

Nullness Analysis of Java Source Code

Master's thesis

Arnout F.M. Engelen

<ARNOUT@ENGELN.EU>

Research number: 548

10th August 2006

Contents

1	Introduction	7
2	Background	11
2.1	Terminology	11
2.1.1	Elements	11
2.1.2	Bias and propagation	12
2.1.3	Assumptions about external code	12
2.2	Java Modeling Language	12
2.2.1	Nullity modifiers and invariants	13
2.2.2	More complicated specifications	14
2.2.3	Non-null by default	14
2.3	Type systems	14
2.3.1	Different approaches to possibly-null values	15
2.3.1.1	Traditional (nullable) pointers	15
2.3.1.2	References with nullness information	15
2.3.1.3	Functional ‘Maybe’ types	18
2.3.2	Type Inference	19
2.4	Program Verification	19
2.4.1	Program Verification and Type Systems	20
2.4.2	Program Verification and Testing	20
3	Existing nullness analysis tools	21
3.1	Nullness checkers	22
3.1.1	Eclipse and IntelliJ IDEA	22
3.1.2	JastAdd Non-Null Checker	23
3.1.3	ESC/Java	23
3.1.4	Spec#	24
3.1.5	FindBugs	24
3.2	Annotation Assistants	27
3.2.1	JastAdd Nullness Inferer	27
3.2.1.1	Example	29
3.2.2	Houdini	30
3.2.2.1	Example	34
3.2.3	CANAPA	34
3.2.3.1	Example	37
3.2.4	Julia	39
3.2.5	Daikon	39

4	Requirements and design considerations	41
4.1	Requirements	41
4.2	Design considerations	42
4.2.1	Code external to the analysis	42
4.2.2	Limits of type-based approaches	42
4.2.2.1	Combining type- and logic-based analysis	44
4.2.2.2	Nullness type inference	45
4.2.3	Impact of incorrectly inferred annotations	47
4.2.4	Initial Annotations	47
4.2.4.1	Using a type-based analysis for pre-processing	48
4.2.5	Termination	48
4.2.6	Interactivity	49
4.2.7	Scalability	49
4.2.8	Iterative, Modular Interaction	50
5	Annotation Assistance Algorithm	51
5.1	Algorithm description	51
5.2	Interactivity	51
5.3	Termination	52
5.4	A note about soundness	53
5.5	Implementation	53
5.5.1	The codebase (F)	53
5.5.2	Parsing Java	53
5.5.3	Initial annotations	54
5.5.4	File selection	54
5.5.5	Running ESC/Java	54
5.5.6	Processing conflicts	55
5.5.7	From conflict to suggestion	55
5.5.7.1	Suggestions about external files	56
5.5.8	Applying suggestions	56
5.5.9	Example	57
6	Case studies and Benchmarks	59
6.1	Benchmarking annotation assistants	59
6.1.1	Resource usage	59
6.1.2	Quality	60
6.1.2.1	Determining whether an annotation is correct	60
6.1.3	Taking into account user interaction	60
6.2	DigiD Gateway	60
6.2.1	Introduction of modularity	60
6.2.1.1	Resource usage	61
6.2.1.2	Quality	61
6.3	Promedico ASP	61
6.3.1	Generated code	62
6.3.2	Dependencies required	62
6.3.3	toString() methods	62
6.3.4	Pre-processing performance	62
6.3.4.1	Quality	63
6.3.4.2	Performance	63

7	Conclusions	65
7.1	Further Research	65
7.1.1	Interoperability of nullness analysis tools	65
7.1.2	Heuristic type inference	66
7.1.3	User Interface improvements	66
7.1.4	Non-null by default	66
A	Example tool output	69
A.1	Clean Java files	69
A.1.1	External.java	69
A.1.2	Test1.java	69
A.1.3	Test2.java	70
A.1.4	Test3.java	70
A.1.5	Test4.java	71
A.2	JastAdd inferer results	71
A.2.1	External.java	71
A.2.2	Test1.java	71
A.2.3	Test2.java	72
A.2.4	Test3.java	73
A.2.5	Test4.java	73
A.3	Houdini results	74
A.3.1	External.java	74
A.3.2	Test1.java	74
A.3.3	Test2.java	75
A.3.4	Test3.java	75
A.3.5	Test4.java	76
A.4	CANAPA results	76
A.4.1	External.java	76
A.4.2	Test1.java	76
A.4.3	Test2.java	77
A.4.4	Test3.java	77
A.4.5	Test4.java	78

Chapter 1

Introduction

These days, software is everywhere: it drives not only personal computers, but all kinds of processes ranging from your mobile phone or media center to pace-makers to huge critical processes in industry.

Software development is an expensive and complicated process, in which speed (to cut costs and improve time-to-market) and reliability are key factors. In order to improve on those points, it is important that techniques are developed that can identify (potential) bugs early and reliably.

We can distinguish two kinds of software errors: errors which are found by statically analysing the source code (i.e., without actually having to start the application), and errors which are found at run-time. The latter variation can be particularly vicious because they might be found at a later stage, possibly even after the product has been distributed to customers and put to use in production environments. Because of this, it is worthwhile to try and find as many bugs as possible by static analysis.

Many bugs stem from inconsistencies: two parts of a system can both be locally correct, but if one part makes an incorrect assumption about the other, an error is born. Software generally consists of many parts, often written by many developers over a long period of time, so explicitly and precisely stating the specifications (both assumptions and guarantees) is vital.

Expressing those assumptions and guarantees in the documentation and comments of the code is good, but even better is to write them down in a formal language. This prevents ambiguity and makes tool support for writing and checking specifications possible. Such formal languages exist: for example for Java there is the Java Modeling Language described in section 2.2.

Once the specification of the code is formalized, it becomes possible to automatically check that the code indeed adheres to this specification. For Java, the ESC/Java2 tool (described in section 3.1.3) statically verifies a large subset of the JML specifications.

Many languages like C++ or Java have a concept of a *pointer* or *reference* that can either point at some object or at nothing (the latter case is often called a *null pointer*) — this is not known statically. When such a reference is specified to be non-null, this means this reference should always point to some object, in other words, it should never be null. This thesis will focus on these specific

‘nullness specifications’.

Nullness specifications are relatively simple, but very important. Most operations on an object, such as reading its fields or calling its methods, are only valid when the object reference is not null. When such an operation is performed on a null reference, this almost always has an undesirable effect: in the more primitive languages such as C++, the program might crash entirely. Languages that handle this more robustly, for example by throwing an exception like Java does, obviously allow the developer to recover from these errors more gracefully. However, it might also potentially lead to even more dangerous problems: when the exception is inadequately handled by the developer, the program might keep running in an unexpected unstable state, possibly corrupting data or compromising security. Because unexpected `NullPointerException`s are such a common source of bugs [15], this is a real danger.

Another advantage of nullness analysis is the ability to omit some run-time nullness checks if they can be statically eliminated, improving performance. This is not really feasible for systems currently in general use, but certainly a long-term goal.

When seeking to apply nullness specification checking to existing Java source code (or when converting code to a language that supports nullness attributes natively), many annotations of the nullability of references will have to be added. This can be a very cumbersome and time-consuming task, as often (for example due to function calls across objects) many different files need to be annotated. As such, it seems desirable to have tool support for this process, at least partly automating it. We refer to such tools as *annotation assistants*.

The research question for this project is thus as follows:

What is the current state of the art in annotation assistance, and what are the most relevant requirements and design considerations when implementing an annotations assistant?

To be able to answer this question, we first selected and reviewed several tools with their accompanying research papers. One tool in particular, Houdini, looked like it had been quite successful in the past judging by the positive evaluation in the papers published on it, but had been abandoned for a long time. We took up the challenge to revive this tool, bringing it up to speed with the current version of ESC/Java2, the checking tool with which it is tightly integrated.

Based on these experiences, we formulated requirements and design considerations, and finally constructed an annotation assistance algorithm based on these.

Finally, we made a partial proof-of-concept implementation and used it to test some of our ideas in practice, and evaluated our choices based on the data gathered from running it on two case studies.

Summary

We now give a description of the contents of each chapter in this thesis.

Chapter 2: Background

This chapter introduces the background required to understand the rest of the thesis.

First, some useful terminology is introduced, explained and related to the terminology used in other research.

The Java Modeling Language, JML, is briefly introduced.

Type systems are an important aspect of computer languages. It is common for languages to have pointers or reference types to allow for efficient programming. There are several approaches to typing possibly-null values (for example nullable pointers as found in Java and C++, option types as found in the Nice programming language or a polymorphic Maybe type as found in functional languages such as Haskell).

The Java type system lacks types for non-null references. However, using JML annotations we can specify nullness attributes of references and much more.

A static checker such as ESC/Java for JML makes the picture complete. Static code analysis is an analysis that is performed by a tool without actually executing the program built from that code. Rather, it works on the source code (or possibly some form of object code such as Java class files). The type checking phase which is performed when compiling statically typed languages such as C, Java or ML is a well-known example of simple static checking.

Chapter 3: Existing nullness analysis tools

This chapter gives an overview of the currently available software tools. We distinguish between *checkers* which verify the nullness properties of an already annotated program, and *inferred* which attempt to automatically extract such annotations. We will explain this distinction is not always easy to make.

Chapter 4: Requirements and design considerations

In this chapter we describe the requirements for an annotations assistant, and identify key design choices found in the existing annotation tools we have reviewed. These design choices are evaluated in the context of the requirements for an annotations assistant.

Based on the insights gained from this analysis, we make several recommendations about the way an annotation assistant for nullness properties should be built.

Chapter 5: Annotation Assistance Algorithm

Based on the insights from the previous chapter, we have developed an Annotation Assistance Algorithm.

We put some of our recommendations to the test by implementing a proof-of-concept nullness annotation assistant called INAPA. This proof of concept implementation is described in some detail.

Chapter 6: Case Studies and Benchmarks

This chapter discusses two case studies which have been used throughout the project to identify areas that require improvements, and to evaluate whether

changes to the assistance techniques indeed have the desired effect.

The first case study is the DigiD Gateway, a J2EE servlet that is part of the DigiD system used by the Dutch government to authenticate citizens when communicating with government institutions online.

The other case study is the Promedico ASP HIS, which implements an online information system including web interface for health care centres. This is a large body of commercial production code.

Chapter 7: Conclusions

Adding annotations to an existing project is quite different from using extended static checking from the beginning while writing the code. This makes the requirements for tool support for this process different from checkers. In this chapter we summarize the most important aspects we have identified and evaluate the impact of the design choices in the proof-of-concept annotation assistance tool we developed in the process.

We also describe our experiences with nullness annotation and annotation assistance in general.

Chapter 2

Background

2.1 Terminology

Research on nullness analysis has come in many shapes and forms over the years, and has been referred to with various names. The ESC/Java¹ research generally talks about *nullness* and *non-nullness*. The same goes for the Spec#² documents, which is not surprising as there are several ESC/Java researchers now in the Spec# team. Most JML documentation, such as [17], refers to nullness as *nullity*. Recent research by Bertrand Meyer [20] on nullness in Eiffel³ introduced the terminology of *attached* and *detachable* for non-null and nullable references, since non-null references are guaranteed to be ‘attached’ to some object. The Eclipse project refers to its implementation as the ‘Null reference analysis’. In the database world (for example in the documentation on the Hibernate Object-Relational mapper⁴), the term *nullability* is often seen.

For this thesis, we will use the term *nullness analysis* for the general analysis and *nullable* and *non-null* for the two possible kinds of reference types. The *nullness* of a type or variable determines whether it is nullable or non-null.

2.1.1 Elements

It has proved useful to introduce one term to refer to all things that might be nullable or non-null. We chose to call these *elements*. Elements are references used as:

- object member fields
- method parameters
- method return values
- local variables

¹<http://kindsoftware.com/products/opensource/ESCJava2>

²<http://research.microsoft.com/specsharp>

³<http://www.eiffel.com>

⁴<http://www.hibernate.org>

2.1.2 Bias and propagation

When classifying nullness inference tools, we introduce a couple of terms to make some useful distinctions:

A tool can be *nullable-biased* or *non-null-biased*. When there is no indication that an element is nullable, but on the other hand there is also no indication that it is non-null, a nullable-biased inferrer will consider it to be nullable, while a non-null-biased inferrer will assume it should be non-null.

Some tools clearly deploy a *nullable propagation* or a *non-null propagation* approach. Such tools will, after possibly adding a set of initial annotations, mainly work by propagating the fact that certain elements are nullable or non-null, respectively. Not every tool clearly falls into one of these two categories, some do a bit of both.

2.1.3 Assumptions about external code

When dealing with calls to external code, inference tools will have to make some assumptions about their nullness properties.

These assumptions can be roughly divided into *optimistic* and *pessimistic* assumptions. Inferreders which make optimistic assumptions will assume a minimum of requirements and a maximum of guarantees from external methods. In other words, they will assume that external methods can always be passed nullable values as parameters, and always guarantee that they return a non-null object. Pessimistic inferreders will assume the opposite: that external methods must always be passed non-null values, but always return nullable values. These are two extremes with many other possibilities between them: ESC/Java, for instance, assumes nullable by default for both method parameters and return values. That is pessimistic for return values, but optimistic for parameters.

Aside from this division, we also distinguish between *minimal* and *maximal* assumptions: a tool that makes minimal assumptions only makes an assumption when it appears to be required, whereas a tool that makes maximal assumptions will start out by making assumptions about everything beforehand, even assumptions that might not be necessary.

2.2 Java Modeling Language

As we have seen in the introduction, it is highly desirable to be able to specify properties of a part of a program. To reduce ambiguity and to make tool support possible, it is useful to express these specifications in a formal language.

For Java, the Java Modeling Language (JML for short)⁵ [17] is such a language. JML was designed to be expressive and unambiguous, but at the same time accessible to any Java developer.

JML specifications can be added to a Java source file, or stored in separate so-called *spec files*. They are represented as a special kind of comment beginning with an `@` symbol, much like how JavaDoc⁶ documentation is marked with an extra `*` or `/` character.

⁵<http://www.cs.iastate.edu/~leavens/JML//index.shtml>

⁶<http://java.sun.com/j2se/javadoc>

```

class SummedPair {
    //@invariant sum == left + right;
    int left;
    int right;
    int sum;

    void addLeft (int i) {
        left += i;
        sum += i;
    }
}

```

Figure 2.1: Temporarily breaking an invariant is allowed

Both line comments and block comments can contain JML specifications, for example

```
/*@non_null*/private Object theObject;
```

or

```

/*@assume obj != null
someMethod(obj);

```

For this research, the annotation we will be mainly working with is the simple `/*@non_null*/` modifier. A `/*@nullable*/` modifier also does exist, but `non_null` and `nullable` cannot be mixed: `nullable` tags are only allowed in classes that have *non-null by default* enabled. This is outside the scope of this thesis, but discussed briefly in section 7.1.4.

2.2.1 Nullity modifiers and invariants

The `/*@non_null*/` annotation in the example above is called a *nullity modifier*. Another way to represent the fact that a class field is non-null is to define a class invariant:

```
/*@invariant theObject != null
```

The difference between a nullity modifier and a class invariant is subtle: both are required to be established by all constructors. An invariant is only checked on method boundaries, and as such can be temporarily broken. This is mainly relevant [18] for more complex invariants, as illustrated in Figure 2.1. Some more complex examples are available for example in [22].

For non-null specifications, it is harder to think of a scenario where it would be useful to temporarily break an invariant. It is useful to be aware of this difference however, since it for example explains the unintuitive handling of the helper pragma described in section 2.3.1.2.

2.2.2 More complicated specifications

Aside from these simple nullness annotations, JML supports a very rich scala of complex invariants. For example, it allows the developer to express that an array has only non-null elements (with `\nonnullelements(f)`) and allows logical connectives (enabling conditions like `someboolean ==> f != null`). For more information on the more advanced features of JML, see [17].

2.2.3 Non-null by default

Recent work such as [3] has led to a change in the JML language, changing the default from nullable to non-null reference. This development is still under heavy discussion, and we decided not to take this aspect into account for this thesis.

2.3 Type systems

In early untyped programming languages (such as simple Assembly languages), a value simply consisted of some bits with no additional formally specified meaning. In the context of the application, these bits are generally a representation of some value: this could be anything, a (bounded) natural number, a floating point number, a memory address, a character, or something entirely different. The developer had to manually make sure that the value was interpreted the way it was intended: interpreting the binary representation of a floating point number *1.00000* as a natural number will not, in general, produce *1*.

More high-level languages introduced the much-anticipated ability to assign a *data type* to a value. This can be done in two ways: it is possible to add type information to the run-time representation of the value. This is called *dynamic typing*, and is popular in interpreted programming languages. The main downside of this approach is that type conflicts occur only at run time. Another approach is to add type information to the source code of the program, and have the compiler check that it is consistent. This has the big advantage of being statically checkable, but is less flexible and can be more complicated. Java takes a middle road by mainly using static typing, but still allowing dynamic run-time type tests.

In its simplest form, the types in a static type system are simply a set of markers, and each value is of exactly one type. The types of the variables and functions in the program must be identified in the source code, and the type checker checks that this the program is indeed consistent.

More advanced systems, most prominently those in object-oriented languages, introduce subtyping: the types are no longer simply a set of markers, but a partial ordering $<$ is defined on them. Suppose we have the types *box* and *shape*, the ordering *box* $<$ *shape* specifies that every *box* is also a *shape*, and as such all functions defined on *shapes* can also be used on *boxes*. Because *box* is the more specific type, it is possible to assign a *box* to a variable of type *shape*, but not the other way around.

2.3.1 Different approaches to possibly-null values

Most programming languages allow for a natural way for representing the type of a variable that might or might not contain something at run-time.

2.3.1.1 Traditional (nullable) pointers

A **struct** in traditional languages like C is represented at run-time by the memory address of the represented data. It quickly became common to use the special memory address '0' to represent the fact that the pointer isn't currently pointing to any structure. This seemed like a reasonable thing to do and maps well to the way assembly language was traditionally written.

Objects in C++ were also represented as (nullable) pointers, and even though Java has a more abstract representation for them, still included a special "null" case. The main difference between null pointers in C++ and Java is that in C++, a null dereference will lead to undefined behaviour and (most likely) a segmentation fault crash. In Java, the Virtual Machine will spot the null dereference and a `NullPointerException` will be thrown. While this is clearly a considerable improvement, it is not without problems: in practice such exceptions might be inadequately handled because they are not expected. In that case the program might keep running in an unexpected unstable state and do much more harm than it would have if the program would have simply terminated.

One of the reasons `NullPointerException`s tend to be inadequately handled is because they are so-called *unchecked exceptions*, whose main characteristic is that a method that may throw such an exception is not obliged to state that in the *throws* clause of their definition. Because of this a developer might not be aware that a given call might throw a `NullPointerException`, and not handle it.

2.3.1.2 References with nullness information

It is currently becoming more common to add nullness information to the typing of mainstream programming languages: not only by adding an extra specification layer to the compile-time checks (like ESC/Java and Spec# do), but also by incorporating it in the language itself.

One elegant example of a Java-like language that includes nullability information in the type system is Nice⁷. This language, developed as a spin-off of academic research on object-orientation [2, 1], compiles to Java byte code and supports generics and explicit nullness typing.

All references are non-null references by default in Nice, for nullable references (referred to as *option types* in the Nice documentation) the type name must be prefixed with a question mark. This is a clear and short way to express nullability, which has also been adopted by Spec#. Eiffel is using [20] this notation as well, though not without criticism from the community, so it might move to keywords in the future.

Object construction In Object-Oriented languages, classes have fields which may have non-null types. It is common for classes to have a *constructor method* which is called when an object of that class is created. The semantics of a non-null field is not entirely obvious: it could mean the field should be non-null

⁷<http://nice.sourceforge.net>

```

class Foo {
  //@invariant this.s != null;
  String s;
  public Foo () {
    derefS();
    s = "foo";
  }
  private /*@helper*/ void derefS () {
    s.toString(); // ESC/Java will warn about this
  }
}

```

Figure 2.2: `s.toString()` will yield an ESC/Java error

```

class Foo {
  //@invariant this.s != null;
  String s;
  public Foo () {
    s = "foo";
    derefS();
  }
  private /*@helper*/ void derefS () {
    s.toString(); // ESC/Java will not warn about this
  }
}

```

Figure 2.3: `s.toString()` will not yield an ESC/Java error

even before the constructor method is called, or it could require that the field is non-null after the constructor has completed.

From a theoretical standpoint, the former might seem like an obvious choice. However, in practice fields which are designed to be non-null often cannot be statically initialized, and are initialized in the constructor. For this reason, both ESC/Java and Spec# use the latter semantics for non-nullness of fields.

This interpretation does cause some problems when normal methods are called from the constructor, or (worse still) the object being constructed is leaked to external code. The latter may happen when `this` is assigned to a global variable or globally reachable field, or passed on to an external method. The modular analysis of the external code would assume the field to be non-null, while it might not yet have been initialized when called from the constructor. This problem gets even worse in case of inheritance [19].

JML-based tools like ESC/Java might resolve some of those issues by requiring that the class methods called by the constructor are specified to be *helper* methods — though the current version of ESC/Java does not warn about this. Helper methods are private methods for which the object invariants are not checked before or after calling them [17]. Helper methods are not checked modularly, but quasi-inlined wherever they are called. This means the code in Figure 2.2 will yield an ESC/Java error, but the code in Figure 2.3 will not.

A limitation of this approach is that nullity modifiers such as `/*@non_null*/`


```

class Foo {
  /*@non_null*/ String s;
  public Foo () {
    s.toString();
    s = "foo";
  }
}

```

Figure 2.4: An explicitly inlined null dereference is correctly caught

```

class Foo {
  /*@non_null*/ String s;
  public Foo () {
    derefS();
    s = "foo";
  }
  public /*@helper*/ void derefS () {
    s.toString();
  }
}

```

Figure 2.5: A null dereference quasi-inlined by using `/*@helper*/` goes unnoticed

are not strictly considered to be part of the class invariants. Even for helper methods, ESC/Java will assume nullity modifiers of class fields have been satisfied. This leads to the situation that Figure 2.4 is correctly reported as erroneous, but the error in Figure 2.5 is missed.

When external code is called from the constructor, some leak analysis may be performed to verify that `this` has not been exposed yet. Unfortunately, the current version of ESC/Java2 (version 2.0a9) does not appear to implement such an analysis yet: the erroneous Foo class in Figure 2.6 is accepted without warnings.

Spec# fixes this issue by extending the model with *raw* and *partially raw* types as described in [10]. This seems like an elegant way of extending the type system with various degrees of partial initialization.

The Nice programming language approaches this problem in a rather different way: classes in Nice have an automatically generated ‘default’ constructor which requires instantiations for all non-null fields, and allows values for any fields which are nullable or have a static default value. This is sufficient in many cases, removing the need to write the constructor in the first place. For initialisation code that is not needed to fill any required fields, every class may contain an ‘initializer’ method which is automatically called after construction. If even more control over the construction is needed, a custom constructor can be defined. Such a constructor, however, does not yet have direct access to `this` or the non-static member functions of the class, and requires that the last statement of the custom constructor is a call of another constructor for the same type (typically the default constructor). For example, the Java class shown in

```

class Foo {
    /*@non_null*/ String s;
    Foo () {
        Bar b = new Bar();
        b.derefS(this);
        s = "foo";
    }
}
class Bar {
    public void derefS (/*@non_null*/Foo f) {
        f.s.toString();
    }
}

```

Figure 2.6: The current version of ESC/Java2 does no leak analysis

```

class Foo {
    /*@non_null*/String name;
    /*@non_null*/String filename;
    Foo (/*@non_null*/String name) {
        this (name, name + ".txt");
    }
    Foo (/*@non_null*/String name, /*@non_null*/String filename) {
        this.name = name;
        this.filename = filename;
        System.out.println ("A Foo has been constructed");
    }
}

```

Figure 2.7: Example class in Java

Figure 2.7 corresponds to the Nice class in Figure 2.8.

2.3.1.3 Functional ‘Maybe’ types

Many functional language take another approach to possibly absent structures: these languages often have a very natural BNF-like way of defining (possibly polymorphic) data types. A polymorphic linked list might be defined as follows (where ‘a’ is a polymorphic type variable):

```

:: List a = Node a (List a) | EmptyList

```

In this case an expression of type `List Int` will either look like a `Node` with an `Int` and another expression of type `List Int`, or hold the constant `EmptyList`.

In order to be able to represent a possibly absent expression, the `Maybe` type can be easily introduced:

```

:: Maybe a = Just a | Nothing

```

```

new Foo (String userdefined) {
    this (userdefined: userdefined, derived: userdefined + “.txt”);
}
class Foo {
    { println(“A Foo has been constructed”); }
    String userdefined;
    String derived;
}

```

Figure 2.8: Example class in Nice

An expression of the type `Maybe Int` will now either contain a `Just` node with an `Int`, or the constant `Nothing`. Functional languages tend to include such a construct in their standard library, along with some convenience methods to manipulate the construct⁸.

2.3.2 Type Inference

Once you have formally described a language’s type system, there are algorithms to find out whether a program or expression is well-typed without needing to explicitly specify the types of all variables.

Type inference is commonly implemented with a variation of the Hindley-Milner type inference algorithm [7]. This algorithm works in two phases: first, it introduces type variables for all locations of variables, parameter positions, etcetera. Then, it scans through the source code and records constraints on the types in the form of type equality or subtyping constraints. For example, if a location with type variable a is assigned to a location with type variable b , the inferrer would typically record the constraint $a <:= b$. This means a must be either equal to b , or a more specific type. After all these constraints have been generated, an algorithm is applied that tries to solve all these equations. If this succeeds, which means we can find types so that none of the constraints are violated, the program is obviously well-typed.

Most type inference algorithms work reasonably well for well-typed expressions. The main differences between algorithms are in the kind of type systems they support (for example whether they allow polymorphism), and in the quality of the error messages generated when trying to infer types in a program which contains a typing conflict.

2.4 Program Verification

Program Verification describes the field of tools that reason about software to verify that it is correct with respect to a given specification or property, often by proven it correct using formal methods of mathematics. Traditionally, program verification has usually been used to manually prove that a (usually small) piece of code correctly implements its full functional specification.

Program verification is a *static code analysis*: an analysis that is performed by a tool without actually executing the program built from that code. It works

⁸for example <http://www.haskell.org/onlinereport/maybe.html>

on the source code or possibly some form of object code (for instance Java class files). The type checking process described in section 2.3 is a simpler example of a static analysis. By using static analysis not only the common case is verified, but also many potential problems in hypothetical cases (which are unlikely to be completely covered by traditional testing) are covered.

Extended static checking is a form of program verification that, in contrast to traditional program verification, uses an *automatic* theorem prover and checks not a complete functional specification, but a given set of interesting properties. These properties are usually defined by augmenting the code with formalisations of assumptions which traditionally would have been left implicit, or merely noted as comments. In the case of Java, such formalisations could be expressed in JML. The Extended Static Checking software can now verify that these assertions indeed must hold, for instance by deploying various automatic theorem proving techniques.

2.4.1 Program Verification and Type Systems

Some of the problems that can be solved by program verification techniques could also be solved by extending the type system, and vice versa. Type systems have the advantage of being a well-understood technique that can be implemented with proven algorithms which work very efficiently.

Program verification techniques tend to require more effort to specify and more resources to automatically check, but can be applied locally instead of in an all-or-nothing manner: it is more easy to perform a detailed analyses of the critical part of the system while leaving the less critical parts alone.

When using a programming language with a very rich and detailed type system, the entire program must be written in such a way that the program conforms to the type system. With program verification, it can be very well imagined that the critical parts of the system are richly annotated with complicated preconditions, postconditions and invariants, while other parts contain almost no formal specifications at all.

2.4.2 Program Verification and Testing

Testing practices (such as Unit Testing and Test-Driven Development) and program verification are highly complementary techniques. Typically, a static analysis can rule out whole classes of errors at compile time, removing the need to cover those in software testing. The null pointer dereference problems are a good example of such a class of errors. On the other hand, there will always be properties that are hard to specify formally, but can be effectively covered by testing.

Ideally, program verification will be able to actually *prove* the absence of these errors, instead of merely increasing the confidence in the absence of certain errors.

Chapter 3

Existing nullness analysis tools

In this chapter we will look at a number of existing nullness analysis tools. We will highlight some important aspects and evaluate and compare those in more detail in the next chapter. The tools we investigated are the analysis integrated in Eclipse¹ and IDEA², the JastAdd Nullness Checker and Inferer³, ESC/Java⁴, Spec#⁵, Houdini, Daikon⁶ and Julia⁷.

While the tools are different in approach and task, they share a common goal:

Reducing the number of `NullPointerException`s thrown at run-time, by warning the developer at compile-time.

Nullness analysis tools can be roughly divided into two categories: nullness checkers and nullness annotation assistants (or *inferers*). Both checkers and annotation assistants try to analyze the nullness properties of code and point out possible problems.

The distinction is actually subtle and not always very clear. The JastAdd Non-Null Checker, the type checkers in languages like Nice and Eiffel and the nullness analysis integrated into Eclipse and IntelliJ IDEA are clearly purely checkers: they perform the checking in a straight-forward way and rely on explicit annotations if this analysis is insufficient. The JastAdd Non-Null Inferer clearly purely infers nullness attributes, and does not even warn in case of possible problems. FindBugs and ESC/Java are mainly used as checkers to identify problems with code, but also perform intelligent analysis. The line between these more intelligent checkers and annotation assistants is blurry: indeed, the annotation assistant for ESC/Java, Houdini, can not only be used as a separate tool, but it can also be used as a front-end. When used in that way, it calls ESC/Java only in the background, invisibly to the user.

¹<http://www.eclipse.org>

²<http://www.jetbrains.com/idea>

³<http://jastadd.cs.lth.se/>

⁴<http://kindsoftware.com/products/opensource/ESCJava2/>

⁵<http://research.microsoft.com/specsharp>

⁶<http://pag.csail.mit.edu/daikon/>

⁷<http://profs.sci.univr.it/~spoto/julia/>

The discriminating difference between checkers and annotations assistants seems to be that with an annotation assistant, the added annotations are visible to the developer, and are intended to correspond to the design of the application. While checkers might infer and even temporarily add annotations in the background, these are merely used to guide the analysis process, and should be seen as properties of the code rather than specifications.

The FindBugs checker and the checkers integrated in the Eclipse and IDEA IDE's are different from the other checkers because they do not strive for soundness in the rigorous way the other projects pursue this: they happily sacrifice it by making some bold assumptions, for example that all parameters passed into a method are non-null⁸. This is a trade-off: in return they are able to cut down immensely on the number of false positives, even when no annotations are provided by the developer. This means when these checkers report a problem, it is very likely that there is indeed in fact a problem with the code. On the other hand, the *absence* of warnings means little or nothing.

The other tools strive to be much more (but not entirely) sound in the sense that the absence of warnings should give a strong confidence in the absence of problems. To make the problem feasible often some of the more subtle possible sources of unsoundness are not taken into consideration⁹. The distinction between these tools is mainly found in the expressiveness of the specification language: ESC/Java and Spec# are much more expressive than the solutions that only use type annotations like JastAdd, FindBugs and IDEA or even no annotations at all like Eclipse.

Daikon and Houdini are both meant to infer annotations for ESC/Java, but take different approaches: Daikon works by observing the actual program behaviour when it is executed. It then tries to extract invariants from the observed behaviour, and represents those as ESC/Java annotations which can be statically checked.

Julia was unfortunately too sparsely documented and unstably implemented to perform an in-depth analysis at this time.

3.1 Nullness checkers

3.1.1 Eclipse and IntelliJ IDEA

Null reference analysis is quite a hot topic at the moment, in academia but it is also making inroads in industry: the commercial IntelliJ IDEA¹⁰ Java IDE has included nullness analysis for some time¹¹, and with Eclipse 3.2¹² this popular Open-Source Java IDE also includes some simple null reference analysis¹³.

⁸Actually, FindBugs marks parameters as NCP, Null on a Complex Path. However, it produces no warnings when NCP values are dereferenced. To produce a warning, a dereferenced value must be marked with a stronger indication of nullability such as NSP, Null on a Simple Path.

⁹See, for example, appendix C of the ESC/Java User's Manual for a list of such issues in ESC/Java. This manual can be found at <http://kindsoftware.com/products/opensource/ESCJava2/docs.html>

¹⁰<http://www.jetbrains.com/idea/>

¹¹<http://www.jetbrains.com/idea/documentation/howto.html>

¹²<http://www.eclipse.org>

This is very recent work: eclipse 3.2 was released on the 30th of June, 2006.

¹³<http://blogs.infosupport.com/peterhe/archive/2006/03/06/4163.aspx>

Eclipse currently does not allow annotation of the source code, and only warns the user in case of obvious mistakes: for example, if a local variable is dereferenced after being set or checked to be null.

IDEA is a bit more advanced, allowing interprocedural checks with `@Nullable` and `@NotNull` Java 1.5 tags, but also performs a rather primitive flow analysis. The company behind IDEA, JetBrains, has suggested to include these annotations in the standard Java SDK to allow interoperability with other checkers. This issue is still pending, and some of the issues cropping up are discussed in section 7.1.1. Eclipse is eager to add support for some annotations mechanism, but this is intentionally moving forward slowly due to the lack of standardisation in this field so far.

Even though the analysis performed by IDEA and Eclipse is limited, we expect this will be a great boost for the more advanced tools such as those on which this thesis focuses. These new features, as simple as they are, are being very enthusiastically received, and are gaining widespread use. Hopefully this will show how useful formalizing specifications is, and convince developers of the virtues of adopting tools supporting more powerful specification languages like JML.

3.1.2 JastAdd Non-Null Checker

JastAdd¹⁴ [13] is a Java-based compiler compiler system. It is well-suited to create extensible compilers for Java-like languages and tools. A Java 1.4 front-end and backend are available.

As an example of the flexibility of extending JastAdd-based compilers, an extension of the Java 1.4 front-end has been created that extends the language with non-null types¹⁵ [9]. JastAdd is the only Java-based checker we have seen that takes into account class invariants not only after construction, but also for partially constructed objects, as we have discussed in section 2.3.1.2. They have adopted a solution based on the research in [10].

It currently uses the modifiers `[NotNull]`, `[Raw]`, `[MaybeNull]` and even fine-grained raw types such as `[Raw(Upto=S.class)]`, but like most nullness tools this project also plans to use Java 5 annotations (like `@NotNull`) in the future.

Type of reasoning

The JastAdd Checker uses a classical type system to check nullness properties. The standard Java type system is extended, adding a non-null class `T-` for every class `T`. Then we add the obvious subtyping relation rules: `T- <: T` and `T- <: S- iff T <: S`. In the standard Java type system, `null` was considered to be a more specific instance of any reference type. This should now only hold for nullable references, not for non-null ones: `null <!: T-`.

3.1.3 ESC/Java

ESC/Java is a static checker for JML annotations (as described in section 2.2), including but not limited to nullness properties. It does not strictly require

¹⁴<http://jastadd.cs.lth.se/>

¹⁵<http://jastadd.cs.lth.se/examples/NonNullTypesExtension>

annotations, but checks them when present and expects them for any inter-procedural analysis: one of the design choices that make ESC/Java so scalable is that it works in a modular fashion. This means that while checking one method ESC/Java does not analyse any code in other methods, it only assumes that the JML specifications of that code is correct.

Where FindBugs tries to report bugs with a minimum of annotation effort and false positives, ESC/Java is more reliable (it makes less questionable assumptions) but requires more annotation effort.

Type of reasoning

ESC/Java works by converting the annotations and code into Verification Conditions that can be passed to an automatic theorem prover, called Simplify, and which hold if the program is indeed correct.

Handling of libraries

ESC/Java makes pessimistic assumptions about the return values in external libraries: it assumes every library method returns a nullable value unless otherwise specified. On the other hand, it is optimistic about parameters: passing a nullable value to a method in an unannotated external class is allowed. ESC/Java requires at least class files of the libraries to be available in the CLASSPATH. Even though because of its modular nature ESC/Java will not look at the code, it uses the class and jar files to find out the exact types of external classes and methods.

3.1.4 Spec#

Spec#, developed at Microsoft, extends the C# languages in a manner much similar to ESC/Java. A notable difference is that it includes some shorthands for annotations that are not encapsulated in comments, and thus break backwards compatibility with C#. However there is always an encapsulated alternative, and the shorthands allow for a very elegant way to express nullness attributes, prefixing the type with an exclamation or question mark much like in Nice.

The similarity between ESC/Java and Spec# is no coincidence: many of the members of the research team that originally developed ESC/Modula and ESC/Java have now been hired by Microsoft to work on Spec#.

Spec# is quite tightly integrated with Visual Studio, which is inconvenient if you don't have valid licenses for that software available.

One of the main differences between the way nullness is handled in Spec# compared to ESC/Java is that Spec# introduces the fine-grained partially initialized 'raw' types [10] which are also implemented in the JastAdd extensions.

3.1.5 FindBugs

FindBugs¹⁶ [8] is a checker that was developed at the University of Maryland. FindBugs takes a pragmatic approach to finding various common suspicious patterns in code, including null pointer dereferences. The FindBugs philosophy is that annotations, which are supported in the form of Java 5 annotations like

¹⁶<http://findbugs.sourceforge.net>


```

void Method (String s) {
    s.toString();
    s = null;
}

```

Figure 3.1: Small example to compare JastAdd and FindBugs

`@NonNull` and `@PossiblyNull`, are mostly meant to suppress a few remaining false warnings. FindBugs aims to find most problems without any user interaction, and does not prominently mention the annotations in the documentation. Like ESC/Java, it analyses only one method at a time, which makes it scale well.

The analysis consists of a data flow analysis on the control flow of each method separately. Like Eclipse, ESC/Java and Spec#, FindBugs works on an intra-procedural basis and allows for annotations to specify interprocedural nullness properties. For this thesis, we will use the term ‘type-based’ loosely, and classify the FindBugs analysis as a type-based approach.

The main difference between a more pure type-based approach like used in JastAdd and the data flow of FindBugs can be explained with the small example of Figure 3.1. JastAdd assigns a type to every element, so in this case `s` would have to be given a type. Because of the assignment `s = null`, the type of `s` should be equal or below the null type of the type lattice in Figure 4.5 — in other words, it should be either null or nullable.

In a data flow analysis, a type marking is added to every occurrence of an element: in this case, the `s` on method entry and both occurrences of `s` within the method. The data flow determines the constraints on the types of the occurrences: the parameter `s` should have the same or a more specific type compared to the first occurrence of `s` within the method. For example, if the lattice in Figure 4.5 would be used, and the first occurrence of `s` within the method was marked non-null, then the parameter `s` should also be non-null. The data in the second occurrence of `s` within the method, however, depends only on the right hand side of the assignment. In the data flow analysis, the nullness of this occurrence is completely independent of the nullness of the previous occurrences, since that data is overwritten. Because of that, unlike JastAdd, a dataflow analysis will have no problems with the assignment of null to a local element which was assumed to be non-null before.

The type lattice used for the dataflow analysis in FindBugs is not as simple as the one in Figure 4.5. To determine which possible violations warrant a warning and which can be ignored to prevent false positives, FindBugs has refined it. Figure 3.2 shows the dataflow lattice as used in FindBugs. To compare it to the type lattice more easily, Figure 3.3 shows it as a type lattice (for some reason in dataflow lattices the more general types are put at the bottom, while in type lattices they are at the top) with the nodes which refine the null, non-null and nullable types clustered.

In practice, the parameter `s` would be marked with the most uncertain variant of nullable: NCP, for Null on a Complex Path. While `s.toString()` is obviously a possible null dereference, FindBugs has chosen not to generate any warnings for dereferences of NCP values, only for the various variants of null,

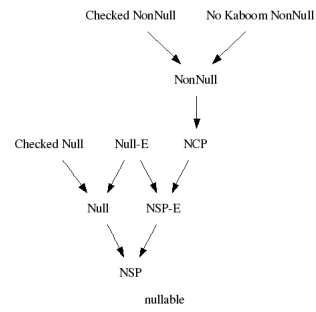


Figure 3.2: The FindBugs dataflow lattice

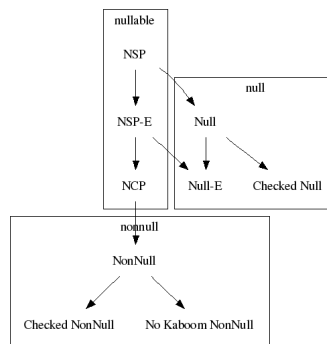


Figure 3.3: The FindBugs lattice as a clustered type lattice

```

class Example {
    void run (String nullable) {
        String external1 = Thread.currentThread().getName();
        String external2 = Thread.currentThread().toString();
        // force assumption that external1 is non-null
        external1.toString();
        String nonnullstring = new String ("Example");
        String t1 = returnParam1(nonnullstring);
        String t2 = returnParam2(nonnullstring);
        // force assumption that t1 is non-null
        t1.toString();
    }
    String returnParam1 (String param) {
        return param;
    }
    String returnParam2 (String param) {
        return param;
    }
    public static void main (String[] args) {
        Example e = new Example();
        // force parameter of e.run() to be nullable
        e.run(null);
    }
}

```

Figure 3.4: Annotation Assistance example, without annotations

NSP (Null on a Simple Path) and a low-priority warning in case of NSP-E (Null on a Simple Path due to an Exception).

3.2 Annotation Assistants

To better illustrate the differences between the main annotation assistants we have looked at (JastAdd, Houdini and CANAPA), we have constructed the small example of Figure 3.4. The example shows calls to external code, one that is assumed to return non-null and one that is not. It also includes two methods where the nullness of their return values depend on the nullness of their parameter. They are passed a non-null object, and for one of them the non-nullness of the return value is in fact also assumed by the calling code.

3.2.1 JastAdd Nullness Inferer

The JastAdd Nullness Inferer¹⁷ [9] is a tool that performs an analysis akin to type inference on the Java source code, inferring nullness attributes. Like the JastAdd nullness checker discussed in section 3.1.2, it has been implemented as a compiler based on the JastAdd compiler compiler system: in this case, the

¹⁷<http://jastadd.cs.lth.se/examples/NonNullTypesExtension/>

```

class Test {
    String st = new String("");
    void one() {
        two(false, null);
    }
    void two(boolean sisnonnull, String s) {
        if(sisnonnull) {
            st = s;
        }
        three(st);
    }
    void three(String stp) {
        stp.toString();
    }
}

```

Figure 3.5: Error creep in JastAdd

source language is Java, where the target language is Java with JastAdd nullness tags inserted.

Although it does not explicitly use type variables, it defines the nullness of a variable in the Abstract Syntax Tree by looking at the nullness of the assignments to it, much in the same way the type inference equations would have been generated. For fields, it also checks that all constructors provide a value to the field. This means the system will infer non-nullness of variables and fields when possible, but if there is only one place where a nullable value is assigned to it, it will immediately be assumed to be nullable also. Because of this, a single variable that has erroneously been assumed to be nullable can have far-reaching impact on the analysis of the rest of the system. This can be illustrated with the small example in Figure 3.5.

JastAdd does not annotate a single element for this example. JastAdd obviously does not know about the implicit precondition that `s` is non-null if `sisnonnull` is true. This is normal: most inferrers would not have caught this. However, it has great impact on the rest of the analysis: it will not recognise that the assignment in `two()` assigns a non-null value to `st`. As a result, this assignment causes `st` to remain nullable. Because JastAdd effectively propagates nullness rather than non-nullness, this means the `stp` parameter of `three()` also remains nullable — regardless of its body.

For practical use, the recently developed JastAdd inferrer still not very usable: for example, apparently the abstract syntax tree includes a call to `<init>()` at the start of the constructor. This might be useful for any processing of the AST, but obviously it should not be included in the output as it is now. Also, a string assignment like `s = "foo";` will leave `s` as a nullable field, while `s = new String("foo")` correctly allows `s` to be a non-null element. Another practical issue is that it will remove any comments from the code.

In spite of those practical issues, the inferrer serves well to demonstrate how many different kinds of tools can be elegantly developed using the novel

```

class Example {
    void run(String nullable) {
        [NN]String external1 = Thread.currentThread().getName();
        [NN]String external2 = Thread.currentThread().toString();
        external1.toString();
        [NN]String nonnullstring = new String("Example");
        [NN]String t1 = returnParam1(nonnullstring);
        [NN]String t2 = returnParam2(nonnullstring);
        t1.toString();
    }
    [NN] String returnParam1([NN]String param) {
        return param;
    }
    [NN] String returnParam2([NN]String param) {
        return param;
    }
    public static void main([NN]java.lang.String[] args) {
        [NN]Example e = new Example();
        e.run(null);
    }
    Example(){
        <init>();
    }
}

```

Figure 3.6: Annotation Assistance example, JastAdd output

aspect-oriented approach implemented in the JastAdd system. It also gives a reasonable idea of the kind of results a type-based inferer might produce.

3.2.1.1 Example

When inferring the nullness properties of the example of Figure 3.4, JastAdd produces the output in Figure 3.6.

The points based on which the rest of the code is analysed are the two external calls, which are optimistically assumed to return non-null, and the fact that the `Example` and `String` constructors assigns a non-null value to `e` in the `main` function and `nonnullstring` in the `run` method, respectively.

All other non-null annotations are derived from these four points: for example, when deciding how to annotate `t2`, the JastAdd inferer would look at all assignments to this variable. In this case, this is only the return value of `returnParam2()`, the nullability of which is determined completely by the parameter in `returnParam2()`. This in turn is determined by all call sites of `returnParam2()`. In this case, the only place where `returnParam2()` is called is in `run()`, and there it is called with a non-null parameter because `nonnullstring` is assigned a freshly constructed `String`. Therefore, all these locations are marked non-null.

Notice that annotating `t2`, the return value of `returnParam2()` and the parameter of `returnParam2()` as nullable would also have been perfectly valid.

However, since `returnParam2` is here always called with non-null parameter, the non-null annotation is chosen. This clearly shows the JastAdd inferer is non-null-biased: as long as there is no indication a location might be null, it will always be annotated as non-null.

3.2.2 Houdini

The research group that developed ESC/Java also created an annotation inference tool to go with it: Houdini [11, 6], named after the great ‘ESCape’ artist. This seemed to be the most relevant past research in annotation assistance. While the papers on Houdini claimed great success, the code had not been touched since the original ESC/Java group was disbanded in 2000.

Houdini’s revival

Because in the mean time ESC/Java itself had evolved due to the work of Joe Kiniry and David Cok [5], and the Houdini code is very tightly coupled with ESC/Java, Houdini could not be built with the current ESC/Java2. To be able to get hands-on experience with Houdini and to possibly be able to extend it later in this project, we decided to revive Houdini and bring it back in line with ESC/Java2.

This was not a trivial job: even though Houdini was released by HP, it was not in a usable state, not even with the old version of ESC/Java as released by HP at the same time. Not only does the documentation seem out of date, incomplete and generally not written to be understood by anyone outside of the team, the tool was even missing an entire required Perl script. We were able to reconstruct this script by looking at the input that was given to it and the code that tried to read its output.

Another tricky bug that we solved occurred at some points where the Simplify theorem prover was invoked. Before passing the theorems we were interested in to the prover, it would be fed a set of ‘background predicates’ from a file. After some long debugging sessions, it turned out that apparently a copyright message was mass-added to several files, including the file with background predicates. Unfortunately, this copyright message was commented out with a `#` character, instead of with a `;` which is how Simplify comments are required to be marked. Because of that, Simplify would try to prove the copyright message correct and obviously failed.

Our efforts now have allowed to run Houdini on small- and medium-sized examples again, though there are still cases that cannot be handled yet. Most notably, we have not been able to succeed to have the CopyLoaded module support external jar files. Because of this, we are not yet able to demonstrate Houdini on real production code. We have been able to experiment with the way Houdini handles external code, however, because it does support handling external code that is in the Java standard library. We could also run Houdini on itself, with frankly rather unsatisfying results, especially in terms of performance: even though the Houdini code was already annotated with various ESC/Java annotations, the analysis took many hours.

A start has been made to contribute our changes back to the ESC/Java community.

Type of reasoning

Houdini works by first adding many possible annotations to the program. Then, ESC/Java is invoked repeatedly to refute invalid guessed annotations. This process ends when no more annotations can be revoked. In other words, the analysis implemented by Houdini implements is non-null-biased and employs nullness-propagation.

Use

The user is mainly expected to use Houdini as a front-end of ESC/Java. Houdini will output pretty-printed hyperlinked code, including (in grey) any revoked guessed annotations. When dealing with an ESC/Java error, the user finds out why an annotation was revoked by clicking on it, hopefully leading to the real error in only a few steps.

Handling of libraries

Houdini supports and promotes the use of specification files for libraries. When such specifications are not available, however, Houdini makes *optimistic* assumptions about the library: it assumes for each library method that it guarantees to return a non-null value, and that it is okay to pass null as a parameter of any library method. This contrasts to ESC/Java, which makes *pessimistic* assumptions about return values in absence of library annotations. In order to implement optimistic assumptions Houdini generates a .spec file for any unannotated external class, specifying the optimistic assumptions for every method in this class whether it is needed or not. In other words, Houdini makes *maximal* assumptions.

Evaluation

Despite our high expectations due to the positive evaluation of Houdini in its corresponding research papers, this might not be the most convenient technique to use in practice. The analysis is rigorous and able to catch subtle bugs, but the process is monolithic and the output can be cryptic.

Houdini needs to finish running on the whole codebase before it generates a report. As can be expected from such a rigorous logic-based analysis, it appears to be rather resource-intensive, judged from our tests running Houdini on its own source code. Since the annotation process is inherently iterative and interactive (the analysis tool can only infer so much, after every run there are bugs to be fixed and annotations to be corrected manually), this way the annotation process remains prohibitively time-consuming.

The process is not easily modularized, since when running Houdini on only part of a codebase it will often see a method but not (all of) its call sites, and because of that fail to remove many annotations.

When running Houdini on the code in Figure 3.7, we were quite surprised to be presented the result in Figure 3.8.

The `requires false` annotation for `WordStat::increaseCount()` was puzzling: certainly `s.increaseCount()` was called in `addWord()`? The annotation is, however, in fact correct: because the stub implementation of the `findStats()` method always returns null, `s` is always null in `WordCounter::addWord`,

```
class WordStat {
    WordStat (String s) { }
    void increaseCount () { }
}
class Dictionary {
    WordStat findStats (String word) {
        return null;
    }
    void addStat (WordStat stat) { }
}
class WordCounter {
    void addWord (Dictionary dict, String word) {
        WordStat s = dict.findStats (word);
        if (s == null) {
            WordStat new_stat = new WordStat (word);
            dict.addStat (new_stat);
        } else {
            s.increaseCount();
        }
    }
}
class Test11 {
    public static void main (String[] args) {
        Dictionary mydict = new Dictionary();
        WordCounter C = new WordCounter ();
        C.addWord (mydict, "Henk");
    }
}
```

Figure 3.7: Houdini example, not annotated


```

class WordStat {
    /*@(houdini:constructor) requires false; */
    WordStat (/*@(houdini:parameter:constructor) non_null */ String s) {}
    /*@(houdini:instance method) */final/* */
    /*@(houdini:instance method) requires false; */
    void increaseCount () { }
}
class Dictionary {
    /*@(houdini:defaultconstructor) */public Dictionary(){}
    /* Explicating default constructor here */
    /*@(houdini:instance method) */final/* */
    /*@(houdini:instance method) ensures \result != null; */
    /*@(houdini:instance method) ensures \fresh(\result); */
    /*@(houdini:instance method) ensures word == null ==> \result != null; */
    /*@(houdini:instance method) ensures word != null ==> \result != null; */
    WordStat findStats (
        /*@(houdini:parameter:instance method) non_null */ String word) {
        return null;
    }
    /*@(houdini:instance method) */final/* */
    /*@(houdini:instance method) requires false; */
    void addStat (/*@(houdini:parameter:instance method) non_null */ WordStat stat) {}
}
class WordCounter {
    /*@(houdini:defaultconstructor) */public WordCounter(){}
    /* Explicating default constructor here */
    /*@(houdini:instance method) */final/* */
    void addWord (/*@(houdini:parameter:instance method) non_null */ Dictionary dict,
        /*@(houdini:parameter:instance method) non_null */ String word) {
        WordStat s = dict.findStats (word);
        if (s == null) {
            WordStat new_stat = new WordStat (word);
            dict.addStat (new_stat);
        } else {
            s.increaseCount();
        }
    }
}
class Test11 {
    /*@(houdini:defaultconstructor) */public Test11(){}
    /* Explicating default constructor here */
    /*@(houdini:parameter:static method) requires \nonnullelements(args); */
    public static void main (String[] args) {
        Dictionary mydict = new Dictionary();
        WordCounter C = new WordCounter ();
        C.addWord (mydict, "Henk");
    }
}

```

Figure 3.8: Houdini example, automatically annotated

so the else-branch is dead code and `ensures false` is not revoked. Even though it is interesting that this was caught by Houdini, we believe it is rather hard to interpret Houdini’s output correctly. Because annotation assistance is usually performed only once on a codebase (after which the checker is used on a regular basis to verify changes), we believe tools should preferably have a shorter learning curve.

Even more importantly, this behaviour makes it cumbersome to run Houdini in a modular fashion, since `WordState.increaseCount()` might be callable from outside. For clarity, we put the `WordStat` class in the same Java file as the `Test11` class here, but the result is identical if we make it a public class and move it to its own file — regardless of whether we run it on both `Test11.java` and `WordStat.java` at once, or separately for each.

3.2.2.1 Example

First, Houdini adds candidate annotations as shown in Figure 3.9. After adding these candidate annotations, ESC/Java is ran over this code. This will obviously lead to many warnings, as the candidate annotations include invalid annotations which certainly will not be satisfied. For example, ESC/Java will warn that the precondition *requires false* is not met by the statement `e.run(null)` in `main`. In response to this warning, Houdini will remove this precondition, after which ESC/Java will warn that `e.run(null)` violates the requirement that the `nullable` parameter should be non-null. This requirement is also removed, which shows that Houdini propagates nullness. This process repeats itself until ESC/Java reports no more errors¹⁸, after which the example looks like the code in Figure 3.10.

We notice that Houdini infers many annotations, including relatively advanced ones like `/*@houdini:instance method) ensures param != null ==> \result != null; */`. The requirement that the parameter of `returnParam1` is always non-null, however, is too strong. Also, the advanced annotation `/*@houdini:instance method) ensures param == null ==> \result != null; */` is incorrect if the parameter is indeed allowed to be null. Since `param` is here required to be non-null this annotation is, strictly speaking, correct — but logically empty because the condition `param == null` never holds.

It can be clearly seen from the `returnParam` methods that Houdini is non-null-biased. The maximal optimistic assumptions Houdini makes about the environment are not immediately visible in the output, but show when you look at the spec files that have been generated in the process: not only `Thread.currentThread().getName();` is specified to return non-null, but also `Thread.currentThread().toString();`

3.2.3 CANAPA

CANAPA, like Houdini, invokes ESC/Java iteratively. However, instead of assuming many annotations and working from there, it runs ESC/Java on the raw Java code, and tries to fix any warnings by doing some fairly conservative

¹⁸In fact, Houdini does not really call ESC/Java each time: some optimizations have been implemented which allow Houdini to directly manipulate the ESC/Java verification conditions and call the Simplify prover itself. These optimizations are described in [21]. They do not, however, change the process from a conceptual point of view, so for this thesis we will describe the (much simpler) original design.

```

class Example {
  /*@(houdini:defaultconstructor) */public Example(){}
  /* Explicating default constructor here */
  /*@(houdini:instance method) */final/* */
  /*@(houdini:instance method) requires false; */
  void run (/*@(houdini:parameter:instance method) non_null*/String nullable) {
    String external1 = Thread.currentThread().getName();
    String external2 = Thread.currentThread().toString();
    // force assumption that external1 is non-null
    external1.toString();
    String nonnullstring = new String ("Example");
    String t1 = returnParam1(nonnullstring);
    String t2 = returnParam2(nonnullstring);
    // force assumption that t1 is non-null
    t1.toString();  }
  /*@(houdini:instance method) */final/* */
  /*@(houdini:instance method) ensures \result != null; */
  /*@(houdini:instance method) requires false; */
  /*@(houdini:instance method) ensures \fresh(\result); */
  /*@(houdini:instance method) ensures param == null ==> \result != null; */
  /*@(houdini:instance method) ensures param != null ==> \result != null; */
  String returnParam1 (
    /*@(houdini:parameter:instance method) non_null */ String param) {
    return param;
  }
  /*@(houdini:instance method) */final/* */
  /*@(houdini:instance method) ensures \result != null; */
  /*@(houdini:instance method) requires false; */
  /*@(houdini:instance method) ensures \fresh(\result); */
  /*@(houdini:instance method) ensures param == null ==> \result != null; */
  /*@(houdini:instance method) ensures param != null ==> \result != null; */
  String returnParam2 (
    /*@(houdini:parameter:instance method) non_null */ String param) {
    return param;
  }
  /*@(houdini:parameter:static method) requires \nonnullelements(args) */
  public static void main (String[] args) {
    Example e = new Example();
    // force parameter of e.run() to be nullable
    e.run(null);
  }
}

```

Figure 3.9: Annotation assistance example with Houdini’s candidate annotations

```

class Example {
  /*@ (houdini:defaultconstructor) */public Example(){
    /* Explicating default constructor here */
    /*@ (houdini:instance method) */final/* */
    void run (String nullable) {
      String external1 = Thread.currentThread().getName();
      String external2 = Thread.currentThread().toString();
      // force assumption that external1 is non-null
      external1.toString();
      String nonnullstring = new String ("Example");
      String t1 = returnParam1(nonnullstring);
      String t2 = returnParam2(nonnullstring);
      // force assumption that t1 is non-null
      t1.toString();  }
    /*@ (houdini:instance method) */final/* */
    /*@ (houdini:instance method) ensures \result != null; */
    /*@ (houdini:instance method) ensures param == null ==> \result != null; */
    /*@ (houdini:instance method) ensures param != null ==> \result != null; */
    String returnParam1 (
      /*@ (houdini:parameter:instance method) non_null */ String param) {
      return param;
    }
    /*@ (houdini:instance method) */final/* */
    /*@ (houdini:instance method) ensures \result != null; */
    /*@ (houdini:instance method) ensures param == null ==> \result != null; */
    /*@ (houdini:instance method) ensures param != null ==> \result != null; */
    String returnParam2 (
      /*@ (houdini:parameter:instance method) non_null */ String param) {
      return param;
    }
    /*@ (houdini:parameter:static method) requires \nonnullelements(args) */
    public static void main (String[] args) {
      Example e = new Example();
      // force parameter of e.run() to be nullable
      e.run(null);
    }
  }
}

```

Figure 3.10: Annotation assistance example: Houdini output

inferences. For example, it only works on method parameters, local variables and function results. It never infers the non-nullness of a class attribute.

Type of reasoning

CANAPA runs ESC/Java with the '-Suggest' flag, which makes ESC/Java provide a suggestion on how to fix each problem. CANAPA works by running ESC/Java with the '-Suggest' option enabled. In case of nullness conflicts, it recognises suggestions such as:

```
tolk/GoToDigiD.java:33: Warning: Possible null dereference (Null)
    applChosen = request.getParameter("appl");
                        ^
Suggestion [33,22]: perhaps declare parameter 'request'
at 47,55 in tolk/TolkServlet.java with 'non_null'
```

Whenever it sees such a suggestion, it adapts that file as suggested and re-runs ESC/Java, until no new suggestions become visible.

This is a nullable-biased system using non-null propagation (as ESC/Java only suggests the addition of invariants, never their removal), even if there might be strong suggestions that reference is in fact nullable. Like standard ESC/Java, it makes pessimistic assumptions about external libraries: suggestions to add annotations to external code are ignored by CANAPA.

Use

CANAPA is meant to be run one single time over the unannotated source code. After that, the developer will have to run the checker and fix any remaining issues by hand.

3.2.3.1 Example

When running the original CANAPA over our example class, the pessimistic assumptions ESC/Java makes about the environment: the first line of the run method causes an error because `Thread.currentThread()` is assumed to return a nullable reference. Since ESC/Java generally only shows the first warning per method, CANAPA would return immediately after showing this warning, without adding any annotations.

We decided to help CANAPA on the way a bit by adding a `//@nowarn` pragma to the external calls. This time it could run unobstructed, and resulted in the annotated code in Figure 3.11.

The effect of CANAPA being nullable-biased shows clearly: there are considerably fewer spurious annotations added. Of course the annotations of `external1` and `t1` are not strictly necessary any more, though they might serve a purpose when the annotations about `returnParam1()` or `Thread.getName()` would be revoked. The annotation of the return value of `returnParam1` is not entirely correct, of course: the application does assume it to be non-null, but not necessarily in all occasions. The developer would later have to refine this specification.

Even though we had to manually intervene to suppress the warnings about external code, the CANAPA output is much cleaner, which is a desirable property for an annotation assistant. Even though fewer annotations have been

```

class Example {
    void run (String nullable) {
        /*CANAPA*//*@ non_null @*/String external1 =
            Thread.currentThread().getName(); //@nowarn
        String external2 = Thread.currentThread().toString(); //@nowarn
        // force assumption that external1 is non-null
        external1.toString();

        String nonnullstring = new String ("Example");
        /*CANAPA*//*@ non_null @*/String t1 = returnParam1(nonnullstring);
        String t2 = returnParam2(nonnullstring);
        // force assumption that t1 is non-null
        t1.toString();
    }
    /*CANAPA*//*@ non_null @*/String returnParam1 (String param) {
        return param;
    }
    String returnParam2 (String param) {
        return param;
    }
    public static void main (String[] args) {
        Example e = new Example();
        // force parameter of e.run() to be nullable
        e.run(null);
    }
}

```

Figure 3.11: Annotation assistance example: CANAPA output

added, the tool does infer sufficient annotations to allow ESC/Java to verify the absence of `NullPointerException`s.

3.2.4 Julia

Julia¹⁹ [25], the ‘Java Universal Interpreter through Abstraction’, is a generic static analysis tool that provides a framework for applying the abstract interpretation technique to Java bytecode. It was developed by Fausto Spoto at the University of Verona, Italy. Abstract domains can be ‘plugged into’ Julia to specialize its behaviour. One of the analyses claimed to be implemented with Julia is a non-nullness analysis.

Unfortunately, the tool is not very easy to use and undocumented. Even though the research paper describes the general abstract interpretation implementation in some detail, the implementation of the non-nullness analysis is not described at all. Due to the lack of documentation we have not been able to install and test this tool in practice.

3.2.5 Daikon

Daikon²⁰ [23] was developed by the Program Analysis Group at the MIT Computer Science and Artificial Intelligence Laboratory. It takes a dynamic approach to inferring invariants: instead of analysing the code itself, it runs the program and tries to find likely invariants by observing the actual values of fields.

While this approach is certainly interesting, it has some drawbacks for our situation. It requires that the software under consideration can be executed with reasonable code coverage. This might not be a problem for small codebases, but for large projects that for example might communicate with a database or other software this can be hard to do. On the whole, Daikon seems to be most suitable for analysing small amounts of complex code: Daikon output can be rather verbose and complicated, and generally does not perform very well on large programs resource-wise either.

Using Daikon will generally require some effort: the critical code will have to be identified and a test suite constructed that runs this part of the code achieving a reasonable code coverage.

This thesis focuses on assisting with the tool-supported annotation of an existing, previously unannotated codebase. As this means we want to reduce the amount of manual intervention required, and because the codebase might contain a considerable amount of code, Daikon does not seem to be a suitable tool for our specific problem.

¹⁹<http://profs.sci.univr.it/~spoto/julia/>

²⁰<http://pag.csail.mit.edu/daikon>

Chapter 4

Requirements and design considerations

This chapter will first look at high-level requirements of an annotation assistant. Subsequently we will take a more in-depth look at the design space, and evaluate what choices were made in the projects we described in the previous chapter. For each design choice we make a recommendation on what choice would seem to be the most suitable for an annotation assistant.

4.1 Requirements

The main goal of annotating software is to find as many bugs as possible, and get some guarantees that certain problems will not occur at runtime. The annotations should reflect the application’s design as closely as possible.

An annotation assistant is a tool which is designed to make the task of adding annotations to code easier. This is mainly important when processing an existing codebase which was not checked with a checker like ESC/Java before. In this scenario the codebase is relatively stable and tested by traditional means, but lacking annotations. Without an annotation assistant, adding these annotations is a tedious and time-consuming task: experience shows that especially missing non-null annotations are a large source of false positives. Once the codebase is sufficiently annotated to remove most of the false warnings, the use of an annotation assistant becomes less important: instead, the checker can be run on a regular basis to identify any new problems that might have been introduced. Those problems can probably be easily resolved by hand: they might point to bugs in the recently added (and thus not yet thoroughly tested) code, or suppressible by just a few additional annotations.

The most high-level requirement of an annotation assistant is simple: it should make the task faced by the developer easier.

This requirement can be split into two parts: on the one hand it should actually assist the developer by adding suitable annotations to the code, faster than it could have been done manually. On the other hand it should not “get in the way” of the developer. The latter requirement can be split into some more concrete guidelines:

- The assistant should honour any already manually inserted annotations
- It should keep the code (including formatting and comments) intact as much as possible
- It should keep spurious and incorrect annotations to a minimum

The next section will look at some design considerations in more depth and determine which choice best fits the requirements for an annotation assistant.

4.2 Design considerations

4.2.1 Code external to the analysis

An important aspect is handling unannotated libraries and other code called by the program under analysis. In section 2.1.3, we have introduced some terminology to classify the ways external code can be handled.

As we have seen, the tools that focus on reducing false positives (like Eclipse and FindBugs) make maximal optimistic assumptions. Houdini and the JastAdd Inferer also chose this approach.

Our goal, however was to annotate the code to be more suitable for checking with ESC/Java, which makes pessimistic assumptions about the environment in order to produce more reliable results. Therefore an annotation assistant which makes maximal optimistic assumptions is likely to infer many spurious non-null annotations, especially if it is non-null biased. Making pessimistic assumptions, however, might lead to many missed non-null annotations.

A viable middle road for annotation assistants is, we feel, is to record where an optimistic assumption is actually needed by the application. The optimistic assumption can be used temporarily for the analysis, but will eventually have to be verified in some other way — either manually or possibly also tool-assisted. After verifying and possibly removing an assumption about external code, the annotation assistant should be able to resume its task without this assumption.

4.2.2 Limits of type-based approaches

Type-based nullness analysis such as that performed by JastAdd (seen in section 3.1.2) or Nice (section 2.3.1.2) often introduces a considerable annotation burden, because it is not able to analyse the effect of indirect nullness information.

Consider the simple example of Figure 4.1: in this method, a type-based approach will not be able to infer that `foo` is not null in the bodies of the if-statements. Most type-based approaches are pragmatic and can recognise common checks for null such as the first conditional of the example. More complex conditionals, such as the second one, are generally not caught by such tools. One solution to this might be to force developers to adhere to certain programming patterns. Eiffel requires the developer to write the check in a special syntax, like this:

```
if (x : T) exp then
  ... instructions ...
end
```

```

void Method (Object foo)
{
    // foo can be nullable here
    if (foo != null) {
        // this dereference is safe
        foo.toString();
    }
    boolean fooisnull = (foo == null);
    if (!fooisnull) {
        // this dereference is also safe
        foo.toString();
    }
}

```

Figure 4.1: Example showing limits of type-based approaches

```

void Method (Object foo, boolean fooisnonnull)
{
    if (fooisnonnull)
    {
        foo.toString();
    }
}

```

Figure 4.2: Example of a method with a simple invariant

This evaluates `exp`, and if this evaluates to an attached (non-null) type, the local, read-only variable `x` is filled with the result and the body executed. If the expression evaluates to null, the body is skipped. It might be argued that such restrictions improve the clarity and readability of the code, on the other hand being required to add some code to make the type system happy is something that many developers tend to object to. Especially when retrofitting existing code (which has been developed without these restrictions in mind) refactoring the code to adhere to these new rules might be a cumbersome task which is hard to automate. More rich methods, such as the logic-based analysis of ESC/Java, will generally be able to automatically infer the correctness of constructs like in this example.

More complicated cases such as the one in Figure 4.2 may or may not be caught by a logic-based analysis, but in any case can be resolved by adding an annotation, for example the precondition `//@requires fooisnonnull ==> foo != null`. Such annotations might be too complex to be automatically inferred, but after the developer has specified them it would be desirable that a subsequent analysis by the annotation assistant would take this precondition into account. Type-based methods will generally not be able to take advantage of this information.

If the invariant is too complex for the developer to write down, this probably means it is a good idea to either refactor the code or add a run-time check to prevent errors. As a last resort an `//@assume` statement can be used.

```

public void setMultizoek(String multizoek) {
    this.multizoek = multizoek;
    evaluateMultizoek();
}
(...)
//@requires this.multizoek != null
private void evaluateMultiZoek() {
    (...) multizoek.length() (...)
}

```

Figure 4.3: An example from the Promedico HIS case study

Another example that shows a situation where a type-based approach is not sufficiently powerful is shown in Figure 4.3, taken from the Promedico HIS case study: the `PatientZoekCriteria` class contained a field called `multizoek` which was quite definitely nullable. However, if filled, some processing and validation was performed. The private method which performed this step, `evaluateMultiZoek()`, was called from the setter method for this field. Unsurprisingly, it assumed the `multizoek` field was non-null, which is not the case in general. In a type-based setting, probably the code would need to be changed: possibly by adding a spurious check for nullity or by passing the value as a parameter rather than via the field. The latter solution could arguably be a more elegant solution in this case, but in general it would be desirable not to be required to change the code. With the logic-based approach as provided by ESC/Java, we could simply specify the precondition `this.multizoek != null` for `evaluateMultiZoek()`.

Note that for example a non-modular (interprocedural) dataflow-based analysis could also have verified the correctness of this case. The logic-based approach shown here, however, results in code with a clear, formal specification (of this aspect), that remains checkable in a modular way for performance, scalability and reusability.

4.2.2.1 Combining type- and logic-based analysis

A possible way to stretch the limits of type-based methods could be to combine it with logic-based techniques. Type-based annotation assistants will generally not be able to take into account the more advanced invariants added by the developer. This could be worked around by adding assertions to the code: revisiting the example in section 4.2.2, now adding an assertion as shown in Figure 4.4.

The type-based algorithm would treat the assertion as an assumption, and therefore `foo.toString()` would be accepted by the type-based algorithm. The logic-based system could complete the system by treating the assertions as proof obligations. This way, the type system guarantees the absence of `NullPointerException`s assuming that the assertions hold, and the logic-based system verifies that these assumptions indeed hold. Hence, these techniques combined would prove the absence of `NullPointerException`s.

```

void Method (Object foo, boolean fooisnonnull)
{
    if (fooisnonnull)
    {
        assert foo != null
        System.out.println(foo.toString())
    }
}

```

Figure 4.4: Code with a simple invariant, revisited

4.2.2.2 Nullness type inference

The technique described in the previous section is not very suitable for implementing an annotation assistant: the annotation burden of adding non-nullness assertions actually looks a lot like the problem we wanted to solve in the first place. Since the kind of checkers we are targeting will do more ‘intelligent’ things than just type checking (though they might deploy some typechecking techniques transparently ‘under the hood’), most of the the assertions will probably be spurious once all non-null and other annotations are properly added. Hence, it does not seem to be a good idea to require the user to spend a lot of effort adding these assertions.

However, if the assertions could be automatically inferred, this might still be a viable route.

Adding a ‘complex non-null’ type One possible way of inferring these assertions was thought to be a classical type inference algorithm — though one with a ‘twist’ in that the nullability-type of a variable can change due to a conditional comparing the value to null. We would have a very simple type system consisting of the types ‘non-null’, ‘nullable’ and ‘complex non-null’. The trick would be that it would be safe to assign a nullable value to a complex non-null location, and the nullity would then generate a verification obligation for that location. Checking the verification obligation would be left to the logic-based part of the system.

It is, unfortunately, easy to see that this approach will not work with a traditional type inference algorithm. The normal nullness type lattice is trivial: it simply contains only ‘nullable’ and ‘non-null’, where $\text{non-null} <: \text{nullable}$ (since non-null values are ‘special cases’ of the more general set of nullable values. You can assign a non-null value to a nullable field, but not vice-versa). We could add a ‘null’ element to the lattice (with $\text{null} <: \text{nullable}$), but it seems this merely makes the type lattice more complicated without providing any additional value.

The ‘complex non-null’ type cannot be correctly added to this lattice: it would have to be assignable to both nonnull and nullable types (and thus $\text{complex-non-null} <: \text{non-null}$), but it should also be possible to assign nullable values to it (these are the assignments that would need to be checked later), requiring that $\text{nullable} <: \text{complex-non-null}$. By this reasoning, the type lattice has become a graph, since $\text{nullable} <: \text{complex-non-null} <: \text{nonnull} <: \text{nullable}$.

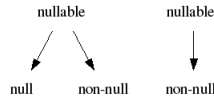


Figure 4.5: Normal nullness type lattice with and without specific null type

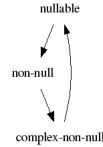


Figure 4.6: complex-non-null does not fit into the lattice

More complex type lattices FindBugs, which we evaluated in section 3.1.5, uses the more complex lattice shown in Figure 3.3 to guide its analysis, representing various degrees of certainty about nullability. This works well because FindBugs is performing a more fuzzy analysis mainly targeted at reducing false warnings, without providing guarantees about the correctness of the code in the absence of warnings. Such a more fine-grained lattice is less suitable for the more solid analyses like ESC/Java performs, but might be useful in an annotation assistant to determine whether or not to propagate an annotation, or to choose which annotation to propagate in case of conflict.

A heuristic type inference algorithm Another approach would be to adapt the type inference algorithm. Type inference algorithms are generally designed to work on well-typed code, and simply abort with an error message when a type conflict is found. Unfortunately, the location where the first type conflict is found is not necessarily close to the location of the actual problem. This is a well-known problem in languages that do aggressive type inference, such as Haskell or ML. Even though effort is being put in finding the most likely root cause of the type conflict [12] and for improving the error messages themselves [14], this remains a hard problem. The known existing techniques are tuned to a situation where there are few type conflicts and they are corrected manually one by one, where by contrast the code which is analysed by the annotation assistant contains many ‘conflicts’ (which may be checkable by the logic-based checker). This makes the known techniques less suitable for type inference in annotation assistants.

A traditional type inferer generally works in two phases: first, it parses the code, introduces type variables and records restrictions about those type variables. After that, it uses some algorithm to find values for the type variables such that all recorded restrictions hold.

A possible way to do type annotation assistance would be to add type variables and record restrictions as usual, but instead of satisfying the restriction set completely, finding a set of solutions that corresponds to the restrictions best (for some reasonable definition of ‘best’). Then the type checking process

would easily find those spots where the restrictions could not be satisfied, and invoke the intelligent checker to possibly cover those cases.

Type variables and constraints could be represented as a graph with special directed edges. The 'nullability' of each node could be derived from the nullability of the nodes directly connected to it. In case these surrounding nodes contradict each other, an analysis of the 'certainty' of those nodes could be made to make an informed decision.

This seems like an interesting approach, but we chose to abandon this in favour of more checker-driven approaches. We certainly do not wish to dismiss technique, but as we have not been able to find any existing research on this approach (let alone implementations), we have decided not to pursue this idea further in the context of this project. It remains, however, interesting and promising future work.

4.2.3 Impact of incorrectly inferred annotations

An annotation assistant will generally not be able to annotate a program entirely, eliminating all conflicts. Possible sources of conflicts include:

- Actual bugs in the application under analysis
- Dereferences that are guarded by a conditional which indirectly implies the non-nullness of the reference
- Situations where a reference cannot be null because of a complex invariant that can not be inferred automatically

Because this problem is so common, an annotation assistant must be relatively robust against it. This causes problems especially in type-based assistants such as JastAdd: one conflict can have a huge impact on the inferred annotations of other elements, as has been illustrated in section 3.2.1. FindBugs and ESC/Java suffer less from this problem because of their modular nature, which prevents small mistakes to spread across modules.

It is hard to prevent this problem from occurring, but to mitigate it we feel it could be desirable to propagate both 'nullness' and 'non-nullness' information, and to be able to represent the confidence in an annotation. The latter could be done with a more fine-grained nullness type lattice like the FindBugs lattice, or for example by discriminating between annotations that have been added by the developer and those which have been added by the annotation assistant.

When in doubt, we believe it is preferable for an annotation assistant not to add the annotation, and to leave it up to the developer. An annotation assistant that fails to add some annotations is still probably an improvement over no assistance at all. An assistant which adds incorrect annotations, however, can be frustrating to work with and be perceived to do more harm than good.

4.2.4 Initial Annotations

A tool calling the checker to identify problems, like CANAPA, might require several iterations of adding obvious annotations before it gets to the interesting issues. This is problematic because running the checker tends to be resource-intensive, and doing it too often makes the assistant slow and therefore inconvenient to use.

To make processes reach a fixedpoint more quickly, it might be a good idea to start the annotation assistance process by adding some likely initial annotations, possibly removing them again later on.

We have determined some candidates for initial annotations:

- Fields that are directly initialized with some value
- Local variables that are not assigned a literal null
- Return values of methods that never return a literal null

This list could easily be extended with more and more advanced techniques, for example taking into account initialisations in static blocks or constructors, or even running a lightweight annotation inferer like the JastAdd Inferer. Experience will have to show which initial annotations yield the best results.

To get an idea of the effect such initial annotations would have on the analysis, we have added some simple initial annotations to a sample from our Promedico HIS case study and did some tests. The results of those tests are described in section 6.3.4.

4.2.4.1 Using a type-based analysis for pre-processing

Even though many annotations will be missed when using a type-based system like JastAdd, those annotations that are added are almost certainly correct, and the JastAdd analysis is *very* fast compared to logic-based approaches. If some of the practical issues were resolved, and the implementation was extended to support parsing existing annotations, JastAdd could still be a powerful nullness inference tool. Its main weakness is its inability to take into account more complicated invariants, which is possible ESC/Java is used.

It could well be imagined that a system like this would be put to use in the pre-processing step of our annotation algorithm.

4.2.5 Termination

CANAPA starts with an incompletely annotated application, and iteratively runs ESC/Java to add new annotations. CANAPA never removes any annotations, and will not add an annotation to the same place twice. Given that there are a constant number of locations where an annotation can be added, it is trivial to see that this process terminates.

Houdini initially adds a large number of annotations, and after that only removes them. It is easy to see this also terminates.

For approaches that are less clearly monotonic, other ways to ensure termination must be found. A simple yet usable method would be to honour two simple rules:

- Never change an explicit annotation by the developer
- Never add or remove the same annotation twice (also across iterations)

The first rule is required to prevent cases where the annotation assistance algorithm keeps adding an application which is incorrect, and the developer must

keep removing it because it is incorrect. We feel that for an assistance tool it is always a bad idea to override the choice of the developer.

The reason for the second rule is twofold: primarily, the removal of an automatically generated annotation by the developer is also a choice that should not be overridden by the assistant. Also, this rule prevents endless loops where the tool keeps adding and subsequently removing the same annotation.

4.2.6 Interactivity

Even if the annotation assistant has a reasonable response to conflicts, it is still unlikely that the assistant “gets it right” all the time.

Many of the assistants we have seen simply run once, and leave the developer on his own after that. However, because conflicts that cannot be automatically resolved occur so often, it is highly desirable that the tool can be run iteratively. The tool should obviously take into account any annotations from previous runs and those that were added by the developer. Also, if the developer has removed an automatically generated annotation, of course the tool should never add this annotation back again.

The JastAdd inferer does not currently support parsing and assuming existing annotations. This could be added relatively easily, although JastAdd supports merely simple annotations, not complex JML invariants and pre/postconditions. Houdini and CANAPA fully support being re-run on code which already contains annotations. However, because their logic-based analyses are rather resource-intensive, they are not very suitable to be run iteratively. We will elaborate on this issue in section 4.2.8.

4.2.7 Scalability

Unlike type-based systems like the JastAdd inferer, logic-based inferers like Houdini and CANAPA tend to be very resource-intensive. To ensure scalability (both in time and in memory usage), it seems desirable to introduce some more modularity into the analysis.

For example, the CANAPA system runs ESC/Java over the entire codebase under analysis each iteration. This caused some analyses to be needlessly performed multiple times.

This led us to try to first reach a fixedpoint for each source file separately, and then maybe do a final run over the entire codebase. This might change the order in which annotations are added, and therefore it might change which annotations are added at all: for example, suppose there are two locations which assume a certain method to return non-null. The second location consists of a local variable to which the result of this method is assigned, and which is subsequently dereferenced. If the first location is encountered before the second location, the second location is probably accepted without adding any further annotations. However, when the second location is encountered earlier, it might first add an annotation to the local variable and only after that to the method. This is of course a rather harmless example, and our intuition is that most differences will be of such a subtle form.

We have tested this approach on our DigiD test case. The results of this tests were quite positive, though not as radical as we hoped. The results are described in section 6.2.1.

While this introduction of modularity appears to have a positive impact on performance, its real benefit lies in the fact that it makes a higher level of interactivity possible, which will be discussed in the next section.

4.2.8 Iterative, Modular Interaction

Modularity is not only useful to speed up the entire annotation process: it can also prove very useful to improve the interactivity in the annotation process. As we have seen in section 4.2.6, interactivity is an essential aspect of an annotation assistant.

Assistants like Houdini are implemented in a modular way internally, but from the users' perspective they are still monolithic: one has to wait for the entire automated annotation to finish before annotations can be added and bugs can be fixed, after which the entire analysis will have to be run all over again.

An approach that we consider much more practical is to introduce more modularity in the interaction with the tool. As we have seen a lightweight checker-directed assistant like CANAPA can be successfully applied to separate files or even methods before looking at the entire program at once. We can exploit this by designing a user interface which allows the developer to review analysed modules while the assistant is still working on analysing the rest of the application. After making changes to a module it can be added to the queue of modules in need of analysing again, until the developer accepts the result.

Another advantage of this approach is that it might make it easier to see what effects a small change in the code has. However, once you are already working with a reasonably well-annotated version of the source, just running the checker (and not using the annotation assistant at all) will likely be sufficient. Also, applying richer visualisation techniques might be a more suitable way to get an insight of how the nullness aspects of a given piece code relate, which is future work as described in section 7.1.3.

Chapter 5

Annotation Assistance Algorithm

Based on the experiences and observations we gathered while working with the tools described in the previous chapters, we have chosen to modify the CANAPA algorithm to meet the requirements we formulated in section 4.1. In this chapter we will describe the updated algorithm and the rationale behind some of its properties in some detail.

We have also developed a proof-of-concept implementation of the algorithm, in part based on the code from the CANAPA project which was generously distributed as Free, Open Source Software under the GNU General Public License. This implementation, which was also used to get some of the the benchmark measurements of chapter 6.1, is described in more detail in section 5.5.

5.1 Algorithm description

Input is a queue of files F , initially in state ‘initial’. A set of conflicts C is obtained by running the checker over a source file. This set is converted to a set of suggestions S . The process adds annotations to the files and records assumptions about the environment in the set A . During the process a history of previously handled suggestions is stored in the set H .

We have described the algorithm in some ad-hoc pseudo-code in Figure 5.1. We used a hash sign (#) to denote counting the elements of lists (which are denoted by < and > symbols) and sets. We express filtering a list with a set-like | notation: for example, the expression `#<f ∈ F | f.state == pending>` stands for the number of elements in the list of files f from F which are in the ‘pending’ state.

5.2 Interactivity

During the automatic processing the developer can manually set the state of a file to ‘blocked’, temporarily preventing it from being analysed. This gives him time to fix any issues in the code and annotations, after which he can put

```

forall <f ∈ F | f.state == initial>
  preProcess (f)
  f.state := pending
repeat:
  if #<f ∈ F | f.state == pending> == 0:
    wait for manual intervention
  else:
    current_file := head(<f ∈ F | f.state == pending>)
    current_file.state := busy
    repeat:
      C := runChecker (current_file, A)
      S := inferencesuggestions(C)
      n_new_suggestions := #(S \ H)
      for every s ∈ (S \ H):
        add s to H
        if assumption about the environment then
          add the assumption to A
        else
          add an annotation to a source file
      until n_new_suggestions == 0
      current_file.state := done
until manually aborted

```

Figure 5.1: Annotation Assistance Algorithm

it back to ‘pending’ to allow the assistant to incorporate those changes in the analysis.

He can also manipulate the order of the elements in the list of files. This way, he is in control of the order in which the files are analysed.

5.3 Termination

This process never truly terminates, since the developer may always put a files into pending state manually. However, in the absence of interaction, the process will automatically end up in a state where no files are left pending and the system waits for manual intervention.

It is easy to see this is the case: there is a finite number of files which can be pending, and every time the outer loop executes one of those files goes from ‘pending’ to ‘done’.

Of course this assumes the inner loop terminates. Fortunately, it is also easy to see that this is the case: observe that since there’s a finite number of elements in the code, there is also only a finite number of possible suggestions to solve any conflicts. Since every iteration at least one suggestion is implemented and added to the history set H , eventually $S \setminus H$ will be empty.

5.4 A note about soundness

One might notice that some changes to one file will affect other files. For example, a change in the nullability of a method's return value will affect the way the files that call it are verified. Yet, those other files are not put back into 'pending' state. This could be changed (without violating termination properties, even though the proof would get a little less clear), but the impact on the performance of the system outweighs the small improvement in results.

One needs not be concerned about the effect this choice has on the soundness of the system as a whole: soundness is not guaranteed by the annotation assistant, but by the checker. In a sense it is irrelevant whether the assistant is sound. After the initial assistant-aided annotation process is over, the checker can be run over the entire codebase, and will show any remaining problems which have not yet been fixed.

5.5 Implementation

This section gives a more concrete explanation of the algorithm. It can be safely skipped by readers only interested in the general methods and techniques, but may serve to give a better idea of how this algorithm can work in practice.

We based this proof-of-concept implementation on CANAPA, the Completely Automated Non-null Annotation Propagation Application. Since one of the key changes was to introduce a much higher level of interactivity in the annotation assistance algorithm, we dubbed our modified implementation IN-APA: the Interactive Non-null Annotation Propagation Assistant.

5.5.1 The codebase (F)

The queue of files, represented as **F** in the algorithm, is represented as a simple **Vector** of **SourceFile** objects. A **SourceFile** holds some general information, such as the file name, the current state of the file ('initial', 'pending', 'busy' or 'blocked') and the ESC/Java errors shown in the last run for this file.

5.5.2 Parsing Java

CANAPA was written in Java and used Jparse¹ for parsing the Java code. Unfortunately, the abstract syntax tree (AST) provided by Jparse is not very suitable for our needs: it turned out to be hard to find out where exactly in the tree an annotation should be inserted. This led to bugs which in some cases even broke the termination properties, making them tricky to debug.

To be able to continue testing, we have implemented a small Perl script to replace Jparse where it was used in CANAPA. While this is clearly a temporary solution, it does perform more accurately at adding the annotations in the correct place and allows us to run our proof-of-concept on real-world production code.

In the long run, it will be desirable for the application to have access to the Java code as an AST. Jparse provides a primitive tree, but does not support

¹<http://www.ittc.ku.edu/JParse/>, generated from a grammar with the ANTLR[24] language tools.

functionality like resolving bindings. This is required, for example, to find the definition of a method given its call site. To be able to parse the Java code into such an AST, modify it and write the modified code back to a file, we have started implementing functionality with the Eclipse Java Development Tools² (JDT) library where possible. This library can be used independent of the Eclipse IDE, and among other features provides an advanced AST API. This should especially prove worthwhile when implementing more advanced mappings from ESC/Java warnings to actual Suggestions.

In order to be able to record the assumptions about the environment in a way accepted by ESC/Java, the exact type of those external methods has to be looked up in their respective class or jar file. This would probably not have been feasible if we had not started using the Eclipse JDT library.

5.5.3 Initial annotations

The pre-processing step could be quite easily implemented now that we can use Eclipse's RewriteAST interfaces, even though it currently only does a very rudimentary job. It cycles through all SourceFiles, and for each field checks if the following conditions hold:

- Its type is not primitive, but a reference
- The field is not already annotated
- The field has an initializer
- This initializer is not the literal 'null' value

The pre-processor marks all class fields for which those conditions hold with `/*non_null*/`. After pre-processing each SourceFile, it sets its status to 'pending'.

If the annotation effort has been interrupted for some reason, the pre-processing step can be skipped when analysing code that has already been preprocessed.

5.5.4 File selection

As soon as files start to get into Pending state, INAPA will start processing them, one at a time, each time taking the first 'pending' file in the queue.

While a file is being processed, the user can manipulate the order of files in the queue, and change their state manually.

5.5.5 Running ESC/Java

Whenever a SourceFile is selected, ESC/Java is ran over this file. Because of the modular implementation of ESC/Java, this means it takes into account the specifications (annotations) of the other files in the projects, but not their implementations. We invoke ESC/Java with warnings about Casts, Exceptions, Modifies clauses disabled, because we currently have no meaningful response to those.

²<http://www.eclipse.org/jdt>

We give ESC/Java plenty of memory to work with, but limit the running time of the prover to 40 seconds per method, a value slightly below the default. This is useful because Simplify, the theorem prover on which ESC/Java is currently based, uses some heuristics to decide which way to prove a method correct. When the heuristics make an unfortunate choice, the theorem might take a considerable amount of time and memory to finish, if it succeeds at all. When the heuristics perform well, and they have been carefully tuned to perform well in most cases, the answer is usually produced quickly. We abort the Simplify process if it does not produce quick results, since it might not produce results at all. This threshold can be raised when the iterative annotation process has progressed to a more stable state.

5.5.6 Processing conflicts

After ESC/Java output is received, the `engine.Modifier` class is invoked. This class implements the `ConflictListener` interface, and starts the ESC/Java parser which is implemented as a `ConflictReporter` instance. The parser reports any conflicts found in the ESC/Java output back to the `Modifier` class.

A `Conflict` contains the `Location` of the offending code. In case of a `NonNull` warning (where some code violates a non-null specification) the declaration that is violated is also stored. In case of a `Null` warning (where a possibly-null reference is dereferenced) the ESC/Java suggestion is also stored. This suggestion contains the name and kind (for example local variable, parameter or field) of the element that should be annotated as non-null to remove this conflict. If the Java source file for this element is present, the filename and location within this file is reported. When this element is external (i.e. only present in a class or jar file) only the file and class name are reported.

5.5.7 From conflict to suggestion

CANAPA does not handle warnings for which ESC/Java gives no suggestion, such as `NonNull` warnings (where a specification is violated). It could be argued that giving suggestions should be implemented at the ESC/Java level rather than the INAPA level. However, we believe it is suitable for the annotation assistance tool to have more control over the handling of conflicts, so that the automatic inference techniques can be tweaked more easily.

We have implemented one such tweak: normally, CANAPA would have blindly added a non-null suggestion to each field that ESC/Java suggests to be annotated. This is likely not to be desirable for fields that are in fact initialized with a literal non-null value, so in that case we choose not to sustain that suggestion.

An obvious other improvement would be to add automatic annotation propagation for `NonNull` warnings. Some infrastructure for this task has been put into place by converting much of the code to the Eclipse JDT API, but unfortunately implementing the needed functionality to resolve a given identifier is too involved to finish in the context of this project.

5.5.7.1 Suggestions about external files

As we have described in section 4.2.1, we want to make minimal optimistic assumptions about the environment. In practice this means we will have to dynamically add specifications for ESC/Java to read. This can be done by generating so-called .spec files into a local specification repository.

To cut down on the false positives, ESC/Java comes with a set of .spec files for the standard library. Also, there might be an existing set of specifications already present. When looking for a specification, ESC/Java will search through the classpath looking for files with certain names, as described in more detail in section 5.1 of [16] and section 3.1 of [4]. As soon as it finds one such file, it stops.

This is unfortunate, since it makes it hard to add an assumption about a class for which there is already a specification available: it would be desirable to be able to keep the specifications that have been inferred by INAPA separate from the existing more trustworthy annotations, and have ESC/Java merge those automatically. This is not currently implemented.

As a workaround, all existing specifications are copied to a `localspecs` directory, and additional assumptions will be merged into them. In a separate log file a list of the additional assumptions is recorded so they may be verified by the developer.

An alternative workaround would be not to make any assumptions about files for which a .spec file already exists. We feel this is not desirable because it makes it impractical to use partial specification of external code. Being able to use partial specifications is important because it allows a project to incrementally collect specifications, instead of having to provide either a complete specification or no specification at all.

We recommend to use the former workaround for this issue.

5.5.8 Applying suggestions

Before applying a suggestion, the history of past suggestion is referenced to see if this suggestion has been applied before. If that would be the case, the suggestion is not applied again. This prevents loops in which the annotation assistant might keep adding an annotation and subsequently removing it³, and makes sure a previously inferred annotation which has been removed by the developer is not added again. Also, during an iteration over one file there might be several conflicts that lead to the same suggestion. Of course then also it is only applied once.

While applying a suggestion, it will report what part of the code is affected by this change. In our implementation, this will usually simply report that the whole file needs to be checked again. When an annotation is added to a local variable, however, we report that only the method in which the variable is visible is affected. This lays the foundation for reducing the amount of code that has to be checked again in more cases, but we have chosen not to implement that at this point. There is obviously room for improvement here.

³This currently does not occur since none of our rules for converting Conflicts to Suggestions result in the removal of an annotation. Nonetheless it can be imagined that such rules might be added in the future.


```

class Example {
    void run (String nullable) {
        /*INAPA*//*@non_null*/String external1 = Thread.currentThread().getName();
        String external2 = Thread.currentThread().toString();
        // force assumption that external1 is non-null
        external1.toString();
        String nonnullstring = new String ("Example");
        /*INAPA*//*@non_null*/String t1 = returnParam1(nonnullstring);
        String t2 = returnParam2(nonnullstring);
        // force assumption that t1 is non-null
        t1.toString();
    }
    /*INAPA*//*@non_null*/String returnParam1 (String param) {
        return param;
    }
    String returnParam2 (String param) {
        return param;
    }
    public static void main (String[] args) {
        Example e = new Example();
        // force parameter of e.run() to be nullable
        e.run(null);
    }
}

```

Figure 5.2: Annotation assistance example: INAPA result

5.5.9 Example

After constructing INAPA, we revisited the example class we used to illustrate the way JastAdd, Houdini and CANAPA work in chapter 2. The example consists of only one file, and because of that it does not demonstrate the most important advantage of INAPA: its modular, iterative nature. The improvement that INAPA is less strictly non-null-propagating because of the extra step when converting a conflict to a suggestion is also not illustrated here. Nonetheless we can illustrate some other properties:

First, the INAPA Pre-Processor is ran over the file (the original code for which is in Figure 3.4). In this case, however, that does not have any effect: the current implementation of the pre-processor only adds annotations to method fields, and this class has no fields. After running the analysis itself over the example, it produces the output shown in Figure 5.2 after the first run.

One obvious considerable improvement over CANAPA is that we no longer need to manually suppress warnings about external code: since INAPA makes minimal, optimistic assumptions about the environment it will catch this automatically. The assumptions about the environment that were made are in Figure 5.3: it can clearly be seen that INAPA makes no spurious assumptions about external code.

The rest of the output itself is equal to the output of the original CANAPA, already discussed in section 3.2.3.1.

```
package java.lang;
class Thread {
    /*INAPA**/*@non_null*/public static native java.lang.Thread currentThread();
    /*INAPA**/*@non_null*/public final java.lang.String getName();
}
```

Figure 5.3: INAPA’s assumptions about the environment

Chapter 6

Case studies and Benchmarks

This chapter reports the results of several benchmarking tests carried out on two case studies. Section 6.1 contains some general notes on the approach we took. Sections 6.2 and 6.3 contain the actual results of the tests on the DigiD and the Promedico HIS case studies, respectively.

As described in more detail in chapter 6.1, there are various practical problems with automatically testing properties of annotation assistants. Because of the limited time and resources available to us it is not feasible to perform full-blown, statistically meaningful analyses of the interesting properties of the cases. Nonetheless we deemed it appropriate to perform some practical tests to acquire at least some intuition of whether our improvements indeed seem to have the desired effect in practise. Hence, we must be very cautious with the conclusions we draw from the results. They should not be seen as empirical proof of our statements, but serve merely to gain some practical feedback on the proposed changes.

6.1 Benchmarking annotation assistants

There are two main properties based on which we can compare variations on the annotation assistance algorithm: the quality of the output and the performance in terms of resource usage (such as time and memory).

6.1.1 Resource usage

It turns out that the ESC/Java analysis which is part of our annotation assistance algorithm is quite a resource-intensive task. Because of this it is important that effort is also put into reducing the amount of work we leave to ESC/Java.

Measuring resource usage is not always simple, however: measuring the time an analysis takes and measuring it again after a change is made to the analysis technique, as we have done here, gives a reasonable indication. However, a changed analysis technique often also results in different results. In those cases, it is hard to determine the effect of the change on resource usage.

6.1.2 Quality

It is non-trivial to find a suitable measure for the quality of the output of an annotation assistant. It is easy to identify some desirable features that determine the quality of the assistance:

- The number of correct annotations should be high
- The number of incorrect annotations should be low
- The amount of user interaction required should be as low as possible (but now lower)

6.1.2.1 Determining whether an annotation is correct

While it is obvious that the tool should produce many correct and few incorrect annotations, for any given annotation it is hard to tell whether it is ‘correct’. Indeed, if we were able to automatically and reliably determine whether an annotation is correct, annotation assistance would be a much easier task.

6.1.3 Taking into account user interaction

As we have argued in section 4.2.6, user interaction is a key part of the program annotation process. However, it makes automating the benchmarking process almost impossible. Because of this, properly benchmarking the entire annotation assistance process would be an extremely time-consuming task. Because of this we have only performed benchmarks which look at a single iteration, which allows us to ignore user interaction.

6.2 DigiD Gateway

Since the 1st of January, 2005, the Dutch government has implemented a single sign on and central identification service for digitally available government services. This is an interesting project: it can be desirable to be able to digitally authenticate yourself to any municipality using the same credentials. However, it is not required that the municipality knows your credentials, only that it knows you have correctly identified yourself. Requiring civilians to authenticate via the municipality would be a security risk. Instead, the user is redirected to the central DigiD service, authenticates himself there, and gets redirected back to the municipality. Behind the scenes, the central DigiD server confirms to the municipality that you have identified yourself.

The municipality of The Hague developed an ‘exchange platform’ for connecting its websites to DigiD. This platform was released as open source software. We chose to take this code to run some of our benchmarks.

6.2.1 Introduction of modularity

Originally, CANAPA would iteratively run ESC/Java over the entire codebase under analysis at once. While this is undesirable from a user interface point of view, we decided to run some benchmarks to get a feel for the performance impact of modularizing this analysis.

CANAPA runs ESC/Java until a fixedpoint is reached where ESC/Java shows no new errors. We modified this behaviour, trying to reach a fixed-point for each module separately. Because processing a module might introduce warnings in previous modules (for instance when a parameter of a method called by a previous module is annotated as non-null), we finally run the CANAPA again over the whole codebase at once.

6.2.1.1 Resource usage

For the size of a ‘module’ we have tried both a per-file and a per-method modularisation. The running times of the analysis are shown in the following table:

Module size	Modularized analysis	Final run	Total time spent
Full run	n/a	45 minutes	45 minutes
Per file	25 minutes	13 minutes	38 minutes
Per method	27 minutes	21 minutes	48 minutes

A modularization per file, in this case, resulted in a reduction of the total running time of 15%. Modularization per method added 7% to the running time. It is interesting to notice that the modularized analysis per method is only slightly slower compared to per file (2 minutes), probably the result of some overhead when starting the analysis. On the other hand, the final run is much slower: this makes sense, because the methods within a file are likely to be more tightly coupled compared to methods in different classes. Because of this, treating every file as a module seems like a reasonable choice.

6.2.1.2 Quality

Upon closer inspection, it turned out that running the analysis in a modular way in fact reduced the number of spurious annotations. However, we do not conclude that the modularized approach is better in this sense: we expect this to be a coincidence, and probably mainly depends on the order in which methods have been processed. In general, we expect a modularized approach probably does not lead to changes in the quality of the analysis.

6.3 Promedico ASP

Promedico ASP is an advanced web-based information system for health care centres used by more than 2500 users every day. Among other tasks, it takes care of the storage and exchange of patient data among health care institutions, but also performs calendaring and accounting. It is interesting because it is a considerably large amount of real production code, which is bound to bring to the light issues which do not occur in smaller projects.

The code consists of roughly 1425 java files, 287 of which should not be taken into account because they have been generated from some other source rather than written by a developer directly.

6.3.1 Generated code

It is quite common for large Java projects to use a object/relational database persistence layer like Hibernate¹. To guide the transformation between the database entities and objects, a mapping is usually defined in an XML file, and Java classes to access the information in the database can be automatically generated from that mapping definition. Also tools like the Apache Axis WSDL2Java emitter² generate code (67 files in this case).

Because it should be possible to re-generate the Java when the XML file changes, annotations should not be added to those Java files. The annotation assistant should either learn the XML format and make changes there, or consider these Java files ‘external’. The latter is obviously the most pragmatic route to go for now.

6.3.2 Dependencies required

Since ESC/Java2 requires the dependencies of the code (at least in bytecode) to be on the classpath, INAPA also requires this. This turned can be quite an effort for a larger project like this. Luckily, this project used Maven³ which could be used to automatically download all dependencies.

6.3.3 toString() methods

In terms of performance, we found that the `toString()` methods of classes often take up extreme amounts of resources. This might be due to the ESC/Java implementation, which is known not to handle string operations (like concatenation) too well yet. This, combined with the notion that `toString()` methods often are mostly used for debugging purposes, leads to the idea that it might be good to postpone checking the `toString()` methods until after the assisted annotation process.

6.3.4 Pre-processing performance

In section 5.5.3 we described the primitive pre-processing step we implemented. We decided to put this functionality to the test by comparing the analysis process on preprocessed code and on unprocessed code.

We took a sample from the Promedico HIS codebase and recorded a couple of properties:

- the number of iterations
- the time the analysis took per source file
- the number of ESC/Java warnings that remained after INAPA was run.

¹<http://www.hibernate.org>

²<http://ws.apache.org/axis>

³<http://maven.apache.org>

6.3.4.1 Quality

Looking at the number of ESC/Java warnings that remained after INAPA was run, for each file the number of remaining warnings was the same with or without pre-processing. This gives a reasonable confidence in the correctness of the annotations. Apparently, in most cases INAPA was able to verify that the non-nullness of the automatically annotated fields indeed holds.

6.3.4.2 Performance

The main idea behind the pre-processing step was to improve performance by adding annotations up-front that are likely to have been inferred later anyway.

Looking at the number of iterations, at first sight we seem to have gained little: there are as many files that have taken an extra iteration than there are that now finish in fewer annotations. Measured in seconds, however, we do notice we cut down the total running time by 10%. Also, it might well be that an additional iteration in this run would have been necessary at a later stage anyway (i.e. when running INAPA over the file again after manual changes), in which case the pre-processing would have its positive effect on performance at that later stage.

Chapter 7

Conclusions

Static analysis of non-null properties of code is a useful technique that is starting to find its way into the mainstream: the addition of nullness analysis to the Eclipse IDE is a clear sign of this. This might be a stepping stone for more advanced static analysis tools, like the extended static checking of ESC/Java2, towards more general acceptance. This is good, but there is a lot of work to be done: when tools are developed in an academic setting, the resources to finish them as a product ready to be used in industry are often lacking. Even if the tool has reached a certain maturity, as is the case with ESC/Java2, maintaining it can be a problem: for example, ESC/Java2 has not yet been updated to work with Java 5 code and classes, and there are currently no plans to implement this.

As for annotation assistants, we feel we have made some important observations about the way such a tool should work, and we have implemented a proof-of-concept that contains some real improvements over the previously available work. We are highly confident that our tool indeed fits the requirements for an annotation assistant: it really does help the developer in adding nullness annotations to an existing, unannotated codebase faster and more easily — without getting in the way.

While an improvement over existing work is certainly good news, the real question remains: is it good enough? Does an annotation assistant make the work not just easier, but so easy that the cost of annotating software will be clearly outweighed by the benefits? The Promedico ASP case study shows that, for large codebases, adding annotations is still a daunting task. Improvements the checker and the additional propagation techniques that can be built into a tool like INAPA might eventually tip the balance, also for larger projects.

7.1 Further Research

7.1.1 Interoperability of nullness analysis tools

Currently, all tools that support nullness annotations use different ways to represent those annotations. Intuitively, these are different ways to convey much the same information — that is, nullness information.

Interoperability of nullness analysis tools, by way of adopting a common representation of nullness attributes, would probably aid the adoptance of nullness

checkers in general. If a common format could be defined, it would get much more attractive for libraries to produce and distribute formal nullness specifications for their library, as those would not be bound to a particular tool anymore. Eclipse is known to have delayed its support for nullness attributes because of the lack of standardisation in this area.

The addition of user-definable annotations into the Java 5 language is a big step forward: all Java nullness analysis tools are currently considering moving to this format. However, this only solves the difference in presentation, not the differences in semantics. These can be subtle: an obvious example is the difference between object invariants and nullity modifiers in ESC/Java.

We believe interoperability of nullness analysis tools would be beneficial to all, but there are some fundamental issues that still have to be resolved.

7.1.2 Heuristic type inference

We consider the heuristic variation on type inference we described in section 4.2.2.2 a promising approach.

7.1.3 User Interface improvements

This thesis has mainly focused on automatically adding annotations, in order to reduce the work required to start using a tool like ESC/Java on an existing project.

Besides reducing the amount of work that has to be done, we can also make the job itself easier. One of the problems with annotating existing code is that it is often hard to see which references influence the nullness properties of a given element.

This could be made much easier with some user interface extensions visualizing this aspect of the code. For instance, a list could be constructed showing all assignments to a given element. Those could for example be colour-coded with the nullness of their right hand sides, and allow for easy navigation to the locations of the assignments. Doing this recursively might yield a tree giving a map of the ways an element is used.

It would be interesting to see some research into code visualisation techniques for these and other properties.

7.1.4 Non-null by default

There is currently an ongoing debate on considering references to be non-null by default in JML. Throughout this thesis we have followed the traditional point of view that references are nullable by default, like they are in vanilla Java.

However, research has shown that in most cases a declaration is actually expected to be non-null[3], and following that reasoning current versions of JML consider references to be non-nullable unless they are annotated with a `/*@nullable*/` annotation, or the class or interface containing the annotation has been annotated as `nullable_by_default`.

This has been a controversial, recent change, and most tools currently still treat all classes as `nullable_by_default`. Which approach is suitable in which cases remains to be seen. For this reason we have considered the traditional nullable-by-default interpretation for this thesis.

Eiffel has also chosen non-null as the default, but unlike JML (which forbids mixing nullable and non_null annotations) it explicitly encourages using both nullable- and non-null-annotations (represented as ? and !) annotations and using a compiler flag to invert the default for migration purposes.

Appendix A

Example tool output

In this appendix we show the results of various tools on some simple Java source files, to give some intuition of the type of output they provide.

A.1 Clean Java files

A.1.1 External.java

```
class External
{
    public static Object staticMethod (External param) {
        return param;
    }

    public Object method (External param) {
        return param;
    }
}
```

10

A.1.2 Test1.java

```
class WordStat {
    WordStat (String s) {
    }

    void increaseCount () {
    }
}

class Dictionary {
    WordStat findStats (String word) {
        return null;
    }

    void addStat (WordStat stat) {
    }
}
```

10

```

    }

    class WordCounter {
        void addWord (Dictionary dict, String word)
        {
            WordStat s = dict.findStats (word);

            if (s == null)
            {
                WordStat new_stat = new WordStat (word);
                dict.addStat (new_stat);
            } else {
                s.increaseCount();
            }
        }
    }

    class Test1 {
        public static void main (String[] args) {
            Dictionary mydict = null;
            WordCounter C = new WordCounter ();
            C.addWord (mydict, "Henk");
        }
    }

```

A.1.3 Test2.java

```

public class Test2 extends External
{
    public void method (Object param)
    {
        System.out.println(param.toString());
    }
}

```

A.1.4 Test3.java

```

class Test3
{
    void Member ()
    {
        Object var = External.staticMethod (new External());
        System.out.println(var.toString());

        External var2 = new External();
        System.out.println(var2.method(new External()).toString());
    }
}

```

A.1.5 Test4.java

```

class Test4
{
    void member (Object param)
    {
        System.out.println (param.toString());
    }
}

```

A.2 JastAdd inferrer results**A.2.1 External.java**

```

class External
{
    public static Object staticMethod (External param) {
        return param;
    }

    public Object method (External param) {
        return param;
    }
}

```

10

A.2.2 Test1.java

```

class WordStat {
    WordStat(String s){
    }
    void increaseCount() {
    }
}

class Dictionary {
    WordStat findStats(String word) {
        return null;
    }
    void addStat(/*@non_null*/WordStat stat) {
    }
    Dictionary(){
    }
}

class WordCounter {
    void addWord(Dictionary dict, String word) {
        WordStat s = dict.findStats(word);
        if(s == null) {
            /*@non_null*/WordStat new_stat = new WordStat(word);
            dict.addStat(new_stat);
        }
    }
}

```

10

20

```

    }
    else {
        s.increaseCount();
    }
}
WordCounter(){
}
}
}
30

class Test1 {
    public static void main(/*@non_null*/java.lang.String[] args) {
        Dictionary mydict = null;
        /*@non_null*/WordCounter C = new WordCounter();
        C.addWord(mydict, "Henk");
    }
    Test1(){
    }
}
40

class External {
    /*@non_null*/ public static Object staticMethod(/*@non_null*/External param) {
        return param;
    }
    /*@non_null*/ public Object method(/*@non_null*/External param) {
        return param;
    }
    External(){
    }
}
50

```

A.2.3 Test2.java

```

class External {
    /*@non_null*/ public static Object staticMethod(/*@non_null*/External param) {
        return param;
    }
    /*@non_null*/ public Object method(/*@non_null*/External param) {
        return param;
    }
    External(){
    }
}
10

public class Test2 extends External {
    public void method(/*@non_null*/Object param) {
        System.out.println(param.toString());
    }
    public Test2(){

```



```
    }
}
```

20

A.2.4 Test3.java

```
class External {
    /*@non_null*/ public static Object staticMethod(/*@non_null*/External param) {
        return param;
    }
    /*@non_null*/ public Object method(/*@non_null*/External param) {
        return param;
    }
    External(){
    }
}
```

10

```
class Test3 {
    void Member() {
        /*@non_null*/Object var = External.staticMethod(new External());
        System.out.println(var.toString());
        /*@non_null*/External var2 = new External();
        System.out.println(var2.method(new External()).toString());
    }
    Test3(){
    }
}
```

20

A.2.5 Test4.java

```
class Test4 {
    void member(/*@non_null*/Object param) {
        System.out.println(param.toString());
    }
    Test4(){
    }
}
```

```
class External {
    /*@non_null*/ public static Object staticMethod(/*@non_null*/External param) { 10
        return param;
    }
    /*@non_null*/ public Object method(/*@non_null*/External param) {
        return param;
    }
    External(){
    }
}
```

```
}
```

20

A.3 Houdini results

A.3.1 External.java

```
class External
{
    public static Object staticMethod (External param) {
        return param;
    }

    public Object method (External param) {
        return param;
    }
}
```

10

A.3.2 Test1.java

```
class WordStat {
    /*@(houdini:constructor) requires false; */
    WordStat (/*@(houdini:parameter:constructor) non_null */ String s) {
    }

    /*@(houdini:instance method) */final/* */
    /*@(houdini:instance method) requires false; */
    void increaseCount () {
    }
}

class Dictionary {
    /*@(houdini:defaultconstructor) */public Dictionary(){}/* Explicating default constructor here
    /*@(houdini:instance method) */final/* */
    /*@(houdini:instance method) ensures \result != null; */
    /*@(houdini:instance method) requires false; */
    /*@(houdini:instance method) ensures \fresh(\result); */
    /*@(houdini:instance method) ensures word == null ==> \result != null; */
    /*@(houdini:instance method) ensures word != null ==> \result != null; */
    WordStat findStats (/*@(houdini:parameter:instance method) non_null */ String word) {
        return null;
    }

    /*@(houdini:instance method) */final/* */
    /*@(houdini:instance method) requires false; */
    void addStat (/*@(houdini:parameter:instance method) non_null */ WordStat stat) {
    }
}
```

10

20

```

class WordCounter {
    /*@(houdini:defaultconstructor) */public WordCounter(){}/* Explicating default constructor here */
    /*@(houdini:instance method) */final/* */
    /*@(houdini:instance method) requires false; */
    void addWord (/*@(houdini:parameter:instance method) non_null */ Dictionary dict, /*@(houdini:parameter:instance method) non_null */ String word) {
        WordStat s = dict.findStats (word);

        if (s == null)
        {
            WordStat new_stat = new WordStat (word);
            dict.addStat (new_stat);
        } else {
            s.increaseCount();
        }
    }
}

class Test1 {
    /*@(houdini:defaultconstructor) */public Test1(){}/* Explicating default constructor here */
    /*@(houdini:parameter:static method) requires \nonnullelements(args); */ 50
    public static void main (String[] args) {
        Dictionary mydict = null;
        WordCounter C = new WordCounter ();
        C.addWord (mydict, "Henk");
    }
}

```

A.3.3 Test2.java

```

public class Test2 extends External
{
    /*@(houdini:defaultconstructor) */public Test2(){}/* Explicating default constructor here */
    /*@(houdini:instance method) */final/* */
    /*@(houdini:instance method) requires false; */
    public void method (/*@(houdini:parameter:instance method) non_null */ Object param)
    {
        System.out.println(param.toString());
    }
}

```

10

A.3.4 Test3.java

```

class Test3
{
    /*@(houdini:defaultconstructor) */public Test3(){}/* Explicating default constructor here */
    /*@(houdini:instance method) */final/* */
    /*@(houdini:instance method) requires false; */
    void Member ()
    {
    }
}

```

```

{
  Object var = External.staticMethod (new External());
  System.out.println(var.toString());
}

External var2 = new External();
System.out.println(var2.method(new External()).toString());
}
}

```

10

A.3.5 Test4.java

```

class Test4
{
  /*@(houdini:defaultconstructor) */public Test4(){}/* Explicating default constructor here */
  /*@(houdini:instance method) */final/* */
  /*@(houdini:instance method) requires false; */
  void member (/*@(houdini:parameter:instance method) non_null */ Object param)
  {
    System.out.println (param.toString());
  }
}

```

10

A.4 CANAPA results

A.4.1 External.java

```

class External
{
  public static /*CANAPA*//*@ non_null */Object staticMethod (External param) {
    return param;
  }

  public /*CANAPA*//*@ non_null */Object method (External param) {
    return param;
  }
}

```

10

A.4.2 Test1.java

```

class WordStat {
  WordStat (String s) {
  }

  void increaseCount () {
  }
}

class Dictionary {
  WordStat findStats (String word) {
    return null;
  }
}

```

10

```

    }

    void addStat (WordStat stat) {
    }
}

class WordCounter {
    void addWord (/*CANAPA*//*@ non_null */Dictionary dict, String word)
    {
        WordStat s = dict.findStats (word);

        if (s == null)
        {
            WordStat new_stat = new WordStat (word);
            dict.addStat (new_stat);
        } else {
            s.increaseCount();
        }
    }
}

class Test1 {
    public static void main (String[] args) {
        Dictionary mydict = null;
        WordCounter C = new WordCounter ();
        C.addWord (mydict, "Henk");
    }
}

```

A.4.3 Test2.java

```

public class Test2 extends External
{
    public void method (/*CANAPA*//*@ non_null */Object param)
    {
        System.out.println(param.toString());
    }
}

```

A.4.4 Test3.java

```

class Test3
{
    void Member ()
    {
        /*CANAPA*//*@ non_null */Object var = External.staticMethod (new External());
        System.out.println(var.toString());

        External var2 = new External();
        System.out.println(var2.method(new External()).toString());
    }
}

```

```
    }  
}
```

10

A.4.5 Test4.java

```
class Test4  
{  
    void member (/*CANAPA*//*@ non_null */Object param)  
    {  
        System.out.println (param.toString());  
    }  
}
```

Bibliography

- [1] Daniel Bonniot. Type-checking multi-methods in ML (a modular approach). In *The Ninth International Workshop on Foundations of Object-Oriented Languages, FOOL 9*, Portland, Oregon, USA, January 2002.
- [2] Daniel Bonniot. Using kinds to type partially polymorphic multi-methods. In *Workshop on Types in Programming (TIP'02)*, Dagstuhl, Germany, July 2002.
- [3] Patrice Chalin and F. Rioux. Non-null references by default in the java modeling language. 2005.
- [4] David R. Cok. Esc/java2 implementation notes. 2004.
- [5] David R. Cok and Joseph R. Kiniry. Esc/java2: Uniting esc/java and jml.
- [6] Michael Y. Levin Cormac Flanagan, K. Rustan M. Leino. Efficient annotation inference for an extended static checker. 2001.
- [7] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.
- [8] Jaime Spacco David Hovemeyer and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. 2005.
- [9] Torbjörn Ekman and Görel Hedin. Modular implementation of non-null types for java. 2006.
- [10] Manuel Fähndrich and Rustan Leino. Declaring and checking non-null types in an object-oriented language. 2003.
- [11] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. 2001.
- [12] Christian Haack and Joe Wells. Type error slicing in implicitly typed higher-order languages. 2004.
- [13] Görel Hedin and Eva Magnusson. Jastadd: an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, April 2003.
- [14] Bastiaan Heeren and Jurriaan Hage. Type class directives. 2005.

- [15] David Hovemeyer Jaime Spacco and William Pugh. Understanding bugs in student programming projects.
- [16] Greg Nelson K. Rustan M. Leino and James B. Saxe. Esc/java user's manual. 2000.
- [17] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, and Joe Kiniry. Jml reference manual, 2005.
- [18] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. 2005.
- [19] K. Rustan M. Leino and Raymie Stata. Checking object invariants. 1997.
- [20] Bertrand Meyer. Attached types and their application to three open problems of object-oriented programming. *ECOOP*, pages 1–32, 2005.
- [21] Cormac Flanagan Michal Y. Levin and K. Rustan M. Leino. Annotation inference techniques. 2000.
- [22] Peter Müller. Reasoning about object structures using ownership. *Verified Software: Theories, Tools, Experiments*, 2006.
- [23] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating daikon and esc/java. July 23, 2001.
- [24] Terence J. Parr and Russell W. Quong. Antlr: A predicated- ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, July 1995.
- [25] Fausto Spoto. A generic static analyser for the java bytecode. 2005.