Towards Support for Non-null Types and Non-null-by-default in Java

Patrice Chalin

Dept. of Computer Science and Software Engineering,
Dependable Software Research Group,
Concordia University Montréal, Québec, Canada
chalin@cse.concordia.ca

Abstract. This paper begins with a survey of current programming language support for non-null types and annotations, with a particular focus on extensions to Java. With the advent of Java 5 annotations, we note a marked increase in the availability of tools that can statically detect potential null dereferences. For such tools to be truly effective, they require that developers annotate declarations with nullity modifiers. Unfortunately, it has been our experience in specifying moderately large code bases, that the use of non-null annotations is more labor intensive than it should be. Hence, backed by an empirical study, we recently outlined a proposal for adapting the Java Modeling Language (JML) to support a non-null-by-default semantics. This paper extends our earlier work by providing a detailed description of the changes made to JML in support of the new default. Finally, in relation to our survey, we justify the claim that JML is one of the most fully developed solutions for bringing support for non-null annotations/types to Java.

Keywords: non-null annotations, non-null types, Java, JML, non-null by default.

1 Introduction

Null pointer exceptions are among the most common faults raised by components written in mainstream imperative languages. Developers increasingly have at their disposal tools that can detect possible null dereferences (among other things) by means of static analysis of component source. It is well known that such tools can only perform minimal analysis when provided with code alone [1, 2]. On the other hand, given that components and their support libraries are supplemented with appropriate specifications and/or nullity annotations, then the tools are able to detect a large proportion of potential null pointer dereferences while keeping false positives to a minimum.

This paper has two main contributions. Firstly, we present the results of a survey of languages, language extensions and tools supporting non-null types and annotations. While we examined imperative languages in general and object-oriented languages in particular, our focus is on Java. We note that the introduction to Java 5, of a standard way of extending the language by means of annotations, seems to have contributed to an increased emergence of support for static checking of potential null dereferences in Java. Our survey includes half a dozen approaches for Java, not the least of which is preliminary support by two of the most popular Java IDEs, Eclipse and IntelliJ IDEA. Unfortunately, it has been our experience in specifying moderately large code bases, that the use of non-null annotations is more labor intensive than it should be; i.e., we seemed to spend more of our time adding non-null modifiers to declarations than leaving them unannotated.

The results of one of our recent studies demonstrated that the majority[1] of reference type declarations in Java are meant to be non-null, based on design intent, thus confirming our subjective experiences. Hence, we recently [3] outlined a proposal for adapting the Java Modeling Language (JML) to support a non-null-by-default semantics. The second main contribution of this paper extends our earlier work by providing a detailed description of the changes made to JML in support of the new default. We give careful attention to ensuring that the new semantics is compatible with the introduction of non-null *types* into JML (though this will be the subject of a subsequent paper). Finally, in relation to our survey, we justify the claim that JML is one of the most fully developed solutions for bringing support for non-null annotations/types to Java.

The remainder of the paper is organized as follows. The survey is presented in the next section. A description of the adaptations made to JML in order to support non-null-by-default as well as a discussion of the issues that arose, are presented in Section 3. We offer a discussion of the support, in Java, for non-null and non-null-by-default in Section 4, and conclude in Section 5.

2 Survey: Languages and nullity

In this section we present a summary of the languages, language extensions and tools that offer support for non-null types or annotations.

2.1 Languages without pointer types

Early promotional material for Java touted it to be an improvement over C and C++, in particular because “Java has no pointers” [4, §2], hence ridding it of “one of the most bug-prone aspects of C and C++ programming” [4, p.6]. Of course, reference types are implemented by means of pointers, though Java *disciplines* their use—e.g. the pointer arithmetic of C and C++ is prohibited for Java reference types.

Other languages have pushed this discipline even further by eliminating null. Obvious examples are functional languages, including ML which also supports imperative features such as references and assignment. Another noteworthy example from the ML family is the Objective Caml object-oriented language. Though ML and Objective Caml support references, every reference is guaranteed to point to an instance of its base type, because the only way that a reference can be created, is by taking the reference of a *value* of the

[1] Over 63% of declarations were meant to be non-null. The study sample included 161KLOC from over 500 files.

base type [5]. Hence, references are (vacuously) non-null by default in these languages. Of course, a generic "pointer type" can be defined in ML or Objective Caml as a user-defined tagged union type

```
    type 'a pointer = Null | Pointer of 'a ref;
```

Programmers need not go out of their way to define and use such a type since it is very seldom necessary [6]. Similar remarks can be made of early prototypical object-oriented languages like CLU. CLU (vacuously) supported non-null references by default since it did not have an explicit notion of pointers, nor did it have a special "null" value belonging to every reference type. (Our study results will confirm that Java developers, like Objective Caml programmers, need non-null types more often than nullable types.)

2.2 Imperative languages with pointer types

To our knowledge, the first imperative programming language, or language extension, with pointer types to adopt the non-null-by-default semantics is Splint [7, 8]. Splint is a "lightweight" static analysis tool for C that evolved out of work on LCLINT (a type checker of the behavioral interface specification language for C named Larch/C [9, 10]). Splint is sometimes promoted as "a better lint" because it is able to make use of programmer supplied annotations to detect a wider range of potential errors, and this more accurately, than lint. Annotations, like in JML, are provided in the form of stylized comments. In Splint, declarations having pointer types are assumed to be non-null by default, unless adored with `/*@null*/`. Splint does nullity checking at "interface boundaries" [8, §2]: annotations can be applied to function parameters and return values, global variables, and structure fields but not to local variables [11, p.44].

While there are no other extensions to C supporting the non-null-by-default semantics, extensions for non-null annotations or types have been proposed. For example, Cyclone [12], described as a "safe dialect of C", supports the concept of never-NULL pointers, written as "*T* `@`" in contrast to the usual syntax "*T* `*`" for nullable pointers to *T*. As another example, we note that the GNU `gcc` supports a form of non-null annotation for function parameters only; e.g. an annotation of the form

```
    __attribute__((nonnull (1, 3)))
```

after a function signature would indicate that the first and third arguments of the function are expected to be non-null [13, §5.24].

2.3 Object-oriented languages (non-Java)

2.3.1 Eiffel

The recent ECMA Standard of the Eiffel programming language introduces the notions of *attached* and *detachable* types [14]. These correspond to non-null (or non-void types, as they would be called in Eiffel) and nullable types, respectively. By default, types are attached—which, to our knowledge, makes Eiffel the first non research-prototype object-oriented language to adopt this default. Eiffel supports covariance in method return types and invariance of parameter types except with respect to parameter nullity, for which is supports contravariance [14, §8.10.26, §8.14.5]—see Table 1.

Prior to the release of the ECMA standard, types were detachable by default. Hence a migration effort for the existing Eiffel code base has been necessary. Special consideration has been given to minimizing the migration effort in the form of compiler / tool support.

2.3.2 Spec#

Spec# is an extension of the C# programming language that adds support for contracts, checked exceptions and non-null types. The Spec# compiler statically enforces non-null types and generates run-time assertion checking code for contracts [15]. The Boogie program verifier can be used to perform extended static checking of Spec# code [16].

While Spec# code cannot generally be processed by C# compilers, compatibility can be maintained by placing Spec# annotations inside stylized comments (`/*^` … `^*/`) as is done with other annotation languages like JML.

Introduction of non-null types (vs. annotations) requires care, particularly with respect to field initialization in constructors and helper methods [17]. Open issues also remain with respect to arrays and non-null static fields for which the Spec# compiler resorts to run-time checking to ensure type safety [18, §1.0]. For reasons of backwards compatibility, a reference type name *T* refers to possibly null references of type *T*. The notation *T*! (or `/*^ ! ^*/`, with a special shorthand of `/*!*/`) is used to represent non-null references of type *T*.

As of the February 2006 release of the Spec# compiler, it is possible to use a compiler option to enable a non-null-by-default semantics. When this is done, *T*? can be used to denote possibly null references to *T*. We note that of all the languages that were surveyed, Spec# is the only one with annotation suffixes (i.e. that appear after the type name rather than before). Nullity return type and parameter type variance for overriding methods in Spec# conforms to the type (in)variance rules of C#—i.e., types must be the same.

2.3.3 Nice

Nice is a new programming language whose syntax is superficially similar to that of Java. It can be thought of as an enriched variant of Java supporting parametric types, multi-methods, and contracts, among other features [19, 20]. Nullable types are called *option types* in Nice terminology. It is claimed that Nice programs are free of null pointer exceptions. By default, a reference type name *T* denotes non-null instances of *T*. To express the possibility that a declaration of type *T* might be null, one prefixes the type name with a question mark, ?*T* [21].

2.4 Java support for non-null

2.4.1 FindBugs

The FindBugs tool does static analysis of Java class files and reports *common* programming errors; hence, by design, the tool forgoes soundness and completeness for utility (an approach that is not uncommon for static analyzers) [22]. In order to increase the accuracy of error reporting related to nullity and to better be able to assign blame, support for nullity annotations for return types and parameters was recently added—annotations can be applied to local variables but they are effectively ignored. The annotations are: `@NonNull`, used to indicate that the declared entity cannot be null, and `@CheckForNull`, indicating that a null value should be expected and hence, any attempted dereference should be preceded with a check [2].

Although FindBugs has been applied to production code (e.g. Eclipse 3.0 source), nullity annotations have not yet been used on such samples. Our recent study [3] suggests that when this happens, specifiers are likely to find themselves decorating most reference type declarations with `@NonNull`.

2.4.2 Nully and the IntelliJ IDEA

Nully is an IntelliJ IDEA plug-in that can be used to detect potential null dereferences at edit-time, compile-time and run-time. Nully supports the `@NonNull` annotation only. It can be applied to method return types and parameters as well as local variables (but not fields). Nully documentation claims that it supports run-time checking of non-null constraints on local variables; this could not be confirmed, and seems doubtful as no other annotation tool supports this. Non-null checking of parameters is only provided in the form of run-time checks [23].

There has yet to be an official release of Nully and it is not clear whether the tool is still being developed, particularly since the latest release of the IntelliJ IDEA marks the introduction of its own (proprietary) annotations `@NotNull` and `@Nullable` [24]. IDEA supports edit-time and compile-time checks, but not run-time checks of non-null. IDEA supports nullity return type covariance and parameter type contravariance (we note that this is incompatible with Java, which requires invariance for parameter types).

2.4.3 JastAdd

JastAdd is an open source “aspect-oriented compiler construction system” whose architecture promises to support compiler feature development in a more modular fashion than is usually possible [25]. As a demonstration of this flexibility, support for non-null types has been defined as an “add-on” to the JastAdd based Java 1.4 compiler [26]. The implemented type system is essentially that of Fähndrich and Leino [27]. In fact, they make use of the same annotations, which makes the extension incompatible with standard Java (of course, it should be rather easy to rename the annotations to be conformant to Java 5 annotation syntax). Like Spec#, nullity modifiers of overriding methods must match exactly, both for return and parameter types. Finally, we note that the JastAdd compiler currently only does type checking, without apparent support for code generation.

Table 1. Summary of support for non-null

Language / Tool	Type / Annotation	Default	Member declaration modifier (prefix) for		Non-null Annotation (A) and Checking at run-time (R), or statically at compile-time (S). Abbr.: all (✓=ARS); none (✕)					Overriding method type variance w.r.t.		Anno. API of std libraries?	Class modifier?	Compiler option to invert default
			non-null	nullable	method	param-eters	field	local variables	array elements	result nullity	param -eter nullity			
Splint	anno.	**non-null**	/*@notnull*/	/*@null*/	AS	AS	AS	S	✕	N/A	N/A	✕	N/A	✕
Eiffel	type	**non-null**	!	?	✓	✓	✓	✓	✓	covariance	contravar.	✓	✕	✕
Spec#	type	nullable	! (suffix)	? (suffix)	✓	✓	✓	AS	✕	invariance	invariance	(✓)	✕	✓
Nice	type	**non-null**	!	?	AS	AS	AS	AS	✓	covariance	contravar.	✕	✕	✕
Java support														
JML	anno.	**non-null**	/*@non_null*/	/*@nullable*/	✓	✓	✓	AS	✓	covariance	**invariance**	✓	✓	✓
IntelliJ IDEA (≥ 5.1)	anno.	nullable	@NotNull	@Nullable	AS	AS	AS	AS	✕	covariance	contravar.	✕	✕	✕
Nully (IDEA plug-in)	anno.	nullable	@NonNull	N/A	✓	AR	✕	(✓)	✕	no restriction	no restriction	✕	✕	✕
FindBugs (≥ 0.8.8)	anno.	nullable	@NonNull	@CheckForNull	AS	AS	✕	S	✕	no restriction	no restriction	✕	✕	✕
JastAdd + NonNull Extension	anno.	nullable	[NotNull]	[MayBeNull]	AS	AS	AS	AS	✕	invariance	invariance	✕	✕	✕
Eclipse JDT (≥ 3.2?)	anno.	nullable	@NotNull	@Nullable	AS	AS	AS	AS	✕	TBD	TBD	✕	✕	✕

2.4.4 Eclipse JDT

Support for nullity annotations (tentatively `@NotNull`, and `@Nullable`) and null reference analysis is planned for the next dot release of the Eclipse JDT, i.e., 3.2 [28]. Very little has been published yet on this feature, though we suspect that support will likely be similar to that provided by IntelliJ's IDEA.

2.4.5 Java Modeling Language

The Java Modeling Language (JML) originated from Iowa State University (ISU) under the leadership of Gary Leavens. It is currently the subject of study and use by a dozen international research teams [29]. JML is a behavioral interface specification language that, in particular, brings support for Design by Contract (DBC) to Java [30]. Using JML, developers can write complete interface specifications for Java types, i.e. classes and interfaces. JML annotated Java code can be compiled with standard Java compilers because annotations are contained in stylized comments whose first character is @.

JML enjoys a broad range of tool support including [29, 31]:

- Jmldoc that generates documentation in a manner that is similar to `Javadoc`, but incorporating JML specifications.
- `jmlc`, the ISU JML run-time assertion checker compiler.
- ESC/Java2, an extended static checker that provides a compiler-like interface to fully automated checking of JML specifications. Like similar tools, ESC/Java2 compromises soundness and completeness for efficacy and utility.
- LOOP tool that can be used in conjunction with PVS to perform complete verification of JML annotated Java applications.
- JmlUnit, a tool for generating JUnit test suites using JML specifications as test oracles.
- JMLKEY tool that offers support for model-driven design, principally from UML class diagrams, with JML as a design (constraint) specification language. The tool supports the complete JavaCard language.

JML has nullity modifiers (`non_null` and `nullable`) and it recently adopted a non-null-by-default semantics for reference type declarations [3, 32]. Further characteristics of JML will be discussed in the next section.

2.5 Summary

A summary of the languages, extensions and tools covered in this section, is given in Table 1. Focusing our attention on the support for Java we note:

- Unnecessary minor variability in annotation names, particularly for those tools using Java 5 annotations (an exception to this is FindBugs' `@CheckForNull` which actually has an intended semantics that is different from that of `@Nullable`).
- Inconsistency across tools in the kind of type variance supported for the return and parameter type nullities of overriding methods. Only JML is compatible with Java 5, with its covariance in return types and invariance for parameter types.
- The Spec# and JML compilers are the only tools supporting command-line options for reversing the default nullity interpretation of reference type declarations.
- JML is the only Java compatible language providing
    - specifications for standard libraries (e.g. `java.lang`),
    - class-scoped nullity modifiers.

These aspects of JML will be discussed at greater length in Sections 3 and 4.

3 Non-Null by Default in JML

We recently proposed that JML be adapted so that declarations of reference types are interpreted as non-null by default [3]. This new default has the advantages of

- Better matching general practice: the majority of declarations are *correctly* constrained to be non-null.
- Lightening the annotation burden of developers; i.e. there are fewer declarations to explicitly annotate as `nullable`.
- Most importantly, being *safer*.

- Processing of null generally requires extra programming logic and code to be handled correctly. With the new default, an annotation explicitly *alerts* developers that null values need to be considered.
- If a developer delays or forgets to annotate a declaration as nullable, this will at worst limit functionality rather than introducing unwanted behavior (e.g., null pointer exceptions)—also, reduced functionality is generally easier to detect than potential null pointer exceptions.

In the remainder of this section we describe in detail the changes that were made to JML in adopting the non-null-by-default semantics, as well as the changes made in preparation for the shift to supporting non-null types.

3.1 Declarations

In JML, a declaration can be any one of the following

- *field* declaration of a class or interface. In addition to normal (i.e. Java) fields, JML also supports specification only model and ghost fields [32, §2.2].
- *method* or constructor declaration, though only the former is of interest under the current discussion. These declarations also come in two forms: normal and model.
- method or constructor *parameter* declaration (either for normal or model methods and constructors).
- *bound variable* as declared in the following (JML specific extensions to Java `boolean` expressions):
    - quantified expressions built using `\forall` and `\exists`, and generalized quantifiers like, e.g. `\max`, `\min`, `\sum`;
    - set comprehension expressions;

  or a *bound variable* which can be optionally declared as a part of a method contract in
    - `forall` clauses,
    - `old` clauses, and finally
- *local variable* defined in the body of a (normal or model) method or constructor. There are two kinds of local variable declaration: normal and ghost.

The non-null default applies to all of these kinds of declaration *except* for local variables. We exclude local variables here, because their nullity can be automatically inferred. To do otherwise would increase the annotation burden on developers, who would be required to add nullable modifiers unnecessarily.

3.2 Inheritance and overriding methods

Table 2. Rules for method return types (differing nullity)

Method	**RETURN type declared as …**	
supertype	non-null	nullable
subtype	nullable	non-null
New JML semantics	ERROR: contra-variance prohibited	OK, covariance supported (Java 5)
Former jmlc behavior	like the newly proposed semantics	OK, but method **can return null**
Current ESC/Java2	no error, assumes subtype is non-null	like the newly proposed semantics

Table 3. Rules for method parameter types (differing nullity)

Method	**PARAMETER type declared as …**	
supertype	non-null	nullable
subtype	nullable	non-null
New JML semantics	ERROR: contra-variance prohibited	ERROR: co-variance prohibited
Former jmlc behavior	OK; parameter can be null	OK, but parameter **can be null**
Current ESC/Java2	OK; parameter can be null	Caution; parameter can be null

One of the main changes made to JML in the anticipated switch to non-null types has to do with the restrictions imposed on the variability in nullity modifiers for overriding methods and their parameters.

Of course, an overriding method is permitted to have the same nullity modifiers as the corresponding method in its supertype(s). For those cases where they are different, we propose that the variance typing rules of Java 5 be followed. Hence, an overriding method can be declared non-null even if the corresponding supertype method is declared nullable, see Table 2. This corresponds to covariance in the method return type which is supported as of Java 5 [33, §8.4.8]. In the case of the parameters of overriding methods, like Java 5, invariance is enforced—see Table 3. For the purpose of comparison, the tables also show how the JML compiler (used to), and ESC/Java2 (still) behave when faced with differing nullity modifiers.

Notice how the JML compiler used to support what appeared to be a form of covariance in return types, when in fact, it permitted the overriding method to return null. To understand why, consider the specification of the overriding method `B.m()` given in Figure 1(a). The contribution of `B.m()`'s `non_null` modifier, when treated as an annotation, is illustrated in Figure 1(b), which shows the explicit, partially desugared specification of `B.m()`. Notice how the specification case of `m()` from class `B` gets extended with an extra constraint of the form `\result != null` whereas, the specification case contributed by the class `A` is unaffected [34, §3]. Hence, a call to `B.m()` with a positive argument is permitted to return null.

Under the new semantics, we interpret the non-null modifier as if it were a *type* modifier. Hence, having a non-null return *type*, `B.m()` is prohibited from returning null. This is, in effect equivalent to adding "`\result != null`" to *all* specification cases as is illustrated in Figure 1(c). While this may seem to violate the principle of behavioral subtyping, it does not. Behavioral subtyping is preserved. On the other hand, the use of nullity covariance *can* result in a specification that is *unsatisfiable* in those situations where a parent specification case explicitly *prescribes* a null result under some circumstances. But this is no different from the general case: extending the specification of an overriding method can always give rise to an unsatisfiable method contract. Hence, in general, establishing satisfiability is a proof obligation to be discharged by the specifier.

3.3 Specification refinement

JML also supports specification refinement chains. This feature allows specifications to be developed and presented incrementally [32, §16]. An example of a specification refinement is given in Figure 2. It illustrates how the methods in a `.java` file can be

```
public class A {
  //@ requires i >= 0;
  //@ ensures Qa;
  /*@ nullable @*/ String m(int i) { … }
}

public class B extends A {
  /*@ also
    @   requires i <= 0;
    @   ensures Qb;
    @*/
  /*@ non_null @*/ String m(int i) { … }
}
```

(a) JML specification of classes A and B.

```
public class B extends A {
  /*@   requires i >= 0;
    @   ensures Qa;
    @ also
    @   requires i <= 0;
    @   ensures \result != null;
    @   ensures Qb;
    @*/
  /*@ non_null @*/ String m(int i) { … }
}
```

(b) Partial desugaring of `B.m()`'s specification (old semantics)

```
public class B extends A {
  /*@   requires i >= 0;
    @   ensures \result != null;
    @   ensures Qa;
    @ also
    @   requires i <= 0;
    @   ensures \result != null;
    @   ensures Qb;
    @*/
  /*@ non_null @*/ String m(int i) { … }
}
```

(c) Effective "desugaring" of `non_null` (new semantics)

Figure 1. Nullity covariance in method return types

```
public class C {
    //@ requires i > j;
    public int m(int i, int j) {
     return i-j;
    }
}
```
(a) JML specification of class C as given in `C.java`.

```
//@ refines "C.java";
public class C {
    //@ also
    //@   requires i > j;
    //@   ensures \result == i - j;
    public int m(int i, int j);
}
```
(b) File `C.refines-java`, refining `C.java`.

Figure 2. Example of specification refinement

annotated with preconditions only. A file offering a refinement of method specifications (in this case, full method contracts with pre- *and* post-conditions) can be provided separately.

While a method specification may appear in two or more files that are part of a refinement chain, JML generally constrains the method signature to be *exactly* the same in all files. This rule has been extended to apply to the nullity of method return types and method parameters as well.

3.4 Arrays

In Java, multidimensional arrays are realized as arrays of (references to) array objects. In JML, applying a non-null modifier, explicitly or implicitly, to an array declaration constrains the

- declared array variable,
- array elements (if the elements are of a reference type), and
- intermediate sub-arrays

to be non-null. For example, given the following fields (implicitly) declared non-null

```
Integer vector[];
ComplexNumber matrix[][];
```

the constraints imposed on the fields are equivalent to the following:

```
/*@ invariant vector!= null &&
  @   (\forall int i; 0 <= i && i < vector.length; vector[i] != null);
  @
  @ invariant matrix != null &&
  @   (\forall int i; 0 <= i && i < matrix.length; matrix[i] != null &&
  @     (\forall int j;
  @              0 <= j && j < matrix[i].length; matrix[i][j] != null);
  @*/
```

For those infrequent cases when a designer wishes to constrain some dimensions and not others, he or she would have to declare the array as nullable and use an invariant to express the special constraint; e.g.

```
/*@ nullable @*/ ComplexNumber matrix[][];
/*@ invariant matrix != null ==>
  @  (\forall int i; 0 <= i && i <= matrix.length; matrix[i] != null);
  @*/
```

declares `matrix` to be nullable, but if it is non-null, then its sub-arrays must be non-null; no constraints are placed on the `ComplexNumber` elements of the matrix and hence, they could be null.

JML's `\nonnullelements()` operator can be used to assert that an array and its elements (in the first dimension only) are non-null. For example `\nonnullelements(matrix)` is equivalent to [32, §11.4]:

```
matrix!= null &&
(\forall int i; 0 <= i && i < matrix.length; matrix [i] != null)
```

3.5 Migrating projects to the new default

3.5.1 Global tool settings

JML users not wishing to immediately modify their code base, can make use of the command-line options (or settings) of the various JML tools to revert to the old nullable-by-default semantics.

3.5.2 Module-scoped modifiers

To allow the gradual transitioning of project files, JML has two module (i.e., class or interface) scoped declaration modifiers named `nullable_by_default`, and `non_null_by_default`. Applying the first of these modifiers to a module enables developers to recover the nullable-by-default semantics; i.e., all reference type declarations in the module that are not explicitly declared non-null are interpreted as nullable. Note that the scope of the `nullable_by_default` modifier is strictly the module to which it is applied; hence, it is not inherited by subclasses. Such a convention guarantees that readers will always have an explicit visual cue at the start of each module to warn them that the given module still adheres to the nullable-by-default semantics.

The `non_null_by_default` modifier can be used as an optional visual cue to JML newcomers, and is only necessary when using the tools (e.g., `jmlc`) if the non-null-by-default semantics is disabled.

3.5.3 Script to add nullable_by_default

In addition to these module-scoped modifiers, the JML distribution includes a script that enables JML users to add the `nullable_by_default` modifier to all the classes and interfaces of their project source files. This enables the use of JML tools with the new default semantics enabled, and this, without any other changes to the project files. Then, gradually, as needed, files can be reviewed and updated one-by-one to conform to the new default by

- adding `nullable` modifiers,

- optionally removing explicit `non_null` modifiers, and finally,
- removing the `nullable_by_default` modifier.

As JML tool developers, this is the process which we have been following in our gradual migration of the thousands of JML annotated source files which are part of our tool and case study repositories. Of course, such a porting effort also drives home the importance of adopting the right default semantics as early as possible. This leads us to our next topic of discussion.

4 Non-Null by Default in Java

4.1 Java annotations

The ideal solution would be for the Java language standard to be enhanced to support non-null types. There is in fact an official Request For Enhancement (RFE) for this purpose [35] which, incidentally, we encourage supportive readers to vote for. (Actually, we also strongly encourage votes for the DBC RFE [36] which currently figures among the top three RFEs.) Since there is no apparent plan for the inclusion of such a feature in the foreseeable future, the next best solution is to make use of Java 5 annotations.

Based on our approach to the support of nullity in JML we propose that

- At a minimum, there should be concerted effort to standardize the names of nullity annotations. The most naturally readable modifier names (when read in the context of a declaration) are `@NonNull` and `@Nullable`. Of course, specialized annotations like `@CheckForNull`, for use by specific tools, would be retained.
- A definition of module-scoped annotations (for classes and interfaces) used to control the default interpretation of declarations of reference types inside a module. The annotations `@NonNull` or `@Nullable` could be used for this purpose as well.

Unfortunately, the Java 5 annotations feature cannot be used to address the need for a language- (or even project-) wide default nullity semantics. Project meta-data as provided by e.g., property files, is a promising interim solution, though unfortunately, it is unlikely to be universally adopted by Java tool vendors.

4.2 JML

JML currently offers the most complete support for non-null because, in addition to the benefits of its non-null-by-default semantics:

- JML is “non-intrusive”: as was mentioned in Section 2.4.5, JML annotated source files can be compiled with standard Java compilers because all JML annotations are contained inside stylized comments.
- The distribution of JML comes with specifications, and hence nullity annotations, for standard java packages (like `java.*` including `io`, `lang`, `util`, `sql`) among others (e.g. `javax.servlet`).
- JML has comprehensive tool coverage.

This means that designers eager to experiment with the use of tools supporting non-null annotations, and the benefit of a non-null-by-default semantics, can start using JML along with its most popular tools—the JML compiler (jmlc), ESC/Java2, JmlUnit and JmlDoc. Note that ESC/Java2 has yet to be enhanced to support the new non-null-by-default semantics, but it does correctly interpret (explicit) non-null annotations.

We are not claiming that JML and its tools are a panacea. The language design and semantics are still active subjects of research [37] though a stable core syntax and semantics (JML Level 0) have recently been delineated [32]. Among the top JML request-for-enhancements are:

- Java 5 support (it is currently at 1.4),
- tool integration,
- increased efficiency of the JML compiler (for run-time checking).

The first two are being addressed by various research groups (most notably, some MOBIUS research teams [38, 39] and our DSRG [40]) and the last point has been addressed to some extent by recent work on the JML compiler [41].

5 Conclusion and Future work

Our survey provides a comprehensive coverage of languages, extensions and tools supporting non-null types or annotations. While having non-null annotations or modifiers is useful, their use in practice is more work than it should be since a recent study has shown that the majority of declarations of reference types are meant to be non-null in Java. Hence, before too much code is written under the nullable-by-default semantics, we recommend that Java be adapted, or at least a *standard* non-null annotation-based extension be defined, in which declarations are interpreted as non-null by default. In the meantime, we propose use of JML, recently adapted to conform to the new default. While the semantics for JML described in this paper has been fully implemented in the JML compiler, the implementation is still based on nullity annotations. Work has started on a switch to non-null *types*, guided by the work of Fähndrich and Leino [27]. Following this, we plan to work on adapting ESC/Java2 to the new semantics.

Acknowledgments

We are thankful to Gary Leavens for discussions concerning the new nullity semantics during its elaboration, and to the referees for their careful review of the paper. Support for the research described in this paper was provided by NSERC under grant 261573-03.

References

[1] D. Evans, "Annotation-Assisted Lightweight Static Checking," in *First International Workshop on Automated Program Analysis, Testing and Verification*, 2000.

[2] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs," *SIGSOFT Software Engineering Notes*, vol. 31, pp. 13-19, 2006.

[3] P. Chalin and F. Rioux, "Non-null References by Default in the Java Modeling Language," in *Workshop on the Specification and Verification of Component-Based Systems (SAVCBS'05)*. Lisbon, Portugal, 2005.
[4] D. Flanagan, *Java in a Nutshell: A Desktop Quick Reference*: O'Reilly, 1996.
[5] L. C. Paulson, *ML for the Working Programmer*: Cambridge University Press, 1991.
[6] INRIA, "Pointers in Caml," in *Caml Documentation, Specific Guides*, 2006.
[7] D. Evans, "Static Detection of Dynamic Memory Errors," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. Philadelphia, Pennsylvania, United States: ACM Press, 1996.
[8] D. Evans, "Splint User Manual," Secure Programming Group, University of Virginia June 5 2003.
[9] J. V. Guttag and J. J. Horning, *Larch: Languages and Tools for Formal Specification*: Springer-Verlag, 1993.
[10] D. Evans, "Using Specifications to Check Source Code," MIT, MIT/LCS/TR 628, June 1994.
[11] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, vol. 19, pp. 42-51, 2002.
[12] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *USENIX Annual Technical Conference*. Monterey, CA, 2002.
[13] R. Stallman, "Using the GNU Compiler Collection (GCC): GCC Version 4.1.0," Free Software Foundation 2005.
[14] ECMA International, "Eiffel Analysis, Design and Programming Language," ECMA-367, June 2005.
[15] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview," presented at International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004), Marseille, France, 2004.
[16] R. DeLine and K. R. M. Leino, "BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs," Microsoft Research, Technical Report 2005.
[17] M. Fähndrich and K. R. M. Leino, "Non-Null Types in an Object-Oriented Language," in *Workshop on Formal Techniques for Java-like Languages*. Malaga, Spain, 2002.
[18] M. Barnett, R. DeLine, B. Jacobs, M. Faehndrich, K. R. M. Leino, W. Schulte, and H. Venter, "The Spec# Programming System: Challenges and Directions," in *International Conference on Verified Software: Theories, Tools, Experiments*. Zürich, Switzerland, 2005.
[19] D. Bonniot, "The Nice programming language," 2005.
[20] D. Bonniot, "Using kinds to type partially-polymorphic methods," *Electronic Notes in Theoretical Computer Science*, vol. 75, pp. 1-20, 2003.
[21] D. Bonniot, "Type safety in Nice: Why programs written in Nice have less bugs," 2005.
[22] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," *ACM SIGPLAN Notices*, vol. 39, pp. 92-106, 2004.
[23] K. Lea, "Nully," https://nully.dev.java.net/, 2005.
[24] JetBrains, "Nullable How-To," in *IntelliJ IDEA 5.x Developer Documentation*: JetBrains, 2006.
[25] G. Hedin and E. Magnusson, "JastAdd--an aspect-oriented compiler construction system," *Science of Computer Programming*, vol. 47, pp. 37-58, 2003.
[26] T. Ekman and G. Hedin, "Modular implementation of non-null types for Java," Lund University, 2005.
[27] M. Fähndrich and K. R. M. Leino, "Declaring and Checking Non-null Types in an Object-Oriented Language," in *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA'03*: ACM Press, 2003, pp. 302-312.
[28] Eclipse Foundation, "JDT Core R3.2.x Release Plan," http://www.eclipse.org/jdt/core/r3.2, 2006.
[29] G. T. Leavens, "The Java Modeling Language (JML)," http://www.jmlspecs.org, 2006.
[30] G. T. Leavens and Y. Cheon, "Design by Contract with JML," Draft paper 2005.
[31] 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," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, pp. 212-232, 2005.
[32] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin, "JML Reference Manual," 2006.
[33] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed: Addison-Wesley Professional, 2005.
[34] A. D. Raghavan and G. T. Leavens, "Desugaring JML Method Specifications," Department of Computer Science, Iowa State University TR #00-03e, May 2005.
[35] Sun Developer Network, "Add Nice Option types to Java to prevent NullPointerExceptions (Bug ID: 5030232)," 2004.
[36] Sun Developer Network, "Support For 'Design by Contract', beyond 'a simple assertion facility' (Bug ID: 4449383)," 2001.
[37] G. T. Leavens and C. Clifton, "Lessons from the JML Project," in *International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*. Zürich, Switzerland, 2005.
[38] J. R. Kiniry, P. Chalin, and C. Hurlin, "Integrating Static Checking and Interactive Verification: Supporting Multiple Theories and Provers in Verification," in *International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*. Zürich, Switzerland, 2005.
[39] MOBIUS, "The MOBIUS project," http://mobius.inria.fr/.
[40] P. Chalin, "Dependable Software Research Group (DSRG) Web Site," Department of Computer Science and Software Engineering, Concordia University. http://www.dsrg.org, 2006.
[41] F. Rioux, "Effective and Efficient Design by Contract for Java," in *Dependable Software Research Group (DSRG), Faculty of Engineering and Computer Science, Department of Computer Science and Software Engineering*. Montréal, Québec: Concordia University, 2006.