A Feasibility Study of Loop Bound Analysis for Loops with Disjunctive Guards

Nick Gubbels
Student Number: 0710288
N.Gubbels@student.ru.nl

Master Thesis Computer Science
Thesis Number: 666
November 2012

Supervisors:
Prof. dr. Marko van Eekelen
Rody Kersten MSc
Dr. Sjaak Smetsers (Reader)

Radboud University Nijmegen
Contents

1 Introduction
   1.1 AHA project ................................................. 2
   1.2 CHARTER project ............................................ 2
   1.3 ResAna ...................................................... 3
   1.4 Loop Bound Analysis ........................................ 3
   1.5 Goal of the thesis .......................................... 4
   1.6 Related work on Loop Bound Analysis ...................... 4
   1.7 Thesis outline .............................................. 5

2 Loop Bound Analysis with Polynomial Interpolation ........ 7
   2.1 Basic method ................................................ 7
      2.1.1 Extract and instrument the loop .................... 8
      2.1.2 Polynomial interpolation .............................. 9
      2.1.3 Test node generation .................................. 10
      2.1.4 Inferring a ranking function ....................... 10
      2.2 Extensions ............................................... 11
         2.2.1 Step size ............................................ 11
         2.2.2 Branch-splitting ................................... 12
   2.3 Working examples .......................................... 13
      2.3.1 Linear multivariate example ....................... 13
      2.3.2 Quadratic example .................................. 16
   2.4 Soundness .................................................. 18

3 Loop Bound Analysis for Loops with Disjunctive Guards .... 19
   3.1 Piece-wise ranking functions .............................. 19
      3.1.1 Condition Jumping .................................... 21
      3.2 The challenge ............................................ 21
         3.2.1 Example of condition jumping .................... 21
      3.2.2 Solution ............................................... 22
      3.2.3 Multi-jumping ................................-------- 23
   3.3 Implementation ............................................ 23
      3.3.1 Symbolic Execution and SMT solvers ............... 24
      3.3.2 Binary Decision Diagrams ............................ 26
      3.3.3 Comparison .......................................... 28
## Feasibility of Loop Bound Analysis for Loops with Disjunctive Guards in Practice

### 4.1 Global Evaluation

- 4.1.1 Apache harmony
- 4.1.2 Caliper
- 4.1.3 CDx
- 4.1.4 Eclipse IDE
- 4.1.5 GA-Playground
- 4.1.6 GWT-java-math
- 4.1.7 JAGA
- 4.1.8 Data Structures and Algorithm Analysis
- 4.1.9 Lightweight Java Game Library
- 4.1.10 Java-ML
- 4.1.11 JavaNCSS
- 4.1.12 Jembench
- 4.1.13 JGAP
- 4.1.14 Jmatbench
- 4.1.15 NeoBio
- 4.1.16 Weka
- 4.1.17 Totals

### 4.2 Detailed Feasibility Study

- 4.2.1 Detailed Loop Discussion

### 4.3 Categories of Loops with Disjunctive Guards

- 4.3.1 Conjunctive Normal Form Elimination
- 4.3.2 Condition Jumping check error
- 4.3.3 Iterating through data structures with size field or method
- 4.3.4 Iterating through other data structures
- 4.3.5 Breaks
- 4.3.6 Time
- 4.3.7 Threads
- 4.3.8 Unpredictable updates
- 4.3.9 Loops depending on non-numerical values

### 4.4 Summary of the proposed improvements/extensions

### 4.5 Feasibility

## Future work

- 5.1 Improved branch-splitting
- 5.1.1 Using update functions
- 5.2 Loop context

## Conclusion

## A Loops used in the detailed feasibility study

- A.1 While-loops
- A.2 For-loops
In software engineering a program needs to do what it is supposed to do. To ensure this, software should be tested thoroughly. However, for safety-critical, real-time and embedded systems, more aspects of software have to be considered in order to be sure the software is correct. A small bug in the program could lead to fatal situations, e.g. if the software of an airplane reach an error state.

Testing gives insight in what the output will be for given input, but only for the input which is used in the test cases. To get information about more cases, a model of the program can be created and tested for some properties. This is called Model-based testing or model checking and it gives information over larger sets of input/output. However, in the case of safety-critical software systems, it is even desirable to have proofs of certain properties. This is done by theorem proving. Even though theorem proving is most desirable, it takes more effort than model checking, which by itself takes more effort than testing. For this reason most software systems are only tested.

Additionally, testing only addresses the functional aspects of the software, i.e. does the software do what it is supposed to do given this input? However, in the case of safety-critical, real-time and embedded systems, the resources of the system are most likely to be limited; such systems are in general scarce of memory and processing power. It is important that the software of those systems does not exceed the boundaries of its resources, since that could lead to a system crash.

Therefore, it is important to test the system’s use of the resources, or moreover even prove that the system does not exceed its boundaries. This is called resource analysis. One such analysis is Loop Bound Analysis, which focuses on the number of iterations of a given loop and tries to infer an upper bound for that. This information gives insight in the processing time used by the program. Furthermore, it can be used as a base for a termination proof, since if a loop has an upper bound in the number of iterations, it will not iterate infinite many times.

In the following sections a few projects on resource analysis are described.
CHAPTER 1. INTRODUCTION

1.1 AHA project

Amortised Heap Analysis (AHA) project (as described in [32]) was a project which focused on heap consumption. The target of the project was to develop a method to infer non-linear bounds on the heap consumption of a (lazy) functional language.

In [30], [33], [29], [31] and [28] Shkaravska et al. describe a method for size analysis. This method introduces a type system with size annotations. For example, consider the following function type:

\[ L_n \times L_m \rightarrow L_{n+m} \]

This represents a function, which has two lists as input, which contain \( n \) and \( m \) elements respectively. The result is a list of size \( n + m \). This type is typical for an append function.

The type system with size annotations has been created for a small functional language presented in [30, 33, 29, 31, 28]. The size analysis method consists of two parts: type inference and type checking. Note that a type in this context contains the size in the number of elements. For type inference the authors describe a method using testing and polynomial interpolation. To infer the type of some functions, test nodes with known types are created and the functions are executed for those test nodes. Then the results of the functions are used to infer a polynomial, which specifies the type of the function.

Since the method above depends on the selection of test nodes, the method is not sound. Therefore, a type checking method is developed by the authors. They specify formal rules for type checking with size annotations of programs in the functional language presented in [30, 33, 29, 31, 28]. These rules construct a formal derivation tree, which is a proof of the types and sizes of the functions.

1.2 CHARTER project

The CHARTER (Critical and High Assurance Requirements Transformed through Engineering Rigour) project was an ARTEMIS Embedded Computing Systems Initiative project. The target of the project was to develop concepts, methods and tools to assist developers of critical systems to improve the development, verification and certification of software.

Results of the project vary from theoretical methods to fully usable tools for both development and verification of software. JamaicaVM, for example, is one of the development tools. It is a Java virtual machine implementation developed by Aicas especially designed for real-time software and can be used to run and compile Java software. Verification tools are for example ResAna and KeY, which are explained in more depth in the following sections.

The contribution of the Radboud University in the CHARTER project consisted of two parts: 1) develop a method for resource analysis and 2) to create a formal specification of Real-Time Java Libraries. The first will be explained in more detail in the following sections.

---

1 For more information: http://charterproject.ning.com/
2 For more information, see http://www.aicas.com/jamaica.html
1.3 ResAna

One of the results of the CHARTER project is the ResAna tool, as presented by Kersten et al. in [20]. The ResAna tool consists of various resource analysis methods, e.g. heap/stack size analysis and Loop Bound Analysis. For the heap and stack size analysis, it makes use of a tool called COSTA (COst and Termination Analyser) to extract the heap size and another tool Veriflux to extract stack size. Veriflux is an industrial tool by Aicas and is like ResAna part of the CHARTER Project. Veriflux is an automatic static code analysis tool.

The method behind COSTA is described in an article by Albert et al. [3]. The method analyses Java byte code to extract certain properties, e.g. it does cost analysis given various cost models. One of such models is the heap cost model, i.e. what is an upper bound for the size of the heap given a piece of code? Another cost model is the number of instructions, this is a model on execution length of a piece of code, i.e. what is an upper bound on the number of instructions executed. In combination with the cost analysis, the tool tries to prove termination as well. Indeed if an upper bound on the number of instructions exists and it is finite, termination can be guaranteed. An extension of this method is proposed by Montenegro et al. in [24], using the polynomial interpolation method of [30, 33, 29, 31, 28]. In [4], Albert et al. present a method to use COSTA in combination with verification tool KeY [1, 6].

1.4 Loop Bound Analysis

Loop Bound Analysis is another part of the ResAna tool and methods. This focuses on the number of iterations of loops. A basic method of Loop Bound Analysis using polynomial interpolation is presented by Shkaravska et al. in [27] and it is inspired by the method of the same authors for polynomial size analysis described in [30, 33, 29, 31, 28]. This method extracts a loop from the source code and instruments it with variables in order to do tests on the number of iterations. Given the results it infers a symbolic ranking function, which specifies an upper bound on the number of iterations of the loop. Since polynomial interpolation is used, the method in not limited to inferring linear bounds, unlike most other methods (presented in Section 1.6), which are limited to linear bounds. The loop bound analysis method by Shkaravska et al. [27] is implemented as a prototype in the ResAna tool.

The basic Loop Bound Analysis method with polynomial interpolation only infers upper bounds for loops with conditions over conjunctions of numerical (in)equalities. In order to handle disjunctions as well, the method is extended with piece-wise ranking functions, which in essence splits the disjunctions in sets of conjunctions (pieces) for every possible case. However, in [21] Kersten and Van Eekelen describe a challenge with this method: condition jumping. The method assumes the different pieces obtained by splitting to be independent. However, this is not true in some cases. Therefore, a solution to this problem is described in [21].
1.5 Goal of the thesis

This research focuses on Loop Bound Analysis on loops with disjunctive guards in practice. As described above, methods have been proposed, but how accurate and applicable are they in practice and what is needed to improve/extend those methods?

In this study the feasibility of Loop Bound Analysis for loops with disjunctive guards is examined. In other words how feasible is Loop Bound Analysis for loops with disjunctive guards? The base of this study is the Loop Bound Analysis method using polynomial interpolation by Shkaravska et al. [27] and the extension for disjunctive guards by Kersten and Van Eekelen [21].

The following steps have been performed in order to answer the question: how feasible is Loop Bound Analysis for loops with disjunctive guards?

- A global study on loops with disjunctive guards: how many loops contain disjunctive guards?
- A detailed study on loops with disjunctive guards from a large Java program: what would be the optimal ranking function for the loop and how can it be derived?
- Given the results of the detailed study, the loops have been sorted into categories and for each category a solution has been proposed in order to derive a ranking function for loops in that category, if possible. Otherwise an explanation of why this is impossible has been given.

1.6 Related work on Loop Bound Analysis

De Michiel et al. present a method for Loop Bound Analysis for C programs in [11]. It is very different in comparison with the method described here. Instead of a test-based approach, this method analyses the syntax and creates a context tree. Using that it performs execution flow analysis and abstract interpretation.

Ermedahl et al. describe another method for Loop Bound Analysis in [13]. It uses program slicing, abstract interpretation and invariant analysis. It has been implemented in the SWEET tool (SWEdish Execution time Tool), which analyses ANSI-C programs.


Fulara and Jakubczyk describe in [15] a method to derive ranking functions for for-loops. The method uses pattern recognition and derives the ranking functions using predefined patterns. Therefore, this method is less general than the method presented by Shkaravska et al. [27], which is considered for this research.

Hunt et al. describe a method for Worst Case Execution Time Analysis using Data Flow in [19]. It also addresses ranking function and their description in
Java Modeling Language (JML) and the verification of those ranking functions using KeY \cite{1, 6}.

Podelski and Rybalchenko describe in \cite{25} a method for deriving linear ranking functions for loops. This method is complete, i.e. if there exists a linear ranking function for a loop, the method will find it.

Lokuciejewski et al. present in \cite{23} a static analysis method for Loop Bound Analysis using abstract interpretation. It also uses program slicing for efficiency. However, the result of this method for a loop is a concrete/numerical bound instead of a symbolic bound.

1.7 Thesis outline

In Chapter \cite{2} basic Loop Bound Analysis using polynomial interpolation \cite{27} is described. This is followed by a description of loop bound analysis using polynomial interpolation for loops with disjunctive guards \cite{21} in Chapter \cite{3}. Then a feasibility study has been done in Chapter \cite{4}. This feasibility study is divided in three main parts: a) a global study in Section \cite{4.1}, b) a detailed study in Section \cite{4.2} and c) a study on what extensions are needed to derive ranking functions for loops with disjunctive guards in Section \cite{4.3}. Followed by future work in Chapter \cite{5} and finally in Chapter \cite{6} the conclusion.
Chapter 2

Loop Bound Analysis with Polynomial Interpolation

The purpose of Loop Bound Analysis is deriving a ranking function for a given loop. In this context a ranking function is a symbolic upper bound on the number of iterations of a loop. A method to derive a ranking function for a given loop is described by Shkaravska et al. in [27].

The basic method will be described first, followed by an introduction to some extensions and this chapter concludes with some working examples.

2.1 Basic method

Consider the loop in Listing 2.1.

Listing 2.1: Example class

```java
public class Example{
    public void method1 (int i){
        //do something
        while (i < 10)
            i++;  
        //do something else
    }
}
```

A ranking function for this loop is:

\[
\begin{align*}
10 - i & \quad \text{if } i > 0 \\
0 & \quad \text{otherwise}
\end{align*}
\]

This means that if, e.g., \( i = 0 \), that there will be at most 10 iterations according to the ranking function. This is the most precise upper bound for this loop. However, the following ranking function, which overestimates the number of iterations, is also valid, since it is an upper bound:
CHAPTER 2. L.B.A. WITH POLYNOMIAL INTERPOLATION

\[
\begin{cases}
100 - i & \text{if } i > 0 \\
0 & \text{otherwise}
\end{cases}
\]

In [27] a method is described to derive ranking functions for loops automatically. This method is based on testing, i.e. executing the loop given certain input values in order to infer a ranking function. The method consists of the following steps:

1. Extract and instrument the loop
2. Test node generation
3. Inferring a ranking function

In the following sections the method will be explained using the loop in Listing 2.1.

2.1.1 Extract and instrument the loop

Since the loop will be executed to test the number of iterations given certain input values, the loop is extracted from its context, i.e. remove all statements before and after the loop. By extracting the loop and placing it in a testing setting, there is more control over the variables on which the loop depends than there would be when executing the whole program. Since in this case the variables used in the loop can be manipulated directly. However, this is not possible when the whole program is executed, since the variables in the loop, can be dependent of other variables, i.e. the variables used in the loop cannot be manipulated directly, but through manipulating other variables, which makes it more difficult. Moreover it creates a weaker testing method, because the variables can only be manipulated indirectly, i.e. the values of the variables are restricted to which values are possible through the variables they depend on. For example, variable \( j \) is created using variable \( i \): \( j = 2i \). Then the values of \( j \) are restricted to even numbers.

After extracting it from its context, the loop is instrumented with a counter, which will be returned when the loop is finished, i.e. after finishing the loop it returns the number of iterations. Therefore, a new version of the class will be generated, this is shown in Listing 2.2.

**Listing 2.2: Instrumented loop**

```java
public class Example{
    public void method1 (int i){
        int count = 0;
        while(i < 10){
            i++;
            count++;
        }
        return count;
    }
}
```

In [27] a method is described to derive ranking functions for loops automatically. This method is based on testing, i.e. executing the loop given certain input values in order to infer a ranking function. The method consists of the following steps:

1. Extract and instrument the loop
2. Test node generation
3. Inferring a ranking function

In the following sections the method will be explained using the loop in Listing 2.1.

2.1.1 Extract and instrument the loop

Since the loop will be executed to test the number of iterations given certain input values, the loop is extracted from its context, i.e. remove all statements before and after the loop. By extracting the loop and placing it in a testing setting, there is more control over the variables on which the loop depends than there would be when executing the whole program. Since in this case the variables used in the loop can be manipulated directly. However, this is not possible when the whole program is executed, since the variables in the loop, can be dependent of other variables, i.e. the variables used in the loop cannot be manipulated directly, but through manipulating other variables, which makes it more difficult. Moreover it creates a weaker testing method, because the variables can only be manipulated indirectly, i.e. the values of the variables are restricted to which values are possible through the variables they depend on. For example, variable \( j \) is created using variable \( i \): \( j = 2i \). Then the values of \( j \) are restricted to even numbers.

After extracting it from its context, the loop is instrumented with a counter, which will be returned when the loop is finished, i.e. after finishing the loop it returns the number of iterations. Therefore, a new version of the class will be generated, this is shown in Listing 2.2.

**Listing 2.2: Instrumented loop**

```java
public class Example{
    public void method1 (int i){
        int count = 0;
        while(i < 10){
            i++;
            count++;
        }
        return count;
    }
}
```
2.1.2 Polynomial interpolation

Before explaining the next step, test node generation, some explanation is needed about polynomial interpolation. Polynomial interpolation is the interpolation of given data points by a polynomial, i.e. derive a polynomial that goes through those points. Polynomial interpolation itself will be used in the last step of the method: deriving the polynomial, but the conditions presented below are used in the next step.

Deriving a polynomial is finding values for the coefficients of the polynomial. For example, for a polynomial with one variable and degree \( d \)

\[ p(z) = a_0 + a_1 z + \cdots + a_d z^d \]

it is finding values for \( a_1, \ldots, a_d \), in such a way that polynomial \( p \) interpolates the test results.

However, there are certain conditions on the test data, in order for the polynomial to exist and to be unique. In the case of a polynomial with one variable and of degree \( d \) (as in the polynomial shown above), at least \( d + 1 \) different data points are needed. The system of \( d + 1 \) pairwise different points can be written as follows in a matrix form:

\[
\begin{pmatrix}
1 & z_0 & \cdots & z_0^{d-1} & z_0^d \\
1 & z_1 & \cdots & z_1^{d-1} & z_1^d \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
1 & z_{d-1} & \cdots & z_{d-1}^{d-1} & z_{d-1}^d \\
1 & z_d & \cdots & z_d^{d-1} & z_d^d
\end{pmatrix}
\begin{pmatrix}
a_0 \\
a_1 \\
\vdots \\
a_{d-1} \\
a_d
\end{pmatrix}
= 
\begin{pmatrix}
p(z_0) \\
p(z_1) \\
\vdots \\
p(z_{d-1}) \\
p(z_d)
\end{pmatrix}
\]

A matrix in the form of \( A_{i,j} = z_j^{i-1} \), like the matrix above, is called a Vandermonde matrix. For pairwise different points \( z_0, \ldots, z_d \), the matrix has a non-zero determinant, which is called a Vandermonde determinant. Because the determinant has a non-zero value, there exists a unique solution to the system, i.e. a unique polynomial, which interpolates the data points.

However, for multivariate systems the test data need to satisfy more complex conditions in order to get a unique interpolating polynomial. First of all a polynomial \( p(z_1, \ldots, z_k) \) of degree \( d \) and with \( k \) variables, i.e. dimension \( k \), has \( N_k^d = \binom{d+k}{k} \) coefficients. In \[8\] three node configurations are introduced, under which the Vandermonde determinant is non-zero. The simplest one is Node Configuration A. For the 2-dimensional case NCA is defined as follows:

Set \( W \subset \mathbb{R}^2 \) exists of \( N_2^d \) nodes. \( W \) lies in a 2-dimensional NCA if there exists lines \( \gamma_1, \ldots, \gamma_{d+1} \) in \( \mathbb{R}^2 \), in such a way that \( d+1 \) nodes of \( W \) lie on \( \gamma_{d+1} \), \( d \) nodes on \( \gamma_d \setminus \gamma_{d+1} \), \ldots, and 1 node lies on line \( \gamma_1 \setminus (\gamma_2 \cup \cdots \cup \gamma_{d+1}) \).

NCA for any \( k > 2 \) is defined by induction on \( k \):

Set \( W \subset \mathbb{R}^k \) exists of \( N_k^d \) nodes. \( W \) lies in a \( k \)-dimensional NCA if, for any \( 0 \leq i \leq d \), there is a \( (k-1) \)-dimensional hyperplane, in such a way that it...
contains \( N_{d-i}^{k-1} \) nodes, which are on a \((k - 1)\)-dimensional NCA for degree \( d - i \) and these nodes are not lying on any other hyperplane.

2.1.3 Test node generation

The next step in the process of inferring a ranking function, is to generate test nodes. In the previous step, the loop was instrumented with a counter. Therefore it can be executed with different input values and it will return the number of iterations of the loop for each test node, which is a set of input values used for one test execution.

As said in the previous section, in order to have a unique solution for the system of equations using polynomial interpolation, the test data needs to be in NCA. However, for test nodes used to derive a ranking function, there is one extra condition. The test nodes need to satisfy the loop guard as well. Otherwise it would have no sense to execute the test, as it would result in no iterations.

The example (in Listing 2.1) has a loop (guard) with just one variable. Therefore, the only requirement is, that there need to be \( d + 1 \) different nodes, where \( d \) is the expected degree of the polynomial. Given the loop condition \( i < 10 \), it is reasonable to expect a degree of 1, i.e. linear. Therefore, two different nodes are needed:

<table>
<thead>
<tr>
<th>( i )</th>
<th>number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

2.1.4 Inferring a ranking function

After the test nodes are generated and the tests are executed, a polynomial can be inferred, i.e. the system of equations need to be solved. The example has one variable and is of degree 1, so the polynomial has the following form:

\[
a_0 + a_1 i = p(i)
\]

Therefore, the system is as follows:

\[
a_0 = 10
\]

\[
a_0 + a_1 = 9
\]

This gives that \( a_0 = 10 \) and \( a_1 = -1 \). Therefore, the polynomial interpolating the test nodes is:

\[
p(i) = 10 - i
\]

This is the main part of the inferred ranking function. However, there is still another case for which this ranking function is not correct. This is the case
where the loop is never entered, i.e. when the loop condition is not satisfied. Therefore, the inferred ranking function is:

\[
\begin{cases} 
10 - i & \text{if } i > 0 \\
0 & \text{otherwise}
\end{cases}
\]

The method is also implemented in a tool: ResAna, which is part of the CHARTER Project\(^1\).

### 2.2 Extensions

In this section some extensions are presented as given in [27].

#### 2.2.1 Step size

In the example in listing 2.1 the loop variable is incremented by 1. However, there are cases where the loop variable is incremented (or decremented) with another value. The loop in Listing 2.3 is such a case.

<table>
<thead>
<tr>
<th>Listing 2.3: A loop with step size 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 while (i &lt; 20)</td>
</tr>
<tr>
<td>2 i += 4;</td>
</tr>
</tbody>
</table>

Since there is only one variable in the loop and the expected degree is 1, there need to be \(N_1^1 = 2\) test nodes to interpolate the following polynomial in the data:

\[ ai + b = p(i) \]

The following test nodes could be used:

\[ i = 0 \]
\[ i = 1 \]

This would give the following results:

<table>
<thead>
<tr>
<th>(i)</th>
<th>(p(i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

This results in the following polynomial:

\[ p(i) = 5 \]

\(^1\) For more information see [http://charterproject.ning.com/](http://charterproject.ning.com/)
However, this is not a correct ranking function, since the number of iterations of the loop is not constant. It is a correct upper bound for any $i \geq 0$, but it is incorrect for any $i < 0$, e.g., $i = -1$ results in 6 iterations.

The problem is that the test nodes that are chosen for this example are too close to each other. The loop variable is incremented by 4, i.e., the step size of the loop is 4. Therefore, the differences of the values of $i$ in the test nodes need to be a factor of 4. For example, the following test nodes:

\[
i = 0 \\
i = 4
\]

This would give the following results:

<table>
<thead>
<tr>
<th>$i$</th>
<th>$p(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

This results in the following system of equations:

\[
d = 5 \\
4a + d = 4
\]

This gives that $a = -\frac{1}{4}$ and $d = 5$. However, the number of iterations needs to be an integer, so the polynomial needs to be ceiled:

\[
\left\lceil 5 - \frac{1}{4}i \right\rceil\quad \text{if } i < 20 \\
0\quad \text{otherwise}
\]

### 2.2.2 Branch-splitting

Another extension mentioned in [27] handles if-statements inside the loop. Consider the loop in [27].

Listing 2.4: Loop with if-statements as given in [27]

```c
while (i > 0) {
    if (i > 100) 
        i = 10;
    else 
        i --;
}
```

The loop contains two branches, which affect the upper bound differently: when $i > 100$ the value of $i$ drops faster every iteration.

Branch-splitting is a solution to this problem for loops, which have the following worst-case computation branch property:
“For each loop body, there is an execution path such that, for any collections of values of the loop variables, if one follows this execution path in every loop iteration one reaches the worst-case, i.e. the upper bound.” [27]

To do this, multiple new loops are created form the original, one for each branch, as shown in listings 2.5 and 2.6. For all those loops a ranking function is computed and then because of the property described above it is possible to use the maximum of both as a general ranking function for the original loop. Therefore, a ranking function for the loop in listing 2.4 is $i$.

Listing 2.5: A branch from the loop in listing 2.4

```java
while (i > 0){
    i --;
}
```

Listing 2.6: Another branch from the loop in listing 2.4

```java
while (i > 0){
    i = 10;
}
```

2.3 Working examples

2.3.1 Linear multivariate example

In this section a multivariate version of the example in the previous section will be given. The loop used in this example is shown in Listing 2.7.

Listing 2.7: Example loop with 3 variables

```java
public class Example2{
    public void method2 ( int i){
        //do something
        while(x < 10 && y < 10 && z < 10){
            z++;
            if(z == 10){
                y++;
                z = 0;
                if(y == 10){
                    x++;
                    y = 0;
                }
            }
        }
        //do something
    }
}
```
Extract and instrument loop

First the loop is extracted from its context and is instrumented with a counter, as shown in Listing 2.8.

```java
public class Example2 {
    public void method2 (int i) {
        int count = 0;
        while (x < 10 && y < 10 && z < 10) {
            z++;
            if (z == 10) {
                y++;
                z = 0;
                if (y == 10) {
                    x++;
                    y = 0;
                }
            }
            count++;
        }
        return count;
    }
}
```

Test node generation

The example has three variables and it is expected to be linear, i.e. the expected degree is 1. Therefore, there have to be \( N_{3}^{1} = \binom{4}{3} = 4 \) test nodes in NCA, in order to be sure there is a (unique) ranking function.

For a multivariate case with \( k > 2 \) the definition for nodes to be in NCA is given inductively. In this case \( k = 3 \), so for any \( 0 \leq i \leq d \), there have to be a \((k-1)\)-dimensional hyperplane, with \( N_{d-1}^{k-1} \) nodes, which are on a \((k-1)\)-dimensional NCA for degree \( d - i \) and are not on any other hyperplane. In this case there have to be two 2-dimensional hyperplanes in NCA: one of degree 1 (Hyperplane A), with 3 nodes, and one of degree 0 (Hyperplane B), with 1 node.

In order to have a 2-dimensional hyperplane in NCA, it exists of \( N_{d}^{2} \) nodes and there exist lines \( \gamma_{1}, \ldots, \gamma_{d+1} \), in such a way that \( d + 1 \) nodes are on line \( \gamma_{d+1} \), \ldots, and 1 nodes lies on line \( \gamma_{1} \setminus (\gamma_{2} \cup \cdots \cup \gamma_{d+1}) \). For Hyperplane A, there are two lines \( \gamma_{1} \) and \( \gamma_{2} \). Two nodes are on line \( \gamma_{2} \):

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

and one node is on line \( \gamma_{1} \):
For Hyperplane B, there is one line $\gamma_1'$, with one node:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Inferring a ranking function**

The loop has three variables and the ranking function has an expected degree of 1. Therefore, the polynomial of the ranking function has the following form:

$$ax + by + cz + d = p(x, y, z)$$

Given the test nodes, the following results are found:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>$p(x, y, z)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>9000</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>990</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>999</td>
</tr>
</tbody>
</table>

This gives the following system of equations:

$$

d = 1000 \\
\ a + d = 900 \\
\ b + d = 990 \\
\ c + d = 999 
$$

This results in:

$$
\ a = -100 \\
\ b = -10 \\
\ c = -1 \\
\ d = 1000 
$$

Therefore a ranking function for the loop in Listing 2.7 is:

$$
\begin{cases} 
1000 - 100x - 10y - z & \text{if } x < 10 \land y < 10 \land z < 10 \\
0 & \text{otherwise}
\end{cases}
$$
CHAPTER 2. L.B.A. WITH POLYNOMIAL INTERPOLATION

2.3.2 Quadratic example

The examples above were all of degree 1, i.e. linear. The following example will show that the method can also handle higher degrees, in this case degree 2, i.e. quadratic.

Listing 2.9: Loop with a quadratic ranking function

```c
while (x < n && y < n) {
    y++;
    if (y == n) {
        y = 0;
        x++;
        }
    }
```

As said before, the ranking function for the loop will be quadratic. However, to show what happens when the degree is too low, first a ranking function with degree 1 will be inferred.

Degree 1

The loop has three variables. Therefore, there need to be $N_1^3 = 4$ nodes. The polynomial will have the following form:

$$ax + by + cn + d = p(x, y, n)$$

The following test nodes, which are in NCA, could be used:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>n</th>
<th>p(x, y, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This gives the following system of equations:

\[
\begin{align*}
2c + d &= 4 \\
a + 2c + d &= 2 \\
b + 2c + d &= 3 \\
c + d &= 1
\end{align*}
\]

This results in the following polynomial:

$$3n - 2x - y - 2 = p(x, y, n)$$

However, it can easily be shown that this ranking function is incorrect. For example when $x = 0$, $y = 0$ and $n = 3$, the loop has 9 iterations, but according to the polynomial only 7. This underestimates the number of iterations. Therefore it is an incorrect upper bound.
2.3. WORKING EXAMPLES

Degree 2

For three variables and a degree of 2, $N_3^2 = 10$ test nodes are needed. The in the following 10 test nodes, which are in NCA, are shown with the number of iteration as a result of those values.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

The polynomial will have the following form:

$$ax^2 + by^2 + cn^2 + dxy + exn + fyn + gx + hy + in + j = p(x, y, n)$$

This gives the following system of equations:

\[
\begin{align*}
9c + 3i + j &= 9 \\
9c + 3e + g + 3i + j &= 6 \\
4a + 9c + 6e + 2g + 3i + j &= 3 \\
b + 9c + 3f + h + 3i + j &= 8 \\
a + b + 9c + d + 3e + 3f + g + h + 3i + j &= 5 \\
4b + 9c + 6f + 2h + 3i + j &= 7 \\
4c + 2i + j &= 4 \\
a + 4c + 2e + g + 2i + j &= 2 \\
b + 4c + 2f + h + 2i + j &= 3 \\
c + i + j &= 1
\end{align*}
\]

This results in the following polynomial:

$$n^2 - xn - y = p(x, y, n)$$

Therefore, a ranking function for the loop in Listing 2.9 is:

\[
\begin{cases}
  n^2 - xn - y & \text{if } x < n \land y < n \\
  0 & \text{otherwise}
\end{cases}
\]

As shown above, the inferred ranking function will be incorrect, when the expected degree is too low. However, a degree higher than needed, will lead to a correct ranking function, though it leads to more calculations, since the number of test nodes that are needed grows.
2.4 Soundness

The ranking functions found by the method described in this chapter depend on which test nodes are used. Therefore, the method is not sound. However, the inferred ranking functions can be added to the Java code by using the JML clause: `@decreases`, as shown in Listing 2.10.

<table>
<thead>
<tr>
<th>Listing 2.10: Example class with decreases clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 // ensures true;</td>
</tr>
<tr>
<td>2 public class Example{</td>
</tr>
<tr>
<td>3   // assignable i;</td>
</tr>
<tr>
<td>4   // loop_invariant true;</td>
</tr>
<tr>
<td>5   // decreases i&lt;10 ? i-10 : 0;</td>
</tr>
<tr>
<td>6   public void method1 (int i){</td>
</tr>
<tr>
<td>7     // do something</td>
</tr>
<tr>
<td>8     while (i &lt; 10)</td>
</tr>
<tr>
<td>9       i++;</td>
</tr>
<tr>
<td>10     // do something else</td>
</tr>
<tr>
<td>11 }</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

By adding this to the code, it is possible to prove the ranking function, i.e. prove that the loop will never have more iterations than given in the ranking function. There is a tool available that can do this (partly) automatically. The tool is called KeY and is introduced in the article by Ahrendt et al. [1] and is more detailed described by Beckert et al. in [6]. The KeY tool can be used to prove that certain JML clauses are correct given the Java code.

Note that this is the same method as used by Shkaravska et al. [30, 33, 29, 31, 28]. Using polynomial interpolation for inference and use a formal method to verify the polynomial returned by the inference.

Another tool which uses JML annotations is Extended Static Checker for Java (ESC/Java), as described in [14] and with its JML extension in [9]: ESC/Java2. The main purpose of ESC/Java was to find common programming errors. However, ESC/Java2 can prove as well, since it is enhanced with JML and theorem provers.
Chapter 3

Loop Bound Analysis for Loops with Disjunctive Guards

The basic loop bound analysis method as described in section 2.1 handles loops with conditions of the following form:

\[ \bigwedge_{i=1}^{n} (e_{li} \ b \ e_{ri}) \]

where \( b \in \{<, >, =, \neq, \leq, \geq\} \).

However, in [27] an extension to the basic loop bound analysis method is given, in order for the method to handle loops “with as exit condition any propositional logical expression over arithmetical (in)equalities” [21], including disjunctions. This is done by the use of piecewise ranking functions. These are ranking functions consisting of different pieces. The ranking function given at the end of section 2.1.4 is a simple example of a piecewise ranking function:

\[
\begin{cases}
10 - i & \text{if } i > 0 \\
0 & \text{otherwise}
\end{cases}
\]

It consist of two pieces: 1) the loop condition is true and the computed ranking function for the loop is given and 2) when the loop condition is false and the is never entered, i.e. no iterations.

3.1 Piece-wise ranking functions

To allow disjunctions in the loop condition, Shkaravska et al. created the following extension to the basic loop bound analysis method [27] [21].

The first step is to transform the loop condition to disjunctive normal form (DNF):
\[
\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} (e_{ij} \mathbf{b}_{rij})
\]

where \( \mathbf{b} \in \{<, >, =, \neq, \leq, \geq\} \).

Then the transformed condition is split into several conditions with the function \( \text{DNFsplit} : C_d \rightarrow \{C_{nd}\} \), where \( C_d \) is a loop condition in DNF and \( C_{nd} \) is a loop in the form as given at the beginning of this chapter (a conjunction of arithmetical (in)equality). The function \( \text{DNFsplit} \) is defined in [21] as follows:

\[
\text{DNFsplit}(b_1 \land \cdots \land b_n) := \begin{cases} 
\bigwedge_{b_i \in BP} b_i \land \bigwedge_{b_j \in B_{\text{rest}}} \neg b_j \big| B P \in \mathcal{P}([b_1, \ldots, b_n]) \setminus \emptyset \land B_{\text{rest}} = [b_1, \ldots, b_n] \setminus BP
\end{cases}
\]

This function transforms the condition in DNF to a set of \( 2^n - 1 \) pieces (conjunctive conditions, as defined at the beginning of this chapter). The pieces are of a type that can be handled by the basic method (described in Chapter 2). The ranking function is then calculated as follows: for every piece in the set a ranking function is calculated, which are then combined to one ranking function:

\[
\begin{cases} 
RF_{p_1} & \text{if } p_1 \\
\ldots & \text{if } \ldots \\
RF_{p_m} & \text{if } p_m \\
0 & \text{else}
\end{cases}
\]

Consider the example in Listing 3.1. The loop condition in DNF is:

\[
(start < end \land end < 40) \lor (start < end \land end > 100)
\]

Using \( \text{DNFsplit} \) will result in the following pieces:

\[
(start < end \land end < 40) \land \neg (start < end \land end > 100) \\
\neg (start < end \land end < 40) \land (start < end \land end > 100) \\
(start < end \land end < 40) \land (start < end \land end > 100)
\]

However, the last piece can be omitted, since there is no value for \( end < 40 \land end > 100 \). Using the basic method for the other two pieces, gives for both the following ranking function: \( end - start \).

Therefore a ranking function for the loop in listing 3.1 is:

\[
\begin{cases} 
end - start & \text{if } (start < end \land end < 40) \land \neg (start < end \land end > 100) \\
end - start & \text{if } (start < end \land end > 100) \land \neg (start < end \land end < 40) \\
0 & \text{otherwise}
\end{cases}
\]
### 3.2. THE CHALLENGE

In the previous section, piecewise ranking functions were introduced as a solution to the challenge that arises when using disjunctions in the loop condition (Section 3.1). However, another challenge arises when using this method. This is mentioned by Kersten and Van Eekelen in [21]. When splitting the disjunctive condition into different pieces, each piece is considered independent, i.e. for every execution of the loop just one piece is true. However, it is possible that during the execution of the loop another piece becomes true. This is called **condition jumping**. In the first section of this chapter the challenge will be explained and in the last section two solutions will be presented.

### 3.2 The challenge

As said above, condition jumping is a challenge that arises when disjunctive loop conditions are used, e.g. \( a \lor b \), which results in the following pieces:

1. \( a \land \neg b \)
2. \( \neg a \land b \)
3. \( a \land b \)

The piece \( \neg a \land \neg b \) is omitted, since it represents the case where the loop is not entered and thus has a ranking function of 0.

Condition jumping arises when at some point in execution one piece is true and after the next iteration another piece is true, e.g. at some point \( a \land \neg b \) holds, but after the next iteration of the loop \( \neg a \land b \) holds. The challenge is that the pieces created for piecewise ranking functions, can be underestimating the actual number of iterations, since the pieces are assumed to be independent. However, when condition jumping arises, this assumption is wrong.

Note that in every (finite) loop some sort of condition jumping arises, the jumping to the piece, for which the loop will be exited \( (\neg a \land \neg b) \). However, this will not be considered condition jumping, since this jumping will result in exiting the loop and will therefore add no difference to the number of iterations.

#### 3.2.1 Example of condition jumping

The loop in Listing 3.2 (as given in [21]) shows an example of condition jumping.

```java
while ((i > 0 && i < 20) || i > 22){
    if (i > 22)
```

---

**Listing 3.1:** Example of a loop condition with a disjunction (as given by Shkaravska et al. in [27])

```java
while (start < end && (end < 40 || end > 100))
```

**Listing 3.2:** Condition jumping example

```java
while ((i > 0 && i < 20) || i > 22){
    if (i > 22)
```
Using piecewise ranking functions, the following pieces will be created:

1. \((i > 0 \land i < 20) \land \neg (i > 22)\)
2. \(\neg (i > 0 \land i < 20) \land (i > 22)\)
3. \((i > 0 \land i < 20) \land (i > 22)\)

As above, the piece for which the loop is not entered \((\neg (i > 0 \land i < 20) \land \neg (i > 22))\) is omitted. Moreover, the third piece can be omitted as well, since there exists no \(i\) such that \((i > 0 \land i < 20) \land (i > 22)\).

3.2.2 Solution

An algorithm to compute all the states which lead to condition jumping from piece \(p_i\) to piece \(p_j\) is described below [21].

In the following a state \(s\) represents a set of values for the variables used in the loop. \(s \rightarrow s'\) represents one iteration of the loop with \(s\) as the state before the iteration and \(s'\) the state after the iteration. \(s \Rightarrow s'\) represents that it is possible to go from state \(s\) to state \(s'\) in one or more iterations.

1. Check whether there is a state \(s\) for which holds: \(s \rightarrow s'\), where piece \(p_i\) is true in state \(s\) and piece \(p_j\) is true in state \(s'\). If there is such a state, add it to the list \(CASES\).
2. Repeat the previous step to find other states \(s \notin CASES\), until there are no such states any more.
3. Pop a state from \(CASES\): \(s\)
4. Check whether there are states which lead to that state and are not already condition jumping cases: \(s' \rightarrow s\) and \(s' \notin CASES\).
5. If so, add the state to \(CASES\). Repeat until no new state is found, then add \(s\) to \(DONE\) and go back to step 3 until \(CASES\) is empty. Finally all states that lead eventually to condition jumping are in \(DONE\).

Steps 1 and 2 search for states which directly lead to condition jumping, i.e. in one step. Steps 3, 4 and 5 search for cases which lead eventually to condition jumping, i.e. \(s \Rightarrow s'\), where in \(s\) piece \(p_i\) is true and in \(s'\) piece \(p_j\) is true.

Executing this algorithm on the example above gives the following: in step 1, the state where \(i = 19\) results in condition jumping, since in that state the first piece is true, but after one iteration the second piece is true, since then \(i = 23\). This is the only direct case of condition jumping. Therefore, the set \(CASES\) is as follows, when reaching step 3: \(i = 19\). The next step is to check if there is a state that reaches \(i = 19\) after one iteration. This is only the case in the state \(i = 15\). The following step is to check if there are states reaching \(i = 15\) etc.

Finally the following cases of condition jumping are found:
3.3. IMPLEMENTATION

- \( i = 19 \)
- \( i = 15 \)
- \( i = 11 \)
- \( i = 7 \)
- \( i = 3 \)

Note that these are all condition jumping cases from the first piece to the second. The cases for condition jumping from the second piece to the first have to be calculated as well. However, for this example, there are no such cases.

These states can be put into one new piece, since there is one case of condition jumping and the other cases lead to that case. However, it is possible that there are more cases of condition jumping, which will result in multiple pieces. The pieces for the example will then be as follows:

- \( i > 0 \land i < 20 \land i \mod 4 \neq 3 \)
- \( i > 0 \land i < 20 \land i \mod 4 = 3 \)
- \( i > 22 \)

Using the basic method for each of these pieces, results in the following ranking function:

\[
\begin{cases}
\left\lceil \frac{(20 - i)}{4} \right\rceil + 1 & \text{if } (i > 0) \land (i < 20) \land i \mod 4 = 3 \\
\left\lceil \frac{(20 - i)}{4} \right\rceil & \text{if } (i > 0) \land (i < 20) \land i \mod 4 \neq 3 \\
i - 22 & \text{if } i > 22 \\
0 & \text{else}
\end{cases}
\]

3.2.3 Multi-jumping

The algorithm above defines how to detect condition jumping from one piece to another. However, it is necessary to check every pair of conditions for condition jumping:

1. DNF-split (see Section 3.1) the loop condition to \( n \) pieces.

2. Apply the algorithm above for any pairwise different \( p_i \) and \( p_j \) pieces in both ways, i.e. condition jumping from \( p_i \) to \( p_j \) and condition jumping from \( p_j \) to \( p_i \).

3.3 Implementation

In the previous section an algorithm to detect condition jumping is given. In this section two different solutions to implement this algorithm are given and compared.
### 3.3.1 Symbolic Execution and SMT solvers

In [21] Kersten et al. describe a method for detecting condition jumping in a loop using symbolic execution and a Satisfiability Modulo Theories (SMT) solver.

**Symbolic Execution**

Symbolic execution will be used to obtain update functions, which define how a variable is updated after one iteration of a given loop.

Symbolic execution is described by King in [22]. It is executing a piece of code in a symbolical way, i.e. instead of executing the program with variables, execute it with symbols. Then the result is not a set of values for input symbols/variables, but a set of symbolic formulas over input symbols.

The update functions obtained with symbolic execution are defined as follows:

\[
\text{next}_{v_i} :: T_{v_1} \rightarrow \cdots \rightarrow T_{v_i} \rightarrow \cdots \rightarrow T_{v_n} \rightarrow T_{v_i}
\]

where \(T_{v_i}\) is the type of variable \(v_i \in LV\) and \(n = |LV|\) and \(LV\) is the set of program variables inside the loop.

These update functions are obtained using symbolic execution [21]. The first step is to gives the variables \(v_1, \ldots, v_n\) the symbolic values \(\alpha_1, \ldots, \alpha_n\). After the symbolic execution each variable \(v_1,\ldots,v_n\) will have a symbolic value: polynomials over \(\alpha_1,\ldots,\alpha_n\), constants and path conditions (to handle branching inside the loop). Update functions can then be derived by substituting the symbolic values \(\alpha_1,\ldots,\alpha_n\) by their corresponding program variables \(v_1,\ldots,v_n\).

For example for the loop in Listing 3.2, \(\phi_i\) is the result of symbolic execution of the loop with \(\alpha_i\) used as symbolic value for \(i\):

\[
\phi_i(\alpha_i) = \begin{cases} 
\alpha_i - 1 & \text{if } \alpha_i > 22 \\
\alpha_i + 4 & \text{if } \neg(\alpha_i > 22)
\end{cases}
\]

When substituting \(i\) for \(\alpha_i\) results in the following update function:

\[
\text{next}_{i}(i) = \begin{cases} 
i - 1 & \text{if } i > 22 \\
i + 4 & \text{if } \neg(i > 22)
\end{cases}
\]

These update functions will be used to determine a value for the variables for which condition jumping occurs. Let \(con_1\) and \(con_2\) be functions representing the two pieces to check for condition jumping with the following type: \(v_1 \ldots v_n \rightarrow \text{Bool}\). If there are values for the variables \(v_1 \ldots v_m \in CV\), where \(CV\) is the set of program variables in the condition and \(m = |CV|\), such that the following holds, then condition jumping occurs:

\[
con_1(v_1, \ldots, v_n) \land con_2(\text{next}_{v_1}(v_1, \ldots, v_n), \ldots, \text{next}_{v_n}(v_1, \ldots, v_n))
\]
SMT scripts

To calculate all values for which condition jumping occurs, an SMT solver can be used. An SMT solver solves Satisfiability Modulo Theories problems, which are a generalisation of boolean satisfiability (SAT) problems. The goal for SAT problems is to determine given a certain set of boolean variables and formulas, if there is a valuation for those variables such that the formulas evaluate to true. SMT problems generalise this by using underlying theories like equality reasoning and arithmetics [7][12].

SMT-LIB is a standard scripting language for SMT solvers, which is describes by Barrett et al. in [5]. The SMT-LIB script in Listing 3.3 is an example script for the loop in Listing 3.2.

Listing 3.3: First SMT-LIB script

```plaintext
(set-logic QF_LIA)
(declare-fun i () Int)
(define-fun nexti ((x Int)) Int (ite (> x 22) (- x 1) (+ x 4)))
(assert (and (and (> i 0) (< i 20)) (> (nexti i) 22)))
(check-sat)
(exit)
```

The script does the following:

1. A variable `i` of type `Int` is created.
2. A function `nexti` is created, which contains the update function for variable `i`.
3. The assertion is made that there is a value for `i`, for which holds that `0 < i < 20` and that after the next iteration `i > 22`, i.e. `nexti(i) > 22`.
4. The SMT solver then checks this assertion.

If the assertion holds, `(get-value (i))` can be added to the script to get the value of `i` for which condition jumping occurs.

The result of running this script will be that for `i = 19` condition jumping occurs. In the next script it will be checked whether there are more values of `i` for which condition jumping occurs. This is shown in Listing 3.4.

Listing 3.4: First SMT-LIB script

```plaintext
(set-logic QF_LIA)
(declare-fun i () Int)
(define-fun nexti ((x Int)) Int (ite (> x 22) (- x 1) (+ x 4)))
(assert (and (and (> i 0) (< i 20)) (> (nexti i) 22)))
(assert (distinct i 19))
(check-sat)
(get-value (i))
(exit)
```

For this example, there are no other cases of condition jumping, so this script is unsatisfiable, i.e. the SMT solver can not find another \( i \) that leads to condition jumping.

The next step is to check if there are values of \( i \) which lead to \( i = 19 \) after the next iteration. This is shown in Listing 3.5.

**Listing 3.5: First SMT-LIB script**

```plaintext
1 (set-logic QF_LIA)
2 (declare-fun i () Int)
3 (define-fun nextI ((x Int)) Int (ite (> x 22) (- x 1) (+ x 4)))
4 (assert (and (and (> i 0) (< i 20)) (= (nextI i) 19)))
5 (assert (distinct i 19))
6 (check-sat)
7 (get-value (i))
8 (exit)
```

This script will return \( i = 15 \). Then this step needs to be repeated for \( i = 15 \) to check whether there are values of \( i \), that will be after the next iteration \( i = 15 \) and so on, until no new values can be added.

### 3.3.2 Binary Decision Diagrams

In this section a second solution to the detection of condition jumping will be presented. The difference with the previous solution, is that it uses Binary Decision Diagrams (BDDs), which are described by Akers in [2], instead of SMT scripts.

BDDs are used to find satisfying assignments for a given formula, like SMT solvers. However, they use different methods. SMT solvers use logic to check the satisfiability of the assumption, whereas BDDs create a decision diagram considering every possible value for each variable and use logic to prune the decision tree, as described in [10].

In this section the same example of condition jumping will be used as in the previous sections, which is shown in Listing 3.2.

One of the challenges of the solution using SMT scripts is the generation of the update functions, which was implemented by the `nextI` function, to calculate the value of \( i \) after one iteration. Kersten and Van Eekelen propose in [21] to use symbolic execution of the loop body.

However, this is not necessary when using BDDs, since the syntax of the loop body can be used in the update functions. This is possible, because there are libraries for using BDDs in a programming language, like Java or C(++), so the update functions can contain the original code of the loop body. Although, there are similar libraries available for SMT solvers\(^2\) it is not possible to use the code of the loop body for update functions, since SMT libraries use their own implementations for data structures for better performance.

In Listing 3.6 a C++ program is given to check if condition jumping occurs in the loop in Listing 3.2 using the BDD library package BuDDy\(^3\).


\(^3\)A C++ library is used, since Java libraries were incomplete and had to be extended first.
Listing 3.6: A C++ program using BDDs to check for condition jumping

```cpp
#include <stdlib.h>
#include <string.h>
#include "bdd.h"
#include "bvec.h"

const int VECTOR_LENGTH = 6;
using namespace std;

int nexti(int i){
    if(i > 22)
        i = --;
    else
        i += 4;
    return i;
}

bvec update_i(bvec vec)
{
    int i = bvec_val(vec);
    i = nexti(i);
    int dif = i - bvec_val(vec);
    if(dif == 0)
        return i;
    else if(dif > 0)
        return bvec_add(vec, bvec_con(VECTOR_LENGTH, dif));
    else
        return bvec_sub(vec, bvec_con(VECTOR_LENGTH, dif*1));
}

bdd con1(bvec i)
{
    return bdd_and(bvec_gth(i, bvec_con(VECTOR_LENGTH, 0)), bvec_lth(i, bvec_con(VECTOR_LENGTH, 20)));
}

bdd con2(bvec i)
{
    return bvec_gth(i, bvec_con(VECTOR_LENGTH, 22));
}

int main()
{
    int domain[1] = {64};
    bdd_init(100,100);
} 
```
In the following, the different functions of the C++ program will be explained. In BDDs integers are implemented using a bit vector.

The pieces of the loop guard are implemented in the \texttt{con1} and \texttt{con2} functions. For example \texttt{con2} is defined as

\[
\text{bvec}_\text{gth}(i, \text{bvec}_\text{con}(\text{VECTOR}_\text{LENGTH}, 22))
\]

where \texttt{bvec}_\text{gth} is the \texttt{>} operator for bit vectors and \texttt{bvec}_\text{con} creates a constant bit vector of length \texttt{VECTOR}_\text{LENGTH} with value 22, i.e. \texttt{con2} returns: \(i > 22\), and similar for \texttt{con1}: \(i > 0 \land i < 20\).

The update function consists of two parts: \texttt{update}_i and \texttt{next}_i. The \texttt{next}_i function consists of the loop body and returns an integer value of \(i\) after one iteration. The \texttt{update}_i function updates the bit vectors according to the \texttt{next}_i function. It first calculates the current integer value of the bit vector, passes it on to the \texttt{next}_i function and calculates the difference after one iteration and adds/subtracts the difference to the bit vector. It is also possible to create a \texttt{next}_i function for bit vectors, so the \texttt{update}_i function is not needed anymore. However, this results in the same challenge as with SMT scripts, since the updates of a variable need to be analysed and transformed to another format e.g. using update functions, which is not needed in the example above, since the loop body is used.

The main statement in the example is:

\[
\text{bdd result} = \text{bdd}_\text{and} (\text{con1}(i), \text{con2}(\text{update}_i(i)))
\]

which calculates the values of \(i\) for which \texttt{con1} is true and in the next state \texttt{con2} is true.

3.3.3 Comparison

In this section a comparison between the two solutions is made and using this comparison a solution will be chosen to be implemented in the prototype ResAna.

The BDD approach has some advantages over the SMT approach. The main advantage of the BDD approach is the use of the loop body of the original code, or at least the statements involving the variable that needs to be updated, instead of doing symbolic execution to analyse the new value after one iteration. In other words, the loop body can be used by itself, whereas update functions first have to be calculated.

Another advantage is that is calculates all satisfiable cases at once. In the example, \(i = 19\) is a satisfiable case, i.e. condition jumping occurs. When using
3.3. IMPLEMENTATION

Table 3.1: Execution time for SMT

<table>
<thead>
<tr>
<th>Value</th>
<th>1 variables z3</th>
<th>2 variables z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>0.005</td>
<td>0.005</td>
</tr>
<tr>
<td>119</td>
<td>0.005</td>
<td>0.005</td>
</tr>
<tr>
<td>1019</td>
<td>0.005</td>
<td>0.005</td>
</tr>
<tr>
<td>10019</td>
<td>0.006</td>
<td>0.006</td>
</tr>
<tr>
<td>100019</td>
<td>0.005</td>
<td>0.007</td>
</tr>
<tr>
<td>1000019</td>
<td>0.006</td>
<td>0.007</td>
</tr>
</tbody>
</table>

the SMT approach, a check has to be done if there is another case of condition jumping, where \( i \neq 19 \). However, the BDD approach will return them all at once, so less executions need to be done.

Another advantage, especially when using a Java library, is that testing by executing a Java program is already used for the Loop Bound Analysis and can easily be extended to execute condition jumping tests as well.

However, the BDD approach has some disadvantages as well. The main disadvantage is that Integers are not used in BDDs, only boolean values. This can be solved by using bit vectors, as is done in the BuDDy package for C(++)

However, the implementation of bit vectors is incomplete for the C(++) package, since only unsigned integers are supported. Other libraries for BDDs, like JDD or JavaBDD (two Java implementations of BDDs), do not have bit vector implementations. Therefore, the BDD libraries all have to be extended to use (signed) integers as bit vectors.

Moreover, using bit vectors rises the assumption that execution time grows exponentially, with the size the integers used. Therefore, an execution time comparison is made between the SMT and BDD approach.

**Execution time comparison**

An execution time comparison has been made, in order to test the extra time needed for running SMT or BDD scripts with increasing the size of the variables.

For both SMT and BDD the size of the condition jumping case 19 has been increased to 119, 1019, . . . , 10000019 for both one variable and two variables. In Table 3.1 the result for the approach SMT are shown. In Table 3.2 the results for BDD approach are shown. Note that these results only involve running the scripts, i.e. the SMT results do not contain the execution time of the symbolic execution, which takes a few seconds for these examples.

However, the results are quite clear. When using larger numbers BDDs become slow and it is assumed that it grows exponentially, whereas the execution time of SMT is static and does not change much for larger values. Therefore, for these timing issues, the SMT approach is regarded as better to create a general implementation. It will be slower for small values, because of symbolic execution and more executions of the SMT scripts. However, the execution time will not grow exponentially and will perform better for larger values than BDDs.

Therefore, the SMT approach is used to solve the condition jumping challenge in the ResAna prototype. Analysing the loop in Listing 3.2, results in the
CHAPTER 3. L.B.A. FOR LOOPS WITH DISJUNCTIVE GUARDS

<table>
<thead>
<tr>
<th>Vector Length (Value)</th>
<th>z3</th>
<th>2 variables z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 (19)</td>
<td>0.011</td>
<td>0.009</td>
</tr>
<tr>
<td>7 (119)</td>
<td>0.012</td>
<td>0.010</td>
</tr>
<tr>
<td>10 (1019)</td>
<td>0.012</td>
<td>0.010</td>
</tr>
<tr>
<td>14 (10019)</td>
<td>0.021</td>
<td>0.016</td>
</tr>
<tr>
<td>17 (100019)</td>
<td>0.030</td>
<td>0.049</td>
</tr>
<tr>
<td>20 (1000019)</td>
<td>0.157</td>
<td>0.297</td>
</tr>
<tr>
<td>24 (10000019)</td>
<td>2.472</td>
<td>4.920</td>
</tr>
<tr>
<td>25 (100000019)</td>
<td>5.006</td>
<td>9.967</td>
</tr>
</tbody>
</table>

Table 3.2: Execution time for BDDs

following ranking function:

\[
\begin{align*}
6 & \text{ if } (i > 0) \land (i < 20) \land (i = 19 \lor i = 15 \lor i = 11 \lor i = 7 \lor i = 3) \\
5 & \text{ if } (i > 0) \land (i < 20) \\
(i - 22) & \text{ if } i > 22 \\
0 & \text{ else }
\end{align*}
\]

This is not the precise ranking function as given before. However, it is a valid upper bound.
Chapter 4

Feasibility of Loop Bound Analysis for Loops with Disjunctive Guards in Practice

In this chapter a feasibility study of Loop Bound Analysis for loops with disjunctive guards will be performed. First a global study will be performed on 16 Java projects. This global evaluation will focus on the number of disjunctive loops in Java projects and will give an answer to the question: how many loops contain disjunctive guards?

The second study will be a detailed loop bound analysis on the loops with disjunctive guards in the source code of Eclipse IDE. Each loop will be examined and a ranking function, if any, will be derived manually. In this chapter an evaluation of the basic method and condition jumping extension will be given. First an analysis of the ratio of condition jumping loops in existing Java programs/projects will be made. This study will give an answer to the question: which loops can be analysed given the current methods?

Given the results of the detailed study the loops will be categorised and for each category an extension, if possible, will be proposed to infer ranking functions for those loops.

4.1 Global Evaluation

In the following sections different Java programs/projects from practice are analysed in order to find out how many loops contain disjunctive guards. This is done by counting first all while-loops and all for-loops and then for both count how many contain disjunctive guards.

4.1.1 Apache harmony

Apache harmony is a Java runtime environment implementation. The source code can be downloaded at [http://harmony.apache.org/](http://harmony.apache.org/).
4.1.2 Caliper
Caliper is a framework for microbenchmarks. The source code can be downloaded at [http://code.google.com/p/caliper/]

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>for-loops</td>
<td>287</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>307</td>
<td>0</td>
</tr>
</tbody>
</table>

4.1.3 CDx
Collision Detector (CDx) is a real-time Java benchmark. The source code can be downloaded at [http://sss.cs.purdue.edu/projects/cdx/]

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>171</td>
<td>1</td>
</tr>
<tr>
<td>for-loops</td>
<td>382</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>553</td>
<td>1</td>
</tr>
</tbody>
</table>

4.1.4 Eclipse IDE
Eclipse is an open-source Integrated Development Environment (IDE) for Java, but through the use of plug-ins several other programming and scripting languages, like C(++) and python, are supported as well. The source code can be downloaded at [http://www.eclipse.org/]

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>9577</td>
<td>125</td>
</tr>
<tr>
<td>for-loops</td>
<td>35672</td>
<td>9</td>
</tr>
<tr>
<td>Total</td>
<td>45249</td>
<td>134</td>
</tr>
</tbody>
</table>

4.1.5 GA-Playground
GA-Playground is a toolkit for genetic algorithms written in Java. The source code can be downloaded at [http://www.aridolan.com/o files/ga/gaa/gaa.aspx]

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>for-loops</td>
<td>222</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>241</td>
<td>1</td>
</tr>
</tbody>
</table>
4.1.6 GWT-java-math

GWT-java-math is an efficient java.math implementation for Google Web Toolkit (GWT). The source code can be downloaded at http://code.google.com/p/gwt-java-math/

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>44</td>
<td>1</td>
</tr>
<tr>
<td>for-loops</td>
<td>152</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>196</td>
<td>1</td>
</tr>
</tbody>
</table>

4.1.7 JAGA

JAGA is a Java API for genetic algorithms. The source code can be downloaded at http://www.jaga.org/.

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>for-loops</td>
<td>139</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>173</td>
<td>1</td>
</tr>
</tbody>
</table>

4.1.8 Data Structures and Algorithm Analysis

Data Structures and Algorithm Analysis is a book by Shaffer [26]. The source code of the examples in the book are used and can be downloaded at http://people.cs.vt.edu/~shaffer/Book/

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>150</td>
<td>0</td>
</tr>
<tr>
<td>for-loops</td>
<td>669</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>819</td>
<td>0</td>
</tr>
</tbody>
</table>

4.1.9 Lightweight Java Game Library

The Lightweight Java Game Library is a library which can be used to write games in Java. The source code can be downloaded at http://lwjgl.org/

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>5135</td>
<td>35</td>
</tr>
<tr>
<td>for-loops</td>
<td>16824</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>21959</td>
<td>35</td>
</tr>
</tbody>
</table>

4.1.10 Java-ML

Java Machine Learning Library (Java-ML) is a collection of machine learning algorithms written in Java. The source code can be downloaded at http://java-ml.sourceforge.net/

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>95</td>
<td>3</td>
</tr>
<tr>
<td>for-loops</td>
<td>891</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>986</td>
<td>5</td>
</tr>
</tbody>
</table>
4.1.11 JavaNCSS

JavaNCSS is a source measurement suite for Java. The source code can be downloaded at [http://www.kclee.de/clemens/java/javancss/](http://www.kclee.de/clemens/java/javancss/).

<table>
<thead>
<tr>
<th></th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>266</td>
<td>1</td>
</tr>
<tr>
<td>for-loops</td>
<td>218</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>484</td>
<td>1</td>
</tr>
</tbody>
</table>

4.1.12 Jembench

Jembench is a Java benchmark for embedded systems. The source code can be downloaded at [http://sourceforge.net/projects/jembench/](http://sourceforge.net/projects/jembench/).

<table>
<thead>
<tr>
<th></th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>52</td>
<td>2</td>
</tr>
<tr>
<td>for-loops</td>
<td>156</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>208</td>
<td>2</td>
</tr>
</tbody>
</table>

4.1.13 JGAP

JGAP is a genetic algorithms and genetic programming framework for Java. The source code can be downloaded at [http://jgap.sourceforge.net/](http://jgap.sourceforge.net/).

<table>
<thead>
<tr>
<th></th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>146</td>
<td>1</td>
</tr>
<tr>
<td>for-loops</td>
<td>727</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>873</td>
<td>1</td>
</tr>
</tbody>
</table>

4.1.14 Jmatbench


<table>
<thead>
<tr>
<th></th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>for-loops</td>
<td>436</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>457</td>
<td>0</td>
</tr>
</tbody>
</table>

4.1.15 NeoBio


<table>
<thead>
<tr>
<th></th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>for-loops</td>
<td>68</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>81</td>
<td>3</td>
</tr>
</tbody>
</table>
4.1.16 Weka


<table>
<thead>
<tr>
<th></th>
<th>Number of loops</th>
<th>Number of disjunctive loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>while-loops</td>
<td>1155</td>
<td>22</td>
</tr>
<tr>
<td>for-loops</td>
<td>6026</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>7181</td>
<td>24</td>
</tr>
</tbody>
</table>

4.1.17 Totals

In total there were 97664 loops in the projects described above, from which only 253 contained disjunctions in their condition, which is less than 0.26%. This shows that loops in which condition jumping can occur are very rare.

4.2 Detailed Feasibility Study

None of the loops in the Java programs/projects described in the previous example, were applicable for the method described in Chapter 3. Therefore, a manual derivation of ranking function will be done for the loops with disjunctive guards of the Eclipse IDE.

The loops used in this detailed feasibility study are listed in Appendix A.

4.2.1 Detailed Loop Discussion

Listing A.1

The loop is bounded by the number of characters returned by `nextChar()`, which is bounded by the size of the array `fSmap`. However, the function `nextChar()` is not well defined: the integer `fPointer` is only checked for equality with the length of `fSmap`. Therefore, it is assuming that `fPointer` is always less than or equal to the length of `fSmap`, resulting in an exception, when this is not true. It would be better to add an extra check, e.g. using `>=` or add an extra if-statement.

**Upper bound:**

\[ fSmap.length \]

**Ranking function:**

\[
\begin{align*}
\text{if } fPointer < fSmap.length & \quad fSmap.length - fPointer \\
\text{if } fPointer = fSmap.length & \quad 1 \\
\text{undefined/exception} & \quad \text{otherwise}
\end{align*}
\]

Listing A.2

The loop is bounded by the integer `length`, which is set to the length of `string`, i.e. the loop is bounded by the length of String `string`. There is no check in the loop if `i` is negative, which would result in an exception, since `i` is out of the bounds of `string`. However, this check is not necessary, since `i` is set to `-1` right before the loop is entered.
CHAPTER 4. FEASIBILITY STUDY

Upper bound: 
length
Ranking function: 
\[
\begin{cases}
  \text{length} - (i - 1) & \text{if } (i - 1) < \text{length} \\
  0 & \text{otherwise}
\end{cases}
\]

Listing A.3
The loop is bounded by the length of the array lexem. However, there is no check if index i exceeds the boundaries of the array, which results in an exception. This happens when the array is filled with only spaces (’ ’) and tabs (’	’). Moreover, the first assignment of variable c is not checked as well, i.e. if lexem is empty, the assignment would result in an exception as well. There is no check for i being negative, but this is not strictly necessary, since it is set to 0.

Upper bound: 
lexem.length
Ranking function: 
\[
\begin{cases}
  \text{lexem.length} - i & \text{if } i < \text{lexem.length} \\
  \text{undefined/exception} & \text{otherwise}
\end{cases}
\]
The loop is better defined in the following way:

Listing 4.1: Improvement of Listing A.3
1  if (lexem.length > 0) {
2    int i = 0;
3    char c = lexem[0];
4    while (++i < lexem.length && (c == ' ' || c == '	')) {
5      c = lexem[i];
6    }
}

Listing A.4
No upper bound/ranking function can be defined, since the exiting of the loop is defined by the type of the current state, e.g. if the current state is a HttpParser.STATE_END state, the loop will be exited.

Listing A.5
No upper bound/ranking function can be defined, since the variables now and end depend on time.

Listing A.6
No upper bound/ranking function can be defined, since it is not possible to predict the value of len in each iteration, it is set using a switch on the variable to_flush of which the value is not known statically, i.e. runtime information is needed.
Listing A.7
No upper bound/ranking function can be defined, since it depends on non-numerical values. Moreover, the check if \textit{id} is a null-pointer can only be done at runtime.

Listing A.8
No upper bound/ranking function can be defined. The loop depends on the elements in \texttt{cache}; the loop iterates until \texttt{cache} is empty. However, how many elements are removed each iteration is unpredictable, since it depends on other objects, from which the value is only known at runtime.

Listing A.9
The number of iteration of this loop is bounded by the size of that data structure for which \textit{iter} has been created, i.e. \texttt{destinationRepos}.

\textbf{Upper bound:}\n\texttt{destinationRepos.size()}
It is not possible to define a ranking function given the current loop. It has to be extended with a counter, which will be explained in Section 4.3.3.

Listing A.10
The number of iterations is bound by the number of elements in the data structure for which \texttt{iteratorIterator} is created. The explanation given in the comments of the Java class, show that it is expected that \texttt{iteratorIterator} to contain iterators of a certain data type. However, the only type known is \texttt{Iterator}. Therefore, it is not possible to give a concrete upper bound or ranking function, since the \texttt{Iterator} class does not have a size method or field.

Listing A.11
The loop itself shows that the number of iterations is probably bound by the number of tokens to be processed: \texttt{nextToken()}. However, the function \texttt{nextToken()}, as defined in the superclass \texttt{ExpressionParser}, gives more information on the number of tokens, which is bounded by the length of the String \texttt{expression}.

\textbf{Upper bound:}\n\texttt{expression.length()}

\textbf{Ranking function:}\n\begin{cases} \texttt{expression.length() - tokenPos} & \text{if } \texttt{tokenPos < expression.length()} \\ 0 & \text{otherwise} \end{cases}

Note that in the function \texttt{nextToken()} the variable \texttt{top} is used to store the value of \texttt{expression.length()}. The latter is used in the upper bound and ranking function, since \texttt{expression} is a field and is globally accessible, whereas \texttt{top} is just a local variable.

Listing A.12
The loop is bounded by the length of \texttt{filterString}. However, there is no check if \texttt{position} is in the right range, i.e. there should be a check of \texttt{position} is positive
and less than the length of `filterString`.

In the following `pos` is the notation for the variable `position`.

**Upper bound:**

`filterString.length()`

**Ranking function:**

\[
\begin{cases}
    filterString.length() - pos & \text{if } pos > 0 \land pos < filterString.length() \\
    \text{undefined/exception} & \text{if } pos > 0 \land pos > filterString.length() \\
    \text{undefined/exception} & \text{otherwise}
\end{cases}
\]

A better loop would be:

```java
if (position > 0 && position < filterString.length()) {
    int begin = position;
    int end = position;
    char c = filterString.charAt(begin);
    while (position++ < filterString.length() && !
        (c == '~' || c == '<' || c == '>' || c == '=' ||
        c == '(' || c == ')')) {
        if (!Character.isWhitespace(c))
            end = position;
        c = filterString.charAt(position);
    }
}
```

Listing A.13

The loop itself shows that the number of iterations is probably bound by the number of tokens to be processed: `nextToken()`. However, the function `nextToken()`, as defined in the superclass `ExpressionParser`, gives more information on the number of tokens, which is bounded by the length of the String `expression`.

**Upper bound:**

`expression.length()`

**Ranking function:**

\[
\begin{cases}
    expression.length() - tokenPos & \text{if } tokenPos < expression.length() \\
    0 & \text{otherwise}
\end{cases}
\]

Note that in the function `nextToken()` the variable `top` is used to store the value of `expression.length()`. The latter is used in the upper bound and ranking function, since `expression` is a field and is globally accessible, whereas `top` is just a local variable.

Listing A.14

No upper bound/ranking function can be defined, since the variable `delay` depends on time.

Listing A.15

The loop condition consists of a conjunction. Therefore, both sides of the conjunction can be used as an upper bound. When only using the right part
(readCnt < len) a ranking function can be found using the method described in Section 2.1.

Upper bound:

len

Ranking function:

\[
\begin{cases} 
  \text{len} - \text{readCnt} & \text{if readCnt} < \text{len} \\
  0 & \text{otherwise}
\end{cases}
\]

The left part of the conjunction can then be used to make the ranking function as described above more precise. However, in this case this is not possible, since both pos and buffer depend on values, which are only available at runtime.

Listing A.16

No upper bound/ranking function can be defined, since it depends on the execution of other threads.

Listing A.17

No upper bound/ranking function can be defined, since it depends on the execution of other threads.

Listing A.18

No upper bound/ranking function can be defined, since the update of \( j \) is restricted by some conditions, which depend on values which are only present at runtime. There is the value of \( j \) unpredictable.

Listing A.19

The loop is bounded by the number of elements in \( \text{resources} \). This is an iterator defined on \( \text{selection} \), which is of the type \( \text{IStructuredSelection} \). This has a method \( \text{size()} \), which returns the number of elements in \( \text{selection} \).

Upper bound:

\( \text{selection.size()} \)

It is not possible to define a ranking function given the current loop. It has to be extended with a counter, which will be explained in Section 4.3.3.

Listing A.20

The loop is bound by \( \text{index} + 1 \), since index decreases with one every iteration.

Upper bound:

\( \text{index} + 1 \)

Ranking function:

\[
\begin{cases} 
  \text{index} + 1 & \text{if index} > -1 \\
  0 & \text{otherwise}
\end{cases}
\]

Listing A.21

No upper bound/ranking function can be defined, since \( \text{scanner} \) depends on a stream, for which the number of elements are unknown.
CHAPTER 4. FEASIBILITY STUDY

Listing A.22

The number of iterations is bounded by the depth of node in the tree, i.e., the number of parents of node. The height/depth of the full tree is the upper bound. However, these numbers can only be specified at runtime, by checking for this object, how many times getParent() can be performed until arriving at the root of the tree.

Listing A.23

The loop is bounded by the number of characters returned by nextChar(). However, the number of characters this method returns depends on a stream. Therefore, no concrete upper bound/ranking function can be given.

Listing A.24

The number of iterations of this loop depends on the values of idx and ptr. However, ptr only updates given some constraints, which depend on runtime values, which make ptr unpredictable.

Listing A.25

To find a ranking function for this loop, the loop condition has to be split into different pieces. For better understanding, consider the simplified version of the loop:

```
Listing 4.3: Simplified version of Listing A.25
1 do {
2     if (fieldStart < methodStart && fieldStart < typeStart) {
3         fieldIndex++; 
4     } else if (methodStart < fieldStart && methodStart < typeStart) {
5         methodIndex++; 
6     } else {
7         typeIndex++; 
8     }
9 } while ((fieldIndex < fieldCount) || (typeIndex < typeCount) || (methodIndex < methodCount));
```

The following notation will be used for the conditions and variables: $A = (fieldIndex < fieldCount)$, $B = (typeIndex < typeCount)$, $C = (methodIndex < methodCount)$, $fC = fieldCount$, $fI = fieldIndex$, $tC = typeCount$, $tI = typeIndex$, $mC = methodCount$ and $mI = methodIndex$. 
4.2. DETAILED FEASIBILITY STUDY

Upper bound:
\[ \max(1, \max(0, (fC - fI) + \max(0, (tC - tI)) + \max(0, (mC - mI))) \]

Ranking function:
\[
\begin{cases} 
  fC - fI & \text{if } A \land \neg B \land \neg C \\
  tC - tI & \text{if } \neg A \land B \land \neg C \\
  mC - mI & \text{if } \neg A \land \neg B \land C \\
  (fC - fI) + (tC - tI) & \text{if } A \land B \land \neg C \\
  (fC - fI) + (mC - mI) & \text{if } A \land \neg B \land C \\
  (tC - tI) + (mC - mI) & \text{if } \neg A \land B \land C \\
  (fC - fI) + (tC - tI) + (mC - mI) & \text{if } A \land B \land C \\
  1 & \text{otherwise}
\end{cases}
\]

Listing A.26

The loop is bounded by the number of tokens, which are subsets of the string \( fContent \). Assuming each token contains at least one character, the loop is bounded by the length of \( fContent \).

Upper bound:
\( fContent.length \)

Listing A.27

The loop is bounded by the number of characters returned by \( \text{nextChar}() \), which is the same as in Listing A.23. However, the number of characters this method returns depends on a stream. Therefore, no concrete upper bound/ranking function can be given.

Listing A.28

The loop iterates over the characters returned by \( \text{reader.read}() \). In the source code it is shown that it either returns a character from \( fDocument \) or \( EOF \), which is \(-1\) after which the loop is exited.

Upper bound:
\( fDocument.getLength() + 1 \)

In the current situation no ranking function can be defined. However, with the use of a special counter it is possible, as will be explained in Section 4.3.3.

Listing A.29

The number of iterations is bounded by the depth of \( node \) in the tree, i.e. the number of parents of \( node \). The height/depth of the full tree is the upper bound. However, these numbers can only be specified at runtime, by checking for this object how many times \( \text{getParent}() \) can be performed until arriving at the root of the tree.

Listing A.30

The number of iterations is bounded by the depth of \( node \) in the tree, i.e. the number of parents of \( node \). The height/depth of the full tree is the upper bound. However, these numbers can only be specified at runtime, by checking for this object how many times \( \text{getParent}() \) can be performed until arriving at the root of the tree.
Listing A.31

Similar to Listing A.28. The loop iterates over the characters returned by reader.read(). In the source code it is shown that it either returns a character from fDocument or EOF, which is −1 after which the loop is exited.

Upper bound:

\[ f\text{Document}.\text{getLength}() + 1 \]

In the current situation no ranking function can be defined. However, with the use of a special counter it is possible, as will be explained in Section 4.3.3.

Listing A.32

The loop condition consists of two booleans. However, the loop can be simplified to the following:

Listing 4.4: Simplified version of Listing A.32

1 while (iLocal < maxLocals || iscope < maxScopes) {
2     if (iscope < maxScopes
3         && (iLocal >= maxLocals || (this.
4             subscopes[iscope].startIndex() <=
5                 iLocal))) {
6         iscope ++;
7     } else {
8         iLocal ++;
9     }
10 }

This makes the loop similar to the one in Listing A.25. The following notations will be used: \(iL = \text{iLocal}, mL = \text{maxLocals}, iS = \text{iscope}, mS = \text{maxScopes}, \text{hasMoreVariables = iLocal < maxLocals}\) and \(\text{hasMoreScopes = iscope < maxScopes}\).

Upper bound:

\[ \max(0, mL - iL) + \max(0, mS - iS) \]

Ranking function:

\[
\begin{align*}
    mL - iL & \quad \text{if } \text{hasMoreVariables} \land \neg\text{hasMoreScopes} \\
    mS - iS & \quad \text{if } \neg\text{hasMoreVariables} \land \text{hasMoreScopes} \\
    (mL - iL) + (mS - iS) & \quad \text{if } \text{hasMoreVariables} \land \text{hasMoreScopes} \\
    0 & \quad \text{otherwise}
\end{align*}
\]

Listing A.33

See Listing A.32

Listing A.34

The loop iterates through the array unitSource. Therefore, the length of that array is the upper bound on the number of iterations. However, it should be checked that the index begin does not exceed the boundaries of the array.
4.2. DETAILED FEASIBILITY STUDY

Upper bound:

unitSource.length

Ranking function:

\[
\begin{cases}
    \text{Undefined/exception} & \text{if } \text{begin} < 0 \\
    \text{unitSource.length} - \text{begin} & \text{if } \text{begin} \geq 0 \land \text{begin} < \text{unitSource.length} \\
    \text{Undefined/exception} & \text{otherwise}
\end{cases}
\]

A better loop would be:

Listing 4.5: Improvement of Listing A.34

```java
1 while (begin > 0 && begin < unitSource.length && ((c = unitSource[begin]) == ' ' || c == '\t')) begin++;
```

Listing A.35

No upper bound/ranking function can be defined, since it depends on non-numerical objects: unclaimedAnnotations.

Listing A.36

The loop iterates through the array unitSource. Therefore, the length of that array is the upper bound on the number of iterations. However, it should be checked that the index begin does not exceed the boundaries of the array.

Upper bound:

unitSource.length

Ranking function:

\[
\begin{cases}
    \text{Undefined/exception} & \text{if } \text{begin} < 0 \\text{unitSource.length} \\
    \text{unitSource.length} - \text{begin} & \text{if } \text{begin} \geq 0 \land \text{begin} < \text{unitSource.length} \\
    \text{Undefined/exception} & \text{otherwise}
\end{cases}
\]

A better loop would be:

Listing 4.6: Improvement of Listing A.36

```java
1 while (begin > 0 && begin < unitSource.length && ((c = unitSource[begin]) == ' ' || c == '\t')) begin++;
```

Listing A.37

No upper bound/ranking function can be defined, since it depends on non-numerical objects: constructorBinding and enclosingTypeBinding.

Listing A.38

No upper bound/ranking function can be defined, since it depends on non-numerical objects: constructorBinding and enclosingTypeBinding.

Listing A.39

No upper bound/ranking function can be defined, since it depends on non-numerical objects: constructorBinding and enclosingTypeBinding.
Listing A.40

No upper bound/ranking function can be defined, since the variables in the loop condition are assigned by objects instead of updated, which makes the variables unpredictable.

Listing A.41

The loop iterates over an array `this.source` with `this.currentPosition` as a starting point.

**Upper bound:**

\[
\text{this.source.length}
\]

**Ranking function:**

\[
\begin{align*}
\text{Undefined/exception} & \quad \text{if } pos < 0 \\
\text{length} - pos & \quad \text{if } pos > 0 \land pos < \text{length} \\
\text{Undefined/exception} & \quad \text{otherwise}
\end{align*}
\]

where \( pos = this.currentPosition \) and \( length = this.source.length \).

The loop would be better defined with the following loop condition:

Listing 4.7: Improvement of Listing A.41

```plaintext
1 while (this.currentPosition > 0 && this.currentPosition < this.source.length && ((this.currentCharacter != '/ ') || (!star)))
```

Listing A.42

See Listing A.41.

Listing A.43

No upper bound/ranking function can be defined, since the variables in the loop condition are assigned by objects instead of updated, which makes the variables unpredictable.

Listing A.44

No upper bound/ranking function can be defined, since it depends on a non-numerical object: `result`.

Listing A.45

It seems that this loop searches for a given value inside an array, possibly starting in the middle of the array and work to the end and repeating at the beginning of the array. However, if the value does not exists in the array, the loop will not stop, since there is no check if a value is checked twice.

A better loop would be:

Listing 4.8: Improvement of Listing A.45

```plaintext
1 int startPosition = index;
```
while (index < this.keyTable.length && ((this.keyTable[index] != 0) || ((this.keyTable[index] == 0) && (this.valueTable[index] != 0)))) {
    if (this.keyTable[index] == key)
        return true;
    if (++index == length) {
        index = 0;
    } else if (index == startPosition)
        break;
}

When arriving at the saved position the loop breaks, since that value has already been checked. The extra check in the loop condition prevents an exception in the first iteration.

**Upper bound:**

*length*

To define a ranking function, an extra counter has to be created, since the *index* counter does not grow monotonically, it switches to 0 if it reaches the end of the array. This method will be explained in Section 4.3.3.

**Listing A.46**

See Listing A.45.

**Listing A.47**

See Listing A.45.

**Listing A.48**

See Listing A.45.

**Listing A.49**

See Listing A.45.

**Listing A.50**

See Listing A.45.

**Listing A.51**

The loop iterates through the array *unitSource*. Therefore, the length of that array is the upper bound on the number of iterations. However, it should be checked that the index *begin* does not exceed the boundaries of the array.

**Upper bound:**

*unitSource.length*

**Ranking function:**

\[
\begin{align*}
\text{Undefined/exception} & \quad \text{if } \text{begin} < 0 \land \text{begin} \geq 0 \land \text{begin} < \text{unitSource.length} \\
\text{unitSource.length} - \text{begin} & \quad \text{if } \text{begin} \geq 0 \land \text{begin} < \text{unitSource.length} \\
\text{Undefined/exception} & \quad \text{otherwise}
\end{align*}
\]
A better loop would be:

```
while (begin > 0 && begin < unitSource.length && ((c = unitSource[begin]) == 'U' || c == '\t')) begin++;
```

Listing A.52
See Listing A.51

Listing A.53
See Listing A.51

Listing A.54
The loop iterates through the array \textit{unitSource}. Therefore, the length of that array is the upper bound on the number of iterations. However, it should be checked that the index \textit{end} does not exceed the boundaries of the array.

\textbf{Upper bound:}\n\textit{unitSource.length}

\textbf{Ranking function:}\n\begin{align*}
\begin{cases}
\text{Undefined/exception} & \text{if } end < 0 \\
\text{unitSource.length - end} & \text{if } end >= 0 \land end < \text{unitSource.length} \\
\text{Undefined/exception} & \text{otherwise}
\end{cases}
\end{align*}

A better loop would be:

```
while (end > 0 && end < unitSource.length && ((c = unitSource[end]) == 'U' || c == '\t')) end++;
```

Listing A.55
The loop by itself should contain some checks to prevent index out of bounds exception. However, these checks are not needed, since the entering condition is obtained in the outer loop and the exit condition inside the loop.

\textbf{Upper bound:}\n\textit{length1 - index1}

\textbf{Ranking function:}\n\begin{align*}
\begin{cases}
\text{length1 - index1} & \text{if index1 < length1} \\
1 & \text{otherwise}
\end{cases}
\end{align*}

Listing A.56
The loop by itself should contain some checks to prevent index out of bounds exception. However, these checks are not needed, since the entering condition is obtained in the outer loop and the exit condition inside the loop.
4.2. DETAILED FEASIBILITY STUDY

Upper bound:
length2 − index2

Ranking function:
\[
\begin{cases}
\text{length2 − index2} & \text{if index2 < length2} \\
1 & \text{otherwise}
\end{cases}
\]

Listing A.57

The loop iterates over an array this.source with this.currentPosition as a starting point.

Upper bound:
this.source.length

Ranking function:
\[
\begin{cases}
\text{Undefined/exception} & \text{if pos < 0} \\
\text{length − pos} & \text{if pos > 0 ∧ pos < length} \\
\text{Undefined/exception} & \text{otherwise}
\end{cases}
\]

where pos = this.currentPosition and length = this.source.length.

The loop would be better defined with the following loop condition:

Listing 4.11: Improvement of Listing A.57

1 while (this.currentPosition > 0 && this.currentPosition < this.source.length && ((this.currentCharacter != '/' || (!star)))

Listing A.58

No upper bound/ranking function can be defined, since it depends on a non-numerical object: field.

Listing A.59

The loop iterates over an array this.source with this.index as a starting point.

Upper bound:
this.source.length

Ranking function:
\[
\begin{cases}
\text{Undefined/exception} & \text{if this.index < 0} \\
\text{length − this.index} & \text{if this.index > 0 ∧ this.index < length} \\
\text{Undefined/exception} & \text{otherwise}
\end{cases}
\]

where length = this.source.length.

The loop would be better defined with the following loop condition:

Listing 4.12: Improvement of Listing A.59

1 while (this.index > 0 && this.index < this.source.length && ((this.currentCharacter != '/' || (!star)))

Listing A.60

See Listing A.59
Listing A.61
See Listing A.59

Listing A.62
No upper bound/ranking function can be defined, since the variables in the loop condition are assigned by objects instead of updated, which makes the variables unpredictable.

Listing A.63
The loop iterates over a number of tokens returned by the variable fScanner, which is an IScanner object. An IScanner object cannot return the number of tokens it contains. However, it can return the source of the scanner, which is an array of characters. Assuming, that one token represents one or more characters in the source, the following upper bound can be defined:

Upper bound:
\[ f\text{Scanner.getSource}().\text{length} \]

Listing A.64
The loop iterates as long \( i \) is smaller than the length of the array cleanUps, where \( i \) is used as index for the array, i.e. the loop iterates over the contents of an array. However, it is never checked whether \( i >= 0 \).

Upper bound:
\[ \text{cleanUps.length} \]

Ranking function:
\[
\begin{align*}
\text{Undefined/exception} & \quad \text{if } i < 0 \\
\text{cleanUps.length} & \quad \text{if } i >= 0 \land i < \text{cleanUps.length} \\
0 & \quad \text{otherwise}
\end{align*}
\]

Listing A.65
No upper bound/ranking function can be defined, since it depends on a non-numerical object: javaElem.

Listing A.66
The number of iterations is bounded by the depth of container in the tree, i.e. the number of parents of node. The height/depth of the full tree is the upper bound. However, these numbers can only be specified at runtime, by checking for this object how many times getParent() can be performed until arriving at the root of the tree.

Listing A.67
The number of iterations is bounded by the depth of node in the tree, i.e. the number of parents of node. The height/depth of the full tree is the upper bound. However, these numbers can only be specified at runtime, by checking for this object how many times getParent() can be performed until arriving at the root of the tree.
4.2. DETAILED FEASIBILITY STUDY

Listing A.68

No upper bound/ranking function can be defined, since it depends on a non-numerical object: \textit{id}.

Listing A.69

The loop iterates over the contents of the array \textit{ZigZag8x8}, until the index \textit{k} arrives at \textit{end}, which probably contains the value of \textit{ZigZag8x8.length}. However, it would be better to check on \textit{ZigZag8x8.length} itself instead of \textit{end} or at least whether \textit{end} < \textit{ZigZag8x8.length} and a check should be added for \textit{k} > 0.

Upper bound:
\[\text{end} + 1\]

Ranking function:
\[
\begin{cases} 
\text{Undefined/exception} & \text{if } k < 0 \\
(\text{end} - k) + 1 & \text{if } k \geq 0 \land k \leq \text{end} \land \text{end} < \text{ZigZag8x8.end} \\
\text{Undefined/exception} & \text{if } k > 0 \land k < \text{end} \land \text{end} > \text{ZigZag8x8.end} \\
0 & \text{otherwise}
\end{cases}
\]

Listing A.70

The loop iterates until \textit{h} < \textit{j}. However, those variables are used to access an array, so it should be checked that \textit{j} < \textit{lengths.length} and \textit{j} > 0.

Upper bound:
\[\left\lceil \frac{h}{j} \right\rceil\]

Ranking function:
\[
\begin{cases} 
\text{Undefined/exception} & \text{if } j < 0 \land j > h \\
\text{if } j > 0 \land j < \text{lengths.length} \land j > h & \text{if } j > 0 \land j < \text{lengths.length} \land j > h \\
\text{Undefined/exception} & \text{if } j > \text{lengths.length} \land j > h \\
0 & \text{otherwise}
\end{cases}
\]

Listing A.71

No upper bound/ranking function can be defined, since it depends on a non-numerical object: \textit{cinfo}.

Listing A.72

The loop iterates over the characters in \textit{fDocument}, in the following abbreviated to \textit{fDoc}. However, the index \textit{end} is not checked to be inside the boundaries, i.e. \textit{end} > 0 \land \textit{end} < \textit{fDoc.getLength()}.

Upper bound:
\[\text{fDoc.getLength()}\]

Ranking function:
\[
\begin{cases} 
\text{Undefined/exception} & \text{if } \text{end} < 0 \\
\text{fDoc.getLength()} - \text{end} & \text{if } \text{end} > 0 \land \text{end} < \text{fDoc.getLength()} \\
\text{Undefined/exception} & \text{if } \text{end} > \text{fDoc.getLength()} \\
0 & \text{otherwise}
\end{cases}
\]
Listing A.73
No upper bound/ranking function can be defined, since it depends on non-numerical objects: $e$ and $p$.

Listing A.74
The loop has to be divided into three different pieces.

**Upper bound:**
\[
\max(\max(0, \text{back} + 1), \max(0, \text{size} - \text{forth}))
\]

**Ranking function:**
\[
\begin{cases}
  \text{back} + 1 & \text{if } \text{back} \geq 0 \land 
  \neg(\text{forth} < \text{size}) \\
  \text{size} - \text{forth} & \text{if } \neg(\text{back} \geq 0) \land \text{forth} < \text{size} \\
  \max(\text{back} + 1, \text{size} - \text{forth}) & \text{if } \text{back} \geq 0 \land \text{forth} < \text{size} \\
  0 & \text{otherwise}
\end{cases}
\]

Listing A.75
No upper bound/ranking function can be defined, since the variables in the loop condition are assigned by objects instead of updated, which makes the variables unpredictable.

Listing A.76
The loop is bounded by $l >= \min$.

**Upper bound:**
\[
l - \min + 1
\]

**Ranking function:**
\[
\begin{cases}
  l - \min + 1 & \text{if } l \geq \min \\
  0 & \text{otherwise}
\end{cases}
\]

Listing A.77
The loop is bounded by $l <= \max$.

**Upper bound:**
\[
\max - l + 1
\]

**Ranking function:**
\[
\begin{cases}
  \max - l + 1 & \text{if } \max <= l \\
  0 & \text{otherwise}
\end{cases}
\]

Listing A.78
The loop is bounded by the number of characters returned by $\text{nextChar}()$, which is the same as in Listing A.23. However, the number of characters this method returns depends on a stream. Therefore, no concrete upper bound/ranking function can be given.

Listing A.79
No upper bound/ranking function can be defined, since the variables in the loop condition are assigned by objects instead of updated, which makes the variables unpredictable.
4.2. DETAILED FEASIBILITY STUDY

Listing A.80

The loop iterates until \textit{count} reaches 0.

\textbf{Upper bound:} \textit{count}

\textbf{Ranking function:}

\begin{equation*}
\begin{cases}
\text{count} & \text{if count > 0} \\
0 & \text{otherwise}
\end{cases}
\end{equation*}

Listing A.81

The loop is bounded by the number of tokens returned by the \textit{tokenizer}. However, the number of tokens is not accessible. This can only be known at runtime by going through all tokens and count them. The number of tokens is likely bounded by the number of characters in the original text. This text is stored in an array of characters in the \textit{Tokenizer}. Assuming one token consists of one or more characters, it is an upper bound to the number of tokens. However, the variable \textit{value} is marked as \texttt{protected} and is not accessible in this loop. Therefore, upper bound \texttt{tokenizer.value.length} cannot be used, unless a getter will be created for the \textit{value} field.

Listing A.82

No upper bound/ranking function can be defined, since it depends on a non-numerical object: \textit{file}.

Listing A.83

No upper bound/ranking function can be defined, since it depends on a non-numerical object: \textit{testShell}.

Listing A.84

Variable \textit{it} is a \texttt{CharacterIterator}, which has methods for returning the begin and end index

\textbf{Upper bound:} \texttt{it.getEndIndex() − it.getBeginIndex()}

Listing A.85

No upper bound/ranking function can be defined, since it depends on objects: \textit{myIter} and \textit{yourIter}.

Listing A.86

The number of iterations is bounded by the depth of \textit{superClass} in the tree, i.e. the number of parents of \textit{node}. The height/depth of the full tree is the upper bound. However, these numbers can only be specified at runtime, by checking for this object how many times \texttt{getSuperclassName()} can be performed until arriving at the root of the tree.
Listings [A.87]
The number of iterations is bounded by the depth of $o$ in the tree, i.e. the number of parents of $node$. The height/depth of the full tree is the upper bound. However, these numbers can only be specified at runtime, by checking for this object how many times $getParent()$ can be performed until arriving at the root of the tree.

Listings [A.88]
The loop is bounded by the number of lines in the IDocument $doc$.

**Upper bound:**

```
doc.getLength()
```  

Listings [A.89]
The loop is bounded by the number of lines in the IDocument $doc$.

**Upper bound:**

```
doc.getLength()
```  

Listings [A.90]
The loop is bounded by the number of characters in $newStream$. However, it is an input stream, for which the end is only known when its reached, i.e. the number of characters can only be known at runtime.

Listings [A.91]
The loop is bounded by the number of characters in $oldStream$. However, it is an input stream, for which the end is only known when its reached, i.e. the number of characters can only be known at runtime.

Listings [A.92]
The number of iterations is bounded by the depth of $parentPath$ in the tree, i.e. the number of parents of $node$. The height/depth of the full tree is the upper bound. However, these numbers can only be specified at runtime, by checking for this object how many times $getParentPath()$ can be performed until arriving at the root of the tree.

Listings [A.93]
No upper bound/ranking function can be defined, since it depends on a non-numerical object: $listener$.

Listings [A.94]
The loop is bounded by the number of lines in $reader$. However, it is a reader over an input stream, for which the end is only known when its reached, i.e. the number of characters can only be known at runtime.
4.2. DETAILED FEASIBILITY STUDY

Listing A.95
The loop is bounded by the number of existing names and reserved names.
Upper bound:
reservednames.size() + contents.length

Listing A.96
The loop is bounded by the number of Items i. The class Item is implemented as a linked list, i.e. adding a pointer to the next Item to the objects. However, the number of items can only be known by iterating all the items and counting them, which can only be done at runtime.

Listing A.97
The loop iterates of the elements of the array a, using i as its index. However, there is no check on i for being larger than the length of the array. Moreover, the check $i \neq -1$ should be $i > -1$, since that would exclude negative numbers, which lead to an exception.
Upper bound:
a.length
Ranking function:
\[
\begin{cases} 
  \text{Undefined/exception} & \text{if } i < -1 \\
  \text{Undefined/exception} & \text{if } i = -1 \\
  i + 1 & \text{if } i > -1 \land i < a.length \\
  \text{Undefined/exception} & \text{otherwise}
\end{cases}
\]

Listing A.98
See Listing A.97

Listing A.99
See Listing A.97

Listing A.100
See Listing A.97

Listing A.101
See Listing A.97

Listing A.102
See Listing A.97

Listing A.103
The loop is bounded by the number of characters in reader.
Upper bound:
reader.size
CHAPTER 4. FEASIBILITY STUDY

Listing A.104
The loop is bounded by the number of characters in reader.

Upper bound:
reader.size

Listing A.105
No upper bound/ranking function can be defined, since it depends on updates done in other threads.

Listing A.106
No upper bound/ranking function can be defined, since it depends on updates done in other threads.

Listing A.107
No upper bound/ranking function can be defined, since it depends on updates done in other threads.

Listing A.108
No upper bound/ranking function can be defined, since it depends on updates done in other threads.

Listing A.109
No upper bound/ranking function can be defined, since the variables in the loop condition are assigned by objects instead of updated, which makes the variables unpredictable.

Listing A.110
No upper bound/ranking function can be defined, since it depends on a non-numerical object: current.

Listing A.111
No upper bound/ranking function can be defined, since it depends on a non-numerical object: bucketTable.

Listing A.112
The loop iterates over an array. Therefore, it should have more checks in the loop condition: offset > 0 and offset < b.length.

Two different ranking functions can be given for this loop.

Upper bound:
remaining

Ranking function:
\[
\begin{cases}
\text{remaining} & \text{if remaining > 0} \\
0 & \text{otherwise}
\end{cases}
\]

The following ranking function uses the array b and its index offset.
4.2. DETAILED FEASIBILITY STUDY

Upper bound:
\( b.length \)

Ranking function:
\[
\begin{align*}
& \text{Undefined/exception} \quad \text{if } offset < 0 \\
& b.length - offset \quad \text{if } offset > 0 \land offset < b.length \\
& \text{Undefined/exception} \quad \text{if } offset = b.length \\
& 0 \quad \text{otherwise}
\end{align*}
\]

Listing A.113
No upper bound/ranking function can be defined, since the variables in the loop condition are assigned by objects instead of updated, which makes the variables unpredictable.

Listing A.114
No upper bound/ranking function can be defined, since it depends on updates done in other threads, since \( t \) is an object of the class \( Thread \).

Listing A.115
The loop is bounded by the number of tokens in the \( StringTokenize\) \( st \).

Upper bound:
\( st.countTokens() \)

Listing A.116
The number of iterations is bounded by the number of readers returned by \( getReader() \), which is bounded by \( readerSources \). However, that is an iterator for which the number of items has to be calculated at runtime.

Listing A.117
The loop is bounded by the number of lines in the reader \( in \). However, this is a stream. Therefore, no upper bound/ranking functions can be defined.

Listing A.118
The loop is bounded by the number of lines in the reader \( in \). However, this is a stream. Therefore, no upper bound/ranking functions can be defined.

Listing A.119
The loop is bounded by the number of lines in the reader \( in \). However, this is a stream. Therefore, no upper bound/ranking functions can be defined.

Listing A.120
The loop iterates over an array. Therefore, it should have more checks in the loop condition: \( offset > 0 \) and \( offset < b.length \).

Two different ranking functions can be given for this loop.
Upper bound:
remaining
Ranking function:
\[
\begin{cases}
\text{remaining} & \text{if remaining} > 0 \\
0 & \text{otherwise}
\end{cases}
\]
The following ranking function uses the array \( b \) and its index \( \text{offset} \).
Upper bound:
\( b.length \)
Ranking function:
\[
\begin{cases}
\text{Undefined/exception} & \text{if} \ \text{offset} < 0 \\
\text{b.length - offset} & \text{if} \ \text{offset} > 0 \land \text{offset} < b.length \\
\text{Undefined/exception} & \text{if} \ \text{offset} \geq b.length \\
0 & \text{otherwise}
\end{cases}
\]

Listing A.121

The loop iterates through two arrays. However, there are no checks if the index is out of boundary, which should be there.
Upper bound:
\( \max(\text{result.length}, \text{m_key.length}) \)
Ranking function:
\[
\begin{cases}
\text{Undefined/exception} & \text{if} \ \text{index} < 0 \\
\text{Undefined/exception} & \text{if} \ \text{ri} < 0 \\
\max(\text{rl} - \text{ri}, \text{ml} - \text{index}) & \text{if} \ \text{index} > 0 \land \text{index} < \text{ml} \land \text{rl} \geq \text{ml} \land \text{ri} < \text{rl} \\
\text{Undefined/exception} & \text{if} \ \text{ri} \geq \text{rl} \\
\text{Undefined/exception} & \text{if} \ \text{index} \geq \text{ml} \\
0 & \text{otherwise}
\end{cases}
\]
where \( ri = rindex, \text{rl} = \text{result.length} \) and \( \text{ml} = \text{m_key.length} \).

Listing A.122

The loop is bounded by \( \text{strength} \).
Upper bound:
\( \text{strength} \)
Ranking function:
\[
\begin{cases}
\text{Undefined/exception} & \text{if} \ \text{strength} < 0 \\
\text{strength} & \text{if} \ \text{strength} > 0 \\
0 & \text{otherwise}
\end{cases}
\]

Listing A.123

The loop needs to be divided in pieces.
In the following: \( m0 = m_{\text{uitlBuffer}}[0], m1 = m_{\text{uitlBuffer}}[1], m2 = m_{\text{uitlBuffer}}[2], A = (cei << 1) < m0, B = cei < m1 \) and \( C = cei < m2 \).
4.2. DETAILED FEASIBILITY STUDY

Upper bound:
\[ \max(0, \max([m_0 - \frac{cei}{2}], \max(m_1 - cei, m_2 - cei))) \]

Ranking function:
\[
\begin{cases}
[m_0 - \frac{cei}{2}] & \text{if } A \land \neg B \land \neg C \\
m_1 - cei & \text{if } \neg A \land B \land \neg C \\
m_2 - cei & \text{if } \neg A \land \neg B \land C \\
\max([m_0 - \frac{cei}{2}], m_1 - cei) & \text{if } A \land B \land \neg C \\
\max([m_0 - \frac{cei}{2}], m_2 - cei) & \text{if } A \land \neg B \land C \\
\max(m_1 - cei, m_2 - cei) & \text{if } \neg A \land B \land C \\
\max([m_0 - \frac{cei}{2}], \max(m_1 - cei, m_2 - cei)) & \text{if } A \land B \land C \\
0 & \text{otherwise}
\end{cases}
\]

Listing A.124

The loop iterates until \texttt{m_optionarg} is greater or equal to \texttt{optionend}.

Upper bound:
\[ \max(0, \texttt{optionend} - \texttt{m_optionarg}) \]

Ranking function:
\[
\begin{cases}
\texttt{optionend} - \texttt{m_optionarg} & \text{if } \texttt{m_optionarg} < \texttt{optionend} \\
0 & \text{otherwise}
\end{cases}
\]

Listing A.125

No upper bound/ranking function can be defined, since it depends on objects: the loop condition contains two booleans, which are set depending on runtime values.

Listing A.126

The loop is bounded by the number of items in the list \texttt{changedItem.parent.children}.

Upper bound:
\[ \texttt{changedItem.parent.children.size()} \]

Listing A.127

The loop iterates over the array \texttt{providers} starting with index \( i = 0 \).

Upper bound:
\[ \texttt{providers.length} \]

Ranking function:
\[
\begin{cases}
\texttt{providers.length} - i & \text{if } \texttt{providers.length} > 0 \\
0 & \text{otherwise}
\end{cases}
\]

Listing A.128

No upper bound/ranking function can be defined, since the variables in the loop condition are assigned by objects instead of updated, which makes the variables unpredictable.

Listing A.129

The loop is similar to the following:
In the following \( cC = columnCount \) and \( c = column \).

**Upper bound:**
\( columnCount \)

**Ranking function:**
\[
\begin{cases} 
1 & \text{if } cC = 0 \\
 cC - c & \text{if } cC > 0 \land c < cC \\
0 & \text{otherwise}
\end{cases}
\]

**Listing A.130**

See Listing A.129

**Listing A.131**

See Listing A.129

**Listing A.132**

The loop is similar to the following:

**Listing 4.14: Similar version of Listing A.132**

```java
int i = 0;
do{
    i ++;
} while ( i < count )
```

**Upper bound:**
\( columnCount \)

**Ranking function:**
\[
\begin{cases} 
1 & \text{if } count = 0 \\
 count - i & \text{if } count > 0 \land i < count \\
0 & \text{otherwise}
\end{cases}
\]

**Listing A.133**

The loop iterates over the array \( toSort \) starting with index \( i = 0 \).

**Upper bound:**
\( toSort.length \)

**Ranking function:**
\[
\begin{cases} 
toSort.length - i & \text{if } toSort.length > 0 \\
0 & \text{otherwise}
\end{cases}
\]
4.3 Categories of Loops with Disjunctive Guards

The loops analysed in the previous section can be sorted into different categories:

- **Conjunctive Normal Form Elimination**: A.2, A.15, A.76, A.77, A.80, A.112, A.122, A.124
- **Condition Jumping check error**: A.25, A.32, A.33, A.74, A.123, A.129, A.130, A.131, A.132
- **Iterating through data structures using an index variable**:
  - Data structure through method calls: A.11, A.13
- **Iterating through data structures without using an index**:
  - Data structure through method calls: A.26, A.28, A.31, A.95
  - Using iterators: A.9, A.63, A.84, A.103, A.104
- **Plain iterators**: A.10, A.116
- **Other iterating classes**: A.81, A.96
- **Break**: A.55, A.56
- **Time**: A.5, A.14
In the following sections those categories are defined and will be presented with an explanation of a possible solution and/or future work/research on the topic.

4.3.1 Conjunctive Normal Form Elimination

The current method (as presented in Chapter 2) only deals with numerical loop conditions. However, many loop conditions contain non-numerical elements, which cannot be handled by the original method, e.g. in the loop in Listing A.2.

In this loop a character c is used. The use of a character is not the problem here, since characters can be regarded as a numerical value, e.g. using its ASCII code. The problem is that the value of character c is unpredictable given the code, since the next value for c depends on the character in String string at position i, but the value of string can only be determined at runtime.

Although c cannot be used to determine a ranking function, a part of the condition does not contain the variable c: ++i < length. Moreover this is connected to the other parts of the condition using a conjunction. This means that to determine an upper bound on the number of iterations, either side of the conjunction can be used, since if either the left or the right part is false the loop is exited. Therefore, for loop bound analysis the loop can be transformed to:

\[
\begin{align*}
\text{while } & (++i < \text{length}) \\
&;
\end{align*}
\]

This results in a ranking function of:

\[
\begin{cases}
  \text{length} - (i - 1) & \text{if } (i - 1) < \text{length} \\
  0 & \text{otherwise}
\end{cases}
\]

General solution

In general if the loop condition contains parts, which cannot be used for determining a ranking function, the following algorithm can be used:

1. Convert the loop condition to Conjunctive Normal Form (CNF).

2. Check which parts of the condition can be used for the loop bound analysis.
   For example by using a syntactical analysis to check if the which parts are of the following form:

\[
\bigvee_{i=1}^{n} (e_i \ b \ e_{ri})
\]

where \( b \in \{<,>,=,\neq,\leq,\geq\} \).

3. For the analysis, remove any parts from the loop condition that cannot be used.
4.3.2 Condition Jumping check error

In Section 3.3.1 a method is given to check for condition jumping in a loop. However, during the detailed study, this method showed insufficient in some cases. The problem is best to be shown with an example. The following is a slightly changed version of the loop in Listing 3.2.

Listing 4.16: Multivariate condition jumping

```java
while ((i > 0 && i < (var - 1)) || i > (var + 1)){
  if (i > (var + 1))
    i --;
  else
    i += 4;
}
```

In the original version the variable `var` was a constant with the value 21, which leads to condition jumping when `i = 19`. Therefore, the SMT solver will (eventually) find the model `(i = 19, var = 21)` as a case of condition jumping. However, it will also find `(i = 1, var = 3), (i = 2, var = 4), (i = 3, var = 5), (i = 4, var = 6), (i = 5, var = 7), (i = 6, var = 8)` etc. In theory the list of condition jumping cases would be endless, since for any value of `var > 2` `i = var - 2` will result in condition jumping. However, in reality this would be bounded by the maximum integer value for the SMT solver.

The problem is that `var` is used as a constant in the loop, since it does not change during the loop. However, when checking for condition jumping it is regarded as a normal variable, so all the different cases will be summed up. To solve this problem another method should be used, since condition jumping occurs when `i = var - 2`, which contains the symbol `var` and SMT solvers cannot handle symbolic variables by itself.

A possible solution could be to use the SMT solver in combination with polynomial interpolation as described in Section 2.1.2 i.e. instead of constant condition jumping cases, use a polynomial to define a set of condition jumping cases. Let there be the possibility to mark a variable as constant, for example `var` in the example above and use that as a variable in the polynomial. Then try to interpolate a polynomial for condition jumping cases of a chosen degree. For the example above a polynomial of degree one (with one variable `var`) suffices, i.e. two nodes are needed. Then use the SMT solver to get two models. In the case of the example above the SMT solvers returns for example: `(i = 1, var = 3)` and `(i = 2, var = 4). Then do polynomial interpolation to find the following polynomial: `a \cdot var + b = i`.

\[
\begin{align*}
3a + b &= 1 \\
4a + b &= 2
\end{align*}
\]

Then `a = 1` and `b = -2`, i.e. `i = var - 2`. 
4.3.3 Iterating through data structures with size field or method

This category includes loops that iterate over elements of a data structure. This can be any data structure, but the data structure needs a function or accessible field which represents its length, e.g. String.length(). This category is split into two subcategories, depending on the use of an index variable, since an index is used to derive a ranking function from the upper bound, i.e. the size of the data structure.

Since these loops iterate over a data structure, its size will be an upper bound on the number of iterations. Using for example a syntactical analysis method to check whether a data structure has a field/method which defines its size, e.g. check if there is an accessible field size or length or a method length() etc.

Instead of using a syntactical analysis method, an annotation can be given to the data structure, which points to the field or method which contains the size information of the data structure. This solution works better than a syntactical analysis, since it is possible to store the size of a data structure in a field which is badly named, i.e. the name of the variable does not suggest a size value. However, this will move the responsibility of finding a size field or method from the analysis method (or tool) to the author of the code.

In the detailed study a difference is made between local data structures and iterating through a method call. For the first an analysis can be done as described above. However, for the latter the analysis becomes more complicated, since the data structure is hidden in a method call, which returns the next value. Therefore, a more complex syntactical analysis is needed, which searches through method calls for the iterating data structure and then use the method above to find the size of that data structure. However, in this case it is also possible to give the responsibility of finding the correct field or method to the author of the code, by using an annotation to the loop.

Another distinction made in the evaluation is iteration through a data structure using an iterator. This subcategory contains the loops using an iterator, for which the iterator is created using a data structure. It is not possible to get the size of standard iterators, since they mainly do not contain size information. However, using the declaration of the iterator, it is possible to derive the data structure through which it iterates. Then use the method above to find the size of that data structure.

With an index variable

When an index variable is present, the derivation of a ranking function is trivial: just subtract the index from the upper bound given by the size of the data structure.

Without an index variable

To obtain a ranking function, when there is no index variable, it is possible to create a new variable which will initial be 0 and increases with 1 every iteration, i.e. this variable simulates an index.

This can be done in two ways:
• add an extra variable which simulates the index. Define a variable before the loop and set it to 0 and increase its value every iteration with 1. For example in Listing A.9:

Listing 4.17: Index with variables

```java
1 int index = 0;
2 Iterator<RepositoryDescriptor> iter = destinationRepos.iterator();
3 while (iter.hasNext() && (artifactRepoDescriptor == null || metadataRepoDescriptor == null)) {
4   RepositoryDescriptor repo = iter.next();
5   if (repo.isArtifact() && artifactRepoDescriptor == null) artifactRepoDescriptor = repo;
6   if (repo.isMetadata() && metadataRepoDescriptor == null) metadataRepoDescriptor = repo;
7   index++;
8 }
```

A ranking function is then:

\[
\begin{cases}
    \text{destinationRepos.size()} - \text{index} & \text{if} \neg \text{destinationRepos.empty()} \\
    0 & \text{otherwise}
\end{cases}
\]

• use Java Modeling Language (JML) in which a ranking function can be defined and use ghost variables to simulate an index. For example in Listing A.9:

Listing 4.18: Index with JML annotations

```java
1 //@ ghost int index;
2 //@ set index = 0;
3 //@ decreases (!destinationRepos.empty()) ? destinationRepos.size() - index : 0;
4 while (iter.hasNext() && (artifactRepoDescriptor == null || metadataRepoDescriptor == null)) {
5   RepositoryDescriptor repo = iter.next();
6   if (repo.isArtifact() && artifactRepoDescriptor == null) artifactRepoDescriptor = repo;
7   if (repo.isMetadata() && metadataRepoDescriptor == null) metadataRepoDescriptor = repo;
8   // @ set index = index + 1;
9 }
```

Note that the solutions above work when the iteration through a data structure visits each element only once. If this is not true the upper bound may
be too small, for example when using both previous() and next() functions in iterators. Using syntactical analysis, it is possible to detect this. Then extra pieces can be added, for which an element will be visited more than once.

### 4.3.4 Iterating through other data structures

This category contains data structures for which no method or field defining the size of the data structure is available. This category is derived in four subcategories.

**Plain iterators**

This subcategory contains loops using an iterator for which the type of the data structure it iterates is not known. For example in Listing A.10, iteratorIterator is set in the constructor of the class. However, it is set by an argument of that constructor, for which the only type known is Iterator, i.e. the original data structure could be of any type that can return an Iterator.

The problem with plain iterators is that they do not contain size information, i.e. no upper bound is present. The only way to derive the size of an iterator is by counting the number of elements by iterating. A way to get an upper bound is to extend the iterator with an extra method, which saves the current location in the data structure, iterates to the end of the data structures, while counting the elements, and then returns to the saved location. Then this function can be used to in the upper bound. For example, if a variable iter is of the type Iterator, which has been extended with a method getSize(), iter.getSize() would be the upper bound. To obtain a ranking function an index variable needs to be added as described in the previous section.

**Streams**

This subcategory contains streams. However, it is not possible to derive an upper bound on the length of a stream. Since the end of a stream is only known when it is reached and it is not possible to go back. Moreover a stream can grow in time, e.g. when using a stream for user import through the keyboard, it is unpredictable when new input will arrive. Therefore it is not possible to calculate the length of the stream beforehand.

**Other data structures**

The last subcategory contains user defined classes which can be iterated. In this case, the author of the classes is responsible for creating a size method or field. However, the following generic approach could be used as well.

**Generic approach**

A generic approach to find the size of a data structure, e.g. linked lists or trees, is to extend the class of the data structure with an extra method. This method counts the number of elements in the data structure reachable from this element (including this element). For example in the case of the following class:
4.3. CATEGORIES OF LOOPS WITH DISJUNCTIVE GUARDS

Listing 4.19: DataStructure Class

```java
public class DataStructure {
    private DataStructure left;
    private DataStructure right;
}
```

This data structure is a tree. The following method calculates the number of elements reachable from an element:

Listing 4.20: Size functions

```java
public int getSize() {
    return 1 + left.getSize() + right.getSize();
}
```

In general, the method to calculate the size, or the number of elements reachable from a given node, is to calculate the recursive sum of 1 (for this element) and the size (or number of elements reachable) from the nodes which this elements points to, i.e. field pointers of the same type as the current element (in the example represented by variables left and right).

However, since this counts all elements, this can result in an overestimation of the upper bound of the number of iterations. For example in a tree structure, iterating through the nodes is mostly a path from root to leaf. Therefore, in such cases the depth of the data structure is more useful than the number of elements. This can be done using the following length function:

Listing 4.21: Max. size functions

```java
public int getSize() {
    return 1 + Math.max(left.getSize(), right.getSize());
}
```

The Tree examples in the detailed feasibility study started at a node in the tree and would iterate to the root. The previous method would work, if the getSize() function would be performed on the root. However, the following would be the applicable for the current node and would be the most precise:

Listing 4.22: Tree

```java
public class DataStructure {
    private DataStructure parent;
}
```

Listing 4.23: Tree size

```java
public int getSize() {
    return 1 + parent.getSize();
}
```

A problem arises when cyclic data structures are used, i.e. if an element of a data structure can be reached more than once when iterating through the
data structure. The solutions presented above will not terminate in the case of cyclic data structures. A solution would be to save which elements already are counted, so they can be skipped. However, in most cases cyclic data structures are not desirable and are likely to be a bug.

4.3.5 Breaks

This section contains loops, which make use of `breaks` in the loop body, e.g. Listing A.55. The loop condition contains two statements, which are non-numerical, i.e. this cannot be used to determine a ranking function. However, since the loop contains a `break` statement, a possibility to derive a ranking function is to disregard the loop condition and try to make use of the breaking inside the loop, i.e. change the loop condition to `true`:

```
Listing 4.24: Replace loop guard
1 while (true) {
2     if (++index1 >= length1) break end;
3 }
```

This simplification can be made, since it will only overestimate the upper bound. However, this kind of loop can be derived using a program transformation:

```
Listing 4.25: Program transformation
1 do {
2 } while(++index1 < length1);
```

This loop can be derived using the basic method presented in Section 2.1, which results in the following:

\[
\begin{cases}
    \text{length1} - \text{index1} & \text{if } \text{index1} < \text{length1} \\
    1 & \text{otherwise}
\end{cases}
\]

4.3.6 Time

It is not possible to define an upper bound on loops that depend on time, since time is an uncontrollable external factor.

4.3.7 Threads

The same holds for loops that depend on the execution of threads, since waiting for the execution of other threads is unpredictable.

4.3.8 Unpredictable updates

This category contains loops, which use numerical variables, but those are updated in an unpredictable way. For example, an integer is updated using the value of an object which can only be determined at runtime or a variable is only updated given certain constraints, which can only be determined given their runtime values. The first is the case in Listing A.6 where integer `len` is updated using a switch. The second is the case in Listing A.24 where `ptr` is
4.4 Summary of the proposed improvements/extensions

When parts of the loop condition are not usable for Loop Bound Analysis try to simplify the loop condition by using Conjunctive Normal Form. If the loop condition in CNF consists of different parts, check if at least one of the parts is useful. If so, remove all the parts that are not usable and simplify the loop condition in such a way that loop bound analysis is possible.

When using multiple variables when checking for condition jumping, a problem could occur, since it is possible that the number of condition jumping cases is infinite in theory. However, instead of checking for all distinct condition jumping cases, use polynomial interpolation to infer a polynomial which represents the set of condition jumping cases.

When iterating over a data structure, the upper bound on the number of iterations is the size of that data structure. However, this is not defined for all data structures. Therefore, it is sometimes needed to extend the code with a method or field containing the size of a data structure. However, there are also data structures for which no upper bound can be given, e.g., streams. When the upper bound is inferred, transform it to a ranking function by using an index, which increases with each iteration.

When the loop condition is not usable and the CNF method cannot be applied, but the loop can be exited by a break-statement in the loop. The following program transformation can be applied: create a do-while loop, where the loop condition contains the branch-condition of the break-statement.

4.5 Feasibility

In the detailed feasibility study (presented in Section 4.2), none of the loops were applicable to the original methods described in Chapters 2 and 3. Therefore, a manual derivation of ranking function was done for 134 loops with disjunctive guards. This was possible for 84 of those loops. For all those loops a solution is proposed, raising the number of loops with disjunctive guards for which a ranking function can be inferred, from 0 to 84, i.e., from 0% to 62.7%. Since the number of loops with disjunctive guards is 0.26% of all loops, the gain for the Loop Bound Analysis method is $0.26\% \times 62.7\% \approx 0.16\%$ of all loops. Note that this only considers the use of the proposed solutions for loops with disjunctive guards. However, some of the solutions are also applicable to loops without
disjunctive guards, e.g. loops that iterate over data structures, resulting in a bigger gain.

The solution with the maximum gain in the detailed study was the solution considering data structures with a size field or method. This solution will raise the number of applicable loops with disjunctive guards to 52, which is 38.8% of the total number of loops considered in the detailed study and 61.9% of the maximum number of loops in the detailed study for which a ranking function can be inferred. In other words, more than half of the gain possible by analysing loops with disjunctive guards can be obtained by the proposed solution considering data structures with a size field or method. Note that the total gain will even be more.

The answer to the question if it is feasible to do Loop Bound Analysis on loops with disjunctive guard, is: yes, it is feasible. However, it is not possible in all the cases, since it is not possible to define an upper bound on the number of iterations in certain cases.
Chapter 5

Future work

This chapter contains other future work as an addition to the future work presented in Section 4.3.

5.1 Improved branch-splitting

In section 2.2.2 branch-splitting is explained. It is used to handle if-statements in the loop body. Consider the following loop:

Listing 5.1: Branch-splitting example

```plaintext
while (i > 0) {
    if (i > 100)
        i --;
    else
        i = 10;
}
```

Using the original method branch-splitting a ranking function would be $\lceil \frac{i}{10} \rceil$. However, this is an overestimation for any $i > 100$. More precise would be:

$$
\begin{cases}
\lceil \frac{i}{10} \rceil & \text{if } i > 0 \land i \leq 100 \\
(i - 100) + 10 & \text{if } i > 100 \\
0 & \text{otherwise}
\end{cases}
$$

This can be done by altering the node generation instead of splitting the if-statements in the loop. Divide the loop into different pieces, for each branch one and generate nodes for which the initial state satisfies the loop condition and the branch-condition. However, this method generates some problems, e.g. for the loop in Listing 5.2. The upper bound of the loop is shown in Figure 5.1.

Listing 5.2: Loop with if-statements as given in [27]

```plaintext
while (i > 0){
    if (i > 100)
        i -= 10;
}
```

69
CHAPTER 5. FUTURE WORK

4     else
5      i--;
5.2. LOOP CONTEXT

If this loop is extracted from its context, i.e. remove the declaration of Object o, there will be compilation errors inside the loop, since the object used inside the loop, o, is not declared then. A simple solution is to remove the statement with object o inside the loop as well. However, this may lead to incorrect ranking functions. This is not the case for the example in Listing 5.3. However, it is possible that the loop variable is altered by an object declared outside the loop (as shown in the examples in Listings 5.4 and 5.5), so the number of iterations is dependent on the object and can for that reason not be removed outside the loop, since it will lead to a wrong ranking function. However, it also does not have the context information it needs, since the object is declared outside the loop and it may be dependent on values that are only available at runtime.

Listing 5.4: Loop counter incremented with object

```java
public void Example(int i) {
    Object o = new Object();
    while (i < 10) {
        o.toString();
        i++;
    }
}
```

Listing 5.5: Incrementor class

```java
public class Incrementor {
    int incValue;

    public Incrementor(int incValue) {
        this.incValue = incValue;
    }

    public int inc(int i) {
        return i + incValue;
    }
}
```

To solve this, two solutions have been devised. The first solution is adding Java Modelling Language (JML) contracts to the methods of the object, which contain information about what the function does to the loop variable, or more generally give information about what the method returns, since numerical loop variables can only be changed through an assignment.

The second solution is creating a new testing environment, which will not extract the loop from its context, but will execute the program in such a way
that the loop will be entered and such that the runtime context information is available.
Chapter 6

Conclusion

In this thesis, the feasibility of Loop Bound Analysis for loops with disjunctive guards has been researched. This gave the following results.

In the global evaluation the number of loops with disjunctive guards is less than 0.26% of all loops, i.e. loops with disjunctive guards are uncommon. Therefore, by adding method to analyse loops with disjunctive guards, result in a small gain in the number of applicable loops.

A solution to the condition jumping challenge (as described in Chapter 3) has been implemented in the ResAna tool. However, none of the loops with disjunctive guards in the global evaluation (Section 4.1) were applicable to the methods described in Chapters 2 and 3. In other words, loops with guards over disjunctions and conjunctions of (in)equalities are even more uncommon.

In the detailed feasibility study, a ranking function and/or upper bound on the number of iterations could be assigned to 62.7% of the loops with disjunctive guards. The other 37.3% of the loops with disjunctive guards were dependent of time, used threads, had unpredictable variable updates or involved non-numerical objects.

Some extensions to the Loop Bound Analysis methods by Shkaravska et al. [27] and Kersten and Van Eekelen [21] have been proposed. Those extensions involved: 1) Conjunctive Normal Form Elimination, which removes unusable parts of the loop guards, 2) polynomial interpolation on condition jumping cases to handle more variables, 3) counting the number of elements of a data structure and 4) do program transformation, in order to use break-statements as exit condition.

The overall gain when adding methods to analyse loops with disjunctive guards is approximately 0.16% of all loops. However, the methods used for loops with disjunctive guards are not exclusive to loops with disjunctive guard, i.e. other loops could be applicable as well. Therefore, the overall gain will be bigger, when considering the gain of loops without disjunctive guards as well. For example, the proposed solution to iterating over data structures is applicable to any loop which involves iterating over a data structure.

Therefore, Loop Bound Analysis on loops with disjunctive guards is feasible for certain cases. Adding methods to analyse loops with disjunctive guard, results in a small gain of analysable loops.
Bibliography


Appendix A

Loops used in the detailed feasibility study

In this chapter the loops that are used in the detailed evaluation (Section 4.2) are listed. Some loops contain extra code for better understanding. Code directly above the loop is code in the same method as the loop, Code separated with white space is from the same class as the loop and if annotated the code is from a superclass.

A.1 While-loops

Listing A.1: ./jdimodelsrc/org/eclipse/jdi/internal/SourceDebugExtension-Parser.java

```java
private char[] fSmap;
private int fPointer;
private char nextChar() {
    if (++fPointer == fSmap.length) {
        fEOF = true;
        return '\000';
    }
    fChar = fSmap[fPointer];
    return fChar;
}
while (fChar == ' ' || fChar == '	') {
    nextChar();
}
```

Listing A.2: ./jdimodelsrc/org/eclipse/jdi/internal/SourceDebugExtension-Parser.java

```java
int i = -1, length = string.length();
char c;
while (i < length && ((c = string.charAt(i)) == ' ' || c == '	')) {
    ...
}
```

Listing A.3: ./jdimodelsrc/org/eclipse/jdi/internal/SourceDebugExtension-Parser.java

```java
```
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

Listing A.4: ./org/eclipse/jetty/http/HttpParser.java

```java
int i = 0;
char c = lexem[0];
while (c == '\n' || c == '\t') {
    c = lexem[++i];
}
```

Listing A.5: ./org/eclipse/jetty/http/AbstractGenerator.java

```java
while (_contentView.length() == 0 && (!_isState(HttpParser.STATE_END) || _isState(HttpParser.STATESEEKING_EOF)) && _endp!=null && _endp.isOpen())
{
    if (!_endp.isBlocking())
    {
        if (parseNext()>0)
        {
            if (!_endp.blockReadable(maxIdleTime))
            {
                _endp.close();
                throw new EOFException("timeout");
            }
        }
        parseNext();
    }
}
```

Listing A.6: ./org/eclipse/jetty/http/HttpGenerator.java

```java
do {
    last_flush=to_flush;
    switch (to_flush)
    {
        case 7: throw new IllegalStateException(); // should never happen;
        case 6: len = _endp.flush(_header, _buffer, null); break;
        case 5: len = _endp.flush(_header, _content, null); break;
        case 4: len = _endp.flush(_header); break;
        case 3: len = _endp.flush(_buffer, _content, null); break;
        case 2: len = _endp.flush(_buffer); break;
        case 1: len = _endp.flush(_content); break;
        case 0: len=0;
        // Nothing more we can write now.
    }
```
A.1. WHILE- LOOPS

if (_header != null)
    _header.clear();

_bypass = false;
_bufferChunked = false;

if (_buffer != null)
    _buffer.clear();

if (_contentLength == HttpTokens.CHUNKEDCONTENT)
{
    // reserve some space for the chunk header
    _buffer.setPutIndex(CHUNKSPACE);
    _buffer.setGetIndex(CHUNKSPACE);

    // Special case handling for small left over buffer from
    // an addContent that caused a buffer flush.
    if (_content != null & & _content.length() < _buffer.space()
        & & _state != STATE_FLUSHING)
    {
        _buffer.put(_content);
        _content.clear();
        _content=null;
    }
}

// Are we completely finished for now?
if (!needCRLF & & !needEOC & & (_content==null
 | | _content.length()==0))
{
    if (_state == STATE_FLUSHING)
        _state = STATE_END;

    if (_state==STATE_END & & _persistent !=
        null & & _persistent & & _status
        !=100 & & _method==null)
        _endp.shutdownOutput();
    else
        // Try to prepare more to write.
        prepareBuffers();
}

if (len > 0)
    total+=len;

to_flush = flushMask();

// loop while progress is being made (OR we have prepared some buffers
// that might make progress)
while (len>0 || (to_flush!=0 & & last_flush==0));
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

```java
long rl = _weakRandom
? (hashCode()) ^ Runtime.getRuntime().freeMemory() ^ random.nextInt()
: random.nextInt();
if (rl < 0)
rl = -rl;
if (workerName != null)
id = workerName + id;
```

Listing A.8: ./org/eclipse/jetty/server/ResourceCache.java

```java
while (cache.size() > 0 && (_cachedFiles.get() > maxCachedFiles ||
_cachesSize.get() > maxCacheSize))
{
    // Scan the entire cache and generate an ordered list by last accessed time.
    SortedSet<Content> sorted = new TreeSet<Content>(
        new Comparator<Content>()
        {
            public int compare(Content c1, Content c2)
            {
                if (c1.lastAccessed < c2.
                    _lastAccessed)
                    return -1;
                if (c1.lastAccessed > c2.
                    _lastAccessed)
                    return 1;
                if (c1.length < c2.length)
                    return -1;
                return c1._key.compareTo(c2._key);
            }
        });
    for (Content content : cache.values())
        sorted.add(content);
    // Invalidate least recently used first
    for (Content content : sorted)
    {
        if (_cachedFiles.get() <= maxCachedFiles && _cachesSize.get() <= maxCacheSize)
            break;
        if (content == _cache.remove(content.getKey()))
            content.invalidate();
    }
}
```

Listing A.9: ./org/eclipse/equinox/p2/internal/repository/tools/AbstractApplication.java

```java
private List<RepositoryDescriptor> destinationRepos = new ArrayList<RepositoryDescriptor>();
while (iter.hasNext() && (artifactRepoDescriptor == null ||
metadataRepoDescriptor == null))
{ RepositoryDescriptor repo = iter.next();
    if (repo.isArtifact() && artifactRepoDescriptor == null)
        artifactRepoDescriptor = repo;
    if (repo.isMetadata() && metadataRepoDescriptor == null)
```
metadataRepoDescriptor = repo;

Listing A.10: ./org/eclipse/equinox/internal/p2/metadata/expression/CompoundIterator.java

```java
private final Iterator<? extends Object> iterator;
public CompoundIterator(Iterator<? extends Object> iterator) {
  this.iterator = iterator;
  while (currentIterator == null || !currentIterator.hasNext()) {
    if (!iterator.hasNext())
      return false;
    Object nextIterator = iterator.next();
    currentIterator = (nextIterator instanceof Iterator<?>) ? (Iterator<T>) nextIterator : RepeatableIterator.<T>create(nextIterator);
  }
```

Listing A.11: ./org/eclipse/equinox/internal/p2/metadata/expression/parser/QLParser.java

```java
// Defined in superclass ExpressionParser:
protected String expression;
protected int tokenPos;
protected int currentToken;
protected void nextToken() {
  tokenValue = null;
  int top = expression.length();
  char c = 0;
  while (tokenPos < top) {
    c = expression.charAt(tokenPos);
    if (!Character.isWhitespace(c))
      break;
    ++tokenPos;
  }
  if (tokenPos >= top) {
    lastTokenPos = top;
    currentToken = TOKEN_END;
    return;
  }
  lastTokenPos = tokenPos;
  switch (c) {
    case '|':
      if (tokenPos + 1 < top && expression.charAt(tokenPos + 1) == '!') {
        tokenValue = OPERATOR_OR;
        currentToken = TOKEN_OR;
        tokenPos += 2;
        break;
      } else {
        currentToken = TOKEN_PIPE;
        ++tokenPos;
      } break;
    case '&':
      if (tokenPos + 1 < top && expression.charAt(tokenPos + 1) == '&') {
        tokenValue = OPERATOR_AND;
        currentToken = TOKEN_AND;
        tokenPos += 2;
      } else {
        currentToken = TOKEN_ERROR;
        break;
      }
    case '=':
      if (tokenPos + 1 < top && expression.charAt(tokenPos + 1) == '=') {
```

APPENDIX A. LOOPS USED IN THE DETAILED STUDY

```c

45     tokenValue = OPERATOR_EQUALS;
46     currentToken = TOKEN_EQUAL;
47     tokenPos += 2;
48 } else 
49     currentToken = TOKEN_ERROR;
50 break;
51
52 case '!' : 
53 if (tokenPos + 1 < top && expression.charAt(
54     tokenPos + 1) == '!' ) { 
55     tokenValue = OPERATOR_NOT_EQUALS;
56     currentToken = TOKEN_NOT_EQUAL;
57     tokenPos += 2;
58 } else { 
59     currentToken = TOKEN_NOT;
60     ++tokenPos;
61 }
62 break;
63
64 case '^' : 
65 if (tokenPos + 1 < top && expression.charAt(
66     tokenPos + 1) == '^' ) { 
67     tokenValue = OPERATOR_MATCHES;
68     currentToken = TOKEN_MATCHES;
69     tokenPos += 2;
70 } else 
71     currentToken = TOKEN_ERROR;
72 break;
73
74 case '>' : 
75 if (tokenPos + 1 < top && expression.charAt(
76     tokenPos + 1) == '>' ) { 
77     tokenValue = OPERATOR_GT_EQUAL;
78     currentToken = TOKEN_GREATER_EQUAL;
79     tokenPos += 2;
80 } else { 
81     currentToken = TOKEN_GREATER;
82     ++tokenPos;
83 }
84 break;
85
86 case '<' : 
87 if (tokenPos + 1 < top && expression.charAt(
88     tokenPos + 1) == '<' ) { 
89     tokenValue = OPERATOR_LT_EQUAL;
90     currentToken = TOKEN_LESS_EQUAL;
91     tokenPos += 2;
92 } else { 
93     currentToken = TOKEN_LESS;
94     ++tokenPos;
95 }
96 break;
97
98 case '?' : 
99     currentToken = TOKEN_IF;
100     ++tokenPos;
101     break;
102
103 case ':' : 
104     currentToken = TOKEN_ELSE;
105     ++tokenPos;
106     break;
107
108 case '.' : 
109     currentToken = TOKEN_DOT;
110     ++tokenPos;
111     break;
112
113 case '$' : 
114     currentToken = TOKEN_DOLLAR;
115     ++tokenPos;
116     break;
117
118 case ']' : 
```

---

**Note:** The above text is a direct transcription from the image and includes all visible content. Please ensure this is the correct context before proceeding with any further action.
A.1. WHILE-LOOPS

```java
A.1. WHILE-LOOPS
85

115 currentToken = TOKEN LC;
116 ++tokenPos;
117 break;
118
119 case '}' : currentToken = TOKEN RC;
120 ++tokenPos;
121 break;
122
123 case '(' : currentToken = TOKEN LP;
124 ++tokenPos;
125 break;
126
127 case ')' : currentToken = TOKEN RP;
128 ++tokenPos;
129 break;
130
131 case '[' : currentToken = TOKEN LB;
132 ++tokenPos;
133 break;
134
135 case ']' : currentToken = TOKEN RB;
136 ++tokenPos;
137 break;
138
139 case ',' : currentToken = TOKEN COMMA;
140 ++tokenPos;
141 break;
142
143 case '"' :
144 case '\': parseDelimitedString(c);
145 break;
146
147 case '/' : parseDelimitedString(c);
148 if (currentToken == TOKEN LITERAL)
149 tokenValue = SimplePattern.compile((String) tokenValue);
150 break;
151
152 default :
153 if (Character.isDigit(c)) {
154 int start = tokenPos++;
155 while (tokenPos < top && Character.isDigit(expression.charAt(tokenPos)))
156 ++tokenPos;
157 tokenValue = Integer.valueOf(expression.substring(start, tokenPos));
158 currentToken = TOKEN LITERAL;
159 break;
160 }
161 if (Character.isJavaIdentifierStart(c)) {
162 int start = tokenPos++;
163 while (tokenPos < top && Character.isJavaIdentifierPart(expression.charAt(tokenPos)))
164 ++tokenPos;
165 String word = expression.substring(start, tokenPos);
166 Integer token = keywordToTokenMap().get(word);
167 if (token == null)
168 currentToken = TOKEN IDENTIFIER;
169 else
170 currentToken = token.intValue();
171 tokenValue = word;
172 break;
173 }
```
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

Listing A.12: ./org/eclipse/equinox/internal/p2/metadata/expression/parser/LDAPFilterParser.java

```java
}  
throw syntaxError();
}

while (currentToken == TOKEN_DOT || currentToken == TOKEN_LB) {
    int savePos = tokenPos;
    int saveToken = currentToken;
    Object saveTokenValue = tokenValue;
    nextToken();
    if (saveToken == TOKEN_DOT) {
        switch (currentToken) {
            case TOKEN_IDENTIFIER :
                name = (String) tokenValue;
                nextToken();
                if (currentToken == TOKEN_LP) {
                    IExpression[] callArgs = parseArray();
                    nextToken();
                    expr = factory.memberCall(expr, name, callArgs);
                } else {
                    expr = factory.member(expr, name);
                    break;
                }
            default :
                tokenPos = savePos;
                currentToken = saveToken;
                tokenValue = saveTokenValue;
                return expr;
        }
        }  
    default :
        tokenPos = savePos;
        currentToken = saveToken;
        tokenValue = saveTokenValue;
        return expr;
    }
    else {
        IExpression atExpr = parseMember();
        assertToken(TOKEN_RB);
        nextToken();
        expr = factory.at(expr, atExpr);
    }
}
```

Listing A.13: ./org/eclipse/equinox/internal/p2/metadata/expression/parser/ExpressionParser.java

```java
protected String expression;
protected int tokenPos;
protected int currentToken;
protected void nextToken() {
    tokenValue = null;
    int top = expression.length();
    char c = 0;
    while (tokenPos < top) {
        c = expression.charAt(tokenPos);
        if (!Character.isWhitespace(c))
            break;
        ++tokenPos;
    }
```

A.1. WHILE-LOOPS

```c
if (tokenPos >= top) {
    lastTokenPos = top;
    currentToken = TOKEN_END;
    return;
}
lastTokenPos = tokenPos;
switch (c) {
    case '|' : 
        if (tokenPos + 1 < top && expression.charAt(tokenPos + 1) == '|') {
            tokenValue = OPERATOR_OR;
            currentToken = TOKEN_OR;
            tokenPos += 2;
        } else {
            currentToken = TOKEN_PIPE;
            ++tokenPos;
        }
        break;
    case '&' :
        if (tokenPos + 1 < top && expression.charAt(tokenPos + 1) == '&') {
            tokenValue = OPERATOR_AND;
            currentToken = TOKEN_AND;
            tokenPos += 2;
        } else {
            currentToken = TOKEN_ERROR;
        }
        break;
    case '=' :
        if (tokenPos + 1 < top && expression.charAt(tokenPos + 1) == '=') {
            tokenValue = OPERATOR_EQUALS;
            currentToken = TOKEN_EQUAL;
            tokenPos += 2;
        } else {
            currentToken = TOKEN_ERROR;
        }
        break;
    case '!' :
        if (tokenPos + 1 < top && expression.charAt(tokenPos + 1) == '=') {
            tokenValue = OPERATOR_NOT_EQUALS;
            currentToken = TOKEN_NOT_EQUAL;
            tokenPos += 2;
        } else {
            currentToken = TOKEN_NOT;
            ++tokenPos;
        }
        break;
    case '˜' :
        if (tokenPos + 1 < top && expression.charAt(tokenPos + 1) == '=') {
            tokenValue = OPERATOR_MATCHES;
            currentToken = TOKEN_MATCHES;
            tokenPos += 2;
        } else {
            currentToken = TOKEN_ERROR;
        }
        break;
    case '>' :
        if (tokenPos + 1 < top && expression.charAt(tokenPos + 1) == '>=') {
            tokenValue = OPERATOR_GT_EQUAL;
            currentToken = TOKEN_GREATER_EQUAL;
            tokenPos += 2;
        } else {
            currentToken = TOKEN_GREATER;
            ++tokenPos;
        }
        break;
```
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

```
case '<' :
    if (tokenPos + 1 < top && expression.charAt(tokenPos + 1) == '=' ) {
        tokenValue = OPERATOR_LT_EQUAL;
        currentToken = TOKEN_LESS_EQUAL;
        tokenPos += 2;
    } else {
        currentToken = TOKEN_LESS;
        ++tokenPos;
    }
    break;

case '?' :
    currentToken = TOKEN_IF;
    ++tokenPos;
    break;

case ':' :
    currentToken = TOKEN_ELSE;
    ++tokenPos;
    break;

case '.' :
    currentToken = TOKEN_DOT;
    ++tokenPos;
    break;

case '$' :
    currentToken = TOKEN_DOLLAR;
    ++tokenPos;
    break;

case '{' :
    currentToken = TOKEN_LC;
    ++tokenPos;
    break;

case '}' :
    currentToken = TOKEN_RC;
    ++tokenPos;
    break;

case '(' :
    currentToken = TOKEN_LP;
    ++tokenPos;
    break;

case ')' :
    currentToken = TOKEN_RP;
    ++tokenPos;
    break;

case '[' :
    currentToken = TOKEN_LB;
    ++tokenPos;
    break;

case ']' :
    currentToken = TOKEN_RB;
    ++tokenPos;
    break;

case ',' :
    currentToken = TOKEN_COMMA;
    ++tokenPos;
    break;

case '"' :
    case '\'
        parseDelimitedString(c);
    break;

case '*' :
    parseDelimitedString(c);
```
A.1. WHILE-LOOPS

Listing A.14: ./org/eclipse/equinox/internal/app/EclipseAppHandle.java

while (!setResult && (delay > 0 || timeout == 0)) {

    if (currentToken == TOKEN_LITERAL)
        tokenValue = SimplePattern.compile((
            String) tokenValue);

    else if (Character.isDigit(c)) {
        int start = tokenPos++;
        while (tokenPos < top && Character.
            isDigit(expression.charAt(tokenPos)))
            ++tokenPos;
        tokenValue = Integer.valueOf(expression.
            substring(start, tokenPos));
        currentToken = TOKEN_LITERAL;
        break;
    }

    else if (Character.isJavaIdentifierStart(c)) {
        int start = tokenPos++;
        while (tokenPos < top && Character.
            isJavaIdentifierPart(expression.
            charAt(tokenPos)))
            ++tokenPos;
        String word = expression.substring(start,
            tokenPos);
        Integer token = keywordToTokenMap().get(
            word);
        if (token == null)
            currentToken = TOKEN_IDENTIFIER;
        else
            currentToken = token.intValue();
        tokenValue = word;
        break;
    }

    throw syntaxError();
}

while (currentToken == TOKEN_DOT || currentToken == TOKEN_RB) {
    int savePos = tokenPos;
    int saveToken = currentToken;
    Object saveTokenValue = tokenValue;
    nextToken();
    if (saveToken == TOKEN_DOT) {
        switch (currentToken) {
        case TOKEN_IDENTIFIER :
            name = (String) tokenValue;
            nextToken();
            expr = factory.member(expr, name);
            break;
        default :
            tokenPos = savePos;
            currentToken = saveToken;
            tokenValue = saveTokenValue;
            return expr;
        }
    } else {
        IExpression atExpr = parseMember();
        assertToken(TOKEN_RB);
        nextToken();
        expr = factory.at(expr, atExpr);
    }
}
wait(delay); // only wait for the specified amount of time
if (timeout > 0)
    delay -= (System.currentTimeMillis() - startTime);
}

Listing A.15: ./org/eclipse/equinox/console/common/ConsoleInputStream.java

    public synchronized int read()
    {
        while (current == null && buffer.isEmpty() && !isClosed)
        {
            try {
                wait();
            } catch (InterruptedException e) {
                return -1;
            }
        }
        if (isClosed) {
            return -1;
        }
        try {
            if (current == null) {
                current = buffer.remove(0);
                return current[pos++] & 0xFF;
            } else {
                return current[pos++] & 0xFF;
            }
        } finally {
            if (current != null) {
                if (pos == current.length) {
                    current = null;
                    pos = 0;
                }
            }
        }
    }
}

while ((pos > 0 || !buffer.isEmpty()) && readCnt < len) {
    if (i == -1) {
        return (readCnt > 0) ? readCnt : i;
    }
    b[currOff] = (byte) i;
    currOff++; readCnt++;
}

Listing A.16: ./org/eclipse/equinox/log/internal/BasicReadWriteLock.java

while (writing != null || writersWaiting != 0) {
    try {
        if (writing == Thread.currentThread())
            throw new IllegalStateException("Attempted to nest read lock inside write lock"); // NON-NLS-1$
        wait();
    } catch (InterruptedException e) {
        if (resetInterruptedState) keep waiting
        Thread.currentThread().interrupt();
    }
}

Listing A.17: ./org/eclipse/equinox/log/internal/BasicReadWriteLock.java

while (writing != null || currentReaders.size() != 0) {
    try {
A.1. WHILE-LOOPS

```java
if (writing == Thread.currentThread() || currentReaders.contains(Thread.currentThread()))
    throw new IllegalStateException("Attempted to nest write lock inside a read or write lock"); //NON-NLS-1

wait();

} catch (InterruptedException e) {
    // reset interrupted state but keep waiting
    Thread.currentThread().interrupt();
}
```

Listing A.18: ./org/eclipse/ui/internal/quickaccess/QuickAccessContents.java

```java
do {
    // will be set to false if we find a provider with remaining
    // elements
    done = true;
    for (int i = 0; i < providers.length
        && (showAllMatches || countTotal < maxCount); i++) {
        if (entries[i] == null) {
            entries[i] = new ArrayList();
            indexPerProvider[i] = 0;
        }
        int count = 0;
        QuickAccessProvider provider = providers[i];
        if (filter.length() > 0
            || provider.isAlwaysPresent()) {
            QuickAccessElement[] elements = provider
                .getElementsSorted();
            int j = indexPerProvider[i];
            while (j < elements.length
                && (showAllMatches || (count <
                        countPerProvider &&
                        countTotal < maxCount))) {
                QuickAccessElement element = elements[j];
                QuickAccessEntry entry;
                if (filter.length() == 0) {
                    entry = new QuickAccessEntry(
                        element, provider,
                        [0][0],
                        new
                        [0][0]) ;
                } else {
                    entry = null;
                }
                else {
                    entry = element.match(filter,
                        provider);
                }
                if (entry != null) {
                    entries[i].add(entry);
                    count++;
                    countTotal++;
                    if (i == 0 && entry.element ==
                        perfectMatch) {
                        perfectMatchAdded = true
                    }
                    maxCount =
                        MAX_COUNT_TOTAL;
                }
            }
    }
```
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

```java
j++;
}
indexPerProvider[i] = j;
if (j < elements.length) {
    done = false;
}

// from now on, add one element per provider
while ((showAllMatches || countTotal < maxCount) && !done);
```

Listing A.19: ./org/eclipse/ui/internal/navigator/resources/actions/ResourceMgmtActionProvider.java

```java
IStructuredSelection selection = (IStructuredSelection) getContext().getSelection();
Iterator resources = selection.iterator();
while (resources.hasNext() && !hasOpenProjects || !hasClosedProjects || hasBuilder || isProjectSelection) {
    Object next = resources.next();
    IProject project = null;
    if (next instanceof IProject) {
        project = (IProject) next;
    } else if (next instanceof IAdaptable) {
        project = (IProject) ((IAdaptable) next).getAdapter(IProject.class);
    }
    if (project == null) {
        isProjectSelection = false;
        continue;
    }
    if (project.isOpen()) {
        hasOpenProjects = true;
        if (hasBuilder && !hasBuilder(project)) {
            hasBuilder = false;
        }
    } else {
        hasClosedProjects = true;
        hasBuilder = false;
    }
}
```

Listing A.20: ./org/eclipse/ui/internal/console/ConsoleDocumentAdapter.java

```java
while (index > -1 && (line.charAt(index) == '\n' || line.charAt(index) == 'r')) {
    index--;
}
```

Listing A.21: ./org/eclipse/jdt/ui/text/folding/DefaultJavaFoldingStructureProvider.java

```java
while (terminal != ITerminalSymbols.TokenNameEOF && ! (terminal == ITerminalSymbols.TokenNameclass || terminal == ITerminalSymbols.TokenNameinterface || terminal == ITerminalSymbols.TokenNameenum || (foundComment && (terminal == ITerminalSymbols.TokenNameimport || terminal == ITerminalSymbols.TokenNamepackage)))) {
    if (terminal == ITerminalSymbols.TokenNameCOMMENT_JAVADOC || terminal == ITerminalSymbols.TokenNameCOMMENT_BLOCK || terminal == ITerminalSymbols.TokenNameCOMMENT_LINE) {
        if (foundComment)
            headerStart = scanner.getCurrentTokenStartPosition();
```
A.1. WHILE-LOOPS

Listing A.22: ./org/eclipse/jdt/core/dom/rewrite/ASTRewrite.java

```java
while (start <= currStart || end > currEnd) { // go up until a node
    node = node.getParent();
    currStart= node.getStartPosition();
    currEnd= currStart + node.getLength();
}
```

Listing A.23: ./org/eclipse/jdt/internal/formatter/comment/HTMLEntity2JavaReader.java

```java
//As defined in superclass SubstitutionTextReader:
protected int nextChar() throws IOException {
    if (this.fReadFromBuffer == (this.fBuffer.length() > 0)) {
        char ch= this.fBuffer.charAt(this.fIndex++);
        if (this.fIndex <= this.fBuffer.length()) {
            this.fBuffer.setLength(0);
            this.fIndex= 0;
        }
        return ch;
    } else {
        int ch= this.fCharAfterWhiteSpace;
        if (ch == -1) {
            ch= this.fReader.read();
        }
        if (this.fSkipWhiteSpace && ScannerHelper.isWhitespace((char)ch)) {
            do {
                ch= this.fReader.read();
            } while (ScannerHelper.isWhitespace((char)ch));
            if (ch != -1) {
                this.fCharAfterWhiteSpace= ch;
                return -1;
            }
        } else {
            this.fCharAfterWhiteSpace= -1;
        }
        return ch;
    }
}
```

Listing A.24: ./org/eclipse/jdt/internal/formatter/FormatJavadocText.java

```java
while (idx<=this.separatorsPtr || (this.htmlNodesPtr != -1 && ptr <=
    this.htmlNodesPtr)) {
    if (idx > this.separatorsPtr) {
        // last node
        FormatJavadocNode node = this.htmlNodes[ptr++];
        node.toStringDebug(buffer, source);
        return;
    }
    int end = (int) this.separators[idx] >>> 32);
    if (this.htmlNodesPtr >= 0 && ptr <= this.htmlNodesPtr &&
        end > this.htmlNodes[ptr].sourceStart) {
        FormatJavadocNode node = this.htmlNodes[ptr++];
```
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

```
Listing A.25: ./org/eclipse/jdt/internal/formatter/CodeFormatterVisitor.java

```
A.1. WHILE-LOOPS

```java
if (!method.isDefaultConstructor() && !method.isClinit()) {
    members[index++] = method;
}
if (++methodNameIndex < methodCount) { // find next method
    if any
    methodStart = (method = typeDeclaration.methods[
        methodIndex]).declarationSourceStart;
} else {
    methodStart = Integer.MAX_VALUE;
}
else {
    // next member is a type
    members[index++] = type;
    if (++typeIndex < typeCount) { // find next type if any
        typeStart = (type = typeDeclaration.memberTypes[
            typeIndex]).declarationSourceStart;
    } else {
        typeStart = Integer.MAX_VALUE;
    }
} while ((fieldIndex < fieldCount) || (typeIndex < typeCount) || (methodIndex < methodCount));
```

Listing A.26: ./org/eclipse/jdt/internal/ui/text/spelling/PropertiesFileSpellCheckIterator.java

```java
protected final String fContent;
protected String nextToken() {
    String token = null;
    fPrevious = fPredecessor;
    fStartsSentence = false;
    nextBreak();
    boolean update = false;
    if (fNext != fPrevious) {
        if (fSuccessor != BreakIterator_DONE && fContent.charAt(
            fPrevious) == IJavaDocTagConstants.JAVADOC_TAG_PREFIX) {
            nextBreak();
            if (Character.isLetter(fContent.charAt(fPrevious + 1))) {
                update = true;
                token = fContent.substring(fPrevious, fNext);
            } else {
                fPredecessor = fNext;
            }
        } else if (fSuccessor != BreakIterator_DONE && fContent.
            charAt(fPrevious) == IHtmlTagConstants.HTML_TAG_PREFIX &&
            Character.isLetter(fContent.
                charAt(fNext)) || fContent.
            charAt(fNext) == '/') {
            if (fContent.startsWith(IHtmlTagConstants.
                HTML_CLOSE_PREFIX, fPrevious)) {
                nextBreak();
            }
        } else {
           .nextBreak();
        }
    } if (fSuccessor != BreakIterator_DONE && fContent.
        charAt(fNext) == IHtmlTagConstants.
        HTML_TAG_POSTFIX) {
        nextBreak();
        if (fSuccessor != BreakIterator_DONE) {
            update = true;
        }
    }
```

// Defined in the superclass SpellCheckIterator:
protected final String fContent;
protected String nextToken() {
    String token = null;
    fPrevious = fPredecessor;
    fStartsSentence = false;
    nextBreak();
    boolean update = false;
    if (fNext != fPrevious) {
        if (fSuccessor != BreakIterator_DONE && fContent.charAt(
            fPrevious) == IJavaDocTagConstants.JAVADOC_TAG_PREFIX) {
            nextBreak();
            if (Character.isLetter(fContent.charAt(fPrevious + 1))) {
                update = true;
                token = fContent.substring(fPrevious, fNext);
            } else {
                fPredecessor = fNext;
            }
        } else if (fSuccessor != BreakIterator_DONE && fContent.
            charAt(fPrevious) == IHtmlTagConstants.HTML_TAG_PREFIX &&
            Character.isLetter(fContent.
                charAt(fNext)) || fContent.
            charAt(fNext) == '/') {
            if (fContent.startsWith(IHtmlTagConstants.
                HTML_CLOSE_PREFIX, fPrevious)) {
                nextBreak();
            }
        } else {
            nextBreak();
        }
    } if (fSuccessor != BreakIterator_DONE && fContent.
        charAt(fNext) == IHtmlTagConstants.
        HTML_TAG_POSTFIX) {
        nextBreak();
        if (fSuccessor != BreakIterator_DONE) {
            update = true;
        }
    }
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

36 token = fContent.substring(fPrevious, fNext);
37 }
38 }
39 } else if (fSuccessor != BreakIterator.DONE && fContent.charAt(fPrevious) == HTMLTagConstants.HTML_ENTITY_START && (Character.isLetter(fContent.charAt(fNext)))
40 nextBreak();
41 if (fSuccessor != BreakIterator.DONE && fContent.charAt(fNext) == HTMLTagConstants.HTML_ENTITY_END)
42 nextBreak();
43 if (isToken(fContent.substring(fPrevious, fNext), HTMLTagConstants.HTML_ENTITY_CODES))
44 skipTokens(fPrevious, HTMLTagConstants.HTML_ENTITY_END);
45 update = true;
46 } else token = fContent.substring(fPrevious, fNext);
47 }
48 else token = fContent.substring(fPrevious, fNext);
49 update = true;
50 }
51 } else if (!isWhitespace(fPrevious, fNext) && isAlphaNumeric(fPrevious, fNext)) {
52 if (isUrlToken(fPrevious))
53 skipTokens(fPrevious, WHITE_SPACE_TOKEN);
54 else if (isToken(IJavaDocTagConstants.JAVADOC_PARAM_TAGS))
55 fLastToken = null;
56 else if (isToken(IJavaDocTagConstants.JAVADOC_REFERENCE_TAGS))
57 fLastToken = null;
58 skipTokens(fPrevious, fDelimeter.charAt(0));
59 } else if (fNext - fPrevious > 1 || isSingleLetter(fPrevious) && !fIsIgnoringSingleLetters)
60 token = fContent.substring(fPrevious, fNext);
61 update = true;
62 }
63 if (update && fSentenceBreaks.size() > 0) {
64 if (fPrevious >= nextSentence()) {
65 while (fSentenceBreaks.size() > 0 && fPrevious >= nextSentence())
66 fSentenceBreaks.removeFirst();
67 ifStartsSentence = (fLastToken == null) || (token != null);
68 }
69 return token;
70 }
71 while (fSuccessor != BreakIterator.DONE && (token == null || fContent.charAt(fNext) == '&')) {
72 if (token != null) {
73 if (previous == -1)
74 previous = fPrevious;
75 String nextToken = nextToken();
76 if (nextToken != null)
A.1. WHILE-LOOPS

Listing A.27: ./org/eclipse/jdt/internal/ui/text/javadoc-/JavaDoc2HTMLTextReader.java

```java
A.1. WHILE-LOOPS

Listing A.27: ./org/eclipse/jdt/internal/ui/text/javadoc-/JavaDoc2HTMLTextReader.java

protected int nextChar() throws IOException {
    if (this.fBuffer.length() > 0) {
        char ch = this.fBuffer.charAt(this.fIndex++);
        if (this.fIndex >= this.fBuffer.length()) {
            this.fBuffer.setLength(0);
            this.fIndex = 0;
        }
        return ch;
    } else {
        int ch = this.fCharAfterWhiteSpace;
        if (ch == -1) {
            ch = this.fReader.read();
        }
        if (this.fSkipWhiteSpace && ScannerHelper.isWhitespace((char)ch)) {
            do {
                ch = this.fReader.read();
            } while (ScannerHelper.isWhiteSpace((char)ch));
            this.fCharAfterWhiteSpace = ch;
            return ' ';  // Simulate whitespace character
        } else {
            this.fCharAfterWhiteSpace = -1;
        }
    }
    return ch;
}

while (c == '.' || c != -1 && Character.isLetter((char)c)) {
    buffer.append((char)c);
    c = nextChar();
}
```

Listing A.28: ./org/eclipse/jdt/internal/ui/text/java/hover/JavaSourceHover.java

```java
// As defined in superclass SubstitutionTextReader:
protected int nextChar() throws IOException {
    this.fReadFromBuffer = (this.fBuffer.length() > 0);
    if (this.fReadFromBuffer) {
        char ch = this.fBuffer.charAt(this.fIndex++);
        if (this.fIndex >= this.fBuffer.length()) {
            this.fBuffer.setLength(0);
            this.fIndex = 0;
        }
        return ch;
    } else {
        int ch = this.fCharAfterWhiteSpace;
        if (ch == -1) {
            ch = this.fReader.read();
        }
        if (this.fSkipWhiteSpace && ScannerHelper.isWhitespace((char)ch)) {
            do {
                ch = this.fReader.read();
            } while (ScannerHelper.isWhiteSpace((char)ch));
            this.fCharAfterWhiteSpace = ch;
            return ' ';  // Simulate whitespace character
        } else {
            this.fCharAfterWhiteSpace = -1;
        }
    }
    return ch;
}

while (c == '.' || c != -1 && Character.isLetter((char)c)) {
    buffer.append((char)c);
    c = nextChar();
}
```
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

19 // a comment starts, advance to the comment end
20 ++ fOffset;
gotoCommentEnd();
22 continue;
23 } else if (next == '/') {
24 // '/'/*-comment starts, advance to the line end
25 gotoLineEnd();
26 continue;
27 }
28 return current;
29 }
30 case "\":
31 case "\":
32 if (fSkipStrings) {
33 gotoStringEnd(current);
34 continue;
35 }
36 return current;
37 }
38 return current;
39 }
40 } return EOF;
41 }
42 private int readBackwards() throws BadLocationException {
43 while (0 < fOffset) {
44 -- fOffset;
45 handleSingleLineComment();
46 char current = fDocument.getChar(fOffset);
47 switch (current) {
48 case '/':
49 if (fSkipComments && fOffset > 1) {
50 char next = fDocument.getChar(
51 fOffset = 1);
52 if (next == '*) {
53 // a comment ends, advance to the comment start
54 fOffset = 2;
55 gotoCommentStart();
56 continue;
57 }
58 return current;
59 }
60 return current;
61 case '\"':
62 case '\\':
63 if (fSkipStrings) {
64 gotoStringStart(current);
65 continue;
66 }
67 return current;
68 return current;
69 }
70 }
71 while (c != -1 & (c == '\r' || c == '\n')) {
72 c = reader.read();
73 }
A.1. WHILE-LOOPS

Listing A.29: ./org/eclipse/jdt/internal/ui/text/java/hover/JavadocHover.java

```java
while (loc == QualifiedType.NAME_PROPERTY || loc == QualifiedName.NAME_PROPERTY || loc == SimpleType.NAME_PROPERTY || loc == ParameterizedType.TYPEPROPERTY) {
    node = node.getParent();
    loc = node.getLocationInParent();
}
```

Listing A.30: ./org/eclipse/jdt/internal/ui/text/java/JavaAutoIndentStrategy.java

```java
while (node != null && (relativeOffset == node.getStartPosition() || relativeOffset == node.getStartPosition() + node.getLength()))
    node = node.getParent();
```

Listing A.31: ./org/eclipse/jdt/internal/ui/infoviews/SourceView.java

```java
// Defined in the superclass JavaCodeReader
public static final int EOF = -1;
public int read() throws IOException {
    try {
        return fForward ? readForwards() : readBackwards();
    } catch (BadLocationException x) {
        throw new IOException(x.getMessage());
    }
}

private int readForwards() throws BadLocationException {
    while (fOffset < fEnd) {
        char current = fDocument.getChar(fOffset++);
        switch (current) {
        case '/':
            if (fSkipComments && fOffset < fEnd) {
                char next = fDocument.getChar(fOffset);
                if (next == '*') {
                    // a comment starts, advance to the comment end
                    ++ fOffset;
                    gotoCommentEnd();
                } else if (next == '/') {
                    // '//' comment starts, advance to the line end
                    gotoLineEnd();
                } continue;
            }
            return current;
            case '/*':
            case '¥':
                if (fSkipStrings) {
                    gotoStringEnd(current);
                } continue;
            }
            return current;
        }
        return EOF;
    }
```
handleSingleLineComment();

char current = fDocument.getChar(fOffset);
switch (current) {
  case '/':
    if (fSkipComments && fOffset > 1) {
      char next = fDocument.getChar(fOffset - 1);
      if (next == '*') {
        // a comment ends, advance to the comment start
        fOffset = 2;
        gotoCommentStart();
        continue;
      }
    }
    return current;
  case ' "':
    case '\':
    if (fSkipStrings) {
      --fOffset;
      gotoStringStart(current);
      continue;
    }
    return current;
  case ' /':
    return current;
}
return EOF;
}
}
}
while (c != -1 && (c == 'r' || c == 'n' || c == 't')) {
  c = reader.read();
}

Listing A.32: ./org/eclipse/jdt/internal/compiler/lookup/BlockScope.java

while (hasMoreVariables || hasMoreScopes) {
  if (hasMoreScopes
    && !(hasMoreVariables || (this.subscopes[isceop].startIndex() <= ilocal))) {
    // consider subscope first
    if (this.subscopes[isceop].instanceof BlockScope) {
      BlockScope subscope = (BlockScope) this.subscopes[isceop];
      int subOffset = subscope.shiftscopes == null ? this.offset : subscope.maxShiftedOffset();
      subscope.computeLocalVariablePositions(0, subOffset, codeStream);
      if (subscope.maxOffset > this.maxOffset)
        this.maxOffset = subscope.maxOffset;
    }
  } else {
    // consider variable first
    LocalVariableBinding local = this.locals[ilocal]; // if no local at all, will be locals[ilocal]==null
    // check if variable is actually used, and may force it to be preserved
    boolean generateCurrentLocalVar = (local.useFlag > LocalVariableBinding.UNUSED && local.constant() == Constant.NotAConstant);
    // do not report fake used variable
    if (local.useFlag == LocalVariableBinding.UNUSED

A.1. WHILE-LOOPS

```java
&& (local.declaration != null) // unused (and non secret) local
&& ((local.declaration.bits & ASTNode.IsLocalDeclarationReachable) != 0)) { //
declaration is reachable
if (! (local.declaration instanceof Argument)) // do not report unused catch arguments
    problemReporter().unusedLocalVariable(
        local.declaration);
}
// could be optimized out, but does need to preserve unread variables ?
if (!generateCurrentLocalVar) {
    if (local.declaration != null && compilerOptions().preserveAllLocalVariables) {
        generateCurrentLocalVar = true; // force it to be preserved in the generated code
        if (local.useFlag ==
            LocalVariableBinding.UNUSED)
            local.useFlag =
            LocalVariableBinding.USED;
    }
    // allocate variable
    if (generateCurrentLocalVar) {
        if (local.declaration != null) {
            codeStream.record(local); // record user-defined local variables for attribute generation
        }
        // assign variable position
        local.resolvedPosition = this.offset;
        if ((local.type == TypeBinding.LONG) || (local.
type == TypeBinding.DOUBLE)) {
            this.offset += 2;
        } else {
            this.offset++;
        }
        if (this.offset > 0xFFFF) { // no more than 65535 words of locals
            problemReporter() . noMoreAvailableSpaceForLocal(
                local,
                local.declaration == null ? (ASTNode)methodScope() .
                referenceContext : local .
declaration);
        }
    } else {
        local.resolvedPosition = -1; // not generated
    }
    hasMoreVariables = ++iloc < maxLocals;
}
while (hasMoreVariables || hasMoreScopes) {
    if (hasMoreScopes && (!hasMoreVariables || (this.subscopes[isc] .
        startIndex() <= ilocal))) {
        // consider subscope first
        Scope subscope = this.subscopes[isc] .
        if (subscope.kind == Scope.BLOCK_SCOPE) { // do not dive
```
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

```java
localDeclarations = ((BlockScope)subscope).findLocalVariableDeclarations(position);
if (localDeclarations != null) {
    return localDeclarations;
}

hasMoreScopes = ++iscope < maxScopes;
else {
    // consider variable first
    LocalVariableBinding local = this.locals[i local]; // if no local at all, will be locals[i local]==null
    if (local != null) {
        LocalDeclaration localDecl = local.declaration;
        if (localDecl != null) {
            if (localDecl.declarationSourceStart <= position) {
                if (localDeclarations == null) {
                    localDeclarations = new LocalDeclaration[maxLocals];
                }
                localDeclarations[declPtr++] = localDecl;
            } else {
                return localDeclarations;
            }
        } else {
            return localDeclarations;
        }
    }
    hasMoreVariables = ++ilocal < maxLocals;
    if (!hasMoreVariables && localDeclarations != null) {
        return localDeclarations;
    }
}
```

Listing A.34: ./org/eclipse/jdt/internal/compiler/apt/util/Util.java

```java
while ((c = unitSource[begin]) == \n | | c == '\
 begin++;
```

Listing A.35: ./org/eclipse/jdt/internal/compiler/apt/dispatch/RoundDispatcher.java

```java
while (_searchForStar || !_unclaimedAnnotations.isEmpty()) {
    ProcessorInfo pi = _provider.discoverNextProcessor();
    if (null == pi) {
        // There are no more processors to be discovered.
        break;
    }
    handleProcessor(pi);
}
```

Listing A.36: ./org/eclipse/jdt/internal/compiler/tool/Util.java

```java
while ((c = unitSource[begin]) == \n | | c == '\
 begin++;
```

Listing A.37: ./org/eclipse/jdt/internal/compiler/ast/JavadocAllocationExpression.java
A.1. WHILE-LOOPS

while (!constructorBinding.isValidBinding() &
    (enclosingTypeBinding.isMemberType() ||
     enclosingTypeBinding.isLocalType())) {
    enclosingTypeBinding = enclosingTypeBinding.enclosingType();
    constructorBinding = scope.getConstructor(enclosingTypeBinding,
                                           argumentTypes, this);
}

Listing A.38: ./org/eclipse/jdt/internal/compiler/ast/JavadocMessageSend.java

while (!methodBinding.isValidBinding() &
    (enclosingTypeBinding.isMemberType() ||
     enclosingTypeBinding.isLocalType())) {
    methodBinding = scope.getMethod(enclosingTypeBinding, this,
                                  selector, argumentTypes, this);
}

Listing A.39: ./org/eclipse/jdt/internal/compiler/ast/JavadocMessageSend.java

while (!constructorBinding.isValidBinding() &
    (enclosingTypeBinding.isMemberType() ||
     enclosingTypeBinding.isLocalType())) {
    if (CharOperation.equals(this.selector, enclosingTypeBinding.
                             shortReadableName())) {
        constructorBinding = scope.getConstructor((ReferenceBinding)enclosingTypeBinding,
                                           argumentTypes, this);
    }
}

Listing A.40: ./org/eclipse/jdt/internal/compiler/Compiler.java

do {
    // extract units to process
    int length = top - bottom;
    CompilationUnitDeclaration[] currentUnits = new
    CompilationUnitDeclaration[length];
    int index = 0;
    for (int i = bottom; i < top; i++) {
        CompilationUnitDeclaration currentUnit = this.unitsToProcess[i];
        if ((currentUnit.bits & ASTNode.IsImplicitUnit) == 0) {
            currentUnits[index++] = currentUnit;
        }
    }
    if (index != length) {
        System.arraycopy(currentUnits, 0, currentUnits =
                         new CompilationUnitDeclaration[index], 0, index);
    }
    this.annotationProcessorManager.processAnnotations(currentUnits,
                             binaryTypeBindingsTemp, false);
    ICompilationUnit[] newUnits = this.annotationProcessorManager.
    getNewUnits();
    newUnitSize = newUnits.length;
    ReferenceBinding[] newClassFiles = this.annotationProcessorManager.getNewClassFiles();
    binaryTypeBindingsTemp = newClassFiles;
    newClassFilesSize = newClassFiles.length;
    if (newUnitSize != 0) {
        ICompilationUnit[] newProcessedUnits = (ICompilationUnit[])
        newUnits.clone(); // remember new units in case
                           a source type collision occurs
        try {
            this.lookupEnvironment.isProcessingAnnotations =
            true;
            internalBeginToCompile(newUnits, newUnitSize);
        } finally {
            this.annotationProcessorManager.processAnnotations(newUnits,
                        binaryTypeBindingsTemp, false);
        }
    }

}
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

Listing A.41: ./org/eclipse/jdt/internal/compiler/parser/Scanner.java

```java
while ((this.currentCharacter != '/ ') || (!star)) {
    if (this.currentPosition >= this.eofPosition) {
        throw new InvalidInputException(UNTERMINATED_COMMENT);
    }
    if ((this.currentCharacter == '\r') || (this.currentCharacter == '\n')) {
        if (isUnicode) {
            pushUnicodeLineSeparator();
        } else {
            pushLineSeparator();
        }
    }
    switch (this.currentCharacter) {
    case '*':
        star = true;
        break;
    case '@':
        if (firstTag == 0 && this.isFirstTag()) {
            firstTag = previous;
        }
        //SFALL-THROUGHS default case to set star to false
        default:
            star = false;
        }
    //get next char
    previous = this.currentPosition;
    if (((this.currentCharacter == this.source[this.currentPosition +1]) == '\\')) {
        //unicode traitement
        getNextUnicodeChar();
    } else {
        isUnicode = true;
    }
    } else {
        isUnicode = false;
    }
    //handle the \u case manually into comment
    if (this.currentCharacter == 'u') {
        if (this.source[this.currentPosition] == '\u') {
            //\u case
            this.currentPosition++;
        } else {
            this.currentPosition++;
        }
    } //jump over the \u
}
```

Listing A.42: ./org/eclipse/jdt/internal/compiler/parser/Scanner.java

```java
while ((this.currentCharacter != '/ ') || (!star)) {
    if (this.currentPosition >= this.eofPosition) {
        return;
    }
}
```
A.1. WHILE-LOOPS

if ((this.currentCharacter == '\r') || (this.currentCharacter == '\n')) {
    if (this.recordLineSeparator) {
        if (isUnicode) {
            pushUnicodeLineSeparator();
        } else {
            pushLineSeparator();
        }
    }
}

switch (this.currentCharacter) {
    case 'a':
        star = true;
        break;
    case 'b':
        if (firstTag == 0 && this.isFirstTag()) {
            firstTag = previous;
        }
        // $FALL-THROUGH$ default case to set star to false
        default:
            star = false;
    }

    // get next char
    previous = this.currentPosition;
    if ( ((this.currentCharacter = this.source[this.currentPosition++]) == '\n')
        && (this.source[this.currentPosition] == 'u') ) {
        getNextUnicodeChar();
        isUnicode = true;
    } else {
        isUnicode = false;
    }
    // handle the \u case manually into comment
    if (this.currentCharacter == '\\') {
        if (this.source[this.currentPosition] == '\\\n            this.currentPosition++;
    } // jump over the \\

Listing A.43: ./org/eclipse/jdt/internal/compiler/parser/diagnose/DiagnoseParser.java

while (act > ERROR_ACTION || act < ACCEPT_ACTION) { // SHIFT−REDUCE action or SHIFT action:
    this.nextStackTop = this.tempStackTop + 1;
    for (int i = next_pos + 1; i <= this.nextStackTop; i++)
        this.nextStack[i] = this.tempStack[i];

    for (int i = pos + 1; i <= this.nextStackTop; i++) {
        this.locationStack[i] = this.locationStack[this.stateStackTop];
        this.locationStartStack[i] = this.locationStartStack[this.stateStackTop];
    }
}

// If we have a shift−reduce, process it as well as // the goto−reduce actions that follow it.
if (act > ERROR_ACTION) {
    act = ERROR_ACTION;
    do {
        this.nextStackTop = (Parser.rhs[act]−1);
        act = Parser.ntAction(this.nextStackTop, Parser.lhs[act]);
    } while (act <= NUM_RULES);
    pos = pos < this.nextStackTop ? pos : this.nextStackTop;
}

if (this.nextStackTop + 1 >= this.stackLength)
    reallocateStacks();
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

```java
26 this.tempStackTop = this.nextStackTop;
27 this.nextStack[++this.nextStackTop] = act;
28 next_pos = this.nextStackTop;
29
30 // Simulate the parser through the next token without
31 // destroying STACK or next_stack.
32 this.currentToken = this.lexStream.getToken();
33 tok = this.lexStream.kind(this.currentToken);
34 act = Parser.tAction(act, tok);
35 while (act <= NUMRULES) {
36 // ... Process all goto-reduce actions following
37 // reduction, until a goto action is computed ...
38 //
39 do {
40 int lhs_symbol = Parser.lhs[act];
41 if (DEBUG) {
42 System.out.println(Parser.name[Parser.
43 non_terminal_index[lhs_symbol]]);
44 }
45 this.tempStackTop -= (Parser.rhs[act] - 1);
46 act = (this.tempStackTop > next_pos
47 ? this.tempStack[this.
48 tempStackTop
49 : this.nextStack[this.
50 tempStackTop]);
51 act = Parser.ntAction(act, lhs_symbol);
52 } while (act <= NUMRULES);
53
54 // ... Update the maximum useful position of the
55 // (STATE)STACK, push GOTO state into stack, and
56 // compute next action on current symbol ...
57 //
58 // if (this.tempStackTop + 1 >= this.stackLength)
59 reallocateStacks();
60 next_pos = next_pos < this.tempStackTop ? next_pos :
61 this.tempStackTop;
62 this.tempStack[this.tempStackTop + 1] = act;
63 act = Parser.tAction(act, tok);
64 } while (act <= NUMRULES);
65
66 // ... Update the maximum useful position of the
67 // (STATE)STACK, push GOTO state into stack, and
68 // compute next action on current symbol ...
69 //
70 // if (this.tempStackTop + 1 >= this.stackLength)
71 reallocateStacks();
72 //
73 // if (act != ERRORACTION) {
74 this.prevStackTop = this.stateStackTop;
75 for (int i = prev_pos + 1; i <= this.prevStackTop; i++)
76 this.prevStack[i] = this.stack[i];
77 prev_pos = pos;
78 }
```

A.1. WHILE-LOOPS

```
while (result == this.readInProcessMarker || result == null) {
    // let the readingThread know we're waiting
    // System.out.print('|');
    this.contentsRead[this.readyToReadPosition] = null;
    try {
        wait(250);
    } catch (InterruptedException ignore) { // ignore
        if (this.caughtException != null) {
            // rethrow the caught exception from the readingThreads
            in the main compiler thread
            if (this.caughtException instanceof Error)
                throw (Error) this.caughtException;
            throw (RuntimeException) this.caughtException;
        }
    }
    result = this.contentsRead[this.readyToReadPosition];
}
```

```
while ((this.keyTable[index] != 0) || ((this.keyTable[index] == 0) && (this.valueTable[index] != 0))) {
    if (this.keyTable[index] == key)
        return true;
    if (++index == length) {
        index = 0;
    }
}
```

```
while ((this.keyTable[index] != 0) || ((this.keyTable[index] == 0) && (this.valueTable[index] != 0))) {
    if (this.keyTable[index] == key)
        return this.valueTable[index] = value;
    if (++index == length) {
        index = 0;
    }
}
```

```
while ((this.keyTable[index] != 0) || ((this.keyTable[index] == 0) && (this.valueTable[index] != 0))) {
    if (this.keyTable[index] == key)
        return this.valueTable[index];
    if (++index == length) {
        index = 0;
    }
}
```
LISTING A.48: ./org/eclipse/jdt/internal/compiler/codegen/IntegerCache.java

```java
while ((this.keyTable[index] != 0) || ((this.keyTable[index] == 0) && (this.valueTable[index] != 0))) {
    if (this.keyTable[index] == key)
        return true;
    if (++index == length) {
        index = 0;
    }
}
```

LISTING A.49: ./org/eclipse/jdt/internal/compiler/codegen/IntegerCache.java

```java
while ((this.keyTable[index] != 0) || ((this.keyTable[index] == 0) && (this.valueTable[index] != 0))) {
    if (this.keyTable[index] == key)
        return true;
    if (++index == length) {
        index = 0;
    }
}
```

LISTING A.50: ./org/eclipse/jdt/internal/compiler/codegen/IntegerCache.java

```java
while ((this.keyTable[index] != 0) || ((this.keyTable[index] == 0) && (this.valueTable[index] != 0))) {
    if (this.keyTable[index] == key)
        return true;
    if (++index == length) {
        index = 0;
    }
}
```

LISTING A.51: ./org/eclipse/jdt/internal/compiler/problem/DefaultProblem.java

```java
while ((c = unitSource[begin]) == '\n' || c == '\t') begin++;
```

LISTING A.52: ./org/eclipse/jdt/internal/compiler/batch/Main.java

```java
while ((c = unitSource[begin]) == '\n' || c == '\t') begin++;
```

LISTING A.53: ./org/eclipse/jdt/internal/compiler/batch/Main.java

```java
while ((c = unitSource[begin]) == '\n' || c == '\t') begin++;
```

LISTING A.54: ./org/eclipse/jdt/internal/compiler/batch/Main.java

```java
while ((c = unitSource[end]) == '\n' || c == '\t') end--;
```

LISTING A.55: ./org/eclipse/jdt/internal/compiler/classfmt/ClassFileReader.java

```java
int length1 = currentMethodInfos == null ? 0 : currentMethodInfos.length;
int length2 = otherMethodInfos == null ? 0 : otherMethodInfos.length;
int index1 = 0;
int index2 = 0;
```
A.1. WHILE-LOOPS

Listing A.56: ./org/eclipse/jdt/internal/compiler/classfmt/ClassFileReader.java

```java
int length1 = currentMethodInfos == null ? 0 : currentMethodInfos.length;
int length2 = otherMethodInfos == null ? 0 : otherMethodInfos.length;
int index1 = 0;
int index2 = 0;
```

Listing A.57: ./org/eclipse/jdt/internal/codeassist/complete/CompletionScanner.java

```java
while ((this.currentCharacter != '/') || (!star)) {
  if ((this.currentCharacter == '\r') || (this.currentCharacter == '\n')) {
    checkNonExternalizedString();
    if (this.recordLineSeparator) {
      isUnicode = true;
    }
    pushLineSeparator();
  }
  switch (this.currentCharacter) {
    case '*':
      star = true;
      break;
    case '@':
      if (firstTag == 0 && this.isFirstTag()) {
        firstTag = previous;
        //FALL-THROUGH default case to set star to false
        default:
          star = false;
      }
      //get next char
      if (this.currentCharacter == this.source[this.currentPosition++] == '\' ) {
        //---unicode traitement ---
        getNextUnicodeChar();
        isUnicode = true;
      } else {
        isUnicode = false;
      }
      //handle the \u case manually into comment
      if (this.currentCharacter == '\u') {
        if (this.source[this.currentPosition] == '\u') {
          this.currentPosition++;
          //jump over the \u
        }
      }
    
    break;
  }
}
```

Listing A.58: ./org/eclipse/jdt/internal/core/jdom/DOMField.java

```java
while (field.isVariableDeclarator() || field.hasNextMultipleVariableDeclarators()) {
```
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

if (field.fNextNode instanceof DOMField && ((DOMField)field.fNextNode).isVariableDeclara(t) {  
  field= (DOMField)field.fNextNode;
  
} else {  
  break;
  
}

Listing A.59: ./org/eclipse/jdt/internal/core/util/BindingKeyParser.java

while (this.index < this.source.length && (this.source[this.index] == '<' || this.source[this.index] == '%')) {  
  this.index++;
}

Listing A.61: ./org/eclipse/jdt/internal/core/util/PublicScanner.java

while (((this.currentCharacter != '/') || (!star)) {  
  if (this.currentPosition >= this.eofPosition) {  
    throw new InvalidInputException(UNTERMINATED_COMMENT);
  }
  if (isUnicode) {  
    isUnicode = true;
  } else {  
    isUnicode = false;
  }
  if (this.currentCharacter == 'u') {  
    pushUnicodeLineSeparator();
  } else {  
    pushLineSeparator();
  }
  // handle the \u case manually into comment
  if (this.currentCharacter == '\\') {  
    isUnicode = true;
  } else {  
    isUnicode = false;
  }
}

Listing A.60: ./org/eclipse/jdt/internal/core/util/PublicScanner.java

while ((this.currentCharacter != '/') || (!star)) {  
  if (this.currentPosition >= this.eofPosition) {  
    return;
  }
  if (this.currentCharacter == '\\') {  
    nextUnicodeChar();
  } else if (this.currentCharacter == 'u') {  
    pushUnicodeLineSeparator();
  } else {  
    pushLineSeparator();
  }
  // get next char
  if (((this.currentCharacter == this.currentPosition++) == '\\')  
    && (this.source[this.currentPosition] == 'u')) {  
    isUnicode = true;
  } else {  
    isUnicode = false;
  }
  // jump over the \\
}

Listing A.61: ./org/eclipse/jdt/internal/core/util/PublicScanner.java
A.1. WHILE-LOOPS

```java
if ((this.currentCharacter == '\r') || (this.currentCharacter == '\n')) {
    if (this.recordLineSeparator) {
        if (isUnicode) {
            pushUnicodeLineSeparator();
        } else {
            pushLineSeparator();
        }
    }
}

switch (this.currentCharacter) {
    case '*':
        star = true;
        break;
    case '@':
        if (firstTag == 0 && this.isFirstTag()) {
            firstTag = previous;
        }
        // $FALL-THROUGH$ default case to set star to false
        default:
            star = false;
    }

    // get next char
    previous = this.currentPosition;
    if (((this.currentCharacter == this.source[this.currentPosition + 1] == '\' && (this.source[this.currentPosition] == 'u')) {
        getNextUnicodeChar();
        isUnicode = true;
    } else {
        isUnicode = false;
    }
}

    // handle the \u case manually into comment
    if (this.currentCharacter == '\\') {
        if (this.source[this.currentPosition] == '\\') {
            // jump over the \\n        }
    }
}
```

Listing A.62: ./org/eclipse/jdt/internal/core/JavadocContents.java

```java
while ((index = CharOperation.indexOf(JavadocConstants.ANCHOR_PREFIX_START, this.content, false, fromIndex)) != -1 && (index < indexOfSectionBottom || indexOfSectionBottom == -1)) {
    fromIndex = index + 1;
    int anchorEndStart = index + JavadocConstants.ANCHOR_PREFIX_START_LENGTH;
    this.tempLastAnchorFoundIndex = anchorEndStart;
    if (CharOperation.prefixEquals(anchor, this.content, false, anchorEndStart)) {
        return computeChildRange(anchorEndStart, anchor, indexOfSectionBottom);
    } else {
        if (this.tempAnchorIndexes.length == this.tempAnchorIndexes.length) {
            System.arraycopy(this.tempAnchorIndexes, 0, this.tempAnchorIndexes = new int [this.tempAnchorIndexesCount + 20], 0, this.tempAnchorIndexesCount);
            this.tempAnchorIndexes[this.tempAnchorIndexesCount++] = anchorEndStart;
        }
    }
}
```

Listing A.63: ./org/eclipse/jdt/internal/corext/dom/TokenScanner.java
private IScanner fScanner;

private int readNextWithEOF(boolean ignoreComments) throws CoreException {
    int curr = 0;
    do {
        try {
            curr = fScanner.getNextToken();
        } catch (InvalidInputException e) {
            throw new CoreException(createError(
                LEXICAL_ERROR, e.getMessage(), e));
        }
    } while (ignoreComments && isComment(curr));
    return curr;
}

while (curr == ITerminalSymbols.TokenNameCOMMENT_LINE || curr ==
        ITerminalSymbols.TokenNameCOMMENT_BLOCK) {
    int currStartLine = getLineOfOffset(getCurrentStartOffset());
    int linesDifference = currStartLine - prevEndLine;
    if (linesDifference > 1) {
        return prevEndPos; // separated comments
    }
    if (curr == ITerminalSymbols.TokenNameCOMMENT_LINE) {
        prevEndPos = getLineEnd(currStartLine);
        prevEndLine = currStartLine;
    } else {
        prevEndPos = getCurrentEndOffset();
        prevEndLine = getLineOfOffset(prevEndPos - 1);
    }
    if (sameLineComment) {
        if (linesDifference == 0) {
            res = prevEndPos;
        } else {
            sameLineComment = false;
        }
    }
    curr = readNextWithEOF(false);
}

Listing A.64: ./org/eclipse/jdt/internal/corext/fix/CleanUpRefactoring.java

do {
    ICompilationUnit cleanUp = cleanUps[i];
    ICompilationUnitFix fix;
    if (slowCleanUps != null) {
        long timeBefore = System.currentTimeMillis();
        fix = cleanUp.createFix(context);
        if (System.currentTimeMillis() - timeBefore >
            SLOW_CLEAN_UP_THRESHOLD)
            slowCleanUps.add(cleanUp);
    } else {
        fix = cleanUp.createFix(context);
    }
    if (fix != null) {
        CompilationUnitChange current = fix.createChange(null);
        TextEdit currentEdit = current.getEdit();
        if (solution != null) {
            if (TextEditUtil.overlaps(currentEdit, solution.getEdit())) {
                undoneCleanUps.add(cleanUp);
            } else {
                CleanupChange merge = new CleanupChange(
                    FixMessages.CleanupRefactoring.cleanup_multi_change_name,
                    context.getCompilationUnit());
                merge.setEdit(TextEditUtil.merge(
                    currentEdit, solution.getEdit()));
                copyChangeGroups(merge, solution);
            }
        }
    }
}

Listing A.65: ./org/eclipse/jdt/internal/corext/fix/CleanUpRefactoring.java

do {
    ICompilationUnit cleanUp = cleanUps[i];
    ICompilationUnitFix fix;
    if (slowCleanUps != null) {
        long timeBefore = System.currentTimeMillis();
        fix = cleanUp.createFix(context);
        if (System.currentTimeMillis() - timeBefore >
            SLOW_CLEAN_UP_THRESHOLD)
            slowCleanUps.add(cleanUp);
    } else {
        fix = cleanUp.createFix(context);
    }
    if (fix != null) {
        CompilationUnitChange current = fix.createChange(null);
        TextEdit currentEdit = current.getEdit();
        if (solution != null) {
            if (TextEditUtil.overlaps(currentEdit, solution.getEdit())) {
                undoneCleanUps.add(cleanUp);
            } else {
                CleanupChange merge = new CleanupChange(
                    FixMessages.CleanupRefactoring.cleanup_multi_change_name,
                    context.getCompilationUnit());
                merge.setEdit(TextEditUtil.merge(
                    currentEdit, solution.getEdit()));
                copyChangeGroups(merge, solution);
            }
        }
    }
}
A.1. WHILE-LOOPS

```java
24     copyChangeGroups(merge, current);
25     solution= merge;
26     }  
27     
28     } else {  
29         solution= new CleanUpChange(current.getName(),
30             context.getCompilationUnit());  
31         solution.setEdit(currentEdit);
32     }
33     copyChangeGroups(solution, current);
34     }
35 }
36 }  
37 }
38 while (i < cleanUps.length && (context.getAST() == null || !cleanUps[i].getRequirements().requiresFreshAST()));
```

Listing A.65: ./org/eclipse/jdt/internal/corext/buildpath/ClasspathModifying.java

```java
do {
    if (container instanceof IFolder)
        javaElem= JavaCore.create((IFolder) container);
    if (container.getFullPath().equals(project.getPath())) {
        javaElem= project;
        break;
    }
    container= container.getParent();
    if (container == null)
        return null;
}
while (javaElem == null || !(javaElem instanceof IPackageFragmentRoot)) ;
```

Listing A.66: ./org/eclipse/jdt/internal/corext/refactoring/reorg/RefactoringModifications.java

```java
while (container != null && ! (container.exists() || getResourceModifications().willExist(container))) {
    getResourceModifications().addCreate(container);
    container= container.getParent();
}
```

Listing A.67: ./org/eclipse/jdt/internal/corext/refactoring/code/ExtractMethodRefactoring.java

```java
do {
    node= node.getParent();
}
while (node != null && ! (node instanceof AbstractTypeDeclaration || node instanceof AnonymousClassDeclaration));
```

Listing A.68: ./org/eclipse/jdt/internal/debug/ui/jres/InstalledJREsBlock.java

```java
do {
    id= String.valueOf(System.currentTimeMillis());
}
while (vmType.findVMInstall(id) != null || id.equals(fgLastUsedID));
```

Listing A.69: ./org/eclipse/swt/internal/image/JPEGFileFormat.java

```java
while ((zeros < r || dataUnit[zzIndex] != 0) && k <= end) {
    if (dataUnit[zzIndex] != 0) {
        dataUnit[zzIndex] = refineAC(dataUnit[zzIndex], approxBit);
    }
```
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

Listing A.70: ./org/eclipse/swt/internal/image/PngHuffmanTable.java

```java
while ( j >= h && ( lengths[j - h] > v || (lengths[j - h] == v && codeValues[j - h] == codeValuesTemp)) ) {
    lengths[j] = lengths[j - h];
    codeValues[j] = codeValues[j - h];
    j -= h;
}
```

Listing A.71: ./org/eclipse/swt/internal/image/JPEGDecoder.java

```java
while ( cinf.input.scan_number < cinf.output.scan_number ||
    cinf.input.scan_number == cinf.output.scan_number &&
    cinf.input.IMCU_row <= cinf.output.IMCU_row ) {
    if (consume_input(cinfo) == JPEG_SUSPENDED)
        return JPEG_SUSPENDED;
}
```

Listing A.72: ./org/eclipse/jface/text/formatter/ContentFormatter.java

```java
while ('\t' == c || ' ' == c)
    c = fDocument.getChar(++end);
```

Listing A.73: ./org/eclipse/jface/text/rules/RuleBasedPartitioner.java

```java
do {
    --first ;
    if ( first < 0 )
        break;
    p = category[first ];
} while ( p.overlapsWith(e.getOffset() , e.getLength()) ||
    (e.getOffset() == fPreviousDocumentLength &&
    (p.getOffset() + p.getLength() ==
    fPreviousDocumentLength)));
```

Listing A.74: ./org/eclipse/jface/text/AbstractDocument.java

```java
while ( back >= 0 || forth < size ) {
    if ( back >= 0 ) {
        if ( position == positions.get(back) ) {
            positions.remove(back);
            return;
        }
        back--;
    }
    if ( forth < size ) {
        if ( position == positions.get(forth) ) {
            positions.remove(forth);
            return;
        }
        forth++;
    }
```
A.1. WHILE-LOOPS

Listing A.75: ./org/eclipse/jface/text/source/DefaultCharacterPairMatcher.java

```java
while ((pos1 => lowerBoundary && !lowerFound) || (pos2 < upperBoundary && !upperFound)) {
    for (int i= 0; i < counts.length; i++) {
        counts[i][0] = counts[i][1] = 0;
    }

    outer1: while (pos1 => lowerBoundary && !lowerFound) {
        final char c = doc.getChar(pos1);
        final int index = getCharacterIndex(c, document, pos1);
        if (index != -1 & doc.inPartition(pos1)) {
            counts[index / 2][0]--; // start
        } else {
            counts[index / 2][0]++; // end
        }
        for (int j= 0; j < counts.length; j++) {
            if (counts[j][0] == -1) {
                lowerFound = true;
                break outer1;
            }
        }
        pos1 = doc.getNextPosition(pos1, false);
    }

    outer2: while (pos2 < upperBoundary && !upperFound) {
        final char c = doc.getChar(pos2);
        final int index = getCharacterIndex(c, document, pos2);
        if (index != -1 & doc.inPartition(pos2)) {
            counts[index / 2][1]++; // start
        } else {
            counts[index / 2][1]--; // end
        }
        for (int j= 0; j < counts.length; j++) {
            if (counts[j][1] == -1 & j == 0) {
                upperFound = true;
                break outer2;
            }
        }
        pos2 = doc.getNextPosition(pos2, true);
    }

    if (pos1 > start || pos2 < end - 1) {
        // match inside selection => discard
        pos1 = doc.getNextPosition(pos1, false);
        pos2 = doc.getNextPosition(pos2, true);
        lowerFound = false;
        upperFound = false;
    }
}
```

Listing A.76: ./org/eclipse/jface/text/source/LineChangeHover.java

```java
while (l => min & info != null && (info.getChangeType() == ILineDiffInfo.CHANGED || info.getChangeType() == ILineDiffInfo.ADDED)) {
    info = differ.getLineInfo(--1);
}
```

Listing A.77: ./org/eclipse/jface/text/source/LineChangeHover.java

```java
while (l <= max & info != null && (info.getChangeType() == ILineDiffInfo.CHANGED || info.getChangeType() == ILineDiffInfo.ADDED)) {
```
Listing A.78: ./org/eclipse/jface/internal/text/html/HTML2TextReader.java

```java
protected int nextChar() throws IOException {
    if (this.fReadFromBuffer) {
        char ch = this.fBuffer.charAt(this.fIndex++);
        if (this.fIndex >= this.fBuffer.length()) {
            this.fBuffer.setLength(0);
            this.fIndex = 0;
        }
        return ch;
    } else {
        int ch = this.fCharAfterWhiteSpace;
        if (ch == -1) {
            ch = this.fReader.read();
        }
        if (this.fSkipWhiteSpace && ScannerHelper.isWhitespace((char)ch)) {
            do {
                ch = this.fReader.read();
                while (ScannerHelper.isWhiteSpace((char)ch));
                if (ch != -1) {
                    this.fCharAfterWhiteSpace = ch;
                }
            } else {
                this.fCharAfterWhiteSpace = -1;
            }
        }
        return ch;
    }
    while (Character.isLetterOrDigit((char)ch) || ch == '#') {
        buf.append((char) ch);
        ch = nextChar();
    }
    return ch;
}
```

Listing A.79: ./org/eclipse/team/internal/c-cvs/ssh2/CVSSSH2ServerConnection.java

```java
while (firstTime || tryAgain) {
    tryAgain = false; // reset the try again flag
    session = JSchSession.getSession(location, location.getUsername()
        .password, location.getHost(), location.getPort(), monitor);
    channel = session.openSession().openChannel("exec"); //NON-NLS-1
    ((ChannelExec) channel).setCommand(COMMAND);
    channel_out = channel.getOutputStream();
    channel_in = channel.getInputStream();
    try {
        channel.connect();
    } catch (JSchException ee) {
        // This strange logic is here due to how the JSch client
        // shares sessions.
        // It is possible that we have obtained a session that
        // thinks it is connected
        // but is not. Channel connection only works if the
        // session is connected so the
        // above channel connect may fail because the session is
down. For this reason,
        // we want to retry if the connection fails.
        try {
            if (firstTime && (isSessionDownError(ee) ||
                isChannelNotOpenError(ee))) {
                tryAgain = true;
```
A.1. WHILE-LOOPS

```java
while (count > 0 && (val[begin + count - 1] == '\n' || val[begin + count - 1] == '\t'))
    count --;
```

Listing A.80: ./org/eclipse/osgi/framework/internal/core/Tokenizer.java

```java
while (count > 0 && (val[begin + count - 1] == '\n' || val[begin + count - 1] == '\t'))
    count --;
```

Listing A.81: ./org/eclipse/osgi/util/ManifestElement.java

```java
while (c == '=' || c == ' :
    while (c == ' :')
        // may not really be a :=
        c = tokenizer.getChar();
    if (c != '=')
        String restOfNext = tokenizer.getToken("="); // $NON-NLS-1$
    else
directive = true;
}
// determine if the attribute is the form attr:List<
String preserveEscapes = null;
if (!directive && next.indexOf("List") > 0) // $NON-NLS-1$
    Tokenizer listTokenizer = new Tokenizer(next);
    String attrKey = listTokenizer.getToken("="); // $NON-NLS-1$
if (attrKey != null && listTokenizer.getChar() == ':' &&
    "List".equals(listTokenizer.getToken("<"))) { // $NON-NLS-15/$NON-NLS-25
    // we assume we must preserve escapes for , and
    preserveEscapes = "\\:"; // $NON-NLS-1$
}
String val = tokenizer.getString(";", preserveEscapes); // $NON-
if (val == null)
    throw new BundleException(NLS.bind(Msg.
        MANIFEST_INVALID_HEADER_EXCEPTION, header, value), BundleException.MANIFEST_ERROR);
if (Debug.DEBUG_MANIFEST)
    Debug.print(";" + next + ";=" + val); // $NON-NLS-15/
    SNON-NLS-25
try {
    if (directive)
        manifestElement.addDirective(next, val);
    else
        manifestElement.addAttribute(next, val);
directive = false;
} catch (Exception e) {
    throw new BundleException(NLS.bind(Msg.
        MANIFEST_INVALID_HEADER_EXCEPTION, header, value), BundleException.MANIFEST_ERROR, e);
}
```

```java
A.1. WHILE-LOOPS

117

e117
{19
if (!tryAgain) {
    throw ee;
}
}
finally {
    // Always dispose of the current session when a
    // failure occurs so we can start from scratch
    session.dispose();
}

firstTime = false; // the first time is done
```
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

Listing A.82: ./org/eclipse/ant/internal/ui/launchConfigurations/Ant-LaunchShortcut.java

```java
while ( file == null || file.getType() != IResource.FILE ) {
    for (int i = 0; i < names.length; i++) {
        String string = names[i];
        file = lparent.findMember(string);
        if ( file != null && file.getType() == IResource.FILE ) {
            break;
        }
    }
    lparent = lparent.getParent();
    if (lparent == null) {
        return null;
    }
}
```

Listing A.83: ./org/eclipse/e4/ui/internal/workbench.swt/PartRenderingEngine.java

```java
while ((testShell != null && !testShell.isDisposed()) || (!theApp.getChildren().isEmpty() && someAreVisible(theApp.getChildren())))
        && !display.isDisposed()) {
    try {
        if (!display.readAndDispatch()) {
            runContext.processWaiting();
            if (spinOnce)
                return;
            advisor.eventLoopIdle(display);
        }
    }
    catch (ThreadDeath th) {
        throw th;
    }
    catch (Exception ex) {
        handle(ex, advisor);
    }
    catch (Error err) {
        handle(err, advisor);
    }
}
```

Listing A.84: ./org/eclipse/e4/core/services/util/JSONObject.java

```java
while (Character.isDigit(c) || c == '-' || c == '+' || c == '.' || c == 'e' || c == 'E') {
    buffer.append(c);
    c = it.next();
}
```

Listing A.85: ./org/eclipse/compare/rangedifferencer/RangeDifferencer.java

```java
while (myIter.fDifference != null || yourIter.fDifference != null) {
    DifferencesIterator startThread;
    myIter.removeAll();
```
5    yourIter.removeAll();
6    //
7    // take the next diff that is closer to the start
8    //
9    if (myIter.difference == null)
10       startThread= yourIter;
11    else if (yourIter.difference == null)
12       startThread= myIter;
13    else { // not at end of both scripts take the lowest range
14       if (myIter.difference.leftStart < yourIter.difference.
15          leftStart) { // 2 => common (Ancestor) change range
16          startThread= myIter;
17       } else if (myIter.difference.leftStart > yourIter.
18          difference.leftStart) {
19          startThread= yourIter;
20       } else {
21          if (myIter.difference.leftLength == 0 &&
22             yourIter.difference.leftLength == 0) {
23             // insertion into the same position is conflict.
24             changeRangeStart= myIter.difference.
25                leftStart;
26             changeRangeEnd= myIter.difference.
27                leftEnd();
28             myIter.next();
29             yourIter.next();
30             diff3.add(createRangeDifference3(factory
31                , myIter, yourIter, diff3, right,
32                , left, changeRangeStart,
33                , changeRangeEnd));
34             continue;
35         } else if (myIter.difference.leftLength == 0)
36         { // insertion into a position, and
37             // modification to the next line, is not conflict.
38             startThread= myIter;
39         } else if (yourIter.difference.leftLength == 0)
40         { startThread= yourIter;
41         } else {
42             // modifications to overlapping lines is conflict.
43             startThread= myIter;
44         }
45    }
46
47    changeRangeStart= startThread.difference.leftStart;
48    changeRangeEnd= startThread.difference.leftEnd();
49    startThread.next();
50    monitor.worked(1);
51    // check for overlapping changes with other thread
52    // merge overlapping changes with this range
53    //
54    DifferencesIterator other= startThread.other(myIter, yourIter);
55    while (other.difference != null && other.difference.leftStart
56        < changeRangeEnd) {
57        int newMax= other.difference.leftEnd();
58        other.next();
59        monitor.worked(1);
60        if (newMax > changeRangeEnd) {
61            changeRangeEnd= newMax;
62            other= other.other(myIter, yourIter);
63        }
64    }
65    diff3.add(createRangeDifference3(factory, myIter, yourIter,
66               diff3, right, left, changeRangeStart, changeRangeEnd));
Listing A.86: ./org/eclipse/pde/api/tools/internal/comparator/ClassFileComparator.java

```java
while (superName != null && (!Util.isJavaLangObject(superName) || includeObject)) {
    superClass = superClass.getSuperclass();
    int visibility = VisibilityModifiers.PRIVATE;
    IApiComponent superComponent = superClass.getApiComponent();
    IApiDescription apiDescription = superComponent.getApiDescription();
    IApiAnnotations elementDescription = apiDescription.resolveAnnotations(superClass.getHandle());
    if (elementDescription != null) {
        visibility = elementDescription.getVisibility();
    }
    if (includePrivate || ((visibility & visibilityModifiers) != 0)) {
        list.add(superClass);
    }
    superName = superClass.getSuperclassName();
}
```

Listing A.87: ./org/eclipse/pde/internal/ui/editor/schema/ElementSection.java

```java
while ((o instanceof SchemaElementReference) || (!o instanceof ISchemaElement)) {
    o = o.getParent();
    result++;
}
```

Listing A.88: ./org/eclipse/pde/internal/ui/editor/contentassist/ManifestContentAssistProcessor.java

```java
while ((index = value.indexOf(':')) == -1 || ((value.length() - 1 != index) && (value.charAt(index + 1) == '='))) {
    int startLine = doc.getLineOffset(lineNum);
    value = doc.get(startLine, offset - startLine);
    lineNum--;
}
```

Listing A.89: ./org/eclipse/pde/internal/ui/editor/contentassist/ManifestContentAssistProcessor.java

```java
do {
    startOfLine = doc.getLineOffset(line);
    newValue = doc.get(offset, doc.getLineLength(line) - offset + startOfLine);
    ++line;
    colon = newValue.lastIndexOf(':');
} while ((colon == -1 || (newValue.length() > colon && newValue.charAt(colon + 1) == '=')) && entireHeader || newValue.indexOf(';', colon) == -1) && !doc.getNumberOfLines() == line);
```

Listing A.90: ./org/eclipse/core/internal/localstore/FileSystemResourceManager.java

```java
while (newChar == '\r' || newChar == '\n')
    newChar = newStream.read();
```

Listing A.91: ./org/eclipse/core/internal/localstore/FileSystemResourceManager.java
A.1. WHILE-LOOPS

1 while (oldChar == '\r' || oldChar == '\n')
2   oldChar = oldStream.read();

Listing A.92: ./org/eclipse/debug/internal/ui/viewers/model/TreeModelContentProvider.java
1 while (parentRequests == null || parentRequests.isEmpty()) {
2   parentPath = parentPath.getParentPath();
3   if (parentPath == null) {
4     // no running requests: start request
5     return false;
6   }
7   parentRequests = (List)fRequestsInProgress.get(parentPath);
}

Listing A.93: ./org/eclipse/debug/internal/ui/viewers/model/VirtualFindAction.java
1 while (!listener.fLabelUpdatesComplete || !listener.fViewerUpdatesComplete) && !listener.fProgressMonitor.isCanceled()) {
2   Thread.sleep(1);
}

Listing A.94: ./org/eclipse/debug/internal/core/LaunchManager.java
1 while (line.indexOf('=') < 0 || (line.length() > 0 && !Character.isJavaIdentifierStart(line.charAt(0))) { 
2   value += newLine + line;
3   line = reader.readLine();
4   if (line == null) {
5     // if next line read is the end of the file quit the loop
6     break;
7   }
}

Listing A.95: ./org/eclipse/debug/internal/core/LaunchManager.java
1 protected synchronized String[] getAllSortedConfigNames() {
2   if (fSortedConfigNames == null) {
3     ILaunchConfiguration[] configs = getLaunchConfigurations();
4     fSortedConfigNames = new String[configs.length];
5     for (int i = 0; i < configs.length; i++) {
6       fSortedConfigNames[i] = configs[i].getName();
7     }
8     Arrays.sort(fSortedConfigNames);
9   }
10   return fSortedConfigNames;
11 }
12 public boolean isExistingLaunchConfigurationName(String name) {
13   String[] sortedConfigNames = getAllSortedConfigNames();
14   int index = Arrays.binarySearch(sortedConfigNames, name);
15   if (index < 0) {
16     return false;
17   }
18   return true;
19 }
20 while (isExistingLaunchConfigurationName(newname) || reservednames.contains(newname)) {
21   buffer = new StringBuffer(base);
22   buffer.append(quote); //$NON-NLS-1$
23   buffer.append(String.valueOf(index));
24   index++;

APPENDIX A. LOOPS USED IN THE DETAILED STUDY

```java
buffer.append(')');
newname = buffer.toString();
}
```

Listing A.96: ./org/objectweb/asm/ClassWriter.java

```java
while (i != null && (i.type != key.type || !key isEqualTo (i))) {
    i = i.next;
}
```

Listing A.97: ./org/osgi/framework/ServicePermission.java

```java
while ((i != -1) && ((c = a[i]) == '\n' || c == '\r' || c == '\t' || c == '-'))
    i--;  
```

Listing A.98: ./org/osgi/framework/PackagePermission.java

```java
while ((i != -1) && ((c = a[i]) == '\n' || c == '\r' || c == '\t'))
    i--;  
```

Listing A.99: ./org/osgi/framework/BundlePermission.java

```java
while ((i != -1) && ((c = a[i]) == '\n' || c == '\r' || c == '\t'))
    i--;  
```

Listing A.100: ./org/osgi/framework/CapabilityPermission.java

```java
while ((i != -1) && ((c = a[i]) == '\n' || c == '\r' || c == '\t'))
    i--;  
```

Listing A.101: ./org/osgi/framework/AdminPermission.java

```java
while ((i != -1) && ((c = a[i]) == '\n' || c == '\r' || c == '\t'))
    i--;  
```

Listing A.102: ./org/osgi/framework/AdaptPermission.java

```java
while ((i != -1) && ((c = a[i]) == '\n' || c == '\r' || c == '\t'))
    i--;  
```

Listing A.103: ./org/apache/jasper/compiler/Parser.java

```java
private JspReader reader;
while (Character.isLetter(ch) || Character.isDigit(ch) || ch == '.' ||
    ch == '-' || ch == ':' || ch == ';') {
    buf.append(ch);
    reader.nextChar();
    ch = (char) reader.peekChar();
}  
```
A.1. WHILE-LOOPS

Listing A.104: ./org/apache/jasper/compiler/Parser.java

```java
private JspReader reader;

do {
    // XXX could move this logic to JspReader
    currentChar = reader.nextChar();
    if (currentChar == '\\' &
         (singleQuoted || doubleQuoted)) {
        // skip character following \ within quotes
        reader.nextChar();
        currentChar = reader.nextChar();
    }
    if (currentChar == ')
        err.jspError(start, "jsp.error.unterminated", typeEL);
    if (currentChar == ')')
        doubleQuoted = !doubleQuoted;
    if (currentChar == '
        singleQuoted = !singleQuoted;
} while (currentChar != '}
         || (singleQuoted || doubleQuoted));
```

Listing A.105: ./org/apache/lucene/index/IndexWriter.java

```java
while(writeThread != null || readCount > 0) {
    doWait();
}
```

Listing A.106: ./org/apache/lucene/index/IndexWriter.java

```java
while(readCount > upgradeCount || writeThread != null) {
    doWait();
}
```

Listing A.107: ./org/apache/lucene/index/IndexWriter.java

```java
while(pendingMerges.size() > 0 || runningMerges.size() > 0) {
    doWait();
}
```

Listing A.108: ./org/apache/lucene/index/DocumentsWriter.java

```java
while (!closed &
         ((state != null &
           !state.isIdle) ||
           pauseThreads != 0 ||
           flushPending ||
           aborting)) {
    try {
        wait();
    } catch (InterruptedException ie) {
        // In 3.0 we will change this to throw
        // InterruptedException instead
        Thread.currentThread().interrupt();
        throw new RuntimeException(ie);
    }
}
```

Listing A.109: ./org/apache/lucene/search/spanss/NearSpansOrdered.java

```java
private boolean toSameDoc() throws IOException {
    Arrays.sort(subSpansByDoc, spanDocComparator);
    int firstIndex = 0;
    int maxDoc = subSpansByDoc[subSpansByDoc.length - 1].doc();
    while (subSpansByDoc[firstIndex].doc() != maxDoc) {
        if (!subSpansByDoc[firstIndex].skipTo(maxDoc)) {
            more = false;
            inSameDoc = false;
        } else {
            inSameDoc = true;
            firstIndex = (subSpansByDoc[firstIndex].doc() == maxDoc) ?
                          firstIndex + 1 :
                          Arrays.binarySearch(subSpansByDoc, firstIndex,
                                               maxDoc, docComparator);
maxDoc = subSpansByDoc[firstIndex].doc();
if (++firstIndex == subSpansByDoc.length) {
    firstIndex = 0;
}
}

for (int i = 0; i < subSpansByDoc.length; i++) {
    assert (subSpansByDoc[i].doc() == maxDoc)
    + "NearSpansOrdered.toSameDoc() spans " + subSpansByDoc[i].doc()
    + " but should be at doc " + subSpansByDoc[i].doc();
    }

inSameDoc = true;
return true;
}

private boolean stretchToOrder() throws IOException {
    matchDoc = subSpans[0].doc();
    for (int i = 1; inSameDoc && (i < subSpans.length); i++) {
        while (! docSpansOrdered(subSpans[i-1], subSpans[i])) {
            if (! subSpans[i].next()) {
                inSameDoc = false;
                more = false;
                break;
            } else if (matchDoc != subSpans[i].doc()) {
                inSameDoc = false;
                break;
            }
        }
    }
    return inSameDoc;
}

private boolean shrinkToAfterShortestMatch() throws IOException {
    matchStart = subSpans[subSpans.length - 1].start();
    matchEnd = subSpans[subSpans.length - 1].end();
    possibleMatchPayloads = new HashSet();
    if (subSpans[subSpans.length - 1].isPayloadAvailable()) {
        possibleMatchPayloads.addAll(subSpans[subSpans.length - 1].getPayload());
    }
    Collection possiblePayload = null;
    int matchSlop = 0;
    int lastStart = matchStart;
    int lastEnd = matchEnd;
    for (int i = subSpans.length - 2; i >= 0; i--) {
        Spans prevSpans = subSpans[i];
        if (collectPayloads && prevSpans.isPayloadAvailable()) {
            Collection payload = prevSpans.getPayload();
            possiblePayload = new ArrayList(payload.size());
            possiblePayload.addAll(payload);
        }
    }
    int prevStart = prevSpans.start();
    int prevEnd = prevSpans.end();
    while (true) { // Advance prevSpans until after (lastStart, lastEnd)
        if (! prevSpans.next()) {
            inSameDoc = false;
            more = false;
            break; // Check remaining subSpans for final match.
        } else if (matchDoc != prevSpans.doc()) {
            inSameDoc = false; // The last subSpans is not advanced here.
            break; // Check remaining subSpans for last match in this document.
        } else {
            int ppStart = prevSpans.start();
            int ppEnd = prevSpans.end(); // Cannot avoid invoking .end() here.
            if (! docSpansOrdered(ppStart, ppEnd, lastStart, lastEnd)) {
                break; // Check remaining subSpans.
            }
        }
    }
A.1. WHILE-LOOPS

```java
Listing A.110: ./org/apache/lucene/search/BooleanScorer.java

    do {
        bucketTable.first = null;
        while (current != null) {
            // check prohibited & required
            if ((current.bits & prohibitedMask) == 0 &&
                (current.bits & requiredMask) == requiredMask) {
                if (current.doc >= max)
                    continue;
            } else { // prevSpans still before (lastStart, lastEnd)
                prevStart = ppStart;
                prevEnd = ppEnd;
                if (collectPayloads && prevSpans
                    .isPayloadAvailable()) {
                    Collection payload = prevSpans.
                        getPayload();
                    possiblePayload = new
                        ArrayList(payload.
                            size());
                    possiblePayload.addAll(
                        payload);
                }
            }
            if (collectPayloads && possiblePayload !=
                null) {
                possibleMatchPayloads.addAll(possiblePayload);
            } assert prevStart <= matchStart;
            if (matchStart > prevEnd) { // Only non overlapping
                // spans add to slop.
                matchSlop += (matchStart - prevEnd);
            }
            if (more && (inSameDoc || toSameDoc())) {
                if (stretchToOrder() && shrinkToAfterShortestMatch()) {
                    return true;
                }
            }
        }
    }
```

```java
77 } else { // prevSpans still before (lastStart, lastEnd)
    prevStart = ppStart;
    prevEnd = ppEnd;
    if (collectPayloads && prevSpans
        .isPayloadAvailable()) {
        Collection payload = prevSpans.
            getPayload();
        possiblePayload = new
            ArrayList(payload.
                size());
        possiblePayload.addAll(
            payload);
    }
78 }
79 }
80
81 if (collectPayloads && possiblePayload != null) {
    possibleMatchPayloads.addAll(possiblePayload);
82 } assert prevStart <= matchStart;
83 if (matchStart > prevEnd) { // Only non overlapping
    // spans add to slop.
74 matchSlop += (matchStart - prevEnd);
85 }
86 /\ Do not break on (matchSlop > allowedSlop) here to
87 * make sure
88 * that subSpans[0] is advanced after the match, if any.
89 */
90 matchStart = prevStart;
91 lastStart = prevStart;
92 lastEnd = prevEnd;
93
94 boolean match = matchSlop <= allowedSlop;
95 if (collectPayloads && match && possibleMatchPayloads.size() > 0) {
    matchPayload.addAll(possibleMatchPayloads);
96 }
97 }
98 return match; // ordered and allowed slop
99 }
100 while (more && (inSameDoc || toSameDoc())) {
101 if (stretchToOrder() && shrinkToAfterShortestMatch()) {
102 return true;
103 }
104 }
```

APPENDIX A. LOOPS USED IN THE DETAILED STUDY

18 if (current.coord >= minNrShouldMatch) {
19     bs.score = current.score * coordFactors[current.coord];
20     bs.doc = current.doc;
21     collector.collect(current.doc);
22 }
23 }
24 current = current.next; // pop the queue
25 }
26
27 if (bucketTable.first != null)
28     current = bucketTable.first;
29     bucketTable.first = current.next;
30     return true;
31 }
32 }
33 // refill the queue
34 more = false;
35 end += BucketTable.SIZE;
36 for (SubScorer sub = scorers; sub != null; sub = sub.next) {
37     int subScorerDocID = sub.scorer.docID();
38     if (subScorerDocID != NO_MORE_DOCS) {
39         more |= sub.scorer.score(sub.collector, end, subScorerDocID);
40     }
41 }
42 }
43 current = bucketTable.first;
44 } while (current != null || more);

Listing A.111: ./org/apache/lucene/search/BooleanScorer.java

1 do {
2     while (bucketTable.first != null) { // more queued
3         current = bucketTable.first;
4         bucketTable.first = current.next; // pop the queue
5     }
6     // check prohibited & required, and minNrShouldMatch
7     if ((current.bits & prohibitedMask) == 0 && (current.bits & requiredMask) == requiredMask && current.coord >= minNrShouldMatch) {
8         return doc = current.doc;
9     }
10 }
11 }
12 // refill the queue
13 more = false;
14 end += BucketTable.SIZE;
15 for (SubScorer sub = scorers; sub != null; sub = sub.next) {
16     Scorer scorer = sub.scorer;
17     sub.collector.setScore(scorer);
18     int doc = scorer.docID();
19     while (doc < end) {
20         sub.collector.collect(doc);
21         doc = scorer.nextDoc();
22     }
23     more |= (doc != NO_MORE_DOCS);
24 }
25 } while (bucketTable.first != null || more);

Listing A.112: ./libs/src/org/apache/tools/ant/util/LineOrientationOutputStream.java

1 while (remaining > 0 && (b[offset] == LF || b[offset] == CR)) {
2     write(b[offset]);
3     offset++;
4     remaining--;
5 }
A.1. WHILE-LOOPS

Listing A.113: ./libsrc/org/apache/tools/ant/taskdefs/SQLExec.java

```java
do {
    if (updateCount != -1) {
        updateCountTotal += updateCount;
    }
    if (ret) {
        resultSet = getStatement().getResultSet();
        printWarnings(resultSet.getWarnings(), false);
        resultSet.clearWarnings();
        if (print) {
            printResults(resultSet, out);
        }
    }
    ret = getStatement().getMoreResults();
    updateCount = getStatement().getUpdateCount();
} while (ret || updateCount != -1);
```

Listing A.114: ./libsrc/org/apache/tools/ant/taskdefs/PumpStreamHandler.java

```java
while ((s == null || !s.isFinished()) && t.isAlive()) {
    t.interrupt();
    t.join(JOIN_TIMEOUT);
}
```

Listing A.115: ./libsrc/org/apache/tools/ant/taskdefs/cvslib/CvsVersion.java

```java
while (haveReadAhead || st.hasMoreTokens()) {
    String currentToken = haveReadAhead ? cachedVersion : st.nextToken();
    haveReadAhead = false;
    if (currentToken.equals("Client:")) {
        client = true;
    } else if (currentToken.equals("Server:")) {
        server = true;
    } else if (currentToken.startsWith("(CVS") && currentToken.endsWith("")) {
        cvs = currentToken.substring(5) + currentToken;
    }
    if (!client && !server && cvs != null && cachedVersion == null && st.hasMoreTokens()) {
        cachedVersion = st.nextToken();
        haveReadAhead = true;
    } else if (client && cvs != null) {
        if (st.hasMoreTokens()) {
            clientVersion = st.nextToken() + cvs;
            client = false;
            cvs = null;
        }
    } else if (server && cvs != null) {
        if (st.hasMoreTokens()) {
            serverVersion = st.nextToken() + cvs;
            server = false;
            cvs = null;
        }
    } else if (currentToken.equals("(client/server)") && !client && !server) {
        client = server = true;
        clientVersion = serverVersion = cachedVersion + cvs;
        cachedVersion = cvs = null;
    }
}
```

Listing A.116: ./libsrc/org/apache/tools/ant/taskdefs/Concat.java
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

```java
private Iterator readerSources;

private Reader getReader() throws IOException {
    if (reader == null && readerSources.hasNext()) {
        reader = factory.getReader(readerSources.next());
        Arrays.fill(lastChars, (char) 0);
        return reader;
    }

    while (getReader() != null || needAddSeparator) {
        if (needAddSeparator) {
            cbuf[off] = eolString.charAt(lastPos++);
            if (lastPos >= eolString.length()) {
                lastPos = 0;
                needAddSeparator = false;
            }
            len--;
            off++;
            amountRead++;
            if (len == 0) {
                return amountRead;
            }
            continue;
        }
        int nRead = getReader().read(cbuf, off, len);
        if (nRead == -1 || nRead == 0) {
            nextReader();
            if (isFixLastLine() && isMissingEndOfLine()) {
                needAddSeparator = true;
                lastPos = 0;
            } else {
                if (isFixLastLine()) {
                    for (int i = nRead;
                         i > (nRead - lastChars.length);
                         --i) {
                        if (i <= 0) {
                            break;
                        }
                        addLastChar(cbuf[off + i - 1]);
                    }
                }
                len -= nRead;
                off += nRead;
                amountRead += nRead;
                if (len == 0) {
                    return amountRead;
                }
            }
        }
    }
    return -1;
}
```

Listing A.117: ./libsrc/org/apache/tools/ant/filters/TailFilter.java

```java
while (line == null || line.length() == 0) {
    line = lineTokenizer.getToken(in);
    line = tailFilter(line);
    if (line == null) {
        return -1;
    }
    linePos = 0;
}
```

Listing A.118: ./libsrc/org/apache/tools/ant/filters/HeadFilter.java

```java
while (line == null || line.length() == 0) {
    line = lineTokenizer.getToken(in);
    if (line == null) {
        return -1;
    }
    line = headFilter(line);
```
A.1. **WHILE-LOOPS**

```java
if (eof) {
    return -1;
}
linePos = 0;
```

Listing A.119: ./libsrc/org/apache/tools/ant/filters/TokenFilter.java

```java
while (line == null || line.length() == 0) {
    line = tokenizer.getToken(in);
    if (line == null) {
        return -1;
    }
    for (Enumeration e = filters.elements(); e.hasMoreElements();)
    {
        line = filter.filter(line);
        if (line == null) {
            break;
        }
    }
    linePos = 0;
    if (line != null) {
        if (tokenizer.getPostToken().length() != 0) {
            if (delimOutput != null) {
                line = line + delimOutput;
            } else {
                line = line + tokenizer.getPostToken();
            }
        }
    }
}
```

Listing A.120: ./libsrc/org/apache/tools/ant/DemuxOutputStream.java

```java
while (remaining > 0 && (b[offset] == LF || b[offset] == CR)) {
    write(b[offset]);
    offset++;
    remaining--;
}
```

Listing A.121: ./com/ibm/icu/text/CollationKey.java

```java
while (m_key[index] < 0 || m_key[index] >= MERGE_SEPARATOR) {
    result[rindex++] = m_key[index++];
}
```

Listing A.122: ./com/ibm/icu/text/CollationParsedRuleBuilder.java

```java
while ((CE & STRENGTH_MASK[strength]) != (prevCE & STRENGTH_MASK[strength]) || (prevContCE & STRENGTH_MASK[strength]) != (contCE & STRENGTH_MASK[strength]))
    && (strength != 0) {
    strength--;
}
```

Listing A.123: ./com/ibm/icu/text/CollationParsedRuleBuilder.java

```java
while ((cei <= 1) < m_utilIntBuffer[0] || cei < m_utilIntBuffer[1] || cei < m_utilIntBuffer[2]) {
    if (cei > 0) {
        value = RuleBasedCollator.CE_CONTINUATION_MARKER;
    } else {
        value = 0;
    }
```
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

```java
if ((cei << 1) < m_utilIntBuffer_[0]) {
    value |= ((ceparts[0] >> (32 - ((cei + 1) << 4))) & 0xFFFF) << 16;
}
if (cei < m_utilIntBuffer_[1]) {
    value |= ((ceparts[1] >> (32 - ((cei + 1) << 3))) & 0xFF) << 8;
}
if (cei < m_utilIntBuffer_[2]) {
    value |= ((ceparts[2] >> (32 - ((cei + 1) << 3))) & 0xF);
}
token.m_CE_[cei] = value;
cei++;
```

Listing A.124: ./com/ibm/icu/text/CollationRuleParser.java

```java
while (m_optionarg_ < optionend && (UCharacter.isWhitespace(rules.charAt(m_optionarg_)) || UCharacterProperty.isRuleWhiteSpace(rules.charAt(m_optionarg_)))) {
    // eat whitespace
    m_optionarg_++;
}
```

Listing A.125: ./com/ibm/icu/util/BasicTimeZone.java

```java
while (!bFinalStd || !bFinalDst) {
    txt = getNextTransition(time, false);
    if (txt == null) {
        break;
    }
    time = txt.getTime();
    TimeZoneRule toRule = txt.getTo();
    int ruleIdx = 1;
    for (; ruleIdx < all.length; ruleIdx++) {
        if (all[ruleIdx].equals(toRule)) {
            break;
        }
    }
    if (ruleIdx >= all.length) {
        throw new IllegalStateException("The rule was not found");
    } 
    if (isProcessed.get(ruleIdx)) {
        continue;
    }
    if (toRule instanceof TimeArrayTimeZoneRule) {
        TimeArrayTimeZoneRule tar = (TimeArrayTimeZoneRule) toRule;
        // Get the previous raw offset and DST savings before the very first start time
        long t = start;
        while (true) {
            txt = getNextTransition(t, false);
            if (txt == null) {
                break;
            }
            if (txt.getTo().equals(tar)) {
                break;
            }
            t = txt.getTime();
        }
        if (txt != null) {
            // Check if the entire start times to be added
            Date firstStart = tar.getFirstStart(txt.getFrom().getRawOffset());
```
A.1. WHILE-LOOPS

```java
39  tzt.getFrom().getDSTSavings();
40  if (firstStart.getTime() > start) {
    // Just add the rule as is
    filteredRules.add(tar);
  } else {
    // Collect transitions after the start time
    long[] times = tar.getStartTimes();
    int timeType = tar.getTimeType();
    int idx;
    for (idx = 0; idx < times.length; idx++) {
      t = times[idx];
      if (timeType == DateTimeRule.STANDARD_TIME) {
        t -= tzt.getFrom().getRawOffset();
      }
      if (timeType == DateTimeRule.WALL_TIME) {
        t -= tzt.getFrom().getDSTSavings();
      }
      if (t > start) {
        break;
      }
    }
    int asize = times.length - idx;
    if (asize > 0) {
      long[] newtimes = new long[asize];
      System.arraycopy(times, idx, newtimes, 0, asize);
      TimeArrayTimeZoneRule newtar =
        new TimeArrayTimeZoneRule(tar.getName(),
                                   tar.getRawOffset(),
                                   tar.getDSTSavings(),
                                   newtimes, tar.getTimeType());
      filteredRules.add(newtar);
    }
  }
```

```java
71  ) {
72    AnnualTimeZoneRule ar = (AnnualTimeZoneRule) toRule;
73    Date firstStart = ar.getFirstStart(tzt.getFrom());
74    tzt.getFrom().getDSTSavings();
75    if (firstStart.getTime() == tzt.getTime()) {
      // Just add the rule as is
      filteredRules.add(ar);
    } else {
      // Calculate the transition year
      int[] dfields = new int[6];
      Grego.timeToFields(tzt.getTime(), dfields);
      // Recreate the rule
      AnnualTimeZoneRule newar =
        new AnnualTimeZoneRule(ar.getName(),
                               ar.getRawOffset(),
                               ar.getDSTSavings(),
                               ar.getTimeType(),
                               dfields[0], ar.getEndYear());
      filteredRules.add(newar);
    }
79  ) {
80    // Check if this is a final rule
    if (ar.getEndYear() == AnnualTimeZoneRule.MAX_YEAR) {
      // After both final standard and dst rule are processed
      // exit this while loop.
    }
```
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

92     if (ar.getDSTSavings() == 0) {
93         bFinalStd = true;
94     } else {
95         bFinalDst = true;
96     }
97 }
98 }
99 isProcessed.set(ruleIdx);
100 }

A.2 For-loops

Listing A.126: ./org/eclipse/ui/internal/dialogs/TreeManager.java

1 for (Iterator i = changedItem.parent.children.iterator(); i.hasNext() &&
2     (!checkedFound || !uncheckedFound); ) {
3     TreeItem item = (TreeItem) i.next();
4     switch(item.checkState) {
5         case CHECKSTATE_CHECKED:
6             checkedFound = true;
7             break;
8         case CHECKSTATE_GRAY:
9             checkedFound = uncheckedFound = true;
10            break;
11         case CHECKSTATE_UNCHECKED:
12             uncheckedFound = true;
13             break;
14     }
15 }

Listing A.127: ./org/eclipse/ui/internal/navigator/NavigatorContentServiceDescriptionProvider.java

1 for (int i = 0; i < provider.length && (message == null || message.length() == 0); i++) {
2     if (provider[i] instanceof ICommonLabelProvider) {
3         message = ((ICommonLabelProvider) providers[i]).getDescription(target);
4     }
5 }

Listing A.128: ./org/eclipse/jdt/internal/formatter/Scribe.java

1 for (int idx=0, ptr=0; idx<max || (text.htmlNodesPtr != −1 && ptr <=
2     text.htmlNodesPtr); idx++) {
3     // append text to buffer realigning with the line length
4     int end = (idx > max) ? text.sourceEnd : (int) (text.separators[idx] >>> 32);
5     int nodeKind = 0; // text break
6     if (text.htmlNodesPtr >= 0 && ptr <= text.htmlNodesPtr && end >
7         text.htmlNodes[ptr].sourceStart) {
8         FormatJavadocNode node = text.htmlNodes[ptr];
9         FormatJavadocText htmlTag = node.isText() ?
10            (FormatJavadocText) node : null;
11         int newLines = htmlTag == null ? 0 : htmlTag.linesBefore +
12            (linesAfter > newLines) {
13             newLines = linesAfter
14             if (newLines > 1 && clearBlankLines) {
15                 if (idx < 2 || (text.htmlIndexes[idx−2] & JAVADOC_TAGS_ID_MASK) !=
16                     JAVADOC_CODE_TAGS_ID) {
17                     newLines = 1;
18                 }
19             }
20         }
\begin{algorithm}
\begin{verbatim}
if (textStart < previousEnd) {
    addReplaceEdit(textStart, previousEnd, buffer.toString());
}

boolean immutable = node.isImmutable();
if (newLines == 0) {
    newLines = printJavadocBlockNodesNewLines(block, node, previousEnd);
}

int nodeStart = node.sourceStart;
if (newLines > 0 || (idx > 1 && nodeStart > (previousEnd +1))) {
    printJavadocGapLines(previousEnd+1, nodeStart-1, newLines, clearBlankLines, false, null);
}
if (newLines > 0) textOnNewLine = true;
buffer.setLength(0);
if (node.isText()) {
    if (immutable) {
        // do not change immutable tags, just increment column
        if (textOnNewLine && this.commentIndentation != null) {
            addInsertEdit(node.sourceStart, this.commentIndentation);
            this.column += this.commentIndentation.length();
        }
        printJavadocImmutableText(htmlTag, block, textOnNewLine);
        this.column += getTextLength(block, htmlTag);
        linesAfter = 0;
    } else {
        linesAfter = printJavadocHtmlTag(htmlTag, block, textOnNewLine);
    }
    nodeKind = 1; // text
} else {
  
    if (textOnNewLine && this.commentIndentation != null) {
        addInsertEdit(node.sourceStart, this.commentIndentation);
        this.column += this.commentIndentation.length();
    }
    printJavadocBlock((FormatJavadocBlock)node);
    linesAfter = 0;
    nodeKind = 2; // block
}
textStart = node.sourceEnd+1;
ptr++;
if (idx > max) {
    return linesAfter;
}
else {
    if (idx > 0 && linesAfter > 0) {
        printJavadocGapLines(previousEnd+1, nextStart-1, linesAfter, clearBlankLines, false, buffer);
        textOnNewLine = true;
    }
    boolean needIndentation = textOnNewLine;
    if (idx > 0) {
        if (!needIndentation && text
            .isTextAfterHtmlSeparatorTag(idx-1)) {
            needIndentation = true;
        }
    }
    this.needSpace = idx > 1 && (previousEnd+1) < nextStart;
    // There’s no space between text and html tag or inline block => do not insert space at the beginning
\end{verbatim}
\end{algorithm}
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

of the text
printJavadocTextLine(buffer, nextStart, end, block, idx
==0, needIndentation, idx==*/ opening html tag?*/
|] text.htmlIndexes[|idx-1| ! = -1]);
linesAfter = 0;
if (idx==0) {
    if (isHtmlSeparatorTag) {
        linesAfter = 1;
    }
} else if (text.htmlIndexes[idx-1] ==
    JAVADOC_SINGLE_BREAK_TAGID) {
    linesAfter = 1;
}
// Replace with current buffer if there are several empty lines
between text lines
nextStart = (int) text.separators[idx];
int endLine = Util.getLineNumber(end, this.lineEnds, startLine
-1, this.maxLines);
startLine = Util.getLineNumber(nextStart, this.lineEnds, endLine
-1, this.maxLines);
int linesGap = startLine - endLine;
if (linesGap > 0) {
    if (clearBlankLines) {
        // keep previously computed lines after
    } else {
        if (idx==0 || linesGap > 1 || (idx < max &&
            nodeKind==1 && (text.htmlIndexes[idx-1] &
            JAVADOC_IMMUTABLE_TAGS_ID) !=
            JAVADOC_SINGLE_BREAK_TAGID)) {
            if (linesAfter < linesGap) {
                linesAfter = linesGap;
            }
        }
    }
}
}
textOnNewLine = linesAfter > 0;
// print <pre> tag
if (isCode) {
    int codeEnd = (int) (text.separators[max] >>> 32);
    if (codeEnd > end) {
        if (this.formatter.preferences.
            comment_format_source) {
            if (textStart < end) addReplaceEdit(
                textStart, end, buffer.toString());
            // See whether there's a space before
            the code
        }
    }
    if (linesGap > 0) {
        int lineStart = this.scanner.
            getLineStart(startLine);
        if (nextStart > lineStart) { //
            if code starts at the line,
                then no leading space is
            needed:
                this.scanner.resetTo(
                    lineStart, nextStart-1);
                    try {
                        int token = this
                            .scanner.
                            getNextToken();
                        if (token ==
                            TerminalTokens
                            .TagNameWHITESPACE)
                            {// skip
                            indentation
                                token =
                            this
A.2. FOR-LOOPS

```java
scanner
getNextToken();

if (token ==
    TerminalTokens.TokenNameMULTIPLY)
    nextStart =
        this
        .scanner
        .currentPosition;

catch (InvalidInputException iiie) {
    // skip
}

// Format gap lines before code
int newLines = linesGap;
if (newLines == 0) newLines=1;
this.needSpace = false;
printJavadocGapLines(end+1, nextStart-1, newLines, false/* clear first blank lines inside <pre> tag as done by old formatter */ , false, null);

// Format the code
printCodeSnippet(nextStart, codeEnd, linesGap);

// Format the gap lines after the code
nextStart = (int) text.separators[max];
printJavadocGapLines(codeEnd+1, nextStart-1, 1, false/* clear blank lines inside <pre> tag as done by old formatter */ , false, null);
return 2;
}
else {
    nextStart = (int) text.separators[max];
    if ((nextStart-1) > (end+1)) {
        int line1 = Util.getLineNumber(end+1, this.lineEnds, startLine-1, this.maxLines);
        int line2 = Util.getLineNumber(nextStart-1, this.lineEnds, line1-1, this.maxLines);
        int gapLines = line2-line1-1;
        printJavadocGapLines(end+1, nextStart-1, gapLines, false/* never clear blank lines inside <pre> tag */ , false, null);
        if (gapLines > 0) textOnNewLine = true;
    }
    return 1;
}
// store previous end
previousEnd = end;
```
Listing A.129: ./org/eclipse/jface/viewers/AbstractTableViewer.java

```java
for (int column = 0; column < columnCount || column == 0; column++) {
    ViewerColumn columnViewer = getViewerColumn(column);
    ViewerCell cellToUpdate = updateCell(viewerRowFromItem, column, element);

    // If the control is virtual, we cannot use the cached cell object. See bug 188663.
    if (isVirtual) {
        cellToUpdate = new ViewerCell(cellToUpdate.getViewerRow(), cellToUpdate.getColumnIndex(), element);
    }

    columnViewer.refresh(cellToUpdate);

    // clear cell (see bug 201280)
    updateCell(null, 0, null);

    // As it is possible for user code to run the event loop check here.
    if (item.isDisposed()) {
        unmapElement(element, item);
        return;
    }
}
```

Listing A.130: ./org/eclipse/jface/viewers/AbstractTableViewer.java

```java
for (int column = 0; column < columnCount || column == 0; column++) {
    ViewerColumn columnViewer = getViewerColumn(column);
    ViewerCell cellToUpdate = updateCell(viewerRowFromItem, column, element);

    // If the control is virtual, we cannot use the cached cell object. See bug 188663.
    if (isVirtual) {
        cellToUpdate = new ViewerCell(cellToUpdate.getViewerRow(), cellToUpdate.getColumnIndex(), element);
    }

    columnViewer.refresh(cellToUpdate);

    // clear cell (see bug 201280)
    updateCell(null, 0, null);

    // As it is possible for user code to run the event loop check here.
    if (item.isDisposed()) {
        unmapElement(element, item);
        return;
    }
}
```

Listing A.131: ./org/eclipse/jface/viewers/TableTreeViewer.java

```java
for (int column = 0; column < columnCount || column == 0; column++) {
    String text = null; // NON-NLS-18
    Image image = null;
    if (tprov != null) {
        text = tprov.getColumnText(element, column);
        image = tprov.getColumnImage(element, column);
    } else {
        if (column == 0) {
            ViewerLabel updateLabel = new ViewerLabel(item.getText(), item.getImage());
            buildLabel(updateLabel, element);
        }
```
A.2. FOR-LOOPS

Listing A.132: ./org/eclipse/jface/viewers/ColumnViewer.java

```java
for (int i = 0; i < count || i == 0; i++) {
    Widget owner = getColumnViewerOwner(i);
    if (owner != null && !owner.isDisposed()) {
        ViewerColumn column = (ViewerColumn) owner.getData(ViewerColumn.COLUMN_VIEWER_KEY);
        if (column != null) {
            EditingSupport e = column.getEditingSupport();
            // Ensure that only EditingSupports are wiped that are
            // setup
            // for Legacy reasons
            if (e != null && e.isLegacySupport()) {
                column.setEditingSupport(null);
            }
        }
    }
}
```

Listing A.133: ./org/eclipse/osgi/internal/resolver/StateHelperImpl.java

```java
for (int i = 0; i < toSort.length && (hostIndex == -1 || fragIndex == -1); i++) {
    if (toSort[i] == host) {
        hostIndex = i;
    } else if (toSort[i] == fragment) {
        fragIndex = i;
    }
}
```

Listing A.134: ./org/apache/lucene/search/SloppyPhraseScorer.java

```java
for (int pos = start; pos <= next || !tpsDiffer; pos = pp.position) {
    start = pos;
    if (pos < next && tpsDiffer) {
        // advance pp to min window
        done = true;
        // ran out of a term — done
    }
    if (!pp.nextPosition()) {
```

// As it is possible for user code to run the event
// loop check here.
if (item.isDisposed()) {
    unmapElement(element, item);
    return;
}
text = updateLabel.getText();
image = updateLabel.getImage();
APPENDIX A. LOOPS USED IN THE DETAILED STUDY

6  break;
7  PhrasePositions pp2 = null;
8  tpsDiffer = !pp.repeats || (pp2 = termPositionsDiffer(pp)) == null;
9  if (pp2 != null && pp2 == pp)
10     pp = flip(pp, pp2); // flip pp to pp2
12 }