

Non-null References by Default in the Java Modeling Language

Patrice Chalin, Frédéric Rioux

www.dsrg.org

ENCS-CSE Technical Report 2005-004, revision 3.2
December, 2005

Dependable Software Research Group
Department of Computer Science and Software Engineering
Faculty of Engineering and Computer Science
Concordia University

Keywords: Contracts, Java Modeling Language, JML, reference types, non-null references

D.2.4 [**Software Engineering**]: Software/Program Verification—programming by contract; D.3.3 [**Programming Languages**]; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

Revision History

- June 20, 2005. First release.
- July, 2005. Internal release.
 - Section 2.2.2: of course the assertions can be atomic predicates of the given forms.
- August, 2005. Updates based on SAVCBS referee comments.
- October, 2005. Updates due to change in keyword names: `null` → `nullable`; `non_null_ref_by_default` → `non_null_by_default`; `null_ref_by_default` → `nullable_by_default`.
- November/December 2005. Recording latest design decisions as agreed upon in post-VSTTE discussions.

Table of Contents

1. Introduction.....	1
2. Study.....	2
2.1 Metrics.....	2
2.2 Counting non-null declarations.....	2
2.2.1 JML core and syntactic sugar.....	2
2.2.2 General rules.....	2
2.2.3 Fields.....	2
2.2.4 Method return types.....	2
2.2.5 Method parameters.....	3
2.3 Statistics tool.....	3
2.4 Case study subjects.....	3
2.5 Procedure.....	3
2.5.1 Selection of sample files.....	3
2.5.2 Annotating the sample files.....	4
2.6 Threats to validity.....	4
2.6.1 Internal validity.....	4
2.6.2 External validity.....	4
3. Study results.....	4
4. Adapting JML.....	5
4.1 Non-null by default.....	5
4.2 Non-nullity for Arrays.....	6
4.3 Migration path.....	6
4.4 Implementation.....	6
4.5 Upholding JML design goals.....	6
5. Related work.....	7
5.1 Nullity annotations.....	7
5.2 Non-null types.....	7
5.2.1 Nice.....	7
5.2.2 Eiffel.....	7
5.2.3 Spec#.....	7
6. Conclusion.....	7
7. Appendix.....	9
7.1 JML core and syntactic sugar.....	10
7.2 Fields.....	10
7.3 Method return types.....	10
7.4 Method parameters.....	10

Non-null References by Default in the Java Modeling Language

Patrice Chalin, Frédéric Rioux

Dept. of Computer Science and Software Engineering,
Dependable Software Research Group, Concordia University
1455 de Maisonneuve Blvd. West, Montréal
Québec, Canada, H3G 1M8
{chalin,f_rioux}@cse.concordia.ca

ABSTRACT

Based on our experiences and those of our peers, we hypothesized that in Java code, the majority of declarations that are of reference types are meant to be non-null. Unfortunately, the Java Modeling Language (JML), like most interface specification and object-oriented programming languages, assumes that such declarations are possibly-null by default. As a consequence, developers need to write specifications that are more verbose than necessary in order to accurately document their module interfaces. In practice, this results in module interfaces being left incompletely and inaccurately specified. In this paper we present the results of a study that confirms our hypothesis. Hence, we propose an adaptation to JML that preserves its language design goals and that allows developers to specify that declarations of reference types are to be interpreted as non-null by default. We explain how this default is safer and results in less writing on the part of specifiers than null-by-default. The paper also reports on an implementation of the proposal in some of the JML tools.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—programming by contract; D.3.3 [Programming Languages]; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms

Documentation, Design, Languages, Theory, Verification.

Keywords

Contracts, Java Modeling Language, JML, reference types, non-null references.

1. INTRODUCTION

Null pointer exceptions are among the most common faults raised by components written in mainstream imperative languages like Java. Increasingly developers are able to make use of tools that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

An earlier revision of this paper was presented at *SAVCBS'05*, September 5–6, 2005, Lisbon, Portugal.
Copyright 2005 Patrice Chalin, ACM 1-58113-000-0/00/0004...\$5.00.

can detect possible null dereferences (among other things) by means of static analysis of component source. Unfortunately, such tools can only perform minimal analysis when provided with code alone. On the other hand, given that components and their support libraries are supplemented with appropriate specifications, then the tools are able to detect a large proportion of potential null pointer dereferences. The Java Modeling Language (JML) is one of the most popular behavioral interface specification languages for Java [11, 12]. ESC/Java2 is an extended static checker for Java that uses JML as an interface specification language [4].

While writing Java programs and their JML specifications, it has been our experience, and those of peers, that we generally want declarations of reference types to be non-null. Unfortunately JML, like most interface specification and mainstream object-oriented programming languages, assumes that by default declarations can be null. As a result, specifiers must explicitly constrain such declarations to be non-null either by annotating the declarations with `/*@non_null @*/` or by adding constraints of the form `o != null` to class invariants and/or method contracts. Since most developers tend to write specifications penuriously, in practice this results in module interfaces being left incompletely and inaccurately specified.

In this paper we present the results of a study that confirms the hypothesis that:

In Java programs, the majority of declarations that are of reference types are meant to be non-null, based on design intent.

For this study we sampled over 150 KLOC out of 450 KLOC of Java source. To our knowledge, this is the first formal empirical study of this kind—though anecdotal evidence has been mentioned elsewhere, e.g. [7, 8]. In light of our study results, we propose that JML be adapted to allow developers to specify that declarations of reference types are to be interpreted as *non-null* by default.

The study method and study results are presented in the next two sections. Our proposal to adapt JML to support non-null by default is presented in Section 4 along with a discussion of the way in which the proposal upholds JML's design goals. We offer a discussion of related work and conclude in Sections 4.5 and 6 respectively.

```

public abstract class Greeting
{
    private /*@ spec_public non_null */ String nm;

    /*@ public normal_behavior
       @ requires !aNm.equals("");
       @ modifies nm;
       @ ensures nm == aNm;
    */
    public void set(/*@ non_null */ String aNm) {
        nm = aNm;
    }

    /*@ ensures \result.equals(greeting()+nm);
    public /*@ pure non_null */ String welcome() {
        return greeting() + nm;
    }

    /*@ ensures \result != null;
    /*@      && !\result.equals("");
    public abstract /*@ pure */ String greeting();
}

```

(a) Greeting class

```

public class FrenchGreeting extends Greeting
{
    // constructor omitted

    /*@ also
       /*@ ensures \result.equals("Bonjour ");
    public /*@pure non_null*/ String greeting() {
        return "Bonjour ";
    }
}

```

(b) FrenchGreeting class

Figure 1. Sample class specifications

2. STUDY

2.1 Metrics

Reference types can be used in the declaration of local variables, fields, method (return types) and parameters. In our study we considered all of these types of declaration except for local variables since the non-null annotation of local variables is not yet fully supported in JML. Unless specified otherwise, we shall use the term *declaration* in the remainder of this article to be a declaration other than that of a local variable.

We have two principal metrics in this study, both of which shall be measured on a per file basis:

- d , is a measure of the number of declarations that are of a reference type, and
- m is a measure of the number of declarations specified to be non-null (hence $m \leq d$).

The main statistic of interest, x , will be a measure of the proportion of reference type declarations that are non-null, i.e. m/d . In the next section we explain how m is computed.

2.2 Counting non-null declarations

2.2.1 JML core and syntactic sugar

Like many languages, the definition of JML consists of a core (that offers basic syntactic constructs) supplemented with syntactic sugar that makes the language more practical and pleasant to use. As is often done in these situations, the semantics of JML is defined in terms of a desugaring procedure (that describes how to translate arbitrary specifications into the JML core), and a semantics of the core [13, 18].

As such, it is much simpler to accurately describe how to count non-null declarations relative to the core JML. Unfortunately, such an account would seem foreign to most. Hence, in this paper we have chosen to provide an informal description of counting non-null declarations that is based on “sugared” JML. We refer readers interested in the full details to [5].

2.2.2 General rules

Given a declaration $T o$, where T is a reference type, we can constrain o to be non-null either explicitly or implicitly. We do so explicitly by annotating the declaration with `/*@ non_null */` as is illustrated in Figure 1(a). Notice that the field `nm`, the method `welcome()` and the parameter `aNm` of the method `set()` are explicitly declared non-null.

Generally speaking, we consider that o is implicitly constrained to be non-null if an *appropriate* assertion is of any one of the following forms, or it contains a conjunct of any one of the following forms:

- `o != null`,
- `o instanceof C`,
- `\fresh(o)` which states that o is a reference to a newly allocated object (hence this can only be used in an `ensures` clause), or
- `\nonnull elements(o)`, when o is an array reference.

For example, the `greeting()` method of Figure 1(a) is considered to be implicitly declared non-null because the `ensures` clause constrains the method result to be non-null. Next, for each kind of declaration, we describe the circumstances under which we consider declarations of the given kind to be implicitly declared non-null.

2.2.3 Fields

A non-static (respectively, static) field can be implicitly declared non-null if a non-static (static) class invariant contains a conjunct of the form given in Section 2.2.2. For example, given the declarations of Figure 2, we would count `o1`, `o2` and `a` as non-null but not `o3` (because the `o3 != null` term is an argument to a disjunction rather than a conjunction).

2.2.4 Method return types

```

/*@ non_null */ Object o1;
Object o2;
/*@ invariant o2 != null;
int i;
Object o3;
/*@ invariant i > 0 || o3 != null;
Object a[];
/*@ invariant \nonnull elements(a);

```

Figure 2. Sample field declarations, some non-null

```

/*@ normal_behavior
@   requires i == 0
@   ensures  \result != null
@           && \result.equals("zero");
@ also
@ normal_behavior
@   requires i > 0;
@   ensures  \result != null
@           && \result.equals("positive");
@ also
@ exceptional_behavior
@   requires i < 0;
@   signals(Exception e) true;
@*/
/*@ pure @*/ String m(int i) {
    ...
}

```

Figure 3. Method specification cases separated using ‘also’

The pseudo variable `\result` is used in `ensures` clauses to represent the value returned by a method. A static method or a non-overriding non-static method can be implicitly declared as non-null by constraining `\result` to be non-null in an `ensures` clause as is done for the `greeting()` method of the `Greeting` class—Figure 1(a).

A JML method specification can be given as a list of cases (having different preconditions) separated by the keyword `also` as is illustrated in Figure 3. In such situations, the method can be counted as non-null if and only if: the method is explicitly declared as non-null or, it is implicitly declared as non-null in every `normal_behavior` specification case (and every `behavior` specification case—not discussed here—for which the `ensures` clause is neither false nor `\not_specified`).

Due to behavioral subtyping, the case of overriding methods is slightly more complicated. In JML, an overriding method like `FrenchGreeting.greeting()` of Figure 1(b) must respect the method specifications of its ancestors—which in this case consists only of one method, `Greeting.greeting()`. As a reminder to readers, all overriding method specifications must start with the keyword `also`. Thus, an overriding method `m` in a class `C` can be counted as non-null if and only if `m` is constrained to be non-null in `C`, as well as in all ancestor classes of `C` where `m` is explicitly declared.

2.2.5 Method parameters

The case for method parameters is similar to that for method return types. That is, a parameter of a static or non-overriding method is considered non-null if it is constrained as such in a `requires` clause. On the other hand, a parameter of an overriding method can be counted as non-null if and only if it is declared as non-null in the given class and all ancestor classes.

2.3 Statistics tool

In order to gather statistics concerning non-null declarations, the Iowa State University (ISU) JML checker was extended. The tool uses heuristics similar to those described in the previous section. The metrics gathered are conservative (i.e. when given the choice between soundness and completeness, the tool opts for soundness). The tool gathers data for normal, ghost, and model references. It also warns the user of inconsistent specifications (e.g. pre- or post-conditions trivially simplifying to false in all method specification cases).

Overall Project →	ISU Tools	ESC Tools	SoenEA	Koa	Total
# of files	831	455	52	459	1797
LOC (K)	243	124	3	87	457
SLOC (K)	140	75	2	62	278
Project subsys. →	JML Checker	ESC/Java2	SoenEA	Koa Tally Subsys.	Total
# of files	217	216	52	29	514
LOC (K)	86	63	3	10	161
SLOC (K)	58	41	2	4	104

Table 1 General statistics of study subjects

2.4 Case study subjects

It was actually our work on an ESC/Java2 case study in the specification and verification of a small web-based enterprise application framework named SoenEA [19] that provided the final impetus to initiate the study reported in this paper. Hence, we chose SoenEA as one of our case study subjects. As our three other subjects we chose the ISU JML checker, the ESC/Java2 tool and the tallying subsystem of Koa, a recently developed Dutch internet voting application¹. We chose these projects because:

- We believe that they are representative of typical designs in Java applications and that they are of a non-trivial size (numbers will be given shortly).
- We were familiar with the source code (and/or had peers that were) and hence expected that it would be easier to write accurate JML specifications for it. Too much effort would have been required to study and understand unfamiliar and sizeable projects in sufficient detail to be able to write correct specifications².
- The project sources are freely available to be reviewed by others who may want to validate our specification efforts.
- The sources were at least partly annotated with JML specifications; hence we would not be starting entirely from scratch.

Aside from SoenEA, the other study subjects are actually an integral (and dependant) part of a larger project. For example, the JML checker is only one of the tools provided as part of the ISU tool suite—others include JmlUnit and the JML run-time assertion checker compiler.

Table 1 provides the number of files, lines-of-code (LOC) and source-lines-of-code (SLOC) for our study subjects as well as the projects that they are subcomponents of. Overall the source for all four projects consists of 457 KLOC (278 KSLOC) from over almost 1800 Java source files. Our study subjects account for 161 KLOC from over 500 files.

2.5 Procedure

2.5.1 Selection of sample files

With the study projects identified, our objective was to add JML specifications to all of the source files, or, if there were too many, a randomly chosen sample of files. In the later case, we fixed our

¹ Koa was used, e.g., in the 2004 European parliamentary elections.

² Particularly since projects often lack detailed design documentation.

sample size at 35 (as sample sizes of 30 or more are generally considered “sufficiently large”). Our random sampling for a given project was created by first listing the N project files in alphabetical order, generating 35 random numbers in the range $1..N$, and then choosing the corresponding files.

2.5.2 Annotating the sample files

We then added to the selected files JML specifications consisting essentially of constraints on declarations of reference types, where appropriate. In most situations we added `non_null` declaration modifiers.

An example of a field declaration that we would constrain to be non-null is:

```
static final String MSG1 = "abc";
```

Similarly we would conservatively annotate constructor and method return types as well as parameters based on our understanding of the software applications. As an example, consider the following method:

```
String m(int a[]) {
    String result = "";
    for(int i = 0; i < a.length; i++) {
        result += a[i] + " ";
    }
    return result;
}
```

In the absence of any specification or documentation for such a method we would assume that the designer intended `a` to be non-null (since there is no test for nullity and yet the `length` field of `a` is used). We can also deduce that the method will always return a non-null String.

As was previously explained, constraining the method return type or parameters for an inherited method requires adding annotations to the method’s class as well as to all ancestors of the class in which the method is declared. This was particularly evident in the case of the JML checker code since the class hierarchy is up to 6 levels of inheritance for some of files that we worked on (e.g. `JmlCompiAtionUnit`).

2.6 Threats to validity

2.6.1 Internal validity

We see two threats to internal validity. Firstly, in adding non-null constraints to the sample files we may have been excessive. As was discussed in the previous section, we chose to be conservative in our specification exercise. Furthermore, the code samples (both before the exercise and after) are available for peer review. The JML checker is accessible from SourceForge (sourceforge.net); ESC/Java2 and Koa are available from Joseph Kiniry’s GForge site (sort.ucd.ie) and SoenEA is available from the authors³.

Secondly, our statistics tool may have incorrectly counted a declaration as being non-null. Again, as was previously explained, we chose soundness over completeness during our design of the tool. The tool source (which is part of the ISU JML tool suite) is also available via anonymous CVS for peer review.

2.6.2 External validity

Will we be able to draw general conclusions from our study results? The main question is: can our sample of source files be

³ At the time of writing, the updated Koa source has not yet been committed to GForge; it is available from the authors.

	JML Checker	ESC/Java2	SoenEA	Koa TS	Sum or Average
n	35	35	41	29	140
N	217	216	41	29	503
$\sum d_i$	376	807	231	564	1978
$\sum m_i$	210	499	177	368	1254
$\sum d_i / \sum m_i$	56%	62%	77%	65%	63%
mean (\bar{x})	59%	60%	72%	64%	64%
std.dev.(s)	0.24	0.31	0.37	0.32	-
E ($\alpha=5\%$)	7.4%	9.3%	-	-	-
μ_{min}	52%	51%	72%	64%	60%

Table 2. Distribution of the number of declarations of reference types

taken as representative of typical Java applications? There are two aspects that can be considered here: the design style used in the samples, and the application domains.

Modern object-oriented programming best-practices promote e.g., a disciplined (i.e. moderate) use of `null` with the Null Object pattern recommended as an alternative [9]. Of course, not all Java code is written following recommended best practices; hence our sample applications should include such “non-OO-style” code. This is the case for some of the ESC/Java2 core classes (which were designed quite early in the project history); e.g. some of the classes declare their fields as public (a practice that is discouraged) rather than using getters and setters, making it more difficult to ascertain if a field was intended to be non-null. Also, the class hierarchy is very flat, with some classes resembling a module in the traditional sense (i.e. a collection of static methods) more than a class.

With a four sample set, it is impossible to claim that we have coverage in application domains, but we note that the SoenEA sample represents one of the most popular uses of Java—namely, servlet-based web applications.

3. STUDY RESULTS

A summary of the output of the non-null statistics tool run on our study samples (after having completed our specification exercise) is given in Table 2. As is usually done, the number of files in each sample is denoted by n and the population size by N . Note that for SoenEA, 11 of the files did not contain any declarations of reference types, hence the population size is $41 = 52 - 11$; the reason that we exclude such files from our sample is because it is not possible to compute the proportion of non-null references for files without any declarations of reference types. We see that the total number of declarations that are of a reference type (d) across all samples is 1978. The total number of such declarations constrained to be non-null (m) is 1254. The proportion of non-null references across all files is 63%.

We also computed the mean, \bar{x} , of the proportion of non-null declarations on a per file basis ($x_i = d_i / m_i$). The mean ranges from 59% for the JML checker sample, to 72% for the SoenEA sample. Also given are the standard deviation (s) and a measure of the maximum error (E) of our sample mean as an estimate for the population mean with a confidence level of $1 - \alpha = 95\%$. Hence we can conclude with 95% certainty that the population means are above 50% in all cases.

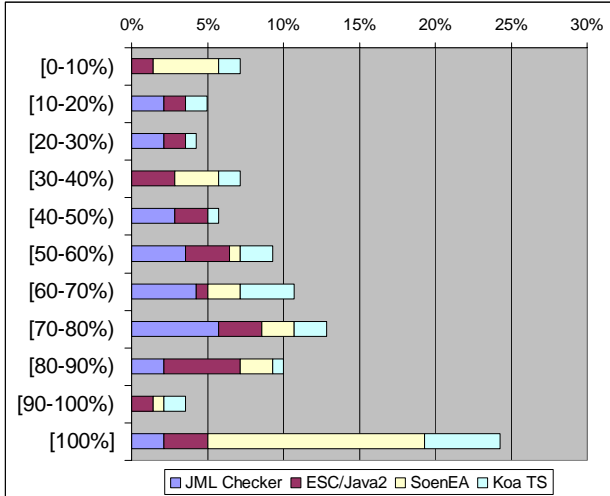


Figure 4. Percentage of files having a value for x (the proportion of non-null declarations) in a given range

It should be noted that for both the JML checker and ESC/Java2 samples, we stopped annotating the files once we had reached a value for μ_{\min} (i.e. $\mu - E$) that was greater than 50% for $\alpha = 5\%$. Hence, it is quite likely that μ is actually higher for these samples. In the case of SoenEA we essentially completed the annotation exercise for all files, and as a result μ is 72%.

A distribution of x , the proportion of non-null declarations, is given in Figure 4—following standard notation, $[a, b)$ represents the interval of values v in the range $a \leq v < b$. The bar length represents the percentage of files for which x is in the given range. We see that the checker has no files with an x in the range $[0-10\%)$. On the other hand, SoenEA has the largest proportion of files in this range as well as for $x = 100\%$.

The mean of x by kind of declaration (fields, methods and parameters) for each of the study samples is given in Figure 5. Almost all samples have a mean for parameters that is higher than for methods. The mean of x for fields is much higher in the case of the JML checker possibly because the checker sample had the smallest number of field declarations.

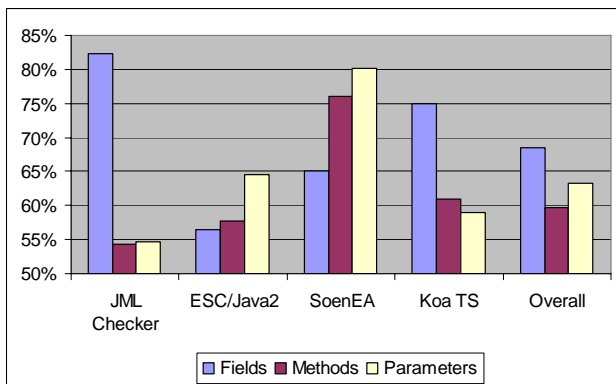


Figure 5. Mean of x , the proportion of non-null declarations, by kind

We believe that the study results support our hypothesis that in Java code, the majority of declarations that are of reference types are meant to be non-null. It is for this reason that we propose a modification to JML as is explained next.

4. ADAPTING JML

4.1 Non-null by default

The study results support the hypothesis that, in general, designers want more than 50% of declarations of reference types to be non-null. Thus, under the current JML semantics, designers must add `/*@ non_null @*/` annotations to the majority of declarations if he or she wants the non-null constraints to be accurately documented. As was remarked in the introduction, since developers tend to write specifications penuriously, in practice this results in module interfaces being left incompletely and inaccurately specified. Thus, module clients might call methods with null arguments when these should be prohibited, resulting in `NullPointerException`'s—one of the most common programming errors.

We propose that JML's semantics be adapted so that all declarations of reference types be implicitly assumed as non-null and that the language be augmented with a `nullable` declaration modifier. Adopting this new default for reference type declarations would allow, on average, over 50% of declarations of reference types in an unannotated source file to be accurately constrained to be non-null. Designers could then gradually add `/*@ nullable @*/` annotations to their specifications as needed. A consequence of forgetting or delaying the addition of `nullable` annotations would, at worst, present a more restrictive interface to a module's clients. This is certainly a safer alternative than the null-by-default semantics. Furthermore, since developers must generally provide special code to handle null, it is best for them to be explicitly informed that a value might be null by the presence of a `nullable` annotation rather than by its absence.

We present in Figure 6 a revised version of the `Greeting` class specification of Figure 1 as it would appear under the new semantics.

```

public abstract class Greeting
{
    private /*@ spec_public */ String nm;

    /*@ public normal_behavior
    @ requires !aNm.equals("");
    @ modifies nm;
    @ ensures nm == aNm;
    */
    public void set(String aNm) {
        nm = aNm;
    }

    /*@ ensures \result.equals(greeting()+nm);
    public /*@ pure */ String welcome() {
        return greeting() + nm;
    }

    /*@ ensures !\result.equals("");
    public abstract
    /*@ pure */ String greeting();
}

```

Figure 6. Greeting specification using the new implicit non-null-by-default semantics

4.2 Non-nullity for Arrays

Under the current definition of JML, the following field declaration

```
/*@ non_null */ Integer vector[];
```

is equivalent to

```
Integer vector[];
/*@ invariant vector != null;
```

Under the newly proposed semantics, `vector`'s elements would be constrained to be non-null as well; i.e.

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

Recall that in Java, multidimensional arrays are realized as arrays of arrays. We propose that when the `non_null` annotation is applied to a declaration of an array of one or more dimensions, then all of the intermediate array “slices” be constrained as non-null.

For example, given fields declared as

```
/*@ non_null */ ComplexNumber matrix[][];
/*@ non_null */ int cube[][][];
```

the expression `matrix[i]` will refer to a (one dimensional) array of `Integers` and `cube[i]` will refer to an array of array's of `ints`. Thus we propose that the `non_null` annotations for `matrix` and `cube` desugar to

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

and

```
cube != null &&
(\forallall int i; 0 <= i && i < cube.length;
 cube[i] != null &&
 (\forallall int j; 0 <= j && j < cube[i].length;
  cube[i][j] != null))
```

respectively.

JML's `\nonnull elements` operator that can be used to assert that an array and its elements (in the first dimension only) are non-null. For example `\nonnull elements(matrix)` is equivalent to ([15], Section 11.4):

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

It may be worth considering a change to the definition of `\nonnull elements` to something compatible with the meaning of `non_null` when applies to array declarations.

4.3 Migration path

Our current JML user base cannot be expected to convert all of their JML annotated source files at the same time. In fact, as JML tool developers, we also find ourselves in need of a gradual migration solution enabling us to convert our thousands of files incrementally.

To this end we introduce a module-scoped declaration modifier named `nullable_by_default`. Adorning a class or interface with this modifier will enable developers to recover the null-by-default semantics. The modifier effectively makes nullable all reference type declarations in the module that are not explicitly declared non-null. Like the `pure` declaration modifier, the scope of `nullable_by_default` is strictly the module that it adorns—i.e. it is not inherited. Such a convention will mean that readers will have an explicit visual cue at the start of each module to warn them that the given module still adheres to the null-by-default semantics.

The `non_null` declaration modifier will continue to be supported though in a future release, tools like the JML checker will likely warn that it, and `nullable_by_default`, are deprecated.

4.4 Implementation

We have implemented the given proposal in the JML checker and the JML Run-time Assertion Checker. Since these tools already supported the notion of possibly-null and non-null declarations, the adaptation was relatively easy (approximately three person-weeks). We expect that the adaptation of other JML tools should be just as straight-forward. The research teams of Joseph Kiniry and the first author are currently implementing the proposal in ESC/Java2.

4.5 Upholding JML design goals

One of the language design goals of JML is to adhere to the semantics of Java to the extent possible. In those situations where JML semantics differ from Java, it should not come as a surprise to Java developers [14]. In our case, the survey results support the idea that developers would like reference type declarations to be non-null by default. Of course, the ideal situation would be for Java to adopt non-null as a default—as has been done in the first official standard release of Eiffel as we describe at further length in the next section.

```

class GenericGreeting : Greeting {
    string! greeting;
    public GenericGreeting(string! n, string! g) {
        base(n);
        // (1)
    }
    // ...
}

```

(a) Partially initialized object (Spec#)

```

class GenericGreeting : Greeting {
    string! greeting;
    public GenericGreeting(string! n, string! g) {
        greeting = g;
        base(n);
        // (1)
    }
    // ...
}

```

(b) Use of initializer (Spec#)

Figure 7. Spec# `GenericGreeting` examples

5. RELATED WORK

5.1 Nullity annotations

Most closely related to our current proposal for JML are the nullity annotations supported by Splint [6]. Splint is a static analysis tool for C programs that evolved out of work on Lclint. Lclint was essentially a type checker for Larch/C, a behavioral interface specification language for C [10]. In Splint, all pointer variables are assumed to be non-null by default, unless adorned with `@null`. Splint can be used to statically detect potential null dereferences.

5.2 Non-null types

In contrast to using assertions or special annotations, some languages have enriched type systems supporting the notion of non-null types. Of the three described here, two are proposing that references be non-null by default.

5.2.1 Nice

The Nice programming language can be seen as an enriched variant of Java supporting parametric types, multi-methods, and contracts among other features [2]. Nice also supports non-null types. 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 [3].

5.2.2 Eiffel

The next major release of the Eiffel programming language [16] will also include support for *attached types* (i.e. non-null types, or non-void types as they might be called in Eiffel) as opposed to *detachable types* (that can be null) [17]. The proposed default for this new release of Eiffel will be attached types. Special consideration has been given to minimizing the migration effort of current Eiffel code.

5.2.3 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 [1]. For reasons of backwards compatibility with C#, a reference type name T refers to possibly null references of type T whereas $T!$ is used to represent non-null references of type T .

The introduction of non-null types naturally complicates the type system and leads to other issues. The main issue has to do with partially initialized objects [7]. Consider the example given in Figure 7(a). At point (1) in the constructor code, the field `greeting` will be null—due to the automatic initialization performed on instance fields. To solve this problem, Spec# allows constructors to provide field initializers as is illustrated in Figure 7(b).

In the case of JML, no such special measures are required since a non-null constraint on a field like `greeting` would be translated into a non-null constraint in a class invariant. Class invariants are not assumed to hold on entry to or during the execution of a constructor body. This is not to claim that JML’s approach is better—especially given the open issues related to the treatment of invariants—but rather than it is an alternative approach.

6. CONCLUSION

In this paper, we report on a novel study of four open projects (totaling over 450 KLOC) taken from various domains of application. The study results support the hypothesis that, by design, the majority of reference type declarations are meant to be non-null in Java.

It would be preferable that Java be adapted so that declarations are interpreted as non-null by default (as will be the case, for example, in the next release of Eiffel). In the meantime, we have suggested an adaptation to JML that would allow specifications to be more concise by interpreting reference types as non-null unless explicitly annotated with the `nullable` declaration modifier. Our proposal results in a safer default since in the absence of declaration annotations, modules simply present stricter interfaces to their clients.

We have implemented our proposal in the JML checker and run-time assertion checker compiler. As future work, we intend to complete the implementation of the proposal in ESC/Java2, and to conduct further studies in an attempt to measure the effectiveness of our new proposed default for reference type declarations.

ACKNOWLEDGMENTS

We are grateful to the anonymous referees for their helpful comments. The authors would like to thank Joseph Kiniry for providing access to, and assistance on the Koa source, and for discussions about Eiffel. We thank Kui Dai for implementing the proposed changes to JML in the RAC, as well as Hao Xi for contributing to the non-null metrics tool. Research support was provided by NSERC of Canada (261573-03) and the Quebec FQRNT (100221).

REFERENCES

- [1] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview." In Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004), Marseille, France, LNCS, vol. 3362, 2004.
- [2] D. Bonniot. *The Nice programming language*, <http://nice.sourceforge.net/>, June 2005.

- [3] D. Bonniot. *Type safety in Nice: Why programs written in Nice have less bugs*, <http://nice.sourceforge.net/safety.html>, June 2005.
- [4] 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)*, 2004.
- [5] P. Chalin and F. Rioux, *Non-null References by Default in the Java Modeling Language*, Dependable Software Research Group, Concordia University, ENCS-CSE TR 2005-004. June, 2005.
- [6] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, vol. 19, no. 1, pp. 42-51, Jan.-Feb., 2002.
- [7] 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.
- [8] C. Flanagan and K. R. M. Leino, "Houdini, an Annotation Assistant for ESC/Java." In *Proceedings of the International Symposium of Formal Methods Europe*, Berlin, Germany, vol. 2021, pp. 500-517, 2001.
- [9] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 1999.
- [10] J. V. Guttag and J. J. Horning, *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
- [11] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: A Notation for Detailed Design," in *Behavioral Specifications of Businesses and Systems*, B. R. Haim Kilov, Ian Simmonds, Ed.: Kluwer, 1999, pp. 175-188.
- [12] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs, "JML: notations and tools supporting detailed design in Java," in *OOPSLA 2000 Companion*, Minneapolis, Minnesota, 2000, pp. 105-106.
- [13] G. T. Leavens, A. L. Baker, and C. Ruby, *Preliminary Design of JML: A Behavioral Interface Specification Language for Java*, Department of Computer Science, Iowa State University TR #98-06-rev27. April, 2005.
- [14] 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*, vol. 55, no. 1-3, pp. 185-208, 2005.
- [15] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry, *JML Reference Manual*, <http://www.cs.iastate.edu/~leavens/JML/jmlrefman/>.
- [16] B. Meyer, *Eiffel: The Language*. Object-Oriented Series. New York. Prentice-Hall, 1991.
- [17] B. Meyer, *Eiffel: The Language*, Draft of future edition, revision 5.00 (June 2005) ed. Unpublished, 2005.
- [18] A. D. Raghavan and G. T. Leavens, *Desugaring JML Method Specifications*, Department of Computer Science, Iowa State University TR #00-03e. May, 2005.
- [19] F. Rioux and P. Chalin, "Improving the Quality of Web-based Enterprise Applications with Extended Static Checking: A Case Study." In *Proceedings of the 1st International Workshop on Automated Specification and*

Verification of Web Sites, Valencia, Spain, Electronic Notes in Theoretical Computer Science, March 14-15, 2005 (to appear).

7. APPENDIX

```
public abstract class Greeting
{
  //@ public model String nm;
  private String _nm; //@ in nm;
  //@ represents nm <- _nm;
  //@ private invariant _nm != null;

  /*@ public behavior
   @ requires aNm != null
   @      && !aNm.equals("");
   @ modifies nm;
   @ ensures nm == aNm;
   @ // ... other clauses omitted
  */
  public void set(String aNm) {
    _nm = aNm;
  }

  // ...

  /*@ public behavior
   @ requires true;
   @ ensures \result != null
   @      && !\result.equals("");
   @ // ... other clauses omitted
  */
  public abstract /*@ pure */ String greeting() { ... }
}
```

(a) Greeting class

```
public class FrenchGreeting extends Greeting
{
  // constructor omitted

  /*@ also public behavior
   @ requires true || true;
   @ ensures
   @   (\old(true) ==>
   @   \result != null
   @   && !\result.equals(""))
   @ && (\old(true) ==>
   @   \result != null
   @   && \result.equals("Bonjour "));
  */
  public /*@ pure */ String greeting() {
    return "Bonjour ";
  }
}
```

(b) FrenchGreeting class

Figure 8. Partly desugared specifications

7.1 JML core and syntactic sugar

As was explained in Section 2.2.1, the definition of JML consists of a core that offers basic syntactic constructs supplemented with syntactic sugar that makes the language more practical and pleasant to use. As is often done in these situations, the semantics of JML is defined in terms of a desugaring procedure (that describes how to translate arbitrary specifications into the JML core), and a semantics of the core.

As an example, we present in

(b) `FrenchGreeting` class

Figure 8. Partly desugared specifications

the result of partially desugaring the `Greeting` and `FrenchGreeting` classes of in Figure 1 (the desugaring is partial since we have chosen not to desugar the pure modifier, since it is not relevant to our presentation).

In the subsections that follow we explain how non-null declarations would be counted for a given a core JML specification. This couples with a description of the desugaring process given elsewhere [13, 18], provides a complete description of how to count declarations that are non-null.

7.2 Fields

The rules for counting fields in the JML core are actually the same as those given in Section 2.2.3.

7.3 Method return types

In its desugared form, a method specification is reduced to a single “behavior” specification block that contains exactly one occurrence of every JML method specification clause. The general form of the `requires` and `ensures` clauses of such a specification block is illustrated in Figure 9—note that the other clauses are not relevant to our discussion and hence are not shown in the figure.

```
/*@ behavior
 @ requires pre1 || pre2 || ... || prek;
 @ ensures (\old(pre1) ==> post1) && ... &&
 @          (\old(prek) ==> postk);
 @ // other clauses omitted ...
 @*/
T m(T1 p1, ..., Tn pn) { ... }
```

Figure 9. Partial method specification (JML core)

A method return type is counted as non-null if every `posti` that is not `false` constrains `\result` to be non-null (as explained in Section 2.2.2). Of course, there must be at least one `posti` that is not `false`.

7.4 Method parameters

In its desugared form, a parameter declaration `Ti pi` is counted as non-null if every `pren` that is not `false` constrains `pi` to be non-null (as explained in Section 2.2.2). Of course, there must be at least one `prei` that is not `false`.

Figure 10 and Figure 11 provide examples of what is counted as non-null and what is not (respectively).

<pre> Parent //@ requires s!=null; //@ ensures post_p; Object m(String s){..} Child //@ also //@ requires s!=null && pre_c; //@ ensures post_c; Object m(String s){..} </pre>
(a) “Sugared” sample specification
<pre> Child //@ requires s!=null s!=null && pre_c //@ ensures \old(s!=null) ==> post_p && //@ \old(s!=null && pre_c) ==> post_c Object m(String s){..} </pre>
(b) Desugared sample specification

Figure 10. Non-null method parameter example

<pre> Parent //@ requires s!=null && pre_p; //@ ensures post_p; Object m(String s){..} Child //@ also //@ requires s!=null pre_c; //@ ensures post_c; Object m(String s){..} </pre>
(a) “Sugared” sample specification
<pre> Child //@ requires pre_p pre_c //@ ensures \old(s!=null && pre_p) ==> post_p && //@ \old(s!=null pre_c) ==> post_c Object m(String s){..} </pre>
(b) Desugared sample specification

Figure 11. Possibly-null method parameter example