On the Tree Width of Ada Programs

Bernd Burgstaller[1], Johann Blieberger[1], and Bernhard Scholz[2]

[1] Institute for Computer-Aided Automation, TU Vienna
Treitlstr. 1–3, A-1040 Vienna, Austria
{blieb,bburg}@auto.tuwien.ac.at
[2] School of Information Technologies
Madsen Building, F09
University of Sydney NSW 2006, Australia
scholz@it.usyd.edu.au

Abstract. The tree width of a graph G measures how close G is to being a tree or a series-parallel graph. Many well-known problems that are otherwise NP-complete can be solved efficiently if the underlying graph structure is restricted to one of fixed tree width.
In this paper we prove that the tree width of goto-free Ada programs without labeled loops is ≤ 6. In addition we show that both the use of gotos and the use of labeled loops can result in unbounded tree widths of Ada programs.
The latter result suggested to study the tree width of actual Ada programs. We implemented a tool capable of calculating tight upper bounds of the tree width of a given Ada program efficiently. The results show that most existing Ada code has small tree width and thus allows efficient automatic static analysis for many well-known problems and – as a by-product – most Ada programs are very close to series-parallel programs.

1 Introduction

The notion *tree width* has originally been introduced by Robertson and Seymour [RS83]. Intuitively, the tree width of a graph G measures how close G is to being a tree or a series-parallel graph. In this way the tree width of a tree equals 1 and the tree width of a series-parallel graph is 2. For more general graphs the tree width increases more and more.

Many well-known problems that are otherwise NP-complete can be solved efficiently (i.e. in polynomial time) if the underlying graph structure is restricted to one of fixed tree width (cf. e.g. [ALS91]). See also [Bod93] for a survey. Our particular interest in the tree width of Ada programs stems from the fact that a well-known approach to find the *worst-case execution time (WCET)* of a program ([PK89]) can be determined in linear time if the underlying *control flow-graph (CFG)* has tree width k. This follows easily from [ALS91] where it is proved that all graph properties definable in monadic second-order logic (MS) with quantification over vertex and edge sets can be decided in linear time for classes of graphs of fixed bounded tree width, because the approach to find the WCET mentioned above is expressible in MS.

In [Tho98] Thorup presents a measure called k-complexity that is equivalent to the original definition of tree width given by Robertson and Seymour [RS83]. Based on the notion of k-complexity Thorup shows in [Tho98] that goto-free Algol and Pascal programs have control flow-graphs of tree width ≤ 3, that Modula-2 programs have control flow-graphs of tree width ≤ 5, and that goto-free C programs have control flow-graphs of tree width ≤ 6.

In particular, Thorup's definition can be used for implementing a parser for a certain programming language in order to compute actual data on the tree width of existing code. This method has been used by Gustedt, Mæhle, and Telle to study the tree width of actual Java programs [GMT02]. We discuss this method in Section 2.

The impact of Ada's very rich set of language features on the resulting tree width of programs is described in Section 3. A restricted class of Ada programs is shown in Section 4 to have tree width ≤ 6. However, in general, Ada programs turn out to have non-bounded tree width, which is proved in Section 5.

In this way it is of paramount interest to study actual Ada programs to find statistics on their tree width. Implementation details of our tool are described in Section 6; the results of the study are given and discussed in Section 7. We conclude our paper in Section 8.

2 On the k-Complexity of Graphs

The notion of tree width was introduced by Robertson and Seymour [RS83]. It is based on a tree-decomposition of a graph:

Definition 1. *A tree-decomposition of a graph $G = (V, E)$ is defined by a tree $T = (I, F)$ together with a family $\{W_i\}_{i \in I}$ of subsets of V such that:*

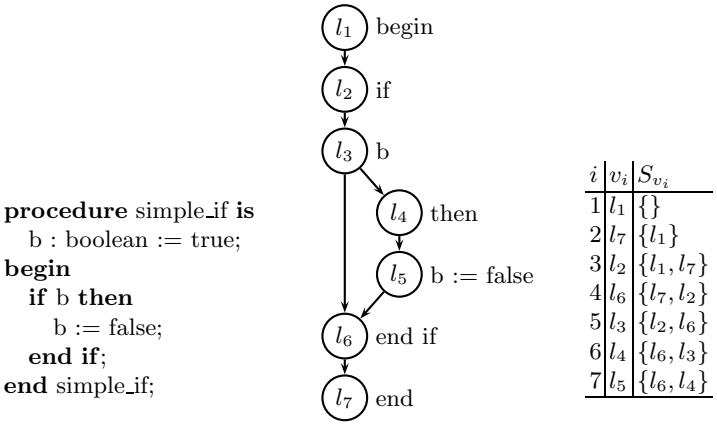
1. $\bigcup_{i \in I} W_i$ *is* V.
2. *for all edges $(v, w) \in E$, there exists an $i \in I$ such that $\{v, w\} \subseteq W_i$.*
3. *for all $i, j, k \in I$, if j is on the path from i to k then $W_i \cap W_k \subseteq W_j$.*

The width of the decomposition is $\max_{i \in I} |W_i| - 1$ and the tree-width of G is the minimal width over all tree-decompositions of G.

Thorup [Tho98] introduced k-complexity of a graph and showed that if a graph is k-complex then the tree width of this graph is k and vice versa. The k-complexity of a graph uses the notion of a listing $L = v_1, \ldots, v_n$ that is an ordered set of the vertices.

Definition 2. *Given a graph G, a* ($\leq k$)-complex listing *is a listing of vertices of G such that for every vertex $v \in V$, there is a set S_v of at most k of the vertices preceding v in the listing, whose deletion from G separates v in G from all the vertices preceding v in the listing. In this case, we say that G is* ($\leq k$)-complex. *The set S_v is referred to as the* separator set *of v in the listing.*

Figure 1 (a) shows a simple program, a corresponding control flow graph (b) and a listing (c) with its separator sets of the vertices. In the example we have



i	v_i	S_{v_i}
1	l_1	$\{\}$
2	l_7	$\{l_1\}$
3	l_2	$\{l_1, l_7\}$
4	l_6	$\{l_7, l_2\}$
5	l_3	$\{l_2, l_6\}$
6	l_4	$\{l_6, l_3\}$
7	l_5	$\{l_6, l_4\}$

Fig. 1. Example: Simple If-Statement

seven control flow nodes. To demonstrate the separator sets we have chosen an arbitrary listing. For this given listing the minimal separator sets can be computed as outlined in Section 4 of Thorup's work [Tho98]. Note that for Definition 2 directed arcs in CFGs have to be replaced with undirected edges.

The importance of k-complexity compared to the original definition of tree width comes from the fact that k-complexity can be computed easily for CFGs based on the syntax of the underlying programming language. In terms of the *Abstract Syntax Tree (AST)* it can be implemented by a simple traversal of the AST.

Definition 3. *Given a graph G, a vertex of the graph and its separator set S_v, we define a S-path from node v to a node $w \in S_v$ to be a path from v to w which does not contain a node $u \in S_v$ $(u \neq w)$. Let G_v be a subgraph of G which contains v and all nodes lying on S-paths from v to nodes in S_v. The nodes of S_v themselves are not contained in G_v.*

In Figure 1 node l_4 has separator set $S_{l_4} = \{l_6, l_3\}$. Graph G_{l_4} contains the nodes l_4 and l_5. Note that l_5 has a greater listing number than l_4. It is easy to check that all nodes of Figure 1 fulfill the condition that G_v only contains vertices whose listing number is greater than the listing number of v.

For the remainder of this section we use the program shown in Figure 2 (a) as a running example. For sake of simplicity we address nodes by their listing number and do not specify the mapping between labels and listing numbers anymore. This simple example consists of 10 nodes and, therefore, there are $10! = 3628800$ different listings according to Definition 2. Generating all possible listings in order to determine the minimum choice with respect to k-complexity constitutes a computationally intractable problem for the general case. For this

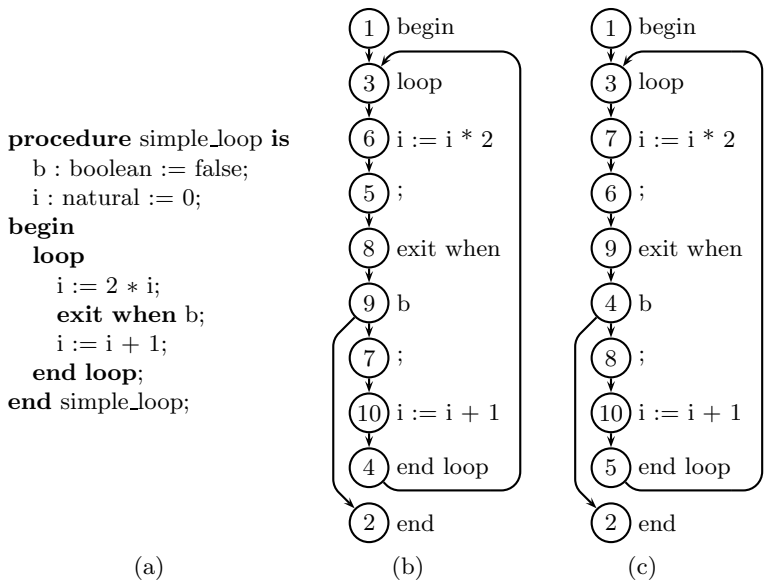


Fig. 2. Example: Simple Loop with Exit Statement

reason Thorup [Tho98] proposes a method for calculating the k-complexity based on the syntax of the underlying programming language. In accordance with Definition 2 this method consists of two steps, namely the derivation of a listing from a given CFG, and the generation of the corresponding separator sets. The following two subsections present an overview of this method. For further details the reader is referred to [Tho98] Section 2.

Derivation of a Listing from a CFG. Each program expressible in a certain programming language can be derived from a starting non-terminal symbol by applying a number of productions. Thorup's central idea is to assign numbers to the terminals of a production when it is applied to a sentential. The numbers are assigned from left to right. It is common to assign numbers also to the semi-colons which separate statements from each other [Tho98].

To return to our example (Figures 2 (a) and 2 (b)), the terminal **begin** of the procedure *simple_loop* gets assigned number 1 and the corresponding terminal **end** gets assigned 2. The next production is the one producing the loop statement. Thus the terminal **loop** gets number 3 and the terminal **end loop** gets 4. The next terminal to consider is the semi-colon after the first statement of the loop body; it gets number 5. The first statement itself, namely `i := 2 * i`, gets assigned 6.

It is not necessary to perform this method on a level deeper than that of "basic" statements such as assignments, null statements, procedure calls and so on. For this method these basic statements are also considered "terminals" of

a production. However it turns out that boolean expressions of if-statements or exit statements have to be examined more carefully because the k-complexity is increased if the boolean expression contains short-circuit operators such as **or else** and **and then** [KP98].

Hence the semi-colon after the exit statements gets assigned 7 and the exit statement itself gets 8. After that the condition of the exit statement is examined. Since in our case it consists of an atom only, this atom **b** gets number 9.

The only remaining statement is the assignment after the exit which gets number 10.

It is clear that generating a k-complex listing by this method can be done in time linear in the size of the program.

Generation of Separator Sets. Thorup's method allows for calculating the separator sets during the generation of the k-complex listing as described above. The separator set of a certain statement (node) can be determined from the CFG predecessors and from the CFG successors of the node. Thorup calls these nodes *potential neighbors*. In fact, neighbors are defined for subgraphs the nodes of which are subtrees of the corresponding AST.

The topmost view of our example consists of the terminal **begin**, the loop-statement, and the terminal **end**. The loop statement consist of nodes 3, 6, 5, 8, 9, 7, 10, and 4 (cf. Figure 2 (b)). The potential neighbors of the loop-statement are therefore node 1, which is the CFG predecessor of this statement, and node 2, its CFG successor. Thorup calls neighbors like the first one *in-neighbors* and those of the second kind *out-neighbors*.

In addition there are also *exit*-neighbors. An exit-neighbor is the target node of the exit-statement of a loop if the exit-condition evaluates to *true*.

In the second view of our example (after applying the production that derives the loop body) we find that the exit-neighbor of the statement sequence between **loop** and **end loop** of the loop body is node 2. The in-neighbor of this statement sequence is node 3 and its out-neighbor is node 4.

In the third view (after the production that derives the first statement of the statement sequence) the in-neighbor of node 6 is node 3 and its out-neighbor is node 5.

Similarly the in-neighbor of the exit-statement (nodes 8 and 9) is node 5 and its out-neighbor is node 7.

Traversing the exit-statement we find that the in-neighbor of node 8 is node 5 and its out-neighbor is node 7. The boolean expression in node 9 has in-neighbor 8 and out-neighbor 7.

Finally, the in-neighbor of node 10 is node 7 and its out-neighbor is node 4.

The separator sets can now be determined based on the type of the statement and its neighbors. For an assignment statement the separator set contains its in- and out-neighbors. In this way, node 6 has separator set $S_6 = \{3, 5\}$.

For an exit-statement the corresponding separator set contains its in-, out-, and exit-neighbors. Hence $S_8 = \{5, 7, 2\}$ and $S_9 = \{8, 7, 2\}$.

For a node corresponding to a semi-colon, the separator set contains its in-, out-, and exit-neighbors. For example $S_5 = \{3, 4, 2\}$.

Summing up we conclude that the separator sets can also be determined in linear time by Thorup's method.

Finally we would like to note that more complex programs require the notion of *stop-neighbors* due to the occurrence of return-statements. In terms of the CFG such a return-statement introduces an edge from the **return** to the **end**.

Discussion of Thorup's Method. Thorup's method is well fitted to determine upper bounds of the tree width for certain programming languages. For example Thorup [Tho98] shows that all Modula-2 programs have tree width ≤ 5 and that goto-free C programs have tree width ≤ 6.

For our running example (cf. Figure 2) Thorup's method calculates a tree width of 3. We did a complete enumeration of all possible listings for this example. Of all 3628800 cases 872 produced a tree width of 3 and 3627928 produced a tree width of 2. So in this case Thorup's method only finds an upper bound for the tree-width!

In the following we discuss some other serious drawbacks of Thorup's method.

1. First, it considers *potential* neighbors and not *actual* neighbors. For example a loop without an exit statement (quite common for non-terminating loops in server tasks) will have potential exit-neighbors. But since there is no exit-statement, the statement succeeding the **end loop** cannot be reached. Thus there is no path from the start node to this node. Nevertheless Thorup's method still lists the start node as an in-neighbor for this "dead" node.
   Employing synthesized attributes [ASU88] for the calculation of the neighbors would enable Thorup's algorithm to derive the *actual* neighbors from the *potential* neighbors. However, the only synthesized attribute used is the listing number (cf. [Tho98]).
2. Second, Thorup's method always assumes that all conditions contain short-circuit expressions. Although he shows how the nodes of CFGs can be renumbered if the boolean expressions do not contain short-circuit expressions, his method silently assumes that conditions consisting of a single atom, are short-circuit. This renumbering in general produces listings with smaller separator sets.
   However, Thorup performs renumbering only on a small number of nodes local to the if-statement. In contrast the renumbering below touches almost all nodes of the loop enclosing the exit statement. This kind of renumbering is not mentioned in [Tho98] and shows again that Thorup's method produces upper bounds only.
   If we introduce the renumbering $6 \to 7$, $5 \to 6$, $8 \to 9$, $9 \to 4$, $7 \to 8$, and $4 \to 5$ in our example (cf. Figure 2 (c)), the separator sets change to: $S_7 = \{3, 6\}$, $S_6 = \{3, 4\}$, $S_9 = \{6, 4\}$, $S_4 = \{3, 2\}$, $S_8 = \{4, 5\}$, $S_{10} = \{8, 5\}$, and $S_5 = \{3, 2\}$. Hence the tree width decreases from 3 to 2 by this renumbering.
   As a side-note we mention that this renumbering also works for more complex exit conditions but the condition must not contain short-circuit expressions.

A simple work-around to get rid of problem (1) is to generate a k-complex listing with Thorup's method, but instead of applying Thorup's neighbor based method use a "greedy" algorithm that calculates the separator set for node v by traversing the paths of graph G_v (Definition 3). This results in a method using time quadratic in the size of the program.

3 Tree Width of Ada Programs

In [Tho98] Thorup studies the following language features that are known to increase the tree width:

- *short-circuit expressions* in conditions of if-statements,
- *multiple returns* from subroutines, and
- *exits* from unlabeled loops.

In addition to the language features studied by Thorup, Ada provides the following features:

- *elsif* and *case statements*,
- both *short-circuit expressions* and *non-short-circuit expressions* in a single boolean expression,
- *for-loops* and *while-loops*,
- *multiple exits* from labeled loops,
- *conditional exits* from loops[3],
- *exceptions* and *exception handlers*, and
- *tasks* and *protected objects*.

Thorup's method can be modified to handle elsif- and case-statements correctly.

Conditions containing both short-circuit expressions and non-short-circuit expressions require a delicate treatment but can also be treated correctly.

While-loops that do not interfere with other language features, in general have a tree width of 3 except when their condition is atomic, i.e., does not contain any of **not**, **and**, **or**, **and then**, and **or else** statements. In this case their tree width is 2. For-loops also result in a tree width of 2.

Multi-exit loops and their consequences are discussed in Section 5.

Conditional exits are handled as if-statements with an unconditional exit in the then branch.

Since our primary interest is in static program analysis, we do not model the exact dynamic semantics of exceptions. Instead we assume that if an exception is raised, it is handled by the nearest statically enclosing exception handler. Note also that we only model explicitly raised exceptions. We therefore do not account for exceptions raised by the runtime system due to faults such as division by zero or failed subtype range checks.

Nevertheless exceptions give rise to a new kind of neighbors, so-called *raise-neighbors*. The raise-neighbor of a certain statement is the node corresponding to the start of the exception handler of the enclosing block.

[3] Thorup uses exit-statements enclosed in if-statements to model conditional exits.

```
procedure Ada_Worst_Case is
  A, B, C, D, E, F, G, H, I, J: boolean:= true;
begin
  if A then return;
  end if;
  if not A then raise constraint_error; end if;
  loop
    if B then
      if C or else D then
        if E then return;
        end if;
        exit when F;
        if not A then raise constraint_error; end if;
      else
        if G then return;
        end if;
        exit when H;
        if not A then raise constraint_error; end if;
      end if;
    end if;
    if I then return;
    end if;
    exit when J;
    if not A then raise constraint_error; end if;
  end loop;
  if not A then raise constraint_error; end if;
exception
  when others =>
     return;
end Ada_Worst_Case;
```

Fig. 3. Example: Restricted Ada Program with Largest Possible Tree Width

Tasks and protected objects are handled correctly, including all different forms of select statements, which are treated like case and if-statements. The abort statement is treated only partially correct. A fully correct treatment would imply that every task must be aware of an abort at any time. We assume that an abort statement is only used to abort the task itself, thus the corresponding CFG has an additional edge from the abort statement to the CFG's end node.

4 The Class of (≤ 6)-Complex Ada Programs

Considering Ada programs without gotos and without labeled loops[4], we know that the tree width of such programs is bounded above by 6. The reason for this is that Ada adds short-circuit expressions, exit statements, multiple return

[4] Gotos and labeled loops will be studied in Section 5.

statements and exceptions to series-parallel programs. Since each of these features adds 1 to the tree width, we get 6 as an upper bound for the tree width of these restricted Ada programs. In fact we can construct a program whose tree width assumes the largest possible value. Such a program is shown in Figure 3.

Program `Ada_Worst_Case` is an extension of a program given in [Tho98]. The program studied there contains a 6-clique which is trivially 5-complex. Our program adds an exception handler and raise-statements which – as can be seen easily – results in a 7-clique being contained in the corresponding contracted CFG[5].

Thus we have proved the following theorem.

Theorem 1. *Ada programs without gotos and without labeled loops are of tree width* ≤ 6. □

5 The Class of Ada Programs with Non-Bounded Tree Width

First of all, Ada has a goto statement. Hence it is possible to write Ada programs whose CFGs contain k-cliques for any k. The tree width of such graphs is $k-1$ and thus can be arbitrarily large.

In the following we restrict our interest to goto-free Ada programs. Statistics for the usage of gotos in Ada programs have been given by Gellerich et al. [GKP96].

Gustedt et al. [GMT02] find that labeled break and continue statements are responsible for the fact that Java programs result in CFGs of non-bounded tree width. In contrast Ada supports some form of labeled break statement, the exit from a labeled loop, but no continue statement. Anyway, we are able to prove that Ada programs result in CFGs of non-bounded tree width.

Theorem 2. *For any value of* $k \geq 0$ *there exists a goto-free Ada program with* k *nested loops such that its control flow-graph has tree width* $\geq k+1$.

Proof. Figure 4 shows four nested loops with several exits statements which have been labeled to facilitate the proof.

In the following we again apply Lemma 6 proved in [Tho98] which can be used to contract nodes of a CFG into one node without increasing the complexity (tree width).

We will show in the following that the statements `I`, `R`, `L3`, `B3`, `B2`, and `B1` form a 6-clique, by looking at them in the above order and arguing that each of them is connected to all the ones following it in the given order.

First, the node `I` is connected to all the other nodes, as the control flows from it to `R` via the last exit-statement, control flows naturally into `I` from `L3`, and for the remaining statements `I` contains exit-statements targeting that node.

[5] A thorough treatment makes use of Lemma 6 of [Tho98] which allows contracting some nodes of a CFG into one node without increasing the complexity (tree width).

```
procedure non_bounded is
  C1, C2, C3, C4 : boolean := true;
begin
  L1: loop
    L2: loop
      L3: loop exit L1 when C1; exit L2 when C2; exit L3 when C3;
        ≪I≫ loop exit L1 when C1; exit L2 when C2; exit L3 when C3;
             exit when C4; end loop;
      ≪R≫ exit L1 when C1; exit L2 when C2; exit L3 when C3; end loop L3;
    ≪B3≫ exit L1 when C1; exit L2 when C2; end loop L2;
  ≪B2≫ exit L1 when C1; end loop L1;
≪B1≫ null;
end non_bounded;
```

Fig. 4. Example: Four Nested Loops

Next, `R` is connected to `L3` as this is the natural flow of control and `R` contains exit-statements targeting the statements following it in the given order. The argument for the remaining statements follows a similar line of reasoning.

Thus the CFG for the example in Figure 4 contains a 6-clique.

It is easy to prove by induction that this is also true for k nested loops. In this case the statements involved are `I`, `R`, `L{k-1}`, `B{k-1}`, ..., `B2`, and `B1`. These form a $k+2$-clique and the theorem is proved. □

Corollary 1. *From Theorem 2 it follows that even goto-free Ada programs do not have bounded tree width.*

Because of Corollary 1 it is of paramount interest to study actual Ada programs to find statistics on their tree width. We have thus decided to implement a (slightly modified) version of Thorup's method for Ada and to calculate the tree width of existing Ada code. As a byproduct our study provides insight into how close Ada programs are to series-parallel programs.

6 Implementation Details

In order to calculate Thorup's k-complexity measure, several approaches can be used:

1. Create the CFG for a given program and use a general purpose algorithm that builds the node listings and separator sets for a general CFG [Bod93].
2. Build a parser for the programming language that – as a byproduct – produces node listings and separator sets.
3. Employ dataflow methods to compute node listings and separator sets.
4. Compute node listings and separator sets by use of an AST built by a compiler.

Approach 1 would have needed to implement a general purpose algorithm such as that mentioned in [Bod93] which we considered an error-prone task. So we did not use this approach.

Approach 2 has been proposed by Thorup [Tho98] and followed by Gustedt et al. [GMT02]. Since however Ada's rich set of language features requires some sort of semantic information in order to calculate the separator sets as exactly as possible, we did not employ this approach.

Approach 3 was also considered error-prone. So we did not use it either.

Instead we extended GNAT and used its AST to calculate the node listings. Because of the semantic information present in the AST we were able to compute the separator sets as exactly as possible. In fact we implemented the "greedy" algorithm mentioned in Section 2.

For example consider a simple if statement such as that depicted in Figure 1. As already pointed out in Section 2, Thorup's original work assumes "potential neighbors" for all nodes, which means that many nodes would have separator sets augmented with the targets of potential exit, return, or raise statements. By traversing the AST, however, it is easy to find that no such statements exist in the code. Thus the cardinality of the separator sets turns out to be smaller than the original Thorup approach suggests, which implies that also the tree width of the example is smaller than the Thorup approach finds.

We also implemented Thorup's method of renumbering the nodes if a boolean expression does not contain short-circuit operators. We did not implement the renumbering shown to solve problem (2) in Section 2 for general loop and exit statements; we did, however, implement it for while- and for-loops without exits, which was much easier to do and covers many practically important cases.

For these reasons we still only compute upper bounds for the tree width of Ada programs, but we think – and this is supported by our results – that the calculated tree width is very tight to the actual tree width, which could only have been found by much more effort, both in time used for implementation and in running time of our tool[6].

Our tool calculates the tree width of so-called *units*. The Ada language features that constitute such units are enumerated below:

- (generic) subroutine bodies,
- task bodies,
- entry bodies,
- declare blocks, and
- (generic) package bodies.

Finally we note that all units containing goto statements are ignored by our tool. However, the total number of goto statments we encountered was less than 1000.

7 Results of Study

In the following we describe the Ada projects we have included in our study.

[6] In general, calculating the tree width of a given graph is NP-complete [ACP87].

AdaBroker is a set of tools and libraries that can be used to develop CORBA applications in Ada. *AdaCGI* is an Ada95 interface to the "Common Gateway Interface" (CGI). *AdaDoc* is a tool to create documentation from a specification package. *AdaGPGME* is a thin Ada 95 binding to GNUPG Made Easy. *Aflex* is a lexical analyzer written in Ada, *ayacc* is a compiler-compiler also written in Ada. *ASIS* is the GNAT implementation of the Ada Semantic Interface Specification. *BC* is the Ada implementation of the Booch components. *BfdAda* provides an Ada API to use the GNU Binutils BFD library. *CMA* is a Configuration Management Assistant for Ada. *Components* is the implementation of some data-structures. *FSFGNAT* is the Free Software Foundation's version of GNAT. *GLADE* is ACT's implementation of the Distributed System Annex. *GNACK* is an Ada binding for the ORBit Corba ORB. *GNADE* is a project to develop an Ada 95 development environment providing a seamless integration of SQL-based databases into Ada 95. *GNAT* is ACT's Ada Compiler. *GPS* is ACT's GNAT Programming System, an integrated programming environment. *GtkAda* denotes an Ada binding to GTK, the GIMP Toolkit, a graphical library. *GVD* is the GNU Visual Debugger, written in Ada. *Intervals* is an implementation of interval arithmetic in Ada. *JGNAT* is ACT's Ada to Java Byte Code translator. *Libra* is a general library of data structures for Ada95 under Unix-like operating systems. *MaRTE* is the Minimal Real-Time Operating System for Embedded Applications and *MAST* is the Modeling and Analysis Suite for Real-Time Applications. Both are developed by the Real-Time Group of the Universidad de Cantabria. *ORK* is the open source implementation of the Ravenscar profile. *PIWG* is the Performance Issues Working Group's test suite. *PPlaner* is a Project Planner written in Ada. *Rail* is a small part of a project planning tool for electronic railway interlocking systems (a compiler) implemented by ARC Seibersdorf research. *Style* is an Ada style checker. *Units* is a units of measurement implementation in Ada.

Table 1 shows percentages of how many units have tree widths 2, 3, 4, and 5. No unit had tree width higher than 5. The last column shows the average tree width of the software projects.

It is not surprising that bindings such as GtkAda are at the lower end of the scale because bindings usually only use a small part of Ada's language features.

It is however remarkable that compilers are at the upper end of the results.

Comparing our results with those of [GMT02] for the tree width of Java applications, we see that the smallest average tree width for Java is 2.48 and the largest tree width is 2.94, while the corresponding values for Ada are 2.06 and 2.43, i.e., the largest value for Ada is smaller than the smallest value for Java.

8 Conclusion

In this paper we have proved that in general Ada programs may result in non-bounded tree width. On the other hand, a study of existing Ada code showed

Name	Version	No. Units	% tw 5	% tw 4	% tw 3	% tw 2	Avg. TW
AdaBroker	1.0	5644	0.00	0.53	16.99	82.46	2.18
AdaCGI	1.6	84	0.00	1.19	17.86	80.95	2.20
AdaDoc	2.01	240	0.00	0.42	14.17	85.42	2.15
AdaGPGME	0.4.2	25	0.00	0.00	12.00	88.00	2.12
aflex	1.4a	259	0.00	0.39	17.37	82.24	2.18
ASIS	3.15p	4148	0.00	0.96	16.20	82.84	2.18
ayacc	1.1	440	0.00	2.27	24.09	73.64	2.28
BC	20030815	1464	0.00	0.82	12.09	87.09	2.13
BfdAda	0.9	203	0.00	0.00	7.88	92.12	2.07
CMA	—	1129	0.00	3.28	23.29	73.43	2.29
Components	1.2	108	0.00	0.93	23.15	75.93	2.25
FSFGNAT	20020814	9713	0.01	3.12	22.46	74.41	2.28
GLADE	3.15p	4113	0.00	1.09	12.13	86.77	2.14
GNACK	1.1c	733	0.00	0.55	27.56	71.90	2.28
GNADE	1.5.0	1855	0.00	0.11	10.24	89.65	2.10
GNAT	3.15p	9933	0.02	3.16	22.46	74.36	2.28
GPS	1.2.2	10816	0.00	0.92	12.33	86.75	2.14
GtkAda	2.2.0	4574	0.00	0.13	5.79	94.08	2.06
GVD	1.2.5	1408	0.00	1.07	24.43	74.50	2.26
Intervals	1.0	115	0.00	0.00	11.30	88.70	2.11
JGNAT	1.1p	8285	0.01	3.07	21.94	74.98	2.28
Libra	0.2.0	405	0.00	0.25	13.83	85.93	2.14
MaRTE	1.2	1271	0.00	0.55	25.96	73.49	2.27
MAST	1.2.2	1289	0.00	1.32	17.77	80.92	2.20
ORK	2.2b	8021	0.01	3.00	21.61	75.38	2.27
PIWG	—	1319	0.00	0.38	6.52	93.10	2.07
PPlanner	—	162	0.00	0.00	19.14	80.86	2.19
Rail	—	2278	0.00	2.02	28.67	69.32	2.32
Style	—	104	0.00	0.96	15.38	83.65	2.17
Units	1.3	157	0.00	0.00	43.95	56.05	2.43

Table 1. Tree Width of Ada Applications

that the actual tree width is quite low. Thus, for example, Ada programs can be analyzed very efficiently by static analysis tools.

In addition we have introduced renumbering schemes that allow to derive tighter upper bounds for the tree width of Ada code. This includes for- and while-loops which have not been treated in [Tho98].

Our results cannot be compared directly to those of [GMT02] because the Java study has used the approach of building a parser and ignored exceptions completely. However, on the average Ada programs turned out to have a smaller tree width than Java programs. This is the case even though Ada programmers utilize all flow affecting constructs the language provides.

Acknowledgments. We are grateful to *ARC Seibersdorf research* for handing over part of their railway code for the study.

References

[ACP87] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski, *Complexity of finding embeddings in a k-tree*, SIAM J. Alg. Disc. Meth. **8** (1987), no. 2, 277–284.

[ALS91] Stefan Arnborg, Jens Lagergren, and Detlef Seese, *Easy problems for tree-decomposable graphs*, Journal of Algorithms **12** (1991), 308–340.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers, Principles, Techniques and Tools*, Series in Computer Science, Addison Wesley, 1988.

[Bod93] Hans L. Bodlaender, *A tourist guide through treewidth*, Acta Cybernetica **11** (1993), 1–21.

[GKP96] Wolfgang Gellerich, Markus Kosiol, and Erhard Ploedereder, *Where does GOTO go to?*, Ada-Europe'96 International Conference on Reliable Software Technologies (Montreux, Switzerland), June 1996, pp. 385–395.

[GMT02] Jens Gustedt, Ole A. Mæhle, and Jan Arne Telle, *The treewidth of Java programs*, Proc. ALENEX'02 - 4th Workshop on Algorithm Engineering and Experiments (San Francisco, CA), LNCS, 2002, pp. 42–51.

[KP98] Sampath Kannan and Todd A. Proebsting, *Register allocation in structured programs*, Journal of Algorithms **29** (1998), 223–237.

[PK89] Peter Puschner and Christian Koza, *Calculating the maximum execution time of real-time programs*, The Journal of Real-Time Systems **1** (1989), 159–176.

[RS83] Neil Robertson and Paul D. Seymour, *Graph minors. I. Excluding a forest*, J. Comb. Theory Series B **35** (1983), 39–61.

[Tho98] Mikkel Thorup, *All structured programs have small tree width and good register allocation*, Information and Computation **142** (1998), no. 2, 159–181.