar needs embellishment: the **`callable`** clause. This clause tells what methods may be called by the method at hand. Methods are denoted by their signatures, if necessary. Some disambiguation by type parameter may also be necessary. The current grammar for a **`callable`** clause contains a list of method names that has the following grammar, in part:

method-name-list := *method-name* [, *method-name*]...
method-name := *method-ref* [(*param-disambig-list*)]
  | *method-ref-start* . *

We allow a *method-name* to be prefixed by an optional list of type arguments, as in

method-name-list := *method-name-gen* [, *method-name-gen*]...
method-name-gen :=
  [< *ActualTypeArgument* [, *ActualTypeArgument*]... >] *method-name*
method-name := *method-ref* [(*param-disambig-list*)]
  | *method-ref-start* . *

where *ActualTypeArgument* is defined in the grammar for Java.

Proposal: *Enhance JML to allow the syntax above for lists of methods in **`callable`** clauses.*

2.6 Specification expressions

2.6.1 \TYPE

The **\TYPE** type is JML's analog of the **`java.lang.Class`** type. Originally **\TYPE** was distinct from **`java.lang.Class`** in order to represent primitive types as well. However, Java evolved to represent primitive types as **`Class`** objects, so currently JML defines **\TYPE** as fully equivalent to **`Class`**. With the introduction of generic types, **\TYPE** could now be defined to be equivalent to **`Class<?>`**.

However, the Java runtime representation of class information erases any type parameter information: both **`List<Integer>`** and **`List<String>`** are simply represented as **`List<?>`**, for example. Thus, in order to retain the full information available statically, it would be better to define **\TYPE** as a fully reified combination of the raw type information in **Class<?>** and the type arguments of a specific instantiation of the raw type; the behavior of **\TYPE** can be defined with appropriate axioms. Retaining this information would allow static checkers to warn about type-unsafe usage in Java that results in runtime exceptions.

In order to have the runtime JML behavior match the static analysis behavior, we need to define an executable representation of this combined information. One possibility is to have **\TYPE** encapsulate the **com.sun.mirror.type** classes. A restriction with this API is that it is meant to model the entities (e.g., types) in a specific declared program, rather than providing a facility to model types in general. Consequently it appears easier to model Java types straightforwardly as a separately declared executable class with appropriate specifications for static reasoning and implemented using Java APIs where they exist.

An expanded type system in JML may require some additional explicit type operators, such as the ability to extract a type parameter from a type object or to construct a parameterized type. However, there is insufficient experience with the specification needs with such a type system or with the proof rules that would be needed to propose a design at this time. We leave that for future work.

In the following sections we use **Class<?>** to mean Java's current runtime representation of class information (with erasure) and **\TYPE** to mean a representation in JML that reifies all statically declarable types.

Proposal: *Represent **\TYPE** as an entity distinct from **Class<?>**, reifying Java's raw type information and the type parameter information.*

2.6.2 \type

The specification expression **\type(**t**)** is currently defined by JML to be equivalent to t**.class** for a type name t. Java does not allow applying **.class** to a type name with parameters, as in **List<Integer>.class**. The runtime type literals do not retain the type parameter information, although it is used for parsing and typechecking. The expression **\type(List<Integer>)**, however, can be allowed. Thus it is consistent with the discussion of the previous section to define the type of **\type(**t**)** as **\TYPE**, allowing it to hold all of the type parameter information in an expression such as **\type(List<Integer>)**.

Proposal: *The type of **\type** is **\TYPE**. The value of **\type(**t**)** is equivalent to t**.class** for any unparameterized type name t. We allow **\type(**t**)** for parameterized type names, even though the types so represented cannot be expressed as Java class literals without erasure occurring.*

2.6.3 \typeof

The **\typeof** predicate returns the dynamic type (a value of type **\TYPE**) of its argument. The argument may be of primitive type. Its analog in Java is **Object.getClass()**. The value of **\typeof(**x**)** is

- undefined if x is null,
- equal to x**.getClass()** if x has nongeneric reference type, and
- equal to t**.class** if x has primitive type t.

We maintain the definition of **\typeof** as returning a value of type **\TYPE**. Then **\typeof** applied to an argument of parameterized type can include the additional type information that **getClass** erases.

Proposal: *The result type of **\typeof** is **\TYPE**; the expression is undefined if the argument is null.*

2.6.4 *subtype operation* (<:)

JML defines a binary operation **<:** between two **\TYPE** values meaning "is a subtype of". With the equivalence of **\TYPE** and **Class**, JML defined **t1 <: t2** as **t2.isAssignableFrom(t1)**. With the introduction of generics, **isAssignableFrom** no longer correctly models subtype relationships as seen by the compiler. We can define **<:** to act on two **\TYPE** values, but the runtime implementation of that operation must be separately implemented. Arguments that represent primitive types can be treated uniformly; a primitive type is not a subtype of anything but itself.

Proposal: *The arguments of **<:** still have type **\TYPE** and can include the **\TYPE** representations of primitive types. The operation is undefined if either argument is null.*

2.6.5 \elemtype

The **\elemtype** function takes an argument of type **\TYPE** and returns a value of type **\TYPE**. If the argument is an array type, the result is the component type of that array. This is equivalent to the method **Class.getComponentType** (for erased types). Consequently it is convenient to also define **\elemtype** to return null if the argument is not an array type, but undefined if the argument is null. Note that the argument is expected to be an array type, not an object of an array type. That is, the common use is, inconveniently, **\elemtype(\typeof(**o**))** for an object o, and not **\elemtype(**o**)**.

Proposal: *The argument and return types of **\elemtype** are **\TYPE**; **\elemtype** is undefined if the argument is null; the value of the expression is null if x**.isArray()** is false for an argument x.*

2.6.6 \nonnullelements

The **\nonnullelements** predicate returns true if its argument is both non-null and an array object all of whose elements are non-null. It has been undefined if the argument is null or not an array object. The semantics can be improved with better typing, such as with the signature **\nonnullelements(Object[] t)** and corresponding signatures for each primitive type. No specifically generic method typing is needed. Multidimensional arrays are handled because any array is an instance of **Object**. The test for undefinedness (because the argument is not an array) is now changed: it was a semantic check on the argument's dynamic type, but now is simply a type check on the argument's static type.

Proposal: *Change the signature of the **\nonnullelements** function to be a set of overloaded functions with argument types of **Object[]** and t**[]** for each primitive type t.*

2.6.7 *set comprehension and* JMLObjectSet

JML has a construct that allows the definition of new sets as expressions. For example, we can write

```
new JMLObjectSet {Integer i; o.contains(i); i > 0},
```

where *o* is a **Collection<Integer>**. The value of this expression is a **JMLObjectSet** that contains exactly the positive elements of *o*.

In current JML, the type of the result of a set comprehension is **org.jmlspecs.models.JMLObjectSet**, which is a set of **Object**s. However, the result type is in the process of being changed to **org.jmlspecs.lang.JMLSetType**, an interface defined in the core language package **org.jmlspecs.lang**. Any type that implements **JMLSetType** may be named in the constructor portion of the set comprehension expression. However, the type of the elements of the set is known from the declaration inside the set comprehension expression. The result should be a parameterized collection; in the example, this would be

```
JMLObjectSet<Integer> s =
   new JMLObjectSet {Integer i; o.contains(i); i > 0)}.
```

So, if C is the generic (without type arguments) type named after the **new** token and T is the element type named in the declaration, then the type of the result is C<T>, which then must implement **JMLSetType<**T**>**.

Proposal: *The JML model interface **JMLSetType<E>** is parameterized by the type of its elements. The set comprehension expression of the form*

new** C **{** T **e; ...; ...}

*has type C<E>, where E is T if T is a reference type and is T's boxed equivalent if T is a primitive type, and where C<E> must implement **JMLSetType<**E**>**. The model types **JMLObjectSet**, **JMLValueSet**, and **JMLEqualsSet** would implement **JMLSetType**.*

2.6.8 `\lockset`

The value of the **\lockset** keyword is the set of all objects whose associated monitor is owned by the thread in which the **\lockset** expression is evaluated. It currently has type **JMLObjectSet**, but should now be **JMLSetType<Object>**.

Proposal: *The type of **\lockset** is **JMLSetType<Object>**.*

2.6.9 `\max`

The **\max** function takes a **JMLObjectSet** as an argument and returns an **Object**. Typically the argument is **\lockset**. The value of the expression is the object in its argument that is the largest (measured by the lock ordering operation) of all the elements in the argument set that are locked by the current thread. Corresponding to previous changes, the signature of the **\max** function is best expressed as

<T> T \max(JMLSetType<T> o) .

Proposal: *The signature of **\max** is*

***<T> T \max(JMLSetType<T> o)**.*

The expression is undefined if the argument is null; the result is null if the argument contains no objects locked by the current thread.

2.6.10 Autoboxing and the lock ordering operations < and <=

JML overrides the less-than (**<**) and less-than-or-equal (**<=**) binary operations to apply to two **Object**s, returning a result according to a user-defined ordering. This feature interacts with Java's auto-boxing. Specifically, the operations are now ambiguous when the arguments are a primitive numeric type and its autoboxed equivalent: for **int i** and **Integer j**, **(i < j)** could be either the numeric comparison between **i** and the unboxed **j** or it could be the lock-order operation between the boxed **i** and **j**. The operations are also ambiguous between two numeric reference types: for **Integer i** and **Integer j**, **(i < j)** could be either the numeric comparison between the unboxed **i** and the unboxed **j** (as it would be in Java) or it could be the lock-order operation between the **i** and **j** (as it would be currently in JML without auto-unboxing).

Proposal: *This issue is currently under discussion[2], but the favored resolution is to deprecate **<** and **<=** as the lock-ordering operators, replacing them with the nonoverloaded new tokens **<#** and **<#=**.*

2.6.11 autoboxing and class literals for `\bigint` and `\real`

JML introduced two new types, **\bigint** and **\real**: **\bigint** is the set of infinite-precision integers; **\real** models the real numbers. Both are intended to provide infinite-precision quantities from mathematics to be used in specifications, rather than only the finite-precision types from programming languages. Although the relevant semantics has been a point of discussion, the definition of JML is simplest if both are interpreted as primitive types. Then we also need to define the boxed equivalents: **java.lang.BigInteger** and a new model type **org.jmlspecs.lang.JMLReal**, respectively.

JMLReal would have a specification that is appropriate for real numbers. Its runtime implementation necessarily needs to approximate the behavior of reals. Also, since there is no infinite-precision primitive integer, the difference between primitive and reference types for **\bigint** must be handled by the type checker, with the executable implementation using **BigInteger** for both.

For each primitive type there is a corresponding class literal. It is different but of the same type as the literal for its boxed type. Thus **int.class** and **Integer.class** are unequal but both have type **Class<Integer>**. The corresponding **Class** values are needed for **\bigint** and **\real**. **Class** objects are typically obtained using native methods from the underlying virtual machine, so one cannot create **Class** objects for new kinds of primitive types. However, we can model these new primitive types as **\TYPE** values.

Proposal: *Define **\bigint** and **\real** as primitive types. Define **java.lang.BigInteger** and **org.jmlspecs.lang.JMLReal** as the corresponding boxed object types, with auto boxing and unboxing conversions corresponding to the other primitive types. Model the literals for these primitive types as **\TYPE** values. Chalin et al. [3] has explored the implications of various semantics of numeric operations in more detail.*

2.6.12 `\only_called`

The arguments of the **\only_called** predicate are method signatures. These must now be allowed to be parameterized method signatures, with either specific types or wildcard types. The same syntax is used for the method signatures as in the **callable** method specification clause.

Proposal: *The arguments of **\only_called** are now instances of* method-name-gen *as defined in section 2.5.*

2.7 Annotations

The annotation feature was a second major change in Java 1.5. In this case existing usage was not changed, but a new capability was created for describing properties of program constructs, and many groups began experimenting with annotations expressing type constraints. In conjunction with annotations, the Java framework provides an API to process annotations as part of compilation. With this API, additional syntactic or semantic checks can be performed that are not part of the compiler (or of pure Java). A number of annotation-related projects may influence the future of JML:

- JML tools are already experimenting with replacing modifiers in declarations (e.g., pure, non_null) with equivalent annotations (**@Pure**, **@NonNull**) from a JML-specific annotation package: **org.jmlspecs.annotations**.

- Taylor [1] experimented with using annotations for all JML specifications. This requires specification expressions to be String arguments to annotations. The approach is feasible but incurs different usability issues than the current JML design. Just as current JML must process comments containing extensions to Java expressions, an annotation-based specification language would need to parse and type-check the String arguments of annotations as extensions to Java expressions. There is currently no compiler or annotation processing support for this language processing.

[2] on the mailing list jmlspecs-interest@lists.sourceforge.net

- The JSR-308 project [9, 11] seeks to allow annotations in conjunction with any use of a type name in Java. This would allow annotations to be used as type modifiers. Then subtypes such as non-null types or readonly types could be easily defined and used uniformly; checkers for them could be built using the annotation processing API, as pure extensions to Java (as has been demonstrated).
- The JSR-305 project [10] seeks to standardize the naming of annotations. Currently, similarly named annotations are used by different groups for similar purposes, but with some differences in semantics. For example, JSR-305 defines **`@NonNull`**, **`@CheckForNull`**, and **`@Nullable`** as three different nullity related annotations, where JSR-308 and JML use just two: **`@NonNull`** and **`@Nullable`**, and IntelliJ uses **`@NotNull`** and **`@Nullable`**. This project would enable the expression of many very specific custom specifications; some examples are that the return value of a method should not be ignored, that a numeric value is positive, that a numeric value is nonzero, and that a collection (or string or array) is not empty.

Proposal: *JML should migrate to using annotations instead of modifiers, particularly if JSR-308 is adopted. (If not, current JML syntax will need to be retained, at least for those syntactic locations where annotations are not allowed.) JML should continue to investigate using annotations for a broader range of specifications.*

In addition the JML community needs to engage with the broader static analysis community in the following ways:

- JSR-308 will allow annotations to be used in more places than they currently are and will allow annotations to replace JML modifiers. It should be supported by the JML community.
- Continue investigation into allowing annotations in other locations in order to support current JML specifications. JML currently allows specifications as statements within method bodies, statement modifiers, and declarations within classes.
- Common fully qualified names for annotations as advocated by JSR-305 would be a good thing, as long as the semantics are also the same. Using the same (unqualified) annotation names with different semantics for different tools is a nuisance, or even with the same semantics but in different packages. There is not yet consensus on the appropriate semantics for each standard annotation name (for example, for nullity annotations).
- JML provides a general mechanism to express a large family of specifications, with a goal of a broad view of static analysis extending as far as possible toward software verification. Annotations in general, particularly if names are standardized by JSR-305, provide a means to define many specific annotations, with a goal of enabling best-effort checks of commonly used, quite specific, specification predicates. It is an open question about how these two approaches should coexist and what combination provides the best and most usable tools for the software developer and specifier.

Proposal: *The JML community should engage more vigorously with both JSR-305 and JSR-308 to enable outcomes that are mutually beneficial and allow a good migration path for JML.*

2.8 Other general changes to Java

2.8.1 Static import

The Java static import statement allows a compilation unit to use static names from another class without qualifying them with a class name. JML has a model import statement corresponding to Java's import. The types imported by a model import statement are only available in JML statements and not in the Java program itself. With the introduction of Java's static import, JML's model import should also have a static option.

This is expected to have little effect on JML implementations. In fact most implementations to date do not distinguish Java from JML imports—all imported names are available in both parts of a compilation unit. This is an incompleteness in the JML implementations, but it rarely causes trouble and problems can be worked around by using fully qualified names.

A proper implementation of JML's model import needs to keep two namespaces of imported names: the Java namespace and the Java+JML namespace. Neither the OpenJDK nor the Eclipse Java compilers can be readily extended to do this. However, the problem is no more difficult with the addition of static imports.

2.8.2 Enum types

The Enum type facility adds true type-safe Enum types to Java. Presuming a JML implementation can use a Java compiler to parse and typecheck JML expressions, this feature is available to JML tools without additional implementation effort.

2.8.3 varargs

The varargs feature allows the declaration of methods that take an arbitrary number of arguments (of the same type). A Java compiler will also provide this capability to JML tools. Keep in mind that model methods should also have the varargs feature. Typically these are parsed in the same way that Java methods are.

2.8.4 Enhanced for statement

The enhanced for statement causes no difficulties for JML tools in itself. However, specifying it is a problem. Traditional while, do, and for statements have an iteration variable that is available to and almost always needed by the specifier in writing loop invariants. For example, a simple for loop might be specified as follows:

```
int sum = 0;
//@ loop_invariant 0<=i && i<=10;
//@ loop_invariant sum == i * (i+1)/2;
for (int i=0; i<10; i++) {
    sum = sum + i;
}
```

It is important to have the loop variable **`i`** available, so that the invariant can be written in terms of the iterations already completed. The loop variant also needs the loop variable to be able to show that the loop is making progress toward termination.

The enhanced for loop provides no such variable. An example of such a loop is this:

```
int[] array = ...
int sum = 0;
for (int element : array) {
    sum = sum + element;
}
```

The loop invariant we would like to write is

```
      sum == (\sum int k; 0<=k && k<i; array[k]),
```

where **`i`** is the index of the next array element to be processed. But this value is not available. Java provides two types of enhanced for statements: one takes an array, as in the example above, the other takes an object of type **Iterable**.

Spec# [6] has solved this problem by introducing a **\values** keyword whose value is a sequence of all of the values that have been iterated over so far. For Java and JML, I propose a corresponding solution, but with two keywords.

• Associated with each enhanced for loop is a new keyword **\index** of type **int**. The keyword represents the 0-based number of the current iteration. For enhanced for loops based on arrays this is also the index in the array of the current array value. The keyword is in scope within the body of the loop and in the loop specifications just prior to the loop. The value of **\index** begins at 0 and increments until equal to $array$**.length**, for array-based loops. The keyword may not be assigned to. Thus the example above is equivalent to (were **\index** a valid Java variable)

```
int[] array = ...
int sum = 0;
for (int \index = 0; \index < array.length; \index++) {
    int element = array[\index];
    sum = sum + element;
}
```

We can specify our example as follows

```
int[] array = ...
int sum = 0;
/*@ loop_invariant sum ==
          (\sum int k; 0<=k && k<\index; array[k]); */
//@ decreasing array.length - \index;
for (int element : array) {
    sum = sum + element;
}
```

• A second new keyword is **\values**. This keyword would have type **org.jmlspecs.lang.JMLList<**T**>**, where T is the type of the iteration variable and **Iterable<**T**>** is the type of the iteration collection. The value of **\values** is a sequence of the values returned so far (prior to the current iteration) by the iterator (autoboxed if the loop is an array-type loop and the array element type is a primitive type). Thus

```
Set<Integer> set = ... // all positive integers
int max = 0;
for (Integer i : set) {
    if (max<i) max = i;
}
```

would be specified as

```
Set<Integer> set = ... // all positive integers
int max = 0;
/*@ loop_invariant max == \values.size() == 0 ? 0 :
              (\max int k; \values.contains(k); k); */
for (Integer i : set) {
    if (max<i) max = i;
}
```

There are two alternatives for when the addition of the loop variable's value to the **\values** list occurs: (a) as part of the update step (after the loop body is executed), or (b) immediately after the value is extracted from the iterator. These two alternatives are being evaluated; the discussion below assumes the first design. Loop invariants are the same in both cases. However, in (a), **\index** always equals the size of **\values**, but in the body of an iteration, the current value of the loop variable is not yet in the **\values** list; if the loop is exited by a break statement, that value will not be in the list. In (b), extracted values are always in the list, but what is true in an invariant is not necessarily true in the body, since **\index** and **\values** are updated at different times. In the first design,

```
//@ loop_invariant ...
for (T element : array) {
    ... body ...
}
```

is equivalent to (where **T'** is **T** or its boxed equivalent)

```
int \index = 0;
JMLList<T'> \values = ... (empty list of T' )...
T element;
for (; \index < array.length ;
                       \index++, \values.add(element)) {
    ... check loop invariant
    element = array[\index];
    ... body ...
}
... check loop invariant ...
```

and

```
//@ loop_invariant ...
for (T element : iterable) {
    ... body ...
}
```

is equivalent to

```
int \index = 0;
JMLList<T> \values = ... (empty list of T )...
Iterator<T> iterator = iterable.iterator();
T element;
for (; iterator.hasNext() ;
                  \index++, \values.add(element)) {
    ... check loop invariant
    element = iterator.next();
    ... body ...
}
... check loop invariant ...
```

Note that if the **Iterable** collection is known to have fixed size, then something like **list.size() - \values.size()** makes an appropriate loop variant. However, the size of an **Iterable** is not necessarily known or fixed.

It is not strictly necessary to define both **\index** and **\values** since **(\index == \values.size())**. However, **\index** is more natural for loops that iterate over arrays, and it seems more readable and easier for reasoning engines to use, hence the proposal here is to define both keywords. If loops are nested, the **\index** and **\values** keywords in the inner loop will hide the corresponding keywords for an outer loop.

Proposal: *Define the keywords* ***\index*** *and* ***\values*** *for enhanced for loops with the semantics described above. Create a parameterized interface* ***JMLList<E>*** *in* ***org.jmlspecs.lang****.*

2.8.5 **SuppressWarnings** *and* **nowarn**

Java 1.5 introduced the **java.lang.SuppressWarnings** annotation as a mechanism for user control over compiler warnings. The arguments of the annotation name the warnings that then will not be issued in the context to which the annotation applies. JML has had a lexical construct, **nowarn**, that offered similar capabilities. Thus the question: can **java.lang.SuppressWarnings** replace **nowarn**?

The short answer is partially. There is a key difference between the two constructs. The **nowarn** token is lexical; it may occur anywhere in the source code and applies to the source code line on which it appears. An annotation is constrained to appear in conjunction with declarations and packages; the **SuppressWarnings** annotation is allowed on type, field, method, constructor, parameter, and local variable declarations. It is also relevant that the **SuppressWarnings** annotation has only source retention and is unavailable at runtime. The JSR-308 proposal would expand the use of annotations to also be allowed on types anywhere they appear.

The warnings from JML tools are of two sorts. Some warnings are compiler-like: misuse of various language constructs. Where these are nonfatal, they can be suppressed just like compiler warnings might be. The more important warnings from JML are runtime or statically found assertion violations. These are associated with nearly every JML construct and implicitly with many Java language features. To be useful, a warning suppressor for JML needs (a) to be more fine-grained than at the method declaration level, and (b) to be able to be applied to any sort of specification construct. Thus the **SuppressWarnings** annotation is not currently an adequate replacement for **nowarn**, although it can provide similar functionality on a coarser scale. A Java annotation that could appear anywhere a comment could appear would be very useful for JML.

Proposal: *JML tools should recognize a common (to be agreed upon) set of warning names for various kinds of assertion violations and recognize their use in* ***SuppressWarnings*** *annotations. The* ***nowarn*** *construct should continue to be used (using the same set of warning names) and should not be deprecated for now.*

2.9 Model classes

JML contains a library of classes (in **org.jmlspecs.models**) intended to model mathematical constructs and to be useful in specifications. Consequently they are designed as types with immutable values and pure, functional methods. The classes have specifications suitable for static analysis and implementations that can be executed at runtime, although they are not necessarily efficient.

Many of these classes implement collections or related constructs and they should be rewritten as generic classes or interfaces, similar to the reimplementation of Java's collection classes, but retaining the design of immutable values. Specifically, the following should be reimplemented (here # stands for one of `Object`, `Equals`, and `Value`):

- Collection classes that should be parameterized by element type:
  JMLCollection, JML#Bag, JML#Sequence, JMLSetType, JML##Pair (e.g., JMLEqualsObjectPair), JML#Set, JML#To#Map, JML#To#Relation, JMLList#Node

- Other types needing generic parameters:
  StringOfObject, JMLComparable, JMLIterator, JMLModelObjectSet, JMLModelValueSet, JMLValueBagSpecs, JMLObjectSequenceSpecs, JMLValueSequenceSpecs, JMLValueSetSpecs

- Enumerations that should be converted to Iterators, with type parameters:
  JMLEnumeration, JMLEnumerationToIterator, JML#BagEnumerator, JML#SequenceEnumerator, JML#SetEnumerator, JML#To#RelationEnumerator, JML#To#RelationImageEnumerator

- Comparison operations needing type parameters (similar to **java.lang.Comparable**):
  org.jmlspecs.models.resolve.*CompareTo

Although not directly a result of the move to generic types, the model classes are not quite appropriately divided between the packages **org.jmlspecs.lang** and **org.jmlspecs.models**. The design is that classes in **org.jmlspecs.lang** are needed by the language features themselves. Consequently **JMLSetType** is there since it is the type of a set comprehension expression and **\lockset**, and **JMLDataGroup** is used for datagroups. In addition, **JMLIterator** and **JMLIterable** are used by **JMLSetType** and should be in **org.jmlspecs.lang**.

Proposal: *The model classes should be reimplemented with generic types. The generic versions should be placed in a new package,* ***org.jmlspecs.genericmodels****.* ***JMLIterator*** *and* ***JMLIterable*** *should be moved to* ***org.jmlspecs.lang****.*

2.10 Existing specifications

There are many JDK classes with at least partial JML specifications (although many more are needed). Many of those classes, particularly collection classes, became generic classes when the language moved from 1.4 to 1.5. The JML specifications for those classes now need to be ported as well. For the most part that work is straightforward, but there is one interesting aspect.

Most of the specifications for nongeneric collections include a ghost field **\TYPE elementType**, intended to hold the dynamic type of the elements of the collection. This is now superseded by the type parameter of the generic collection. Static analysis tools operating on source code can readily use the type information of type parameters. However, current Java erases the generic type information in binary classes; expressions such as **\type(E)** or **E.class** for a type parameter **E** are not legal. Thus runtime checking of JML will still need the **elementType** information.

A runtime assertion checker might correct this deficit by passing the type information into constructors and generic methods as additional parameters. A constructor expression such as, for example, **new HashSet<Integer>()**, would effectively be rewritten as **new HashSet$(Integer.class)**; the type parameter information would be stored in synthetic fields inside the class, equivalent to the **elementType** specification fields. A solution such as this is a matter for future research; it is expected to encounter tricky interactions among proof rules, generic type systems, and Java's current type erasure.

Proposal: *All of the existing JDK specifications need to be ported to Java 1.5. The* ***elementType*** *ghost field previously used in collection, enumeration and iteration types can be deprecated once generic type information is retained in compiled Java. Interim runtime assertion checking implementations can experiment with the auxiliary method parameter solution described above for accessing type parameter information at runtime.*

2.11 The compiler, syntax tree and annotation processing APIs

The compiler, compiler tree and annotation processing APIs together offer a promising step toward better future JML tool generation. The annotation processing API allows user code to process the parse trees of compilation units as parsing occurs. The tool can choose to process all files or only those that are marked with recognized annotations. New compilation units can be generated and entered into the parsing process. The compiler tree API allows the ASTs to be traversed and inspected, and the compiler invocation API allows programmatic control of the compilation process.

However, the current capabilities of these APIs are not yet sufficient for easy construction of JML tools.

- JML needs to parse Java-like expressions, obtaining ASTs representing expressions. Most of JML's expression syntax is the same as Java's, but there are some JML-specific extensions. There is as yet no facility either for extending the compiler or even for invoking the compiler on code fragments. The fact that the public API ignores Java comments, which is where JML specifications currently reside, is an additional complication.

- JML needs to be able to use and extend the name resolution and type checking capabilities of the Java compiler. Those compiler phases happen after annotation processing is performed, so there is currently no way (through the public API) to apply type checking to the JML specification expressions or to extend it for JML extensions.

- For runtime checking, a JML tool needs to modify the syntax tree to represent the source code with assertion checks included. The public APIs currently do not allow replacing one compilation unit with a revision nor revising a compilation unit's AST directly.

However, if the extended parsing and typechecking problems were resolved, the annotation processing API could enable static checking, so future developments in these APIs are worth following. Note that the functionality needed to implement static or runtime checking for JML can be created by direct extension of the publicly available OpenJDK source code, as tools such as the Checker framework [9] and the OpenJML [12] project have done.

3. IMPLEMENTATION AND FUTURE WORK

These enhancements to current JML are implemented on an experimental basis in the OpenJML project. OpenJML is a JML parser and typechecker built by extending the OpenJDK 1.6 source code. Porting the specifications of model types and the JDK is in progress.

Evaluation of JML's specification capabilities against industrial code is an ongoing activity that is being carried out in the context of some issues left unaddressed by this paper: the degree to which full reification of parameterized types is a needed design choice, the need for additional specification constructs for type manipulation, the relationship between the goal of full verification and checks of specific conditions (as, for example, by the FindBugs [5] tool), and the appropriate use of the evolving Java APIs for implementing static analysis tools.

4. CONCLUSIONS

The migration of JML to Java 1.5 and 1.6 has been mostly a task of accommodating the generic type facility of the recent versions of Java. The assessment described in this paper identified a number of areas where typing changes of JML features and a conversion to using generic types throughout JML would be beneficial. JML extensions are needed in the enhanced for statement and changes in the lock ordering operation; generics require some enhancements to refinement resolution; and some careful design work is needed to integrate JML's additional primitive types and to model reified and erased types statically and at runtime. Other changes to Java, such as annotations and some new public APIs, may provide benefits as they evolve, but are not ready to be used for implementing JML itself or to replace existing JML features. Finally, the usefulness of annotations and annotation processing has prompted a number of projects to adopt annotation processing for static analysis; the JML community should engage more fully with those efforts for mutual benefit.

5. REFERENCES

[1] K. P. Boysen. A specification language design for the Java Modeling Language (JML) using Java 5 annotations. Technical Report 08-03, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, Apr. 2008.

[2] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.

[3] P. Chalin. JML support for primitive arbitrary precision numeric types: Definition and semantics. *Journal of Object Technology*, 3(6):57–79, June 2004.

[4] P. Chalin, P. R. James, and G. Karabotsos. An integrated verification environment for JML: Architecture and early results. In *Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007)*, pages 47–53. ACM, Sept. 2007.

[5] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 9–14, New York, NY, USA, 2007. ACM.

[6] B. Jacobs, E. Meijer, F. Piessens, and W. Schulte. Iterators revisited: Proof rules and implementation. In *7th Workshop on Formal Techniques for Java-like Programs (FTfJP)*, July 2005.

[7] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, Mar. 2006.

[8] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, Mar. 2005.

[9] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, Seattle, WA, USA, July 22–24, 2008.

[10] `http://jcp.org/en/jsr/detail?id=305`.

[11] `http://jcp.org/en/jsr/detail?id=308`.

[12] Unpublished information is available at `http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/OpenJML/trunk/OpenJML/README`.