Propagation of JML non-null annotations in Java programs

Maciej Cielecki Jędrzej Fulara

Krzysztof Jakubczyk Łukasz Jancewicz

Institute of Informatics, Warsaw University, ul. Banacha 2, 02-097 Warszawa, Poland

{m.cielecki,j.fulara,k.jakubczyk,l.jancewicz}@students.mimuw.edu.pl

ABSTRACT

Development of high quality code is extremely difficult. Tools that help maintaining the proper quality of code produced by programmers can be very useful: they may increase the quality of produced software and help managers to ensure that the product is ready for the market. One of such tools is ESC/Java2, a static checker of Java Modeling Language annotations. These annotations can be used to ensure that a certain assertion is satisfied during the execution of the program, among the others - to ensure that a certain variable never has a null value. Unfortunately, using ESC/Java2 can be very troublesome and time-consuming for programmers, as it lacks a friendly user interface and a function that propagates annotations.

We present *CANAPA*, a tool that can highly reduce time and effort of eliminating Null Pointer Exceptions in Java code. This tool can automatically propagate JML non-null annotations and comes with a handy Eclipse plug-in. We believe that functionality of *CANAPA* will minimize the effort required to benefit from using the JML non-null checking.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering—*Software/Program Verification*

General Terms

RELIABILITY, VERIFICATION

1. INTRODUCTION

1.1 Coding errors

Ensuring that a piece of software created within a company is free of coding bugs has always been a huge problem. Company managers try to succeed in this area by applying various coding policies and code-checking strategies. These policies can include overnight bug checking (people are hired to work at night and find errors in code that was created during the day), enforcing coding standards, like consisting naming convention or limiting length of methods, using automated tools to verify certain properties of the code either by static code analysis or dynamic assertion checking. The efficiency of the tools of the latter category can usually be improved by annotating the code with some meta information, like pre- and post- condition of methods or invariants of data structure implementations.

There are many languages created to annotate programs, but none of them is actually very popular. For the Java programming language the de facto standard is JML, the Java Modeling Language [11, 3]. Recently Microsoft introduced Spec# [2], the extension of C# targeted at specifying correctness properties of code. Some work has also been done for low-level languages. In particular, the Java Bytecode has its own specification language, called BCSL or BML [4]. The motivation for the latter was ensuring the security of Java applets on SmartCards.

In general, coding errors can be divided into two categories: those that result from a programmer's misunderstanding of the algorithm (for example, adding two variables instead of multiplying) and those that result from a programmer's carelessness (like leaving a variable uninitialized).

Although it is very hard to detect the first kind of bugs, there is a way to avoid a large majority of the second ones, mainly by creating and using automated verification software. It is of course impossible to write a tool that automatically checks the correctness of all programs, but there is a way to check some of its features.

1.2 Avoiding Null Pointer Exceptions in Java

The most common error found in pointer-based object-oriented programming languages occurs when one tries to access an object variable that has not been initialized. In Java, it is the well-known Null Pointer Exception. Null Pointer Exceptions are a very serious threat to the safety of programs, because when they occur at run-time, they cause a total software failure. That is why it is important to support the programmer in detecting and eliminating these kinds of problems. But first, the programmer has to express what features he would expect from his or her software. For this task we use JML, a behavioral interface specification language. One of the key features of JML is the possibility to annotate certain variables as non-null, which means that it was the programmer's intention not to allow to assign a null value to that variable.

1.3 JML

1.3.1 Overview

JML, the Java Modeling Language, is useful for specifying detailed design properties of Java classes and interfaces. JML is a behavioral interface specification language for Java. The behavior of classes and method created by a programmer can be precisely documented in JML annotations. They describe the intended way that programmers should use the code. JML can, for example, list the preconditions and postconditions for methods as well as class invariants.

An important goal for the design of JML is that it should be easily understandable by Java programmers. It is achieved by staying as close as possible to Java syntax and semantics. Several groups worldwide are now building tools that support the JML notation and are involved with the ongoing design of JML. The open, cooperative nature of the JML effort is important both for tool developers and users. For potential users, the fact that there are several tools supporting the same notation is clearly an advantage. For tool developers, using a common syntax and semantics can make it much easier to get users interested, because one of the biggest problem with starting to use a new specification tool is often the lack of familiarity with the specification language.

1.3.2 Non_null annotations

In this paper, we focus on annotating properties of methods, variables etc. which assure that the object under question never has a null value.

In JML, there are two ways to make such an assertion. If we want to make sure that a variable is never null (for example, we would call its method in a moment and it could produce a Null Pointer Exception), we add the `/*@ non_null @*/` annotation (note the `@` sign after the beginning and before the end of the comment):

```
/*@ non_null @*/ String s = "Hi there!";
```

A more interesting example is the method definition. If we want a method argument to be non-null, we could write something like this:

```
public void checkLength(/*@ non_null @*/String s);
```

or, we could add something like that:

```
//requires s != null
public void checkLength(String s);
```

Notice the subtle difference between those examples. In the first one, if the method body would contain the line:

```
s = null;
```

we would get an error. In the second example, as long as at entry point the non-null assertion is fulfilled, the statement won't generate an error. By the way, we find it a bad programming practice to change parameters that way, they should be copied to another variable instead.

1.3.3 JML checking

An annotation language like JML would be quite useless without a tool that can extract information from the annotations and use it to verify some, if not all, of its required features. In general, we divide the checkers into two categories:

- run-time checking tools, like JMLrac [5] — annotations are converted into assertions that are verified when the code they describe is executed
- static checking tools, like ESC/Java and ESC/Java2 [9] — do not require running the program; instead they try to prove that annotations are fulfilled by statically analysing possible execution paths.

Advantages and disadvantages of each method can be clearly seen. Run-time checkers can check any assertion, no matter how complicated, but if a method is never run, its assertions will not be executed and verified. Besides, the execution time is longer due to additional instructions in the code. Static checkers, on the other hand, are limited by their reasoning capabilities. Hence they can sometimes show nonexistent errors (false positives) or fail to find some existing ones (false negatives). The most popular static checker for Java is ESC/Java2 [9].

1.4 ESC/Java2

ESC/Java tool, developed at Compaq Research, performs what is called extended static checking, a compile-time checking that goes well beyond type checking. It can check relatively simple assertions and can check for certain kinds of common errors in Java code, such as dereferencing null, indexing an array outside its bounds, or casting a reference to an impermissible type. ESC/Java supports a subset of JML. ESC/Java2 [9] is an extension to ESC/Java, whose development has ended.

The user's interaction with ESC/Java2 is quite similar to the interaction with the compiler's type checker: the user includes JML annotations in the code and runs the tool, and the tool responds with a list of possible errors in the program. The use of JML annotations enables ESC/Java2 to produce warnings not at the source locations where errors manifest themselves at run-time, but at the source locations where the errors are committed.

The creators of ESC/Java2 wanted it to be as fast as possible, even at the cost of soundness and completeness. ESC/Java2 translates a given JML-annotated Java program into verification conditions, logical formulas that are valid if and only if the program is free of the kinds of errors being analyzed. The verification conditions are fed to an automatic first-order theorem prover Simplify [7], which tries to prove them. Any verification-condition counterexamples found by Simplify are turned into programmer-sensible warning messages, including the kind and source location of each potential error.

2. ANNOTATING THE PROGRAM

The combined usage of JML non-null annotations and ESC/Java2 allows software developers to eliminate all Null Pointer Exceptions from their programs. However, benefits of doing so do not always compensate additional time spent on manually adding the necessary assertions. Several add-on tools were developed to make the process faster, such as the ESCJava2 Eclipse plug-in that highlights places of possible errors. Unfortunately, that is still not enough to convince programmers to use JML.

Using ESC/Java2 to check Null Pointer Exceptions is somewhat cumbersome, because the checker shows us the place in the code where the error might occur, but it does not tell us where to put the /*@ non_null @*/ annotation. Very often the need of inserting another annotation is so obvious, that we would expect it to be done for us.

Let's consider the following example:

```
class Class {
 /*@non_null@*/ String attribute;
 Class() {
  attribute = "eLLo";
 }
 void set(String param) {
  attribute = param;
     //ESCJava2 will point to this line
 }
}
```

ESC/Java2 will signal an error in the assignment inside the method `set()`, so the programmer should add a non-null annotation to the parameter `param`. Then one can discover that, for example, one of `set()` method calls take a parameter which is not annotated non-null. The programmer is forced to correct his or her code and run ESC/Java2 each time he or she does it until all the errors are eliminated.

Other examples of cases when the annotation should and should not be propagated can be found in Section5.

An obvious solution is to create a tool that supports the programmer in annotating his or her code. We would expect the following features from such a tool:

- it should propagate annotations inserted by the user to avoid pointless ESC/Java2 warnings
- it should be fairly easy to use
- it should not require additional annotations in the code to make it work
- it should propagate only those annotations for which it is obvious that they should be propagated
- it should integrate into a popular Java development platform
- its effects should be easily reversible

3. RELATED TOOLS

Our solution, *CANAPA*, is based on the Java Modeling Language and ESC/Java2. There exist several other languages and systems that aim at statical enforcement of program correctness.

In Visual Studio 2005, Microsoft introduced Code Analysis tools [6]. Among other features, these tools can check the program for potential null-pointer dereference errors. There is no support for Java, but one can annotate C++ code or write full specifications of C# programs in Spec#.

There are many static checkers for C language that can check the NULL values to some extent. Many of those are commercial, closed source software, therefore are not broadly available. Nevertheless some of those checkers are very powerful, designed for large codebases, support user defined properties, with very small number of false positives. They are also usually bundled with an entire package of tools that enforce code quality, see eg. [14].

There were many research about the subject of annotations [4], [13]. There is considerable interest in automated annotation propagation, but the approaches considered were different from ours.

There are various tools that were build around ESC/Java. Two most interesting from our point of view are: The Daikon Invariant Detector and The Houdini Tool.

3.1 Daikon Invariant Detector

Daikon [8] is an implementation of dynamic detection of likely invariants; that is, the Daikon invariant detector reports properties that hold at a certain point or points in a program. Daikon runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. It can be used to automatically generate JML annotations in Java Code.

3.2 Houdini

This tool was under development as a research project at Compaq SRC. Houdini infers ESC/Java annotations for a given program and then runs ESC/Java to obtain a set of warnings. This set is considerably smaller, and more interesting, than the set of mostly spurious warnings that ESC/Java would have produced on the unannotated program. Although this process does not provide the same benefit of documented programmer design decisions (it detects the de facto design rather than enforcing de juro design decisions), Houdini greatly reduces the cost of finding common errors in a given program. Non-null annotations are among the annotations generated by Houdini, but the approach taken by the creators of this software is different from ours and does not guarantee that full set of annotations will be generated.

4. OUR SOLUTION

We present *CANAPA*, "Completely Automated Non-null Annotation Propagating Application", a tool to automatically propagate JML annotations that concern being or not being null by variables, method result etc. *CANAPA* is a program that propagates the /*@ non_null @*/ annotations inside the source code "bottom-up". This greatly reduces time and effort to correctly insert non-null annotations into the code.

4.1 Overview

The main idea is that the programmer inserts a JML non-null annotation inside the code, and CANAPA checks what are the obvious implications of such an insertion and inserts additional non-null annotations where ESC/Java2 would expect them. This way, the programmer sees the error in his logic (if any) at its source, and does not have to manually add each assertion to get to the mistake.

CANAPA has the following features:

- the program (*CANAPA*) is idempotent - the result of running it once on a Java code should be the same as running it twice
- from the preceding, it cannot add non-null annotations to class attributes - this would lead to undesirable results - see Use cases
- it changes the code only when it's sure it was the programmer's intention
- it adds its own comment tag to the JML tag in case the programmer wanted to remove the effects of its work

The Usage of CANAPA is fairly straightforward. You simply put `/*@ non_null @*/` annotations in the code where you want them and then run our tool, which propagates those assertions anywhere it is necessary.

CANAPA can be invoked from the command line with a directory parameter, or executed from Eclipse development platform via a plugin. During its work, *CANAPA* adds annotations to selected files and creates their backups, notifying the user which files were modified in the process.

Each annotation added by *CANAPA* is marked with a `/*CANAPA*/` prefix. Annotations existing before running the tool will not have this marker. This way, you can easily find and, if necessary, remove the automatically added annotations.

CANAPA comes with a handy Eclipse plug-in that allows to run it within the Eclipse programming environment. The tool simply adds annotations to the file looked at by the programmer and the text output can be seen in the Console window.

4.2 Implementation Details

The tool consists of a number of elements: the interface to ESC/Java2 (to find errors), a Java code parser to insert needed annotations, a simple text user interface and the Eclipse plug-in.

ESC/Java2 is invoked directly via its main method from the JAR, that's why our software requires Java 1.4 to work.

The parser used in the tool is a slightly modified free JParse [12] tool, which itself is based on ANTLR [1], a free parser generator.

The algorithm invoked by the tool is as follows: ESC/Java2 is run on the code provided by the user. The errors returned by ESC/Java2 are parsed, their solutions (if any) found and appropriate annotations placed to remove ESC/Java2 errors. Then ESC/Java2 is run again (this time it won't detect errors where they were before). If any new errors are detected, the procedure is repeated.

The number of iterations of the algorithm is limited by the depth of the deepest variable and method call dependency in the user's code. It must be noted that, with a "clean" (unannotated) large piece of code running *CANAPA* for the first time may take some time. However, the more annotations are in the code, the faster our tool is. In the ideal working example, when the programmer starts annotating his or her code from the very beginning and runs the tool each time he or she makes a significant addition, *CANAPA* will work very fast, with few iterations.

CANAPA tries to correct the following ESC/Java2 errors:

- assignment of a possibly null item to a non-null annotated variable:
  - assignment of a method parameter
  - assignment of a local variable
  - assignment of a function result
- dereferencing a possibly null item:
  - invoking `variable.someMethod()`
  - invoking `oneMethod().anotherMethod()`

The action taken by *CANAPA* differs depending on the type of item in question:

- if the item is a local variable, annotate its declaration with `/*@ non_null @*/`
- if the item is a formal method parameter, annotate it in the method header with `/*@ non_null @*/`
- if the item is a result of a method, annotate the return type of the method in the method header with `/*@ non_null @*/`
- if the item is a class attribute, do not annotate it - this probably would not be what the programmer wants, as it could cause "top-down" propagation into other methods

It is significant that the tool does not modify the code itself, but only the comments. So the compiler would still work if something went wrong.

The tool does its best to propagate the annotations just as the programmer would. There is, however, one situation in which *CANAPA* fails to predict the right annotation. Let us imagine that the programmer writes a method without annotating its parameter and dereferences it in the method body. It is impossible to know whether the intention of the programmer was to never call this method with the null argument or he simply forgot to put the if clause. Since *CANAPA* cannot guess what to write in the else branch, anyway, it annotates the parameter with non-null.

Fortunately, there is a way to deal with the situation. The Eclipse plug-in provides an option to revert the effects of the last *CANAPA* corrections within a few keystrokes. To avoid programmer confusion about which changes were added in the last *CANAPA* execution, a "commit" option is added that eliminates the `/*CANAPA*/` comments before `/*@ non_null @*/` annotations.

5. USE CASES

In this chapter we will show several basic examples of using *CANAPA*. Each example contains of a short piece of incorrect code and the description how *CANAPA* deals with it.

5.1 Example 1

This example shows how a `/*@ non_null @*/` annotation can be propagated to a method parameter.

```
class Class {
 /*@non_null@*/ String attribute;
 Class() {
  attribute = "Attribute";
 }
 void set(String param) {
  attribute = param;
 }
}
```

The code presented above is not correct: ESC/Java2 will point to the line `attribute = param`. *Attribute* is declared as non-null, and we try to assign *param* to it, so *param* must be declared as `/*@ non_null @*/` too. The easiest way to correct it is to add a `/*@ non_null @*/` annotation to *param* in the method header. Launching *CANAPA* will modify the code as follows:

```
class Class {
 /*@non_null@*/ String attribute;
 Class() {
  attribute = "Attribute";
 }
 void set(/*CANAPA*//*@non_null@*/ String param) {
  attribute = param;
 }
}
```

And that is exactly what our application does. There is another possible way to correct this error - it involves enclosing the assignment in if-else statement. However, it is impossible for the tool to guess what to do if param is null.

5.2 Example 2

This example shows the inference of a `/*@ non_null @*/` annotation to a variable or parameter, of which a programmer invokes a method. Let's consider the following piece of code:

```
class ClassA {
 public A(){}
 public void methodA(){}
}
class ClassB {
 public B(){}
 public void methodB(ClassA a){
  a.methodA();
 }
}
```

This code is invalid, as the parameter `a` of `methodB` could be null. So the method call `a.methodA()` may cause a null pointer exception. To correct the error, one should add a `/*@ non_null @*/` annotation to the parameter in the methodB header. After launching our application, the code will be modified as follows:

```
class ClassA {
 public A(){}
 public void methodA(){}
}
class ClassB {
 public B(){}
 public void
        methodB(/*CANAPA*//*@non_null@*/ClassA a){
  a.methodA();
 }
}
```

Of course the problem concerns not only parameters, but also variables:

```
class Class{
 public Class(){}
 public void method(){
  String str;
  str.substring(1);
 }
}
```

The local variable `str` cannot be null, otherwise the method call `str.substring(1)` would cause a null pointer exception. The solution is to declare `str` as `/*@ non_null @*/`. *CANAPA* will add the appropriate annotation. Of course, the problem persists (`str` is uninitialized), but this time, ESC/Java2 error points the user exactly to the source of the problem.

5.3 Example 3

This example shows how a `/*@ non_null @*/` annotation can be propagated to the method's result.

```
class Class{
 /*@ non_null @*/ String attribute;
 public Class(){
  attribute = "Attribute";
 }
 private String getString(){
  return "This is a string";
 }
 public void set(){
  attribute = getString();
 }
}
```

$\frac{a}{b}$ We assign the result of `getString()` to `attribute`, which is declared as `/*@ non_null @*/`. Until we are not sure that the method `getString()` cannot return a null, this code will be incorrect. The easiest way to solve this problem is to add `/*@ non_null @*/` annotation to the result of `getString()`. The code modified by *CANAPA* will be as follows:

```
class Class{
 /*@ non_null @*/ String attribute;
 public Class(){
  attribute = "Attribute";
 }
 private /*@ non_null @*/ String getString(){
  return "This is a string";
 }
 public void set(){
  attribute = getString();
 }
}
```

5.4 Example 4

This example shows that there are situations, when a propagation should not be done, although one could think that an annotation should be added. Consider following piece of code:

```
class Class{
 String attribute;
 void doSomething(){
  ...
  /*@non_null@*/String str = attribute;
 }
 void setNull(){
  attribute = null;
 }
}
```

One might expect that *CANAPA* will add an annotation to the attribute, modifying the code as follows:

```
class Class{
 /*@non_null@*/ String attribute;
 void doSomething(){
  ...
```

```
    /*@non_null@*/ String str = attribute;
  }
  void setNull(){
   attribute = null;
  }
 }
```

After careful consideration of this code, we can see that the added annotation causes an error in an other method. Namely in the `setNull()` method (we will try to assign null to an attribute that was declared as non-null). This definitely would not be acceptable for most of programmers. Moreover, we claim that in such a situation it is impossible to modify the code automatically in a reasonable way. So we have decided not to add anything to class attributes.

6. SUMMARY

We created *CANAPA*, the tool that highly reduces time and effort of eliminating Null Pointer Exceptions in Java code. This tool automatically propagates JML non-null annotations, whenever this results from the programmer's intension. It also comes with a handy Eclipse plug-in to increase productivity. *CANAPA* is distributed under the GNU LESSER GENERAL PUBLIC LICENSE [10]. It is available from `http://www.mimuw.edu.pl/~chrzaszcz/Canapa/`. It requires a Java Runtime Environment (version 1.4) and the ESC/Java2 checker. To run the CANAPA Eclipse plug-in, version 3.1 or higher of the Eclipse environment is needed.

7. ACKNOWLEDGEMENT

This work was partly supported by the Information Society Technologies programme of the European Commission, under the IST-2005-015905 MOBIUS project. This paper reflects only the authors' views and the Community is not liable for any use that may be made of the information contained therein.

8. ADDITIONAL AUTHORS

Jacek Chrząszcz, Institute of Informatics, Warsaw University, email: `chrzaszcz@mimuw.edu.pl`.

Aleksy Schubert, Institute of Informatics, Warsaw University, Poland and SoS Group, Faculty of Science, University of Nijmegen, email: `alx@mimuw.edu.pl`.

Łukasz Kamiński, Comarch Research and Development Center, email: `Lukasz.Kaminski@comarch.pl`.

9. REFERENCES

[1] Antlr parser generator. http://www.antlr.org/.

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*. Springer, 2004.

[3] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. 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, *FMICS: Eighth International Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, June 5-7 2003.

[4] Lilian Burdy and Mariela Pavlova. Java bytecode specification and verification. In *21st Annual ACM Symposium on Applied Computing (SAC'06)*, Dijon, Apr 2006. ACM Press.

[5] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.

[6] Code analysis for C/C++ — overview. http://msdn2.microsoft.com/en-us/library/d3bbz7tz.aspx.

[7] David L. Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.

[8] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

[9] Extended Static Checker for Java version 2. http://secure.ucd.ie/products/opensource/ESCJava2/.

[10] GNU LESSER GENERAL PUBLIC LICENSE. http://www.gnu.org/copyleft/lesser.html.

[11] The Java Modeling Language (JML). http://www.cs.iastate.edu/ leavens/JML//index.shtml.

[12] JParse: a Java parser. http://www.ittc.ku.edu/JParse/.

[13] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In P. Paradinas and J.-J. Quisquater, editors, *Proceedings of CARDIS'04*, Toulouse, France, August 2004. Kluwer Academic Publishers.

[14] Static source code analysis tools for C. http://www.spinroot.com/static/.