The Treewidth of Java Programs*

Jens Gustedt† Ole A. Mæhle ‡ Jan Arne Telle§

Abstract

Intuitively, the treewidth of a graph G measures how close G is to being a tree. The lower the treewidth, the faster we can solve various optimization problems on G, by dynamic programming along the tree structure. In the paper *M. Thorup, All Structured Programs have Small Tree-Width and Good Register Allocation* [8] it is shown that the control-flow graph of any `goto`-free C program is at most 6. This result opened for the possibility of applying the dynamic programming bounded treewidth algorithms to various compiler optimization tasks. In this paper we explore this possibility, in particular for Java programs.

We first show that even if Java does not have a `goto`, the labelled `break` and `continue` statements are in a sense equally bad, and can be used to construct Java programs that are arbitrarily hard to understand and optimize.

For Java programs lacking these labelled constructs Thorup's result for C still holds, and in the second part of the paper we analyze the treewidth of label-free Java programs empirically. We do this by means of a parser that computes a tree-decomposition of the control-flow graph of a given Java program. We report on experiments running the parser on several of the Java API packages, and the results tell us that on average the treewidth of the control-flow graph of these Java programs is no more than 2.7. This is the first empirical test of Thorup's result, and it confirms our suspicion that the upper bounds of treewidth 6, 5 and 4 are rarely met in practice, boding well for the application of treewidth to compiler optimization.

1 Background

Most structured language constructs such as `while`-loops, `for`-loops and `if-else` allow programs to be recursively decomposed into basic blocks with a single entry and `exit` point, see [1]. Such a decomposition corresponds to a series-parallel decomposition of the control-flow-graph of the program, see [6], and can

*Part of this research was made possible by visiting funds from *The Norwegian Research Council* and the *Région Lorraine*

†LORIA & INRIA Lorraine, France. `Jens.Gustedt@loria.fr`

‡University of Bergen, Norway. `olem@ii.uib.no`

§University of Bergen, Norway. `telle@ii.uib.no`

ease static optimization tasks like register allocation, see [5]. On the other hand, with constructs such as the infamous `goto`, and also short-circuit evaluation of boolean expressions and multiple `exit`, `break`, `continue`, or `return` statements, this nice decomposition structure is ruined, see [5].

However, M. Thorup has shown in a recent article '*All Structured Programs have Small Tree-Width and Good Register Allocation*', see [8], that except for the `goto`, the other constructs listed above do allow for a related decomposition of the control-flow-graph of the program. For each of those language constructs, it was basically shown that regardless of how often they are used, they cannot increase the treewidth of the control-flow graph by more than one. Treewidth is a parameter that measures the 'treeness' of a graph, see [7]. Since a series-parallel graph has treewidth 2, this means that the control-flow-graphs of `goto`-free Algol and Pascal programs have treewidth ≤ 3 (add one for short-circuit evaluation), whereas `goto`-free C programs have treewidth ≤ 6 (add also for multiple `exits` and `continues` from loops and multiple `returns` from functions). Moreover, the related tree-decomposition is easily found while parsing the program, and this structural information can then, as with series-parallel graphs, be used to improve on the quality of the compiler optimization, see *e.g.* [2, 3, 8].

Informally, a graph of treewidth k can be decomposed by taking subsets of vertices of size $\leq k+1$, called bags, as nodes of a tree, in such a way that: 1) any edge of the graph has both endvertices in some bag, and 2) the bags containing any given vertex induce a connected subtree. Most NP-hard graph problems can be solved in linear time when restricted to a class of graphs for which there exists a constant k such that for each graph G in the class the treewidth of G is at most k. Such a linear-time algorithm on G will proceed by dynamic programming on its related width-k tree-decomposition, see [9]. The smaller the value of k, the faster the algorithm, so for the small bounds mentioned above, these algorithms may be quite useful. With unrestricted use of `gotos` one can for any value of k write a program whose control-flow graph has treewidth greater than k, so that the corresponding class of graphs does not have bounded treewidth. These results seem to imply that `gotos` are harmful for static analysis tasks. `Gotos` were originally considered harmful for readability and understanding of programs, see Dijkstra's famous article [4], and languages like Modula-2 and Java have indeed banned their use. Modula-2 instead provides the programmer with multiple `exits` from loops and multiple `returns` from functions with the pleasant consequence that all control-flow-graphs of Modula-2 programs have treewidth ≤ 5. In the paper by Thorup [8], the above-mentioned bounds on the treewidth of `goto`-free Algol, Pascal, C and Modula-2 are all given, but no mention is made of Java.

For a complete proof of the connection between treewidth and structured programs, we refer to [8]. However, by implementing Thorup's result in a Java parser we have gained a good intuition for this connection, which we summarize in the following paragraph.

The fact that control-flow graphs of programs that do not contain any Flow-Affecting Constructs (FACs) such as `goto/break/continue/return` and short-circuit evaluation of boolean expressions are series-parallel, and hence have treewidth 2, is based on the fact that all programming constructs/blocks of such programs have a single entry point and a single exit point, corresponding with the 2 terminals in the series and parallel operations. Also we know that allowing `goto`'s all hope is lost, the control-flow graph can have arbitrarily high treewidth. However, for each of the 'label-free' FACs, `continue/break/return` and short-circuit evaluation, one can show that they respect the nesting structure of the program and also that there exists a tree-decomposition respecting this nesting structure. We take `continue` as an example. The explanation for the other label-free FACs will be very similar. A `continue` statement is related to a certain loop-block (while-loop or for-loop) of a program, and the target of this `continue` is the first statement of the loop-block. Any other `continue` related to this loop-block has the same target. All the statements of a loop-block B, including the target of a `continue`, can be found in the bags of a subtree T_B, and for a nested loop-block N inside of B again we find a similar subtree T_N of T_B, such that the bags of T_N contain no other statements from B except those of N and targets of any `continue`s related to N. Thus, to show Thorup's result it suffices to take the width-2 tree-decomposition whose subtree structure reflects the nesting structure and for each label-free FAC added to the language, simply expand this tree-decomposition by, for each block B, taking the statement which is the target of the FAC at the outer level of block B, and adding it to every bag of T_B that does not belong to an inner nested block. See Figure 2 for an example, as produced by our parser implementing Thorup's result. We thus increase the treewidth by 1 for each label-free FAC added to the language, regardless of how many times each one is used in a program.

Obviously the introduction of labels make things more complicated in that regard, since each label introduces an extra jump target for both `break` and `continue` statements. In Section 2 we will prove technically that by introducing labels even in the restricted form as for Java with `break` and `continue` statements, the corresponding class of control-flow graphs does not have bounded treewidth, and things are in this sense as bad as if `goto`s were allowed. This is in contrast to the introduction of `case`-labels for which it is well known that they don't augment the treewidth by more than one, see [8]. It is also clear that the programming technique of the example we give can be misused to construct Java code that is arbitrarily challenging to understand.

In Section 3 we present our findings on the empirical study of the treewidth of label-free Java programs. In summary, the results are positive, showing an average treewidth of only 2.5. We also discuss the programming examples that have higher treewidth.

2 Not all Java Programs are Structured

As compensation for the lack of a `goto`, the designers of Java decided to add the *labelled* `break` and `continue` statements. The latter two allow labelling of loops and subsequent jumping out to any prelabelled level of a nested loop. Java also contains exception handling, but we don't take this into account here.

```
continue1    l1:while (maybe) {
continue2    l2:   while (maybe) {break l1;
continue3    l3:      while (maybe) {break l1; break l2; continue l1;
innerloop                while (maybe) {break l1; break l2; break l3;
innerloop                               continue l1; continue l2; }
remainder3               break l1; break l2; break l3; continue l1; continue l2;}
break3                break l1; break l2; continue l1;}
break2             break l1;}
break1
```

Figure 1: Skeleton of a Java program whose control-flow graph has treewidth ≥ 7. `Break` and `continue` statements should be conditional, but for the sake of simplicity this has been left out. The left column, in bold font, gives the names of contracted nodes of the control-flow-graph.

The main reason being that the interplay between optimization and exception handling (not only for Java) is quite unclear. On the compiler builder's side, the specification of Java doesn't tell too much about the actual implementation that is expected for exception handling, nor is there any emphasis on performance of the resulting code. Thus, a compiler optimization task like register allocation would apply only to exception-free execution of methods, executing exception-handling without any preallocation of registers.

In the original '*Go To Statement Considered Harmful*'-article, [4], what was in fact specifically objected to by Dijkstra was the proliferation of labels that indicate the target of `goto`s, rather than the `goto`s themselves. In fact, based on the results in this section, that article could aptly have been titled 'Labels Considered Harmful'. We show that, for any value of k, using only k labels, we can construct a Java program whose control-flow graph has treewidth $\geq 2k+1$.

We will view the edges of the control-flow-graph as being undirected. Contracting an edge uv of a graph simply means deleting the endpoints u and v from the graph and introducing a new node whose neighbors are the union of the neighbors of u and v. A graph containing a subgraph that can be contracted to a complete graph on k nodes is said to have a clique minor of size k, and is well-known to have treewidth at least $k-1$, see [7].

The labelled `break` and `continue` statements in Java allows the programmer to label a loop and then make a jump from a loop nested inside the labelled loop. In the case of a `continue` the jump is made to the beginning of the labelled loop, and in the case of a `break` the jump is made to the statement following the labelled loop. In the right-hand side of Figure 1 we show a listing of part of a Java program, with labels l1, l2 and l3, whose control-flow-graph can be contracted to a clique on 8 nodes.

For simplicity we have chosen this code fragment that is obviously not real-life code, though it could easily be augmented to become more natural. For example, `break`s and `continue`s could be `case` statements of a `switch`.

Each of the 8 contracted nodes will naturally correspond to some lines of the corresponding Java program. Each of the 3 first lines of the listed code correspond to a node called, respectively, *continue1*, *continue2* and *continue3*, since they form the targets of the respective `continue` statements labelled l1, l2 and l3. The 4th and 5th lines of the code together form a node that we call *innerloop*, whereas the 6th line we call *remainder3* as it forms the remainder of the loop labelled l3. Lines 7 and 8 of the listing correspond to nodes that we call *break3* and *break2*, respectively, as they form the target of the `break` statements with labels l3 and l2. The target of the `break` labelled l1 is whatever statement that follows the listed code and it will be called *break1*, forming the eighth node.

It should be clear that each of these 8 nodes are obtained by contracting a connected subgraph of the control-flow-graph of the program. We now show that they form a clique after contraction, by looking at them in the order *innerloop, remainder3, continue3, break3, continue2, break2, continue1, break1* and arguing that each of them is connected to all the ones following it in the given order. Firstly, the node *innerloop* is connected to all the other nodes, as the control flows from it into *remainder3* when its loop entry condition evaluates to false, control flows naturally into *innerloop* from *continue3* and for each of the other 5 nodes *innerloop* contains the labelled `break` or continue statement targeting that node. Next, *remainder3* is connected to *continue3* as this is the natural flow of control, and *remainder3* contains the labelled `break` or `continue` statement targeting each of the other 5 nodes following it in the given order. The argument for the remaining nodes follows a similar line of reasoning. Morever, in the same style a larger code example can be made consisting of a method with k labels, a loop nesting depth of $k+1$ and a clique minor of size $2k+2$. The program lines following the line labelled l_k will for this larger example be:

l_k: `while` (maybe) {`break` l1; ... `break` l_{k-1}; `continue` l1; ... `continue` l_{k-2};
  `while` (maybe) {`break` l1; ... `break` l_k; `continue` l1; ... `continue` l_{k-1}; }
  `break` l1;...;`break` l_k; `continue` l1;... `continue` l_{k-1};}
 `break` l1; ... `break` l_{k-1}; `continue` l1; ... `continue` l_{k-2}; }

Theorem 1 *For any value of $k \geq 0$ there exists a Java method with k labels and nesting depth $k-1$ whose control-flow-graph has treewidth $\geq 2k+1$.*

3 Treewidth of Actual Java Programs

If we restrict our focus to Java programs without labels, what can we say about the treewidth? The flow-affecting constructs available in Java are the same as those in C, with the exception that Java does not support the use of the `goto` statement. Thus from Thorup [8] we get the theoretical result that no control-flow graph of such programs have treewidth higher than 6. For a given

Package Name	# Methods	Avg. Treewidth	% tw 2	% tw 3	% tw 4	% tw 5
java.lang	604	2.73	27	73	1	0
java.lang.reflect	50	2.86	14	86	0	0
java.math	96	2.94	7	90	3	0
java.net	279	2.72	31	66	3	1
java.io	620	2.56	47	49	4	0
java.util	990	2.68	32	68	1	0
java.util.jar	93	2.73	28	71	1	0
java.util.zip	157	2.55	45	55	0	0
java.awt	1411	2.66	34	65	1	0
java.awt.event	71	2.74	25	75	0	0
java.awt.geom	527	2.71	30	69	1	0
java.awt.image	623	2.69	30	70	1	0
javax.swing	3400	2.62	39	60	1	0
javax.swing.event	87	2.63	37	63	0	0
javax.swing.tree	379	2.65	35	64	1	0
Total:	9387	Tot. Avg: 2.7				

Table 1: Treewidth of Java API packages

Java method to achieve this high bound, it must contain, in addition to short-circuit evaluation, the flow-affecting constructs `break`, `continue` and `return`. However this is far from sufficient; for the width of the tree-decomposition to be raised by one for each of the constructs they need to "interfere" in the tree-decomposition, i.e. a bag in the decomposition must be affected by all the abovementioned constructs. This gives rise to the natural question of what we can *expect* the treewidth of a given Java method to be. To answer this question we implement a parser that takes as input programs written in Java, computes the corresponding tree-decomposition by Thorup's technique and thereby finds an upper bound on the treewidth of the control-flow graph.

3.1 Treewidth of Java API Packages

The classes of the API are organized in Java packages such as *java.io* and *java.util*. Thus the API is analyzed package-wise. Results from the tests are summarized in Table 1. The four rightmost coloumns shows the percentage-wise distribution of the methods with regard to treewidth, rounded to the nearest integer (except for values below 1, which is rounded to the nearest decimal). For example, package *java.lang* contains 604 methods. 27% of the methods have treewidth ≤ 2, 73% have treewidth 3, while only 1% may have treewidth as high as 4. The treewidth values computed by the parser is an *upper bound* on the treewidth of the control-flow graph of the methods, but we expect this bound to be tight in almost all cases.

While the package test results varies some, the average Java API package

Application Name	# Methods	Avg. Treewidth	% tw 2	% tw 3	% tw 4	% tw 5
MAW D&A	391	2.51	49	51	0	0
MAW D&P	458	2.48	52	47	1	0
JAMPACK	260	2.6	41	58	1	0
Linpack	13	2.69	38	54	8	0
JIU	1001	2.53	48	51	1	0
Scimark2	57	2.59	40	60	0	0
JDSL	955	2.67	307	648	0	0
Total:	3135	Tot. Avg: 2.58				

Table 2: Treewidth of Java application programs

Application Name	Developer	Brief Description
MAW D&A	Mark Allen Weiss	Datastructures/Algorithms
MAW D&P	Mark Allen Weiss	Datastructures/Problem Solving
JAMPACK	G.W. Stewart	JAva Matrix PACKage
Linpack	Jack Dongarra, et.al.	Numerical Computation
JIU	Marco Schmidt	Image processing
Scimark2	Roldan Pozo et.al.	FFT ++
JDSL	Goodrich, Tamassia, et.al.	Data Structures Library

Table 3: Brief description of tested Java applications

typically has a distribution of the methods as follows. 20-40% of the methods have treewidth 2 and 60-80 % have treewidth 3. Only rarely are there more than 2% of the methods that cannot be guaranteed to have treewidth ≤ 3. Surprisingly, only one of the tested Java API packages have methods of treewidth 5, which is the *java.net* package. As it turns out this is only one method, namely *receive(DatagramPacket)* of class *java.net.DataSocket*. A closer look at this method is taken in Section 3.4.

3.2 Treewidth of Java Application Programs

Next we analyze the treewidth of ordinary Java applications. The programs were mostly found on the internet via search engines like Google (www.google.com). The bounds found are similar to those of the Java API classes. As Table 2 shows the results are similar to those of the Java API. Table 3 displays a short description of the applications. The choice of Java packages tested was based on availability of the source code.

3.3 Commonly Used Flow-Affecting Constructs

The 4 flow-affecting constructs (FACs) of Java are `break`, `continue`, `return` and short-circuit evaluation. We know that the treewidth may not necessarily

Name	% using 0 FACs	% using 1 FACs	% using 2 FACs	% using 3 FACs	% using 4 FACs
java.lang	25	62	11	1	0.3
java.lang.reflect	14	80	6	0	0
java.math	7	76	17	0	0
java.net	31	57	11	2	0
java.io	46	41	10	3	0
java.util	31	58	10	1	0.1
java.util.jar	25	59	15	1	0
java.util.zip	45	51	4	0.6	0
java.awt	34	58	7	0.7	0
java.awt.event	25	55	18	1	0
java.awt.geom	27	60	13	0.6	0
java.awt.image	30	57	11	1	0
javax.swing	39	55	6	0.4	0
javax.swing.event	33	60	7	0	0
javax.swing.tree	35	54	11	0.3	0
Total Avg.	29.8	58.9	10.5	0.8	0.03

Table 4: Flow-Affecting statements usage

increase by more than one even though several flow-affecting constructs are used within the same method. For instance a method containing `break`, `continue` and `return` may perfectly well be of treewidth 3. In fact, a program applying short-circuit evaluation may still have treewidth 2. This section examines what kind of flow-affecting structures are most widely used, and also to what extent flow-affecting constructs are used without increasing treewidth.

First we determine for the Java API packages of Section 3.1 how many of the methods use zero, one, two, three or four of the constructs in question, see Table 4. The first thing we observe is that the first columns of Tables 1 and 4 are almost identical. This is expected; the methods that don't utilize any of the flow-affecting constructs have treewidth 2. The differences between the two tables come from the cases where short-circuit evaluation is used without increasing treewidth.

Looking at Table 1 almost all of the methods have treewidth 2 or 3. Comparing this to Table 4 we see that a number of the methods of treewidth 3 are split between having 1 or 2 flow-affecting constructs. In other words, methods commonly have 2 FACs, but treewidth ≤ 3. This is the case for about 10 percent of the methods in the Java API packages.

Next we analyze specifically what kind of flow-affecting constructs are most commonly used. We begin with the smallest of the applications, *Linpack*, for which we will give a somewhat more detailed description than the rest. Since the program doesn't have more than 13 methods we present all of them in Table 5. Throughout the program neither `break` nor `continue` are used at all, bounding the treewidth to 4. This corresponds nicely to our previous analysis

Method Name	tw	FACs
main()	2	
abs()	3	`return`
second()	3	`return`
run_benchmark()	2	
matgen()	3	`return`
dgefa()	3	`return`
dgesl()	2	
daxpy()	4	short-sircuit, `return`
ddot()	3	short-circuit, `return`
dscal()	2	
idamax()	3	`return`
epslon()	3	`return`
dmxpy()	2	

Table 5: Flow-Affecting statements used in the *Linpack* program

of *Linpack*; one method having treewidth 4, the rest 2 or 3. Furthermore we observe, as expected, that the methods that have no flow-affecting contructs at all are exactly those of treewidth 2, while those that use one FAC are a subset of the methods of treewidth 3. Again we see that utilizing more than one flow-affecting construct doesn't necessarily increase treewidth by more than one, as is the case for method *ddot()*, which has 2 FACs, but treewidth 3.

Presenting data in the same manner as Table 5 from each of the methods of the Java API packages would hardly be suitable. Instead Table 6 shows how often the various flow-affecting constructs are used. We can see that by far the most widely used construct is the `return` statement, which is used by 65.5% of the methods in the Java API. Next, used by 13.2%, follows short-circuit evaluation, whereas `break` and `continue` is only found in 3.6 and 0.3% of the methods, respectively. The last coloumn shows how often the labelled `break`/`continue` statements are found. We see that 11 out of 15 packages do not use them at all, while *java.lang* contain either a labelled `break` or `continue` in 1% of the methods.

3.4 A Java API Method of Treewidth 5

As previously mentioned, only one method was found to have treewidth 5. This was method *receive(DatagramPacket)* found in class *java.net.DataSocket*. It is therefore worth to take a closer look at this particular method, and see if we can decide why it achieves such a high bound. The relevant parts of the method are given in Figure 2, together with the generated tree-decomposition.

The excerpt consists of a `while` statement containing an if-else statement in which the expression utilizes short-circuiting. In addition to that, the `then` block of the expression has a `continue` and the `else` block has a `break` statement. As we know, each of these constructs can increase treewidth by one. Since they

Name	% using return	% using break	% using continue	% using Short-Circuit	% using Labelled break/ continue
java.lang	72	2	1	14	1
java.lang.reflect	86	0	0	6	0
java.math	90	0	0	20	0
java.net	64	3	0.4	16	0
java.io	50	4	0.6	15	0.5
java.util	67	2	0.3	12	0.2
java.util.jar	69	3	1	19	0
java.util.zip	43	4	0	13	0
java.awt	60	3	0.2	12	0
java.awt.event	72	18	0	6	0
java.awt.geom	71	2	0.2	14	0.2
java.awt.image	65	7	0.2	11	0
javax.swing	55	2	0.2	11	0
javax.swing.event	64	3	0	7	0
javax.swing.tree	54	1	0	22	0
Total Avg.	65.5	3.6	0.3	13.2	0.13

Table 6: Flow-affecting statements used in Java API packages

are all used within the same statement there exists a bag for which each of these constructs will increase the width, for a total width of 5, accounted for by the node with 3 children, which has a bag of size 6 in Figure 2.

4 Conclusion

Originally Java was designed to be precompiled to bytecode for the *Java Virtual Machine*, so compiler optimization tasks were then not a main issue. Nevertheless, since `goto`s were considered particularly harmful for the conceptual clarity of a program they were completely banned from the specification of Java, and a labelled `break` and `continue` were added. Nowadays, to speed up applications written in Java, there is a strong demand for compiled *and* optimized Java, and so Java-to-native-machine-code compilers are emerging. In this paper we have shown that such compilers must have certain limits that are already inherent in the language itself. Nevertheless, programs that do not utilize labelled `break`/`continue` statements are of low treewidth on average. The experimental results of this paper justifies further research on how the tree-structure of these control-flow graphs can be utilized to improve various algorithms for compiler optimization tasks like register allocation.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] S. Alstrup, P. Lauridsen, and M. Thorup. Generalized dominators for structured programs. In *Proceedings of the 3rd Static Analysis Symposium*, volume 1145 of *LNCS*, pages 42–51, 1996.

[3] H. Bodlaender, J. Gustedt, and J. A. Telle. Linear-time register allocation for a fixed number of registers and no stack variables. In *Proceedings 9th ACM-SIAM Symposium on Discrete Algorithms (SODA'98)*, pages 574–583, 1998.

[4] E. W. Dijkstra. Go to statement considered harmful. *Comm. ACM*, 11(3):147–148, 1968.

[5] S. Kannan and T. Proebsting. Register allocation in structured programs. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–368, San Francisco, California, 22–24 Jan. 1995.

[6] T. Nishizeki, K. Takamizawa, and N. Saito. Algorithms for detecting series-parallel graphs and *D*-charts. *Trans. Inst. Elect. Commun. Eng. Japan*, 59(3):259–260, 1976.

[7] N. Robertson and P. Seymour. Graph minors II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 1986.

[8] M. Thorup. Structured programs have small tree-width and good register allocation. *Information and Computation*, 142(2), 1998.

[9] J. van Leeuwen. *Handbook of Theoretical Computer Science*, volume A, chapter Graph Algorithms - Classes of graphs, pages 545–551. Elsevier, Amsterdam, 1990.

Method Excerpt:

```
public synchronized void receive( DatagramPacket p )
                              throws IOException
{
    while( true )
    {
        if( connectedAddress.equals( peekAddress ) ||
          (connectedPort != peekPort ) )
        {
            continue;
        }
        else
        {
            break;
        }
    }
  }
}
```

Tree–decomposition:

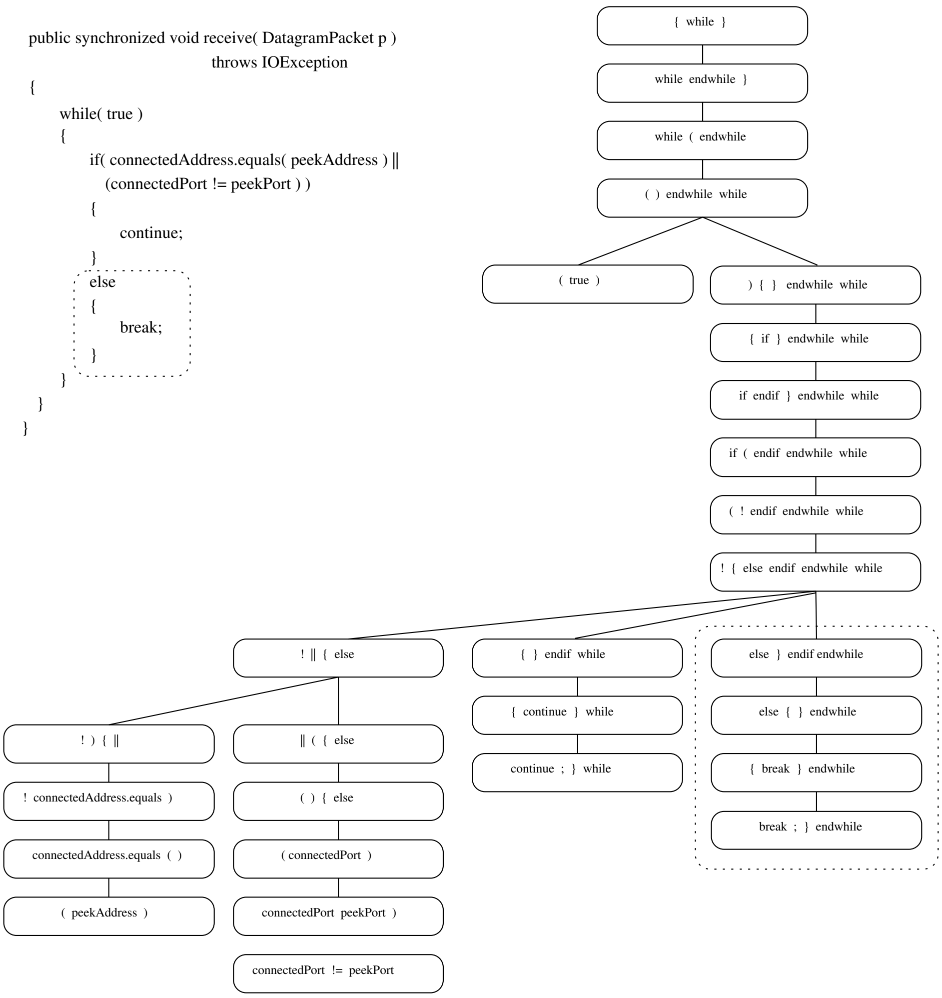

Figure 2: Tree-decomposition of a subset of method *receive(DatagramPacket)* in class *DatagramSocket* of package *java.net*. Note how the indentation blocks (reflecting the scope depth) of the program appear as subtrees in the decomposition. For instance the *else*{ *break;* } block corresponds to the subtree pointed out by the dashed lines in the tree-decomposition.