FAKULTÄT FÜR !INFORMATIK

Detecting Busy Waiting by Means of Static Control Flow Analysis

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Georg Kienesberger

Matrikelnummer 0026887

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Johann Blieberger

Wien, 27.07.2009

(Unterschrift Verfasser) (Unterschrift Betreuer)

Georg Kienesberger, Zentagasse 16/25, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien am 20. Juli 2009,

Detecting Busy Waiting by Means of Static Control Flow Analysis

Georg Kienesberger

July 20, 2009

Abstract

Detecting Busy Waiting by Means of Static Control Flow Analysis

by Georg Kienesberger

Busy waiting occurs whenever a process repeatedly checks a condition until it becomes true without influencing that condition itself, thereby effectively wasting system resources and introducing the risk of system failure due to race conditions. Hence, busy waiting is considered bad programming practice and can be avoided by the use of higher communication facilities.

In the development of critical systems, for which correctness and robustness are of vital importance, software quality assurance is of great value. However, it is difficult and impractical to manually discover busy waiting in existing program code, which is why a static analysis tool is needed for that purpose.

Based on an existing algorithm that targets the detection of busy waiting using methods of static control flow analysis, I developed such software and further improved the analysis methods to increase efficiency and sharpen the results.

Since the Ada programming language is often used for critical applications I selected it as target language for my analysis and also used it for the implementation of the analysis software itself.

The main results of my work are a static analysis tool for detecting busy waiting in Ada programs and a framework providing a powerful CFG-based representation of Ada source code facilitating comprehensive static control flow analysis in general.

Kurzfassung

Detecting Busy Waiting by Means of Static Control Flow Analysis

von Georg Kienesberger

Ein Prozess, in dem Busy Waiting verwendet wird, überprüft wiederholt eine Bedingung, bis sie erfüllt ist, ohne die Bedingung selbst zu beeinflussen, weshalb hierbei Systemresourcen verschwendet werden und sogar Systemversagen auf Grund von Race Conditions möglich ist. Deshalb wird Busy Waiting als schlechte Programmierpraxis angesehen, kann es doch auch durch die Verwendung höherer Interprozesskommunikationsmethoden vermieden werden.

Gerade im Bereich der kritischen Anwendungen, für die Korrektheit und Ausfallsicherheit von entscheidender Bedeutung sind, spielt Softwarequalitätssicherung eine wichtige Rolle. Allerdings ist es diffizil und mühsam, Busy Waiting in bestehendem Quelltext per Hand aufzuspüren, weshalb für diesen Zweck ein Tool zur statischen Analyse unerlässlich ist.

Basierend auf einem existierenden Algorithmus zur Identifizierung von Busy Waiting durch Methoden der statischen Kontrollflussanalyse habe ich eine solche Analysesoftware entwickelt und parallel dazu das Analyseverfahren hinsichtlich Genauigkeit und Effizienz weiterentwickelt.

Nachdem gerade die Programmiersprache Ada häufig für kritische Anwendungen eingesetzt wird, entschied ich, nicht nur Ada-Programme als Ziel meiner Analyse zu wählen, sondern auch die Analysesoftware in dieser Sprache zu entwickeln.

Die wichtigsten Resultate meiner Arbeit sind ein Tool zur statischen Analyse von Ada-Programmen in Hinblick auf Busy Waiting und ein Framework, welches eine mächtige kontrollflussbasierte Darstellung von Ada-Quelltext generiert und so generell umfassende statische Kontrollflussanalyse ermöglicht.

Contents

List of Figures

List of Listings

List of Tables

Chapter 1
INTRODUCTION

A computer program can be represented in various forms where source and machine code are of course the most important ones. The widespread field of control flow and data flow analysis [4, 21, 14, 8, 5, 23], however, relies on the representation in form of a control flow graph (CFG) [3]. The goals of analysing a control flow graph are manifold and range from program optimisation to the detection of specific properties or characteristics of a given program to serve varying purposes.

The Ada programming language, in its different versions, was designed from the beginning to suit the development of large, critical systems for which correctness and robustness are of vital importance. This is also reflected by its current field of application which ranges from aviation and space flight over military technology to medical and financial systems [17].

Clearly, program analysis plays an important role when it comes to ensuring safety and quality of Ada programs. A few years ago, Blieberger et al. [9] invented a static control flow analysis algorithm that detects busy waiting.

To employ busy waiting is generally considered as bad programming practice, as it not only leads to a waste of system resources but may also lead to program failure. It is therefore a threat to the security, robustness and quality of software. Unfortunately, busy waiting is hard to detect without employing a static analysis tool that is capable of locating busy waiting in existing program code.

The goal of my work was to develop such a tool, based on the algorithm proposed by Blieberger et al. [9] and to further improve this algorithm to sharpen the analysis and increase efficiency.

Evidently, being able to transform an arbitrary Ada program into its control flow based representation is the most essential prerequisite for this kind of analysis. However, since no tool that in some way accomplishes this task existed prior to this work, the development of such software was a crucial and extensive part of this project.

As a result, I developed, together with a colleague, the Ast2Cfg framework which is capable of generating powerful CFG-based data structures for an arbitrary Ada program that hold comprehensive information on the original source, such as visibility, package structure, type definitions, etc. and provides means for interprocedural analysis [16, 15].

Upon that framework, I then developed the *Busy Wait Analyser for Ada* (BWAA), a static, CFG-based analysis tool for the detection of busy waiting in Ada programs.

An extensive description of the analysis algorithm with my modifications and the implementation and inner workings of the aforementioned software follows this introduction. The software itself is *free software* as defined by the *Free Software Foundation* [25] and I published it under the terms of the *GNU General Public License.*

Chapter 2
PROBLEM STATEMENT

2.1 Definition of Terms

A *control flow graph* (CFG) is a directed graph [3] with node set N and edge set $E \subseteq N \times N$. The nodes $n \in N$ represent *basic blocks*, which comprise a linear sequence of consecutive statements. There is an edge $(u, v) \in E$ from u to v if v can follow u in some execution sequence. Then, u is called the *predecessor* of v and v is the *successor* of u. Every CFG has a unique start node called *root* node. A *path* from node u_1 to node u_n is a finite list $< u_1, \ldots, u_n >$ of nodes in which successive nodes are connected by an edge [22], so there is an edge (u_i, u_{i+1}) for every $i < n$. If there is a path from u to v, u is an *ancestor* of v and v is a *descendant* of u [11]. A *cycle* is a path where no node is repeated except for the first and last node that are the same. A graph that contains no cycles when the directions of its edges are ignored is called a *tree* [13], and a set of disconnected trees is called a *forest*.

A graph can be represented by an *adjacency list* which consists of an array of lists, one for each node in the graph [11]. So for each node u there is a list which contains all the nodes v such that an edge $(u, v) \in E$ exists.

A *depth first search* (DFS) is a traversal of a graph as it is shown in Listing 2.1. A DFS imposes an ordering [3] on the nodes of a graph as it assigns every node a unique number. An edge (u, v) is called a *retreating edge* iff the DFS-number of u is greater than the DFS-Number of v.

```
procedure DFS (N: Node) is
begin
    if N.Visited = True then
        return;
    end if;

    N.Visited := True;
    N.Number := I;
    I := I + 1;

    for each successor S of N loop
        DFS(S);
    end loop;
end DFS;
```

Listing 2.1: The depth first search algorithm (pseudocode).

A node u *dominates* [3] node v if every possible path from the root node to v includes u. A tree in which the root of the tree equals the root of the CFG, and each node dominates only its descendants in the tree is called *dominator tree*. An edge (u, v) is called a *backedge* if v dominates u.

A control flow graph is *reducible* [3] iff its edges can be partitioned into two disjoint sets: A set of *forward edges* that constitute a graph without cycles in which every node can be reached from the root node and a set of backedges. In a reducible control flow graph the set of backedges is equal to the set of retreating edges.

2.2 Busy Waiting

Busy waiting is often defined as a technique used for synchronisation of concurrent processes. For instance G. R. Andrews defines it as

> “... a form of synchronisation in which a process repeatedly checks a condition until it becomes true ...” [6].

That is, a process that wants to enter a so-called *critical section*, which only one process at a time may enter, has to periodically check whether he is allowed to do so, effectively being unable to perform any useful work but nevertheless consuming processor time [24].

However, in general it can be said that busy waiting occurs whenever a loop exit condition is not influenced from within the loop and therefore the loop is only exited in case the value of a variable is changed from outside [16]. This does not limit busy waiting to a synchronisation technique but for instance also includes processes that loop until a specified time is reached while checking the clock periodically or a process that repeatedly checks whether the user entered some input, etc.

Busy waiting can in general be considered as a bad programming practice or an indication for poor program design. Although, there are a few special cases where busy waiting can be tolerated, like for instance in operating system kernels for reasons of efficiency, the drawbacks of busy waiting make it impossible to justify its use in general.

The most noticeable disadvantage is of course the waste of processor time, however, when used for synchronisation busy waiting may, depending on the implementation, also cause race conditions which in fact could result in program failure. Therefore, higher communication facilities like semaphores, monitors, rendezvous, etc. [18, 24, 7] should be used instead.

As a first example for busy waiting consider Dekker's algorithm for the mutual exclusion of two processes as described by Dijkstra [12]. In Listing 2.2 an implementation in Ada using tasks is shown, and 2.1 depicts the accompanying CFG.

```
procedure Dekker is

    Turn: Integer range 1 .. 2 := 1;
    T1,T2: Boolean := False;
    task Task1;
    task Task2;

    task body Task1 is
    begin
        T1 := True;
        while T2 loop
            if Turn = 2 then
                T1 := False;
            end if;
            while Turn = 2 loop
                null;
            end loop;
            T1 := True;
        end loop;

            -- critical section

        Turn := 2;
        T1 := False;

    end Task1;

...
```

Listing 2.2: Dekker's mutual exclusion algorithm.

Listing 2.2 only shows the code for one of the two tasks, since the second task body is programmed analogously. First, a task sets its flag, `T1` or `T2` respectively, and then checks the other task's flag. In case it is not set, it may enter the critical section immediately. If the second task's flag is set, it next tests whether it is its own turn to insist on entering the critical section or whether it should allow the other task to proceed. In either case, it has to wait until the other task's flag is unset. When leaving the critical section the flag and the `Turn` variable are adjusted accordingly.

Each task has two loops, also highlighted in Figure 2.1 using dotted lines and dashed lines respectively. It can easily be seen, that the exit conditions of both loops, `T2` and `Turn = 2` are not influenced from within the corresponding loop. This is because neither `T2` nor `Turn` are defined inside the loops. Hence, the loops are only exited upon an external event and therefore both loops are *busy wait loops*.

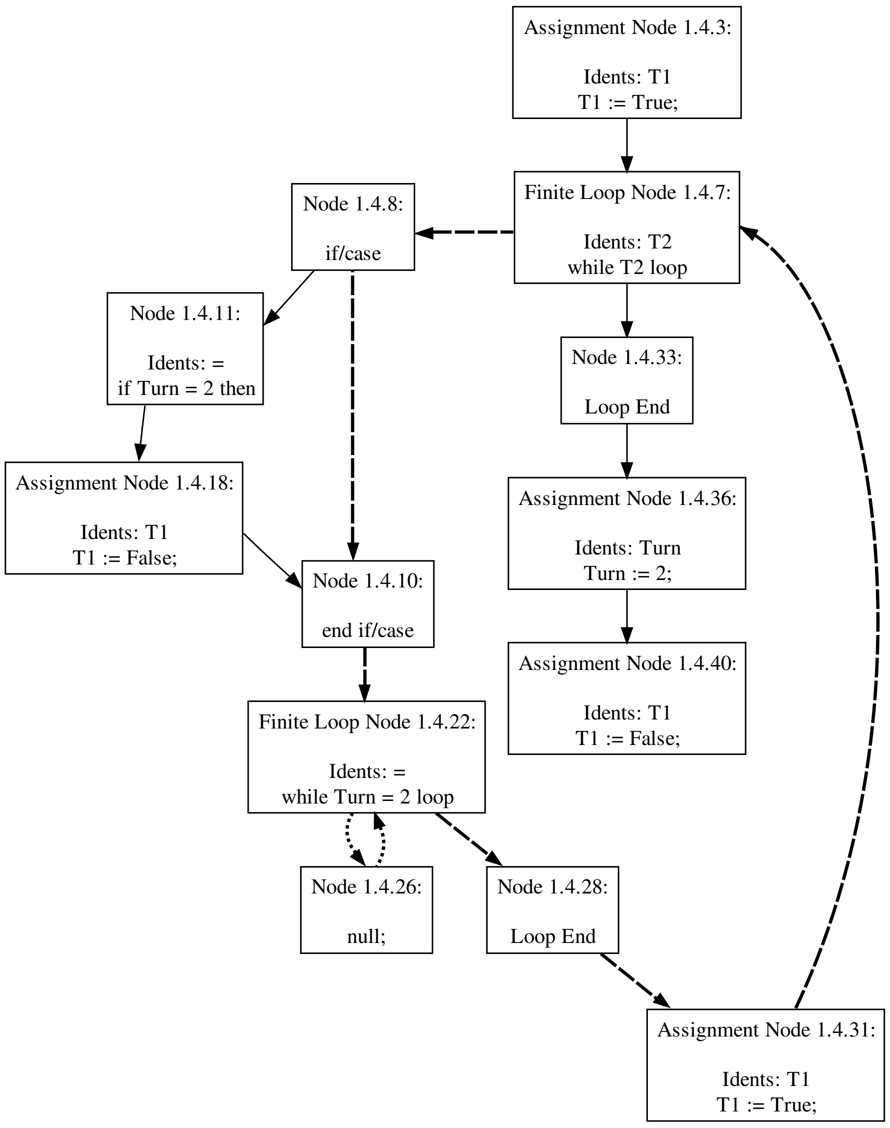


Figure 2.1: The CFG of Dekker's mutual exclusion algorithm.

Employing for example Ada's protected objects this use of busy waiting could have been easily avoided while retaining the same result. However, to manually find busy waiting in existing program code in order to replace it with a proper solution is a very tedious and for projects other than very small ones nearly impossible task.

Therefore, a static analysis tool that automatically finds and reports busy waiting is the only way to ensure that a given program does not contain busy waiting and adheres to state-of-the-art programming style.

To develop such a tool and therefore facilitate software quality assurance of Ada programs is the goal of this thesis.

Chapter 3
ALGORITHM

3.1 Overview

This section gives an overview of the busy wait detection algorithm presented by Blieberger et al. [9] and my modifications to it. The algorithm operates on the CFG of a given program and uses control flow properties and semantic properties of statements to decide whether busy wait is employed in a specific loop.

Since CFGs generated from Ada programs are reducible, the backedges can be used to define the loops of a CFG. So Blieberger et al. define the set of nodes that constitute a loop $L_{(m,n)}$ of a backedge (m, n) as

$$L_{(m,n)} = \{u \mid \exists \pi =< u, \dots, m >: n \notin \pi\} \cup \{n\}.$$

Therefore every node from that m, the source of the backedge, may be reached without going through n is said to be part of he *loop body.* Node n is called the *loop header* because it is part of every path to a node in the loop body and thus dominates every node in the body.

Note, however, that although the target of a backedge is a single loop header, multiple backedges may point to the same header i.e. a single loop may have more than one backedge.

In order to find the loops in a CFG Blieberger et al. suggest to first compute the dominator tree in order to find the backedges. Remember that a backedge is defined as an edge where its source is dominated by its destination. However, I found out that the backedges may be computed much more efficiently. This is because in a reducible CFG, like the ones for Ada programs are, every retreating edge is a backedge. Therefore a simple variant of a DFS is sufficient to compute the set of backedges (see Section 3.2).

Next, a simple work-list algorithm to compute $L_{(m,n)}$ is presented and it is suggested to compute the loop forest in order to be able to start the analysis with the innermost loops. As it turned out, the algorithm I use to compute the loop nesting forest (see Section 3.3) already allows to build $L_{(m,n)}$ along the way, so executing the work-list algorithm is unnecessary.

After that, the statements that affect the termination of a loop have to be found. A *termination statement* is a statement inside the loop that has at least one successor outside the loop. Hence it decides whether to stay in the loop or not and therefore needs to have a branch predicate. The program variables in such a branch predicate may cause busy waiting. If higher communication facilities are not employed and such a variable is only read and not defined within the loop, it is called a *wait variable* [9].

Now, for each of the busy wait candidates it has to be checked whether it might cause the loop to loop forever in case there is no interaction from the outside. Therefore, Blieberger et al. introduce a predicate $defined_{\mathbf{var}}(\pi)$ which holds if variable **var** is *defined* on path π, that is, if **var** is assigned a value in a node of the path. This is then used to define the *busy-var*-predicate $\textbf{busy-var}(L_{(m,n)}, \mathbf{var})$ which holds if **var** is a wait variable:

$$\textbf{busy-var}(L_{(m,n)}, \mathbf{var}) =_{\text{defs}} \exists \pi =< n, \ldots, n >\in L_{(m,n)} : \neg defined_{\mathbf{var}}(\pi)$$

So in case there is a path that starts and ends in the loop header n where the busy wait candidate is not defined, the loop is called a *busy wait loop* and the candidate is in fact a wait variable.

However, as described in Section 3.7, I decided to change this predicate in order to avoid a class of false alarms, and the predicate becomes

$$\textbf{busy-var}(L_{(m,n)}, \mathbf{var}) =_{\text{defs}} \exists \pi =< t, \ldots, t >\in L_{(m,n)} : \neg defined_{\mathbf{var}}(\pi),$$

where t is the termination statement where **var** was found.

Hence, if one variable in a termination statement of some loop does not change and therefore is found to be a wait variable, busy wait is reported. This, however, is the cause for a large class of false alarms because, for example, variables that serve as boundaries in the branch predicate normally do not change, without being responsible for busy wait. So I decided to implement an optional scanning mode that only reports busy wait if all candidates of a termination statement are wait variables.

In order to find a path where the busy wait candidate is not written, Blieberger et al. suggest to construct a subgraph of the loop which is composed only of those nodes that do not define the candidate. Then, a simple reachability check reveals if there is a path π for which $\neg defined_{\mathbf{var}}(\pi)$ holds. However, for the sake of efficiency, I implemented the reachability check right on the full graph which implies only a few additional checks to make sure that only non-writing paths are considered and that the loop is not left.

Finally, Blieberger et al. present a refinement of their algorithm which improves the analysis in a way that more busy wait loops are detected. The refined algorithm also considers variables as busy wait candidates that are used to define existing candidates. So for every candidate all variables that are on the right hand side of an assignment are added to the candidates. This is done until all candidates (and therefore also the new ones) have been handled.

3.2 Finding the Backedges

As already stated, the dominators of a CFG are not needed in order to compute the backedges. This is because every retreating edge in a reducible CFG is a backedge. Retreating edges can be found quite easily using a slightly modified DFS which uses two different marks.

```
procedure Find_Backedges (Cur: Node; Last: Node) is
begin
    if Cur.Mark = 2 then
        return;
    elsif Cur.Mark = 1 then
        -- (Last, Cur) is a backedge
        return;
    end if;

    Cur.Mark := 1;
    Cur.Number := I;
    I := I + 1;

    for each successor S of Cur loop
        Find_Backedges(S, Cur);
    end loop;

    Cur.Mark := 2;
end Find_Backedges;
```

Listing 3.1: The algorithm used to find the backedges (pseudocode).

When the algorithm shown in Listing 3.1 first encounters a node it marks it with `1` and recurses for its successors. After all descendants that were unmarked so far have been handled, the node is marked with `2`. So when a node marked with `1` is encountered, one of its descendants apparently links to it and since the DFS-number of a descendant is greater, a backedge has been found.

3.3 Identifying Loops

In order to identify the loops, i.e. to find the nodes that constitute the loop bodies and also to build the loop-nesting forest, I use Tarjan's classical algorithm [27] for reducible CFGs as it is also presented by Ramalingam [20].

This algorithm for constructing a loop-nesting forest basically takes a CFG as input, and builds up an array (called `Loop_Parent` in Listing 3.2) which maps the corresponding loop header to every node. It also employs a union find data structure as described in Section 3.4 to keep track of the changes to the CFG which itself is not modified.

First the loop parent array is initialised, and every node is placed in a set by itself. Then the algorithm visits every node of the CFG in reverse-DFS-order and checks whether it is the target of a backedge. If so, a loop header was found, and the CFG is traversed backwards until the the header is reached again, adding every node encountered on the way to the loop body. Since in a reducible CFG the loop header dominates the nodes of the body it is guaranteed that the loop is not left during this traversal and the reverse-DFS-order ensures that inner loops are identified first.

After the header and the body have been identified the loop parent array is updated and the nodes of the body are *collapsed* into the header. By collapsing a node w into a node v, w and all its incident edges are deleted, adding an edge (v, x) for each deleted edge (w, x) with $x \neq v$ and (v, x) not already an edge, and adding an edge (x, v) for each deleted edge (x, w) with $x \neq v$ and (x, v) not already an edge [27]. Remember that these changes are reflected by the union find data structure, and so, in order to collapse a node into the header the union operation is used. However, since it is important that the canonical element of such a union find set stays the same all the time, so that the find operation always returns the loop header, in contrast to what is stated by Ramalingam [20], union by rank cannot be implemented.

Now it is easy to build the loop forest either from the loop parent information or by adding custom code to the procedure which is responsible for collapsing.

As an example consider the CFG depicted in Figure 3.1. It has two nested loops which are identified by the backedges $(5, 2)$ and $(6, 1)$. Figure 3.2(a) shows the union find data structure (on the left hand side) and the loop parent array (on the right hand side) right after the initialisation in lines 30-33 of Listing 3.2. The situation after the first loop was found and the body was collapsed into the header is shown in Figure 3.2(b).

```
procedure Collapse (Loop_Body, Loop_Header)
begin
    for every z ∈ Loop_Body loop
        Loop_Parent(z) := Loop_Header;
        UF.Union(z, Loop_Header);
    end loop;
end Collapse;

procedure Findloop (Potential_Header)
begin
    Loop_Body := {};
    Worklist := { UF.Find(y) | y → Potential_Header is a backedge } −
              − {Potential_Header};
    while Worklist is not empty loop
        remove an arbitrary element y from Worklist;
        add y to Loop_Body;
        for every predecessor z of y such that z → y is not a backedge loop
            if (UF.Find(z) ∉ (Loop_Body ∪ {Potential_Header} ∪ Worklist)) then
                add UF.Find(z) to Worklist;
            end if;
        end loop;
    end loop;
    if Loop_Body is not empty then
        Collapse (Loop_Body, Potential_Header);
    end if;
end Findloop;

procedure Tarjans_Algorithm (G)
begin
    for every vertex x of G loop
        Loop_Parent(x) := null;
        UF.Makeset(x);
    end loop;
    for every vertex x of G in reverse-DFS-order loop
        Findloop(x);
    end loop;
end Tarjans_Algorithm;
```

Listing 3.2: Pseudocode for Tarjan's algorithm for constructing the loop-nesting forest as presented by Ramalingam [20].

Finally Figure 3.2(c) illustrates the situation after the algorithm has finished and both loops were found. Note that as far as the loop parent array is concerned the outer loop only has two nodes: 2 and 6. This is because of the collapsing, and has to be taken care of when using the loop parent array.

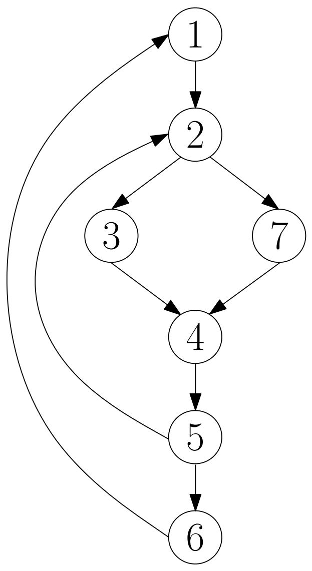


Figure 3.1: A CFG with two loops.

3.4 The Set Union Algorithm

The *set union* or *union find* algorithm as presented by Tarjan [28] maintains a group of disjoint sets under the operation of union. Each set has a unique identifier which is an arbitrary element of the set and is called the *canonical element*. The sets are represented by a tree, where the nodes are the elements of the set and the canonical element is the root. Each node has a pointer to its parent, except for the root node which points to itself.

Tarjan defines three operations on these sets: *makeset*, *find* and *link*. Makeset creates a new set containing only the element given as an argument. The operation find(x) returns the canonical element for the set containing x and link(x,y) creates a new set containing the elements of the sets represented by x and y. The old sets are destroyed and the new canonical element is chosen arbitrarily out of x and y. Cormen et al. [11] in addition introduce a function union(x,y) which works like link(x,y) except that x and y may be elements other than the canonical elements.

When makeset is executed, a node with the given element is created and the pointer is set on itself. A call of find starts at the node containing the argument,

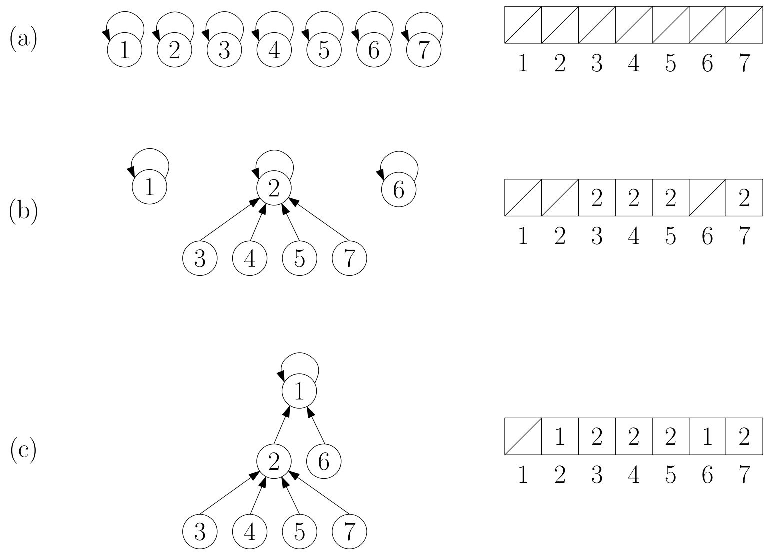


Figure 3.2: The union find data structure and the loop parent array during the algorithm.

and follows the pointers until an element which is its own parent (the root node) is reached. To carry out link(x,y) the pointer to the parent of x is simply set on y. In Figure 3.3(a) the subgraph containing only node a is the result of calling makeset(a), find(g) returns d and Figure 3.3(b) shows the result of link(b,d).

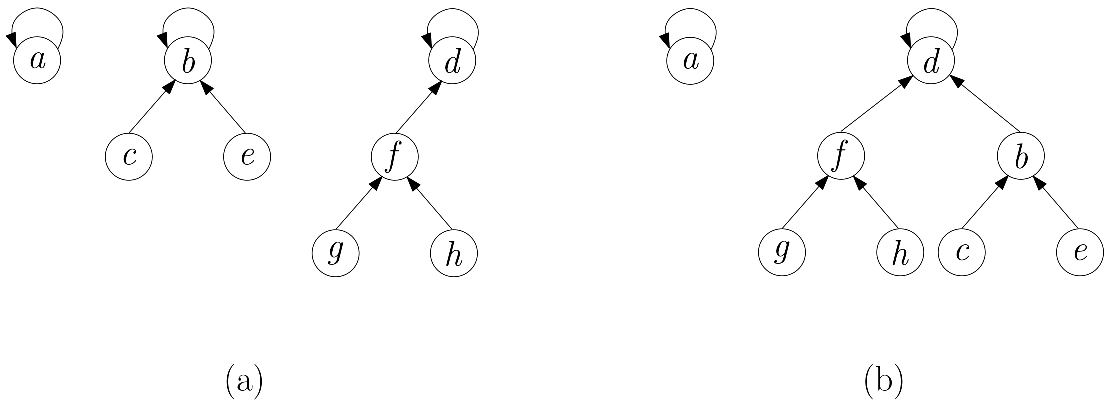


Figure 3.3: A union find data structure before (a) and after (b) link(b,d).

The execution time of the find operation of the algorithm presented so far is in $O(n)$ with n being the total number of elements, which is why two heuristics to improve the efficiency are proposed in [28]. *Path compression* is carried out during a find operation and sets, after the root node has been found, the parent pointers of all nodes encountered during the search directly to the root node (see

Figure 3.4). This does not improve the performance of the current call to find but those of consecutive calls. The second technique is called *union by rank*, and keeps the depth of the trees small. It stores a value *rank* for each node, which is initially 0. When link(x,y) is called the rank of x and y is compared. If the rank of x is smaller than that of y, the pointer of x is set to y. In case the rank of y is greater, then x is the new parent of y. Finally, if the rank is equal x will point to y and the rank of y is increased by one. Now the worst-case bound for the running time of some sequence of the set operations is in $\Theta(m\alpha(m,n)))$, where m is the number of operations, n is the number of elements and α is the functional inverse of the *Ackerman function* which, for practical reasons, can be treated as a constant of four [28].

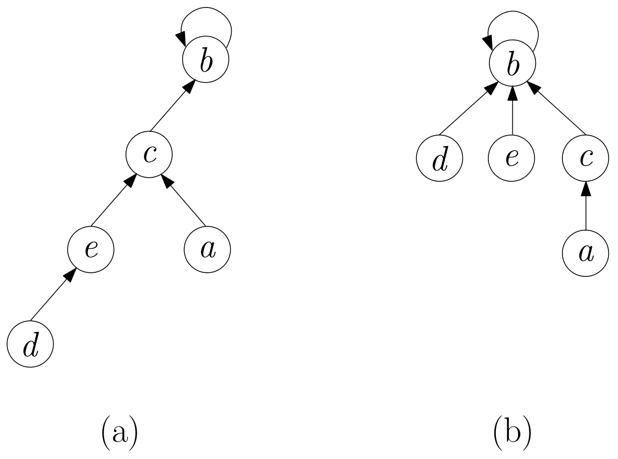


Figure 3.4: A union find data structure before (a) and after (b) find(d) with path compression.

3.5 Finding the Termination Statements

A termination statement exits a loop, and therefore has at least one immediate successor outside the loop. Blieberger et al. [9] define the statements that influence the termination of the loop as the following set of edges:

$$T_{(m,n)} = \{(u,v) \in E \mid u \in L_{(m,n)} \land v \notin L_{(m,n)}\}$$

So the set T of termination statements is the set of all edges (u,v) where u is part of the loop but v is not. Therefore, a node representing such a statement needs to have at least two successors, one inside the loop and one outside. However, this means that it has to be a branching node with a branch predicate that decides whether to stay in the loop or to exit [9].

As an example consider the loops in Figure 3.5 which was generated (without parameter trees) from the source code in Listing 3.3 using Cfg2Dot (introduced in Section 4.2.1). There are two loops, the inner one consisting of nodes 1.1.9, 1.1.15 and 1.1.22, and the outer one additionally containing 1.1.5 and 1.1.26. Note that I added the nodes of the inner loop to the outer one too.

The inner loop with header 1.1.9 has two termination statements, namely 1.1.9 and 1.1.22. 1.1.9 is one because its successor 1.1.26 is not part of the (inner) loop, and 1.1.22 because of node 1.1.27. The outer loop also has two termination statements, 1.1.5 and 1.1.22 since successor node 1.1.27 is not part of the loop.

Note that both loops share the termination statement 1.1.22. This is because the exit statement refers to the loop named `OUTER` and therefore exits the outer and inner loop simultaneously. This is the reason why the nodes of an inner loop also have to be added to the loop body of the outer one during the loop identification phase. Otherwise the outer loop in this example would only have 1.1.5 as a termination statement.

```
OUTER:
    for I in 1 .. 10 loop
        while X < 20 loop
            X := X + 1;
            exit OUTER when X > 10;
        end loop;
    end loop OUTER;
X := 0;
```

Listing 3.3: Code snippet, Figure 3.5 was generated from.

3.6 Busy Wait Candidates

Blieberger et al. [9] define the set $V_{(m,n)}$ of variables that are candidates for wait variables in the loop defined by backedge (m, n) as

$$V_{(m,n)} = \{\mathbf{var} \in bp(u) \,|\, (u, v) \in T_{(m,n)}\}$$

where $bp(u)$ gives the branch predicate of branching node u. So for every termination statement of a loop the branch predicate is determined and the variables contained therein are added to the set of busy wait candidates for that loop.

However, since I decided to implement a slightly modified version of the busy-var-predicate as described in Section 3.7 it is not sufficient to save all candidates

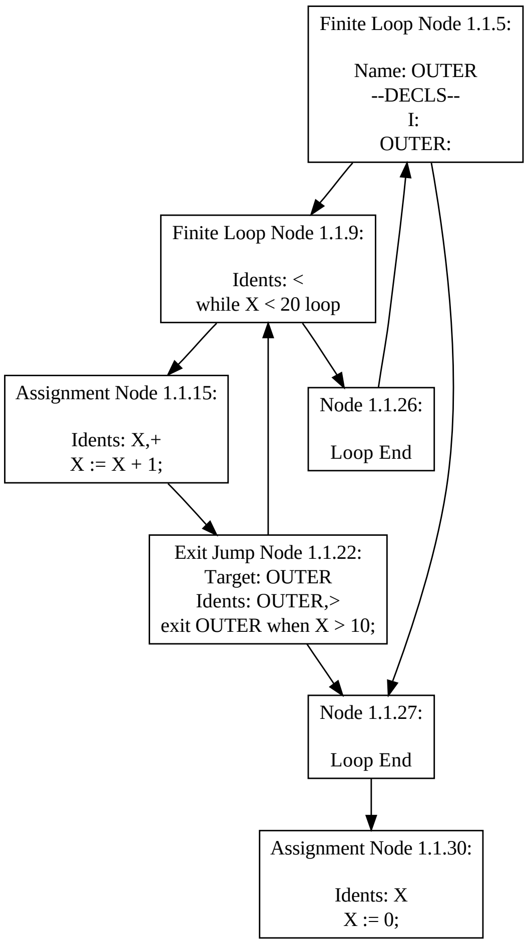


Figure 3.5: Two loops with termination statements.

of a loop in a single set. In fact I need a set of candidates for every termination statement.

3.7 Busy Wait Variables

As already stated in Section 3.1, Blieberger et al. introduce the predicate

$$\textbf{busy-var}(L_{(m,n)}, \textbf{var}) =_{\text{defs}} \exists\pi =< n, \ldots, n > \in L_{(m,n)} : \neg defined_{\textbf{var}}(\pi)$$

which is used to determine whether a loop is a busy wait loop and **var** a wait variable. This predicate holds whenever there is at least one path from the loop header back to itself where **var** is not defined.

However, the code in Listing 3.4 and the accompanying CFGs in Figure 3.6 are an example for a class of false positives when using the loop header as start and end node. The dashed lines together with the dotted ones in Figure 3.6(a) represent the paths that are taken into account when searching for a definition of variable `Y`. In addition, the dashed lines outline a path where `Y` is not defined, which means that the predicate holds and the variable is incorrectly identified as a wait variable.

The problem is that this path does not contain the termination statement in node 1.1.26 and therefore should not be taken into account. Hence, I modified the busy-var-predicate so that the termination statement t serves as start and end node of the paths that are searched for definitions:

$$\textbf{busy-var}(L_{(m,n)}, \textbf{var}) =_{\text{defs}} \exists\pi =< t, \ldots, t > \in L_{(m,n)} : \neg defined_{\textbf{var}}(\pi),$$

Figure 3.6(b) shows the result: Only the path outlined with dashed edges is considered and since `Y` is defined right before the termination statement no busy waiting will be reported.

3.8 Indirect Busy Wait Variables

Consider the example in Listing 3.5. `X` is the only busy wait candidate and, since there is no relevant path in the loop that does not write it, the algorithm does not mark it as a wait variable. However, the assignment in line 5 does not change the value of `X` since `Y` itself is not changed throughout the loop. Therefore `Y` should also be considered as a candidate for a wait variable. Then the algorithm would correctly identify `Y` as an indirect busy wait variable and the loop as a busy wait loop.

```
procedure Foo is
    X: Integer := 0;
    Y: Integer := 0;
begin

    loop
        if X >= 1 then
            Y := Y + 1;
            exit when Y > 10;
        else
            X := X + 1;
        end if;
    end loop;

end Foo;
```

Listing 3.4: Source code illustrating the need for the modified busy-var-predicate.

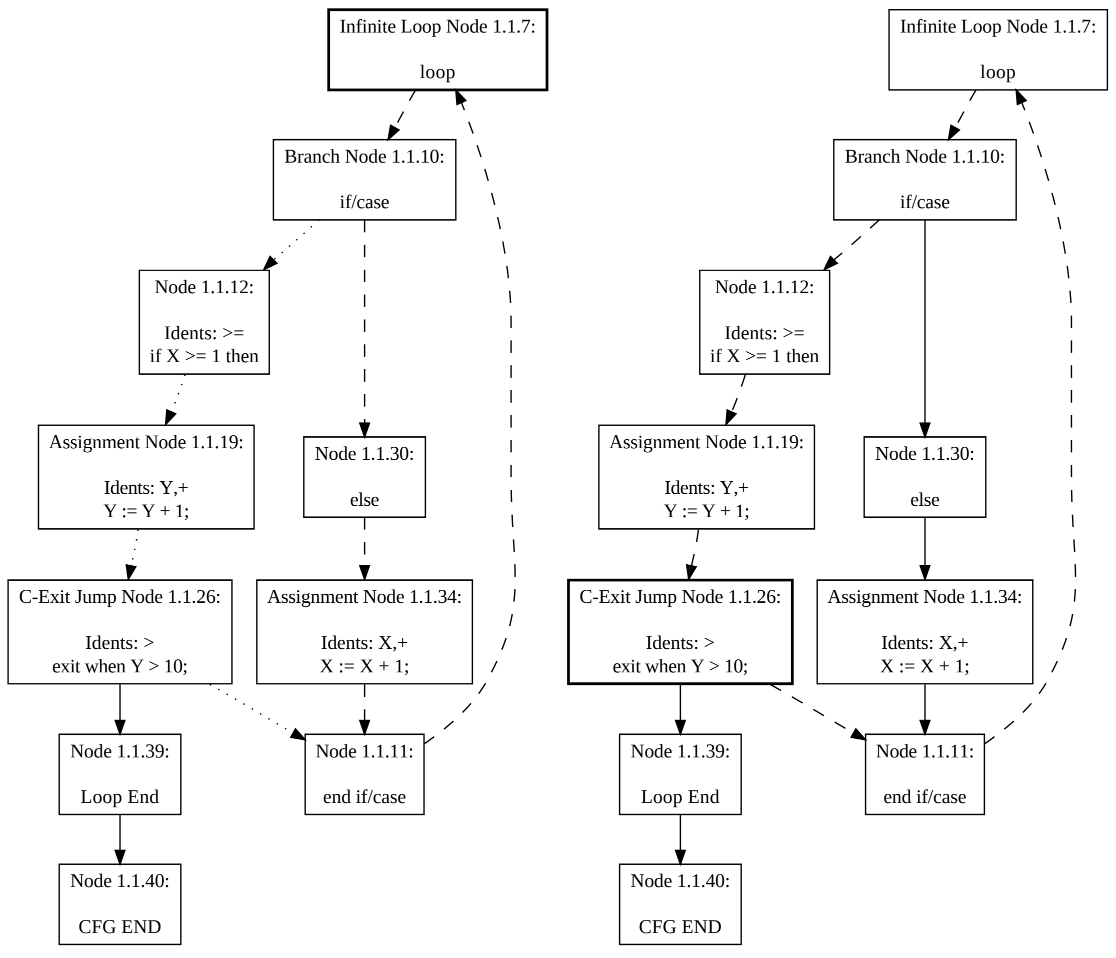


Figure 3.6: Searched paths for (a) the original busy-var-predicate and (b) the modified one.

```
declare
    X,Y: Boolean := False;
begin
    while not X loop
        X := Y;
    end loop;
end;
```

Listing 3.5: Example for an indirect busy wait variable.

Hence Blieberger et al. [9] adapt the definition of $V_{(m,n)}$ as follows:

$$\begin{aligned} V^0_{(m,n)} &= \{\mathbf{var} \in bp(u) \mid (u,v) \in T_{(m,n)}\} \\ V^{k+1}_{(m,n)} &= \{\mathbf{var} \in \text{rhs of assignments in } L_{(m,n)} \text{ to } \mathbf{var} \in V^k_{(m,n)}\} \\ V_{(m,n)} &= \bigcup_{k\geq 0} V^k_{(m,n)} \end{aligned}$$

First the set of busy wait candidates as it was previously defined is composed. Then for every variable in the resulting set the nodes of the loop are searched for assignments to that variable. If one is found, every variable on the right hand side of the assignment is added to the set. After all statements in a loop have been considered, the search continues for the next variable in the set until all variables have been handled and no new candidates are found.

However, this refinement not only leads to the detection of more busy wait loops but may also increase the number of false alarms. For example variable `X` in Listing 3.6 is incremented by one in every iteration of the loop. Nevertheless the loop is wrongly reported to be a busy wait loop, since `Y` does not change throughout the loop.

```
declare
    X,Y: Integer := 0;
begin
    while X < 10 loop
        X := X + Y + 1;
    end loop;
end;
```

Listing 3.6: Example for an indirect busy wait variable.

In any case, as far as reducing the false alarms is concerned, Blieberger et al. refer to symbolic methods as introduced in [10] which will certainly improve the analysis but unfortunately are beyond the scope of this thesis.

3.9 Aliasing

A problem that arose during the implementation of the analysis algorithm was that in Ada the same variable or package may be referenced using different names, which I refer to as *aliases*, depending on package structure and renaming definitions. This implies that simply testing the strings of the variable names for equality is not sufficient.

In the remainder of this chapter I will outline what types of aliases have to be taken into account when dealing with Ada 2005 programs. Moreover, in Section 5.6 I will describe the implementation details and data structures I used.

3.9.1 Paths and Simple Visibility

The most basic form of variable aliases emerges from the fact that, depending on the location of the variable declaration within the nesting of packages and subprograms also addressed in Section 4.3.1 and the location where the variable is referenced, it may be optional or mandatory to add names of enclosing units to the plain variable name. These package or subprogram names separated by a dot constitute a *path* to the declaration of the variable. A *full name* or *absolute path* contains the root of the Pkg/CFG tree in which the variable declaration resides.

As an example consider variable `Y` declared in line 5 of Listing 3.7 which, in procedure `Proc2` may be referenced with the names `Y`, `Bar.Y` and `Foo.Bar.Y`. In procedure `Proc1` the same variable may be referenced only using `Bar.Y` and `Foo.Bar.Y` because it is not directly visible there.

This example also shows that visibility has to be taken into account. In procedure `Proc1` variable `X` declared in line 2 has the names `Foo.X` and `X`, however, in `Proc2` it may only be referenced using `Foo.X` because of the locally declared variable with the same name.

```
 1  package Foo is
 2      X: Integer;
 3
 4      package Bar is
 5          X,Y: Integer;
 6
 7          procedure Proc2;
 8      end Bar;
 9
10      procedure Proc1;
11  end Foo;
```

Listing 3.7: Two variables that may be referenced using different names at different locations.

3.9.2 Variable Renamings

In the Ada 2005 Reference Manual [26] a renaming is defined as follows:

> "A renaming_declaration is a declaration that does not define a new entity, but instead defines a view of an existing entity."

Therefore, a renaming of a variable just defines another name for this variable. In the context of this project, such a name has to be considered an alias exactly the same way as the full name of a variable (cf. Section 3.9.1).

In Listing 3.8 two variables are declared in a package and its subpackage. Furthermore, there are two renamings, one renaming a variable directly (`X2`), and one renaming the renaming of a variable (`X3`). In Table 3.1 the aliases for each variable at the two subprograms of the example are listed.

```ada
package Foo is
    X: Integer;

    package Bar is
        X2: Integer renames X;
        X: Integer;
        procedure Proc2;
    end Bar;

    procedure Proc1;
end Foo;

package body Foo is

    package body Bar is
        procedure Proc2 is
        begin
            null;
        end Proc2;
    end Bar;

    procedure Proc1 is
        X3: Integer renames Bar.X2;
    begin
        null;
    end Proc1;

end Foo;
```

Listing 3.8: Two variables and two variable renamings.

It is important to note, that a variable that is declared after a subpackage, is not visible in the subpackage's specification, even when the full name is used as shown in Listing 3.9.

3.9.3 Package Renamings

A package may be renamed just like a variable, however, the impact is higher because a package renaming also affects the package structure and therefore other package and variable renamings too.

	`Proc1`	`Proc2`
`Foo.X` (line 2)	`X`	`Foo.X`
	`Foo.X`	`X2`
	`Bar.X2`	`Bar.X2`
	`Foo.Bar.X2`	`Foo.Bar.X2`
	`X3`	
	`Proc1.X3`	
	`Foo.Proc1.X3`	
`Foo.Bar.X` (line 6)	`Bar.X`	`X`
	`Foo.Bar.X`	`Bar.X`
		`Foo.Bar.X`

Table 3.1: Aliases for the variables in Listing 3.8.

```
package Foo is

    package Bar is
        X2: Integer renames X; -- illegal because X is undefined here
        X3: Integer renames Foo.X; -- illegal too, for the same reason
    end Bar;

    X: Integer;
end Foo;
```

Listing 3.9: Illegal renaming declarations because of visibility issues.

As an example consider Listing 3.10. Since package `Quuux` in line 15 renames `Foo.Bar.Baz`, `Foo.Bar.Baz.X` is also available using for instance `Foo.Quuux.X`, effectively eliminating one level of the path and providing a shortcut. On the other hand, package `Qux` leads to a series of aliases for `Foo.Baz.X` that add one level.

Moreover, visibility also has to be considered, the renaming in line 8 clearly renames `Foo.Baz` and not `Foo.Bar.Baz`.

Finally, note the circular renaming in line 9 where `Quux` renames `Bar` although itself resides inside `Bar`. This would lead to an infinite set of aliases, which is why I had to introduce a limitation described in Section 5.6.9 and indicated by the finite nature of Table 3.2 which lists all aliases that I consider for the two variables of the example.

```
 1  package Foo is
 2
 3      package Baz is
 4          X: Integer;
 5      end Baz;
 6
 7      package Bar is
 8          package Qux renames Baz;
 9          package Quux renames Bar;
10          package Baz is
11              X: Integer;
12          end Baz;
13      end Bar;
14
15      package Quuux renames Bar.Baz;
16
17      procedure Proc;
18  end Foo;
```

Listing 3.10: An example with a few package renamings.

	`Proc`
`Foo.Baz.X` (line 4)	`Foo.Baz.X`
	`Baz.X`
	`Foo.Bar.Qux.X`
	`Bar.Qux.X`
	`Bar.Quux.Qux.X`
	`Foo.Bar.Quux.Qux.X`
`Foo.Bar.Baz.X` (line 11)	`Foo.Bar.Baz.X`
	`Bar.Baz.X`
	`Foo.Quuux.X`
	`Quuux.X`
	`Foo.Bar.Quux.Baz.X`
	`Bar.Quux.Baz.X`

Table 3.2: Aliases for the variables in Listing 3.10.

Chapter 4

THE AST2CFG FRAMEWORK

As most *control flow* and also *data flow analysis* methods the algorithms I present in this thesis rely heavily on the representation of a program in form of a *control flow graph.* Evidently being able to transform arbitrary Ada programs into a control flow graph was a crucial point in this project.

Since for Ada a tool, that in some way generates the control flow graph for a program was inexistent, I started, together with a colleague, to develop such a software: the Ast2Cfg framework.

First we needed to find a way to handle the input of arbitrary Ada programs. Apart from parsing the source of the input programs, modifying GNAT, the GNU Ada compiler, would have been the most obvious option. However, while parsing the source code would have meant to reinvent the wheel, inserting code into GNAT implies the constant adaption to newer versions, which is a considerable effort especially since at that time GNAT was under heavy development because of the upcoming Ada 2005 standard.

Finally, we decided to use *ASIS-for-GNAT* (see Section 4.1), a library that provides the input program in form of an *abstract syntax tree* (see Section 4.1.1). Ast2Cfg then traverses the resulting abstract syntax tree and at the same time builds up a control flow graph.

However, extensive control flow based analysis requires considerably more information than a simple CFG, for example in form of an adjacency matrix. This is why, in fact, we developed a whole framework which provides comprehensive information on the original Ada source, including, for instance, visibility information, package structure and type definitions. Furthermore it provides means for interprocedural control flow analysis.

In this chapter, which is based on my previous work as a co-author of [15, 16], I will give an overview of the design and inner workings of Ast2Cfg and the related software, but also include the many features I added more recently.

4.1 ASIS

ASIS, the *Ada Semantic Interface Specification* is a standard for an interface between an Ada 95 environment and any tool requiring information from it. The standard is independent of the underlying Ada environment implementations. The ASIS interface consists of a set of types, subtypes, and subprograms which provide a capability to query the Ada compilation environment for statically determinable syntactic and semantic information. The base object in ASIS is the `Asis.Element`, which is an abstraction of entities within a logical Ada syntax tree. So the elements correspond to nodes of a tree representation of an Ada program and therefore represent Ada language constructs. [19]

The usual way of interacting with an ASIS implementation is to traverse this syntax tree and query for information on the visited elements. The ASIS implementation used in this project is *ASIS-for-GNAT*, which is the implementation for use with the GNU Ada compiler GNAT. Although the current ASIS standard targets Ada 95, after the release of the new Ada 2005 standard eventually more and more features of Ada 2005 got implemented in GNAT and ASIS-for-GNAT.

4.1.1 Abstract Syntax Tree

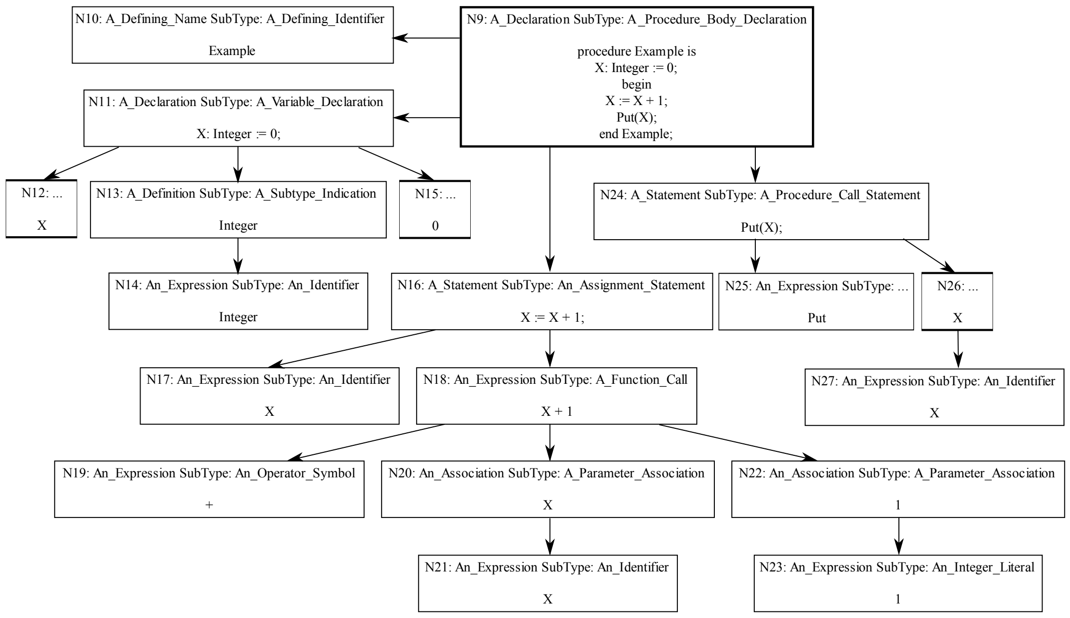
An *abstract syntax tree* (AST) is defined [3] as a tree in which each node represents an operator, and the children of the node represent its operands. As an example consider the AST shown in Figure 4.1 which was generated with Ast2Dot (see Section 4.2) using the source code in Listing 4.1.

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Example is
X: Integer := 0;
begin
   X := X + 1;
   Put(X);
end Example;
```

Listing 4.1: Source code for the ASIS AST examples in Figure 4.1

For reasons of presentation I had to shorten the annotation of some ASIS types and leave out the subtrees for the `with` and `use` clauses, which is why the lowest node identifier is `N9`. To complete the missing information, Node `N12` is of type `A_Defining_Name` and subtype `A_Defining_Identifier`, node `N15` is an expression with subtype `An_Integer_Literal`, `N25` is an identifier and the most specific type of node `N26` is `A_Parameter_Association`.



N9: A_Declaration SubType: A_Procedure_Body_Declaration
procedure Example is
X: Integer := 0;
begin
X := X + 1;
Put(X);
end Example;
N10: A_Defining_Name SubType: A_Defining_Identifier
Example
N11: A_Declaration SubType: A_Variable_Declaration
X: Integer := 0;
N12: ...
X
N13: A_Definition SubType: A_Subtype_Indication
Integer
N15: ...
0
N24: A_Statement SubType: A_Procedure_Call_Statement
Put(X);
N14: An_Expression SubType: An_Identifier
Integer
N16: A_Statement SubType: An_Assignment_Statement
X := X + 1;
N25: An_Expression SubType: ...
Put
N26: ...
X
N17: An_Expression SubType: An_Identifier
X
N18: An_Expression SubType: A_Function_Call
X + 1
N27: An_Expression SubType: An_Identifier
X
N19: An_Expression SubType: An_Operator_Symbol
+
N20: An_Association SubType: A_Parameter_Association
X
N22: An_Association SubType: A_Parameter_Association
1
N21: An_Expression SubType: An_Identifier
X
N23: An_Expression SubType: An_Integer_Literal
1

Figure 4.1: The ASIS AST for the example source code in Listing 4.1.

While the gist of the above definition is still valid as far as ASIS ASTs are concerned, there are quite a few additions to that concept that can be seen in Figure 4.1. The assignment statement `X := X + 1;` in node `N16` has two child nodes, one for the left hand side of the assignment, `X`, and one for the expression on the right hand side, `X + 1`. On the next level of the tree this expression, which is treated like a function call, is then decomposed into the two operands or parameters, `X` and `1`, but also the operator symbol, `+`. So, in general, the first, i.e. leftmost child node of a function call is the name of the function (in this case `+`) and the parameters to the function follow on the same level and in order of their appearance in the function call statement.

ASIS ASTs represent whole subprograms, not only single expressions. Obviously the sequence of the statements also has to be represented in the AST. The root node, node `N9` in the example, contains the whole subprogram which is then decomposed in the following tree levels. The first child node, node `N10`, contains the name of the subprogram and is, at the same level, followed by the declarations (`N11`) and finally each individual statement (`N16` and `N24`).

4.2 Software Overview

From a developer's point of view Ast2Cfg is a library that uses ASIS to get the information that is needed to build the CFG for a given Ada program. When the input has been processed, which is triggered by a library function, the application using Ast2Cfg may access a comprehensive CFG-based representation of the input program. Figure 4.2 depicts the basic structure of the transformation process that is performed by Ast2Cfg.

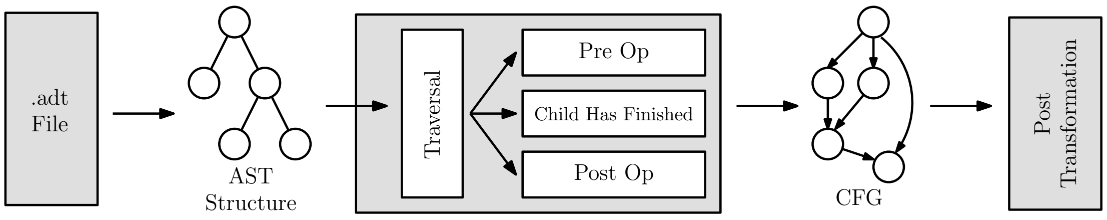


Figure 4.2: The basic structure of the transformation process.

First GNAT has to be used to generate the so-called *tree files* for the given program, which serve as input for Ast2Cfg. This is done using GNAT's `-gnatt` and `-gnatc` options. A tree file contains a snapshot of the compiler's internal data structures at the end of the successful compilation of the corresponding source code [2].

Thereafter Ast2Cfg uses the ASIS-for-GNAT library to extract the AST structure out of the tree files which is then traversed using the *ASIS application template* provided with a typical ASIS installation. This template traverses the ASTs of a given program using a depth first search algorithm. The user of the template has to provide two procedures: one that is executed when a node is visited for the first time (`Pre_Op`), and one that is executed when a node is visited on the way back (`Post_Op`). Additionally, Ast2Cfg introduces a procedure that is called whenever the processing of a successor of some node has finished, that is, `Post_Op` has been executed for this child. These three procedures are the core of the transformation which builds the CFG for the input program.

At this point, a raw version of the CFG-based data structure has already been built. However, it still needs some refinement, which is done during the *post transformation phase*, covered in detail in Section 4.5. Mainly this is because some parts of the transformation can only be accomplished conveniently when a raw CFG already exists. For example consider the situation when a goto statement with a target that has not already been processed is reached during the traversal.

Listing 4.2 shows a simple program that uses Ast2Cfg in order to output the names of all top-level packages that are declared in the context of the tree files `foo.adt` and `bar.adt`.

```
with Ada.Text_IO; use Ada.Text_IO; with Ast2Cfg.Pkgs; use Ast2Cfg.Pkgs;
with Ast2Cfg.Control; with Ast2Cfg.Flow_World; with Ast2Cfg.Output;

procedure Run is
        World: Ast2Cfg.Flow_World.World_Object_Ptr;
        Pkgs: Pkg_Class_Ptr_List.Object;
        Pkg:  Pkg_Class_Ptr := null;
begin
        -- Initialisations
        Ast2Cfg.Output.Set_Level(Ast2Cfg.Output.Warning);
        Ast2Cfg.Control.Init("-CN foo.adt bar.adt");

        -- Fill the World with flow data
        World := Ast2Cfg.Control.Generate;

        -- Output the name of all top-level packages
        Pkgs := Ast2Cfg.Flow_World.Get_Pkgs(World.all);
        Pkg_Class_Ptr_List.Reset(Pkgs);
        while Pkg_Class_Ptr_List.Has_Next(Pkgs) loop
                Pkg_Class_Ptr_List.Get_Next(Pkgs, Pkg);
                Put_Line(Get_Name(Pkg.all));
        end loop;

        -- Finalisation
        Ast2Cfg.Control.Final;
end Run;
```

Listing 4.2: A small application using Ast2Cfg.

Every program that uses Ast2Cfg will have to import `Ast2Cfg.Control` and `Ast2Cfg.Flow_World`. `Ast2Cfg.Control` is needed for initialisation, finalisation and for generating the World Object. Therefore also `Ast2Cfg.Flow_World` is needed, so that a pointer for the World Object may be declared.

By default Ast2Cfg does not output anything, even in case of an error. However, this behaviour can be changed with `Ast2Cfg.Output.Set_Level` which takes an argument of type `Level_Type`. This can be one of `None`, `Error`, `Warning`, `Verbose`, `Debug` and `Max`, where `Warning` is recommended for most applications. Of course all output produced by Ast2Cfg is sent to *standard error*.

`Ast2Cfg.Control` allows to control the transformation in a convenient way. First `Init`, which optionally takes a `Wide_String` with the *ASIS context parameters*, has to be called. The ASIS context parameters determine which tree files are used as input where the default action is to include every tree file in the current directory. For example with the `-CN` option every tree file whose name is in the list given as the next parameter is considered as input [1].

To start the actual transformation process `Ast2Cfg.Control.Generate` has to be called, which also returns an access to the resulting *world object* (see Section 4.3). After the program has completed its own analysis of the transformation results, `Ast2Cfg.Control.Final` has to be called in order to free the memory reserved by the Ast2Cfg structures (including the world object) and to finalise ASIS.

4.2.1 Download and Additional Resources

During the development of Ast2Cfg, there was the need to visualise ASTs and CFGs for testing purposes which is why two small but very useful tools were implemented alongside.

Cfg2Dot is a simple utility that makes use of Ast2Cfg without performing an analysis, but to output a graphical representation of the Control Flow Graphs within a program. This is done using the *dot* format which can be converted to various other graphics formats, including for instance *postscript* and *png*, using the dot program, which is part of the *graphviz* package in most `GNU/Linux` distributions and is also available for other operating systems.

Similarly *Ast2Dot* allows to study the syntax trees provided by ASIS via transformation and output of the ASTs into *dot* files.

Ast2Cfg, Cfg2Dot and Ast2Dot are available under terms of the *GNU General Public License* and may be downloaded at the project website: `http://cfg.w3x.org`.

4.3 The Flow World

All data gathered during the transformation phase is saved into a single object, the *flow world* of type `World_Object`. The flow world basically holds a list of the top-level packages of the input programs. In Ast2Cfg every package of the transformed program is represented by a `Pkg_Object`. Note that control flow information that is not part of a package is contained within an artificial default package.

`Pkg_Object` is derived from the abstract `Flow_Object`. The same applies to `CFG_Object` and `Node_Object`. `CFG_Object` is used to resemble the control flow of various entities in Ada, such as subprograms, blocks or even initialisation sequences of packages. In order to reflect such control flow, a `CFG_Object` uses objects of type `Node_Object`.

As described in detail below, each of these types has a series of subclasses in order to allow for a more fine-grained classification. In Figure 4.3 the complete class hierarchy originating from `Flow_Object` is depicted.

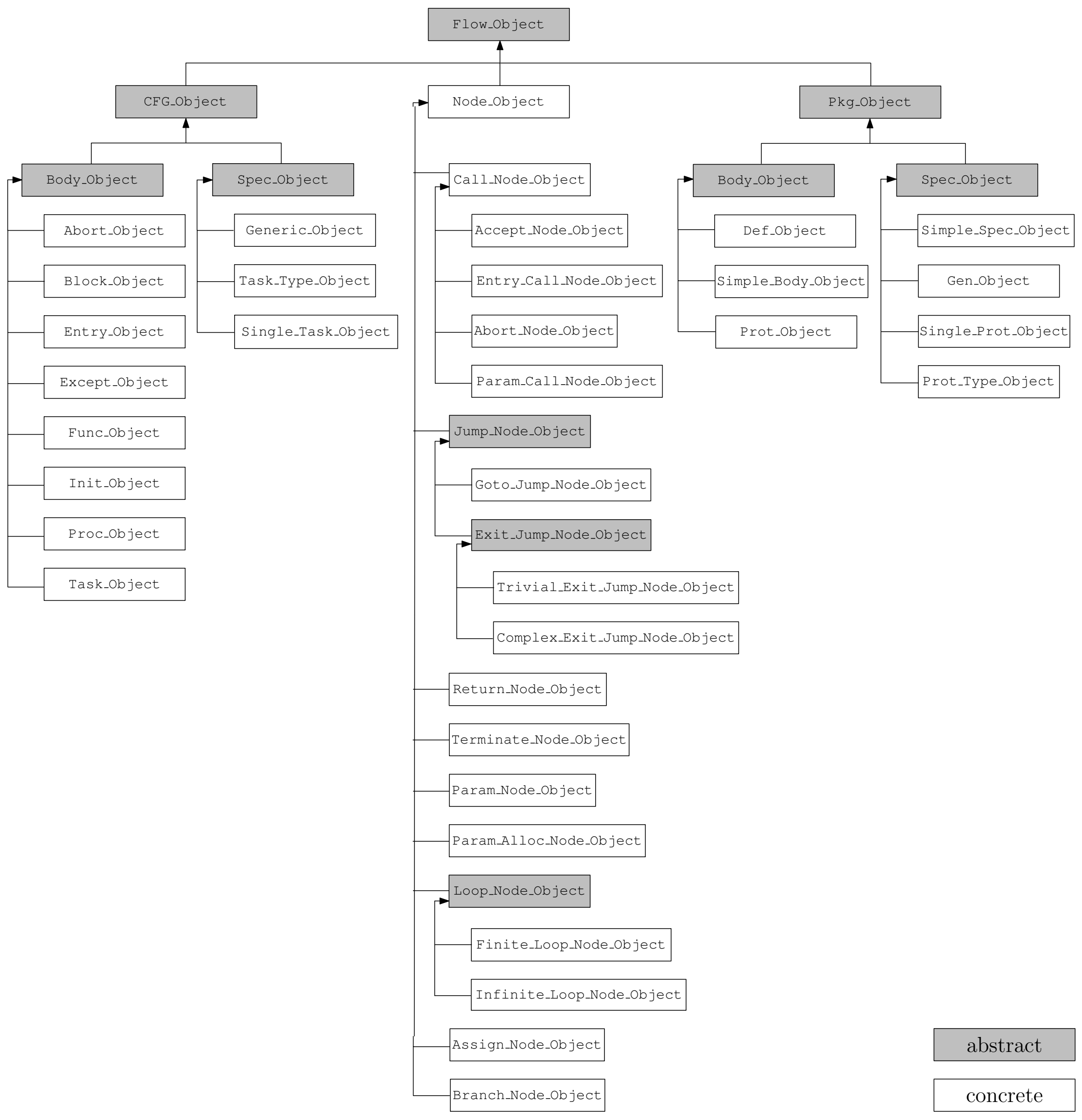


Figure 4.3: Class hierarchy of the flow types.

4.3.1 Pkg/CFG Tree

In Ada subprograms and packages may be declared within each other, which leads to nesting relationships that have to be recorded. Every flow object has a list of predecessors and a list of successors. While node objects use those lists to build up a CFG, in package and CFG objects predecessor and successor links are used to represent the nesting structure. In this context, if B is declared within A, B is a successor of A and A is a predecessor of B. This relation between package and CFG objects imposes a tree structure on the flow world, the *Pkg/CFG tree.*

In practice, there are multiple Pkg/CFG trees in the flow world, because usually there is more than one top-level package in a program.

Figure 4.4 was generated using Cfg2Dot and shows the Pkg/CFG tree for the program in Listing 4.3. Packages are depicted as rectangles and subprograms as circles.

```
procedure Proc1 is

        package Pkg1 is
                procedure Proc2;
                package Pkg2 is
                end Pkg2;
        end Pkg1;

        package body Pkg1 is
                procedure Proc2 is
                begin
                        null;
                end Proc2;
        end Pkg1;

        procedure Proc3 is
        begin
                null;
        end Proc3;
begin
        null;
end Proc1;
```

Listing 4.3: The code from which Figure 4.4 was generated.

4.3.2 Parameter Tree

The *parameter tree*, which is generated whenever an expression is encountered, holds information on the used variables, functions, etc. and their nesting. Therefore, a parameter tree is a complete hierarchical representation of an expression, which allows for comprehensive static analysis. A parameter tree is saved directly in the flow object that contains the expression.

As can be seen in Figure 4.5, Cfg2Dot connects a parameter tree directly to the CFG node containing the expression it originates from, highlighting its nodes with

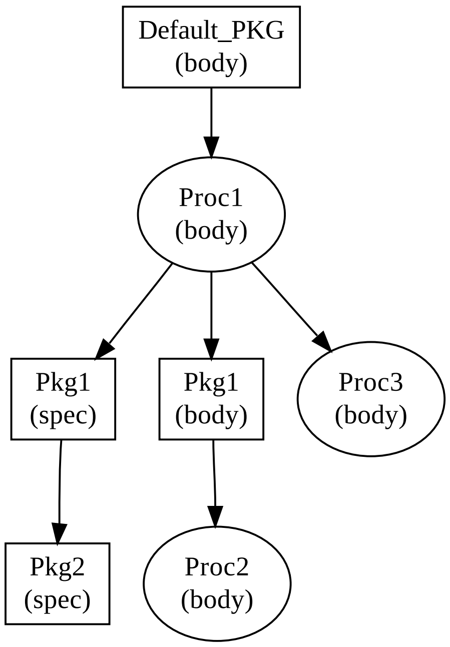


Figure 4.4: The *Pkg/CFG tree* generated from Listing 4.3.

dashed lines. The depicted CFG, which results from the source code in Listing 4.4, only has a single node, that represents the assignment statement on line number 16. This node has two parameter trees, one for the part to the left of the assignment operator and one for the right hand side. Every level in the parameter tree corresponds to a nesting level, thus a child node contains a parameter to its predecessor. As a consequence, a subtree represents a subexpression and the leaf nodes contain either an indivisible expression or nothing in case of a constant value.

```
 1  procedure Test is
 2          function Outer (I: Integer; J: Integer) return Integer is
 3          begin
 4                  return 0;
 5          end;
 6
 7          function Inner (I: Integer) return Integer is
 8          begin
 9                  return 0;
10          end;
11
12          A,B: Integer := 0;
13          C: array (1..10) of Integer := (others => 0);
14  begin
15
16          C(1) := Outer(A, Inner(X => B));
17
18  end Test;
```

Listing 4.4: The code from which Figure 4.5 was generated.

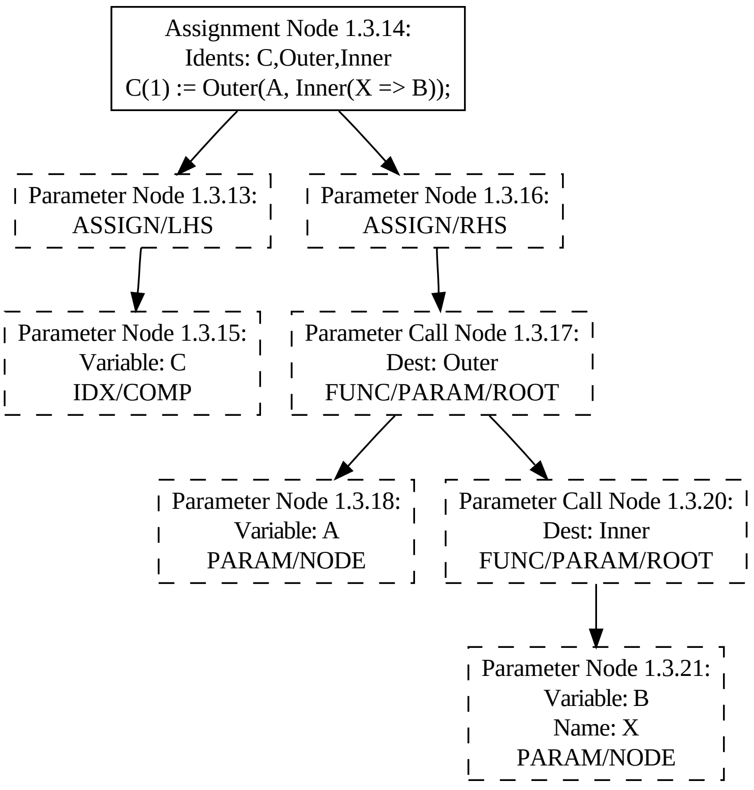


Figure 4.5: A node with two parameter trees.

4.3.3 Flow Object

Every flow object has a unique id which may be sorted and tested for equality. An id basically consists of three numbers: one for the package, one for the CFG and one for the node. Note that such a number is zero in case it is not needed. So for example `1.2.0` is the id of the second CFG in the first package and `1.2.3` is the id of the third node in the second CFG of the first package.

In addition, flow objects have a name. For node objects, however, the name is empty in most cases. This is because nodes, apart from parameter nodes (see Section 4.3.6), only get a name in case the corresponding source line has a label.

As stated previously, every flow object has a list of predecessors and successors, that are used to resemble control flow, or form a Pkg/CFG tree.

Sometimes, there is no other, more adequate place to save a parameter tree, so there is a member variable in the flow object for the special *extra parameter tree*. One example for such an extra parameter tree would be an expression following `when` in an entry body of a protected object. Nevertheless, parameter trees are usually saved in the node object which corresponds to the expression the parameter tree represents.

Furthermore, a flow object contains lists of variable declarations, generic formal parameters, renamings and references such as `with`, `use` or `use type` clauses.

4.3.4 Package Types

The `Pkg_Object` is declared abstract, and all subtypes are derived either from `Body_Object` or `Spec_Object` (see Figure 4.3). The primary function of package objects is to facilitate building the Pkg/CFG tree, consequently the successor and predecessor lists are the most important components. In order to get the contents of a package the subtrees starting at the elements of the successor list have to be traversed. A list of the variables declared within a package can be found in every `Pkg_Object`, except for the `Prot_Object` and the `Def_Object`. Moreover, some package objects convey additional information.

The `Def_Object` is an artificial default package, that contains CFGs that do not have an enclosing package, such as library level procedures. The body of a protected object or type is represented by a `Prot_Object`, which may contain subprograms or entries (see Section 4.3.5). The specification that belongs to such a `Prot_Object` is mapped to a `Prot_Type_Object` in case of a protected type declaration or a `Single_Prot_Object` otherwise. In contrast to the body of a generic package, the corresponding specification requires special handling and therefore is transformed into a `Gen_Object`, which also contains the generic formal parameters.

Finally, ordinary packages, that have not been mentioned above, are mapped to a `Simple_Spec_Object` and, in case there is an accompanying body, a `Simple_Body_Object`.

4.3.5 CFG Types

CFG objects represent the control flow information of different Ada entities like subprograms, blocks, initialisation sequences, etc. by using node objects. Every CFG object contains a reference to root and end node, the total number of nodes (excluding those in parameter trees), and, since many CFG objects represent subprograms, a list of parameters.

As previously stated, every flow object has a member variable that may hold a name. However, as will be described in detail below, it is not possible to give every CFG object a name that is derived from the original Ada source. For instance, a block within some subprogram does not need to have a name. In this case, in connection with call nodes (see Section 4.3.6), a name is generated. To reflect the fact, that such a name is not related to the Ada source, it is saved in an additional string, while the actual name remains `null`.

As can be seen in Figure 4.3, `CFG_Object` itself is declared abstract, which is why all CFG objects have to be of more specific, concrete subtypes. Some of them represent data gained from Ada bodies, while others contain information on Ada specifications, which is reflected through derivation from either `Body_Object` or `Spec_Object`.

In the most basic case, a subprogram has to be represented, which is done by either creating `Proc_Object` for a procedure or, in case of a function, a `Func_Object`. Both types hold a list of the variables declared in the subprogram and its parameters.

For a block statement, which is located within some other CFG object, first a separate `Block_Object` is created. Next, a *call node* (see Section 4.3.6), which represents a subprogram call, is inserted at the position where the block used to be within the enclosing CFG. So, in fact, a block statement is handled like a parameterless procedure, called at the position the block is declared. Of course a list of variables declared in a block statement is available.

An initialisation sequence of a package body is transformed into a so-called `Init_Object`, while an `Except_Object` is created for every exception handler.

For every task body a `Task_Object` is created, where an `accept` statement is transformed separately into an `Entry_Object`. As it was the case with simple blocks, the `Entry_Object` is then linked to its enclosing task body using a call node. Of course, the declared variables are available as usual. Moreover the protected entries of a protected object are also mapped to entry objects.

An `Abort_Object` represents the abortable part of an `select – then abort` statement and is, again, linked into the control flow of the select statement like a parameterless procedure call.

In addition there are three CFG objects, derived from `Spec_Object`, that do not contain any actual control flow information. The main reason why those objects exist is, that by their position in the Pkg/CFG tree, it is possible to track where they are defined, so that visibility information may be gained later on. For a task type declaration a `Task_Type_Object` is created, while a simple task specification (without the `type` keyword) results in a `Single_Task_Object`. The discriminants of a task type declaration are handled like parameters of a subprogram. Finally, a `Generic_Object` is created for every generic procedure or function specification that is encountered. A list of generic formal parameters is available, and the actual parameters of the subprogram may be acquired as usual. Note that the body of a generic subprogram is handled like a normal subprogram body.

4.3.6 Node Types

In contrast to CFG or package objects, `Node_Object` is not declared abstract, so every node of the CFG that has no special properties as described below simply is of type `Node_Object`.

Every node has a reference to the CFG it is part of and in case a statement has a label, then this string is the name of the node representing the statement. Furthermore, a string that holds at least part of the code the node stands for is stored in the `Node_Object`. In addition, all nodes have a handle to a parameter tree, the *right hand side parameter tree*, which resembles the expression contained in the node. It is called that way, because for `Assign_Node_Objects`, which are used to represent assignment statements, there also exists the *left hand side parameter tree* for the part to the left of the assignment operator. Finally, every node contains the `Asis.Element` that was the source of this node. The `Asis.Element` can be seen as a link back into the ASIS AST. Hence, additional information may be gained directly from ASIS by analysing the AST starting at the `Asis.Element` of an arbitrary node. For several reasons the transformation adds nodes that do not correspond to an `Asis.Element`. In that case `Asis.Nil_Element` is used as a placeholder.

As already mentioned, a `Call_Node_Object` is not only used for representing a subprogram call, but also in several situations that are treated alike. Obviously, the most important component of a `Call_Node_Object` is the link to the destination CFG. As depicted in Figure 4.3, there are four subtypes of `Call_Node_Object` which convey additional information on the type of the call. An `Accept_Node_Object`, for example, is used to link an `Entry_Object` to its enclosing task body (see Section 4.3.5), while an `Entry_Call_Node_Object` is used to represent the call of such an entry. Likewise, an `Abort_Node_Object` links the abortable part of a `select - then abort` statement into the control flow of the select statement. Finally, the subtype `Param_Call_Node_Object` is used exclusively within parameter trees to represent a function call.

Whenever a `goto` statement is encountered, a `Goto_Jump_Node_Object` is used to point to the destination of the `goto`. Furthermore, one of the two subtypes of `Exit_Jump_Node_Object` is used for every exit statement within a loop: A `Trivial_Exit_Jump_Node_Object` contains no exit condition, however, it may exit a loop using a label. In contrast, a `Complex_Exit_Jump_Node_Object` does always conditionally exit a loop. It should be noted that the target of an exit jump node is empty in case it exits the innermost enclosing loop.

Of course a `return` statement is also transformed to a special node object, namely the `Return_Node_Object` while a `terminate` statement is represented by a `Terminate_Node_Object`.

The header of a loop is indicated by a `Loop_Node_Object`, and the two concrete subtypes enable the distinction between a `while` or `for` loop, mapped to a `Finite_Loop_Node_Object`, and a simple `loop` statement which is represented by an `Infinite_Loop_Node_Object`.

It is important, however, to note that the naming of these two types just reflects, that an `Infinite_Loop_Node_Object` contains no loop exit condition by itself. So, in fact, an `Infinite_Loop_Node_Object` may stand for a finite loop, because for instance there exists an `exit` statement within the loop. Likewise, a `Finite_Loop_Node_Object` may represent an infinite loop, because the loop condition always holds.

A parameter tree, as presented in Section 4.3.2, is built using nodes of type `Param_Node_Object`, `Param_Alloc_Node_Object` and the already mentioned `Param_Call_Node_Objects`. A `Param_Node_Object` stores the name of the variable that was supplied as a parameter and the name of the parameter itself, in case it is known. In case the variable name is empty a constant was given. A dynamic allocation using the `new` keyword is mapped to a `Param_Alloc_Node_Object`, and the name of the instantiated type is recorded.

Whenever an `if` or `case` statement is encountered, first a special and otherwise empty header node of type `Branch_Node_Object` is created to mark the position where the branching actually happens. Hence, the successors of such a branching node contain the condition for each branch in case there is one (e.g. an `else` branch does not have a condition). Next, the subgraphs for the actual branches follow, until control flow is united again in an end if node. Obviously, in case of an if statement without an else branch there is a direct link from the header node to the end if node.

4.3.7 Declaration Id

The visibility of a declared entity is an important information for numerous applications. To determine whether an entity is declared before or after another one is a common task, which is made possible by the `Declaration_Id` package.

When the declaration of an entity like a package, subprogram, variable or a renaming is encountered, the object created in the flow world immediately is assigned a declaration id.

On that account and because the operators <, >, `=` and `/=` are defined for a `Declaration_Id.Object`, it is, for instance, possible to determine whether a variable `X` is declared before a package `Bar` (cf. Listing 3.9) and therefore is visible in `Bar`.

4.4 Transformation

As already stated in previous sections, control flow data is generated from the AST input using a complex algorithm consisting of two phases. During the first phase, the transformation phase, the information provided by the AST is extracted and the raw flow structure is built. The second part, the post transformation phase (see Section 4.5), further refines this raw structure.

The basic transformation algorithm is constructed upon the inorder traversal skeleton provided by a typical ASIS installation. The AST is traversed one node at a time, generating three types of events:

1. A *preop* event is triggered when the traversal reaches a node for the first time, before any other processing is done.

2. A *postop* event takes place immediately after the traversal has left a node, as soon as all processing involving it has finished.

3. A *child-has-finished* event, which occurs whenever the processing of a node's child has finished. In contrast to the previous events, however, the child-has-finished event is context-sensitive, bearing information on a node's relatives.

This event-based traversal imposes a state-machine-like architecture on the transformation mechanism. While stacks are used to hold the current traversal state, three callback functions, one for each event named above are employed to handle the individual nodes of the AST. Since each method must be able to handle any of the ASIS node types, all three have a symmetrical structure.

One of the strengths of the ASIS abstract syntax trees is that they employ a relatively small set of node types, to describe any program, regardless of its complexity. To achieve this goal, ASIS combines the available types to ample configurations, creating specialised subtrees.

The Ada syntactical constructs can be divided into classes, with the members of each class sharing a common syntax subtree configuration. Usually, each ASIS type has its own `case` branch in the callback functions, but often it is possible take advantage of the tree similarities, by merging the corresponding branches.

As an example consider the variable declarations, the component declarations of the aggregate types and the subprogram parameter specifications. A typical subtree for one of these declarations holds the name of the new entity, its type and, if existent, its initialisation expression. The only node that differs between those subtree classes is the root node, which holds the type.

In most cases, the control flow data can be added to the flow world immediately upon reaching an AST node. The ASIS procedure call statement, for instance, represents a procedure call in the original source code and the subtree rooted in this node describes the statement, comprising possible labels, the name of the called subprogram and its parameters. The standard preop handling is to immediately add a new call node to the current CFG. Later on, upon processing the subtree, a child-has-finished event will be encountered with a procedure call parent and an identifier child, in which case it is clear that the traversal has reached the the name of the called subprogram. This information will then also be saved in the call node which was created earlier. In other cases, however, mainly due to the context-free nature of the transformation, an immediate update of the flow world is not possible.

The Ada language is extremely powerful and versatile, while at the same time facilitating correctness, reliability and good software engineering practices. This, however, results in an ample language set and complex AST structures, which is why the algorithms employed during the transformation phase are rather elaborate as they have to deal with numerous complex details.

Anyhow, since in the context of this thesis the implementation details of the transformation phase are of less interest, the gentle reader is directed to previous publications [15, 16].

4.5 Post Transformation

In this section the different stages of the post transformation phase, that refines the raw CFG-structure generated during the transformation phase will be presented.

4.5.1 Loop Refinement

After the main transformation phase is completed, `while` and `for` loops without `exit` or `return` statements are already represented correctly in the resulting control flow based data structures (see Figure 4.6).

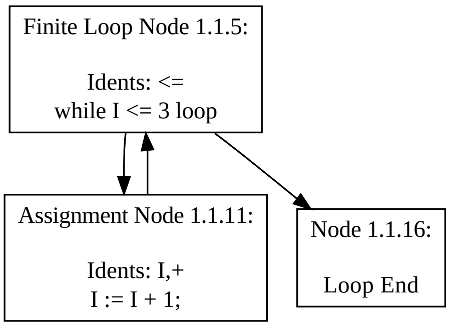


Figure 4.6: A `while` loop.

However, simple loops and loops that contain exit or return statements need to be refined in order to achieve a correct representation. For example consider the simple loop with an exit statement in Figure 4.7. Since there is no condition in the loop header (Node 1.1.5), there should be no edge from the header to the loop end (Node 1.1.19). Furthermore, there should be an edge from the node containing the exit statement (Node 1.1.8) to the loop end. Figure 4.8 shows the same loop after the refinement, this time correctly resembling the original control flow.

As already stated, the loop refinement takes place after the main transformation phase, hence, a preliminary CFG already exists. Anyhow, at this point, there does not exist any information about loops and their components. Indeed ASIS only provides information on the loop headers, but not their bodies. In addition, due to the considerable complexity of the traversal itself, it is easier to construct a raw version of the graph without extensive control flow semantics at first, and compute this information in the post transformation phase.

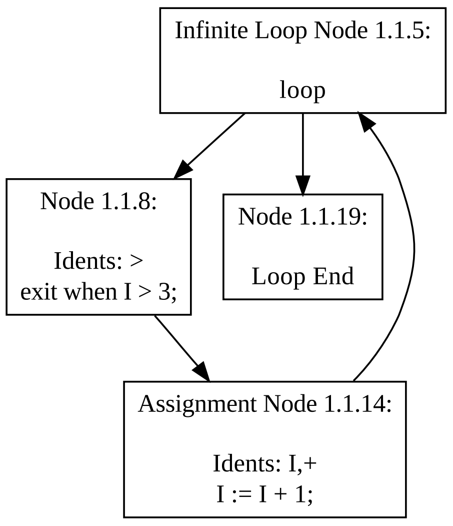


Figure 4.7: A simple `loop` before the loop refinement.

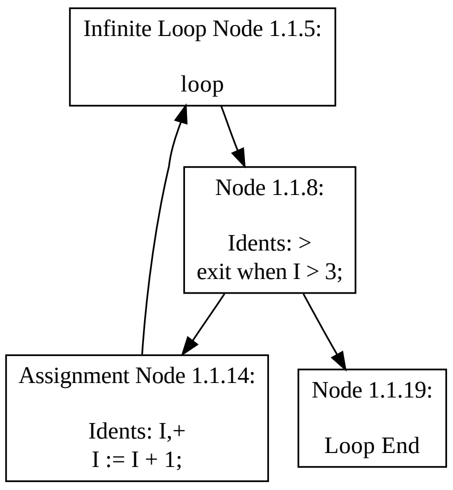


Figure 4.8: A simple `loop` after the loop refinement.

Consequently, the first task is to find the loops, which is done using Tarjan's algorithm for constructing the loop-nesting forest as introduced in Section 3.3.

After the backedges have been located with the algorithm outlined in Section 3.2, a modified version of Tarjan's algorithm for constructing a loop-nesting forest is used to find the loops and perform the refinement.

Actually, the refinement takes place in the `Collapse` procedure of the algorithm. It is called for every discovered loop, with the loop header and body as parameters, and subsequently collapses every node of the body on its header. Thereafter, the exit jump nodes with an empty target, thus the ones that exit the current loop, and those for outer loops are collected separately in two different lists.

Next, the edge from the loop header to the first statement after the loop is identified. This is, since the loop body is already known, accomplished easily by testing whether a successor of the loop header node is not within the body. Note, that in any case, there has to be exactly one such edge.

A trivial exit node, that is a node representing an exit statement without a condition such as `exit` or `exit LABEL` must not have an edge pointing to the unreachable statement right after it. Hence, for every exit node in one of the two aforementioned lists, that happens to be a `Trivial_Exit_Jump_Node_Object` the edge pointing to the node for the unreachable statement is added to the list of edges that will be removed at the end of the algorithm.

Now, every exit jump node without a label is connected to the first statement after the loop. In case the current loop has a label, also the list of labelled exit jump nodes is searched for matching nodes, which then are connected likewise. At this time the list with the unlabelled loop exits is empty, while the list for loop exits with targets may still contain exit nodes for outer loops. Consequently, this list needs to be preserved between different calls of `Collapse`. Note that with Tarjan's algorithm inner loops are always found first, so that in any case an exit node is found before the target loop.

Finally, in case the current loop is a simple loop statement, without a condition in the header, the edge from the loop header to the loop end is scheduled for removal.

After all loops have been processed that way, the edges that were collected for deletion are actually removed from the CFG. However, for example if a loop does not contain an exit statement, and therefore is an endless loop, it is possible that some nodes, representing the unreachable statements right after the loop, are not accessible any more by following only successor links. Likewise, statements following an exit without a condition may cause a similar problem. Apart from the problem of memory leakage, they still may be reached by traversing the CFG backwards, using predecessor links, which would lead to an unacceptable inconsistency. Hence, proper deallocation of those *dangling nodes* is necessary. Therefore, whenever an edge is removed, its target node is recorded and handled at the end of the loop refinement (see Section 4.5.4).

4.5.2 Connecting Gotos

During the transformation phase labels of statements and targets of a `goto` were saved as a string, separated by commas. However, an appropriate edge from the `goto` to the target node was not added. Instead, the node representing the `goto` was connected to the node for the statement right after it. During the post transformation phase, that edge is replaced by one representing the jump to the target correctly.

During a simple DFS, two lists are built: a list of sources containing all found goto jump node objects and a list holding all nodes with at least one label, that is all targets. Note that a goto statement also may have a label, and therefore could be on both lists simultaneously.

Thereafter, for each item in the list of sources the list of targets is searched for the corresponding label. When the target node was found, it is connected to its source, the node representing the goto statement. However, before this edge can be added, all existing edges of the source node have to be deleted first. This is to remove the auxiliary edge that connected the goto jump node to the node for the statement right after it. Until now, that auxiliary edge was needed in order to keep the CFG connected. As with endless loops, or return statements (cf. Section 4.5.1) there may be unreachable statements following a `goto`, which is why the target of the auxiliary edge is stored so that the nodes representing these statements may be removed and deallocated correctly at the end of this algorithm (see Section 4.5.4).

4.5.3 Connecting Returns

Return statements can be treated similar to goto statements since, seen from a control flow perspective, they are basically like a `goto` that has the end of the CFG as a target. Every CFG in the flow world is traversed using a simple DFS. Whenever a `Return_Node_Object` is encountered, the existing edge to the successor, the node representing the statement right after the return statement, is scheduled for removal, and the return node is connected to the end node of the CFG.

This was at first the main reason for the inclusion of an artificial end node in every CFG object, since it simplifies the task of connecting the return nodes significantly. However, at this point, it is not guaranteed that there is an end node, because it may have been removed together with unreachable nodes in a prior step, when the link from the return still was missing. So, in case the reference of a CFG to its end node is `null`, a new one is allocated.

As with other steps in the post transformation phase, at the end nodes that got unreachable are removed (see Section 4.5.4).

4.5.4 Removing Dangling Nodes

As stated previously, subgraphs that are no longer reachable by only following successor links, that is subgraphs representing unreachable statements, are removed after some of the steps in the post transformation phase. Apart from the waste of memory this is an important issue because those nodes would still be misleadingly reachable by a backwards traversal using predecessor links, which numerous algorithms rely on.

The subprogram that is responsible for removing those dangling nodes takes a list of possible root nodes of unreachable subgraphs as an argument. This list is easily built by the calling subprogram, because it just needs to add every target node of an edge that is removed.

First all nodes reachable through a DFS starting at the root node of the CFG are added to a set using the the `Hashed_Sets` implementation in the new container classes of Ada 2005. Subsequently, a DFS is started at every root node of a possibly unreachable subgraph and the encountered nodes are added to another set. Note that such an unreachable subgraph may still by connected to the enclosing CFG somewhere else.

Finally the set containing the nodes of the ordinary DFS is subtracted from the set containing the nodes of all unreachable subgraphs so that only those nodes that are not reachable through a DFS starting at the root node remain. These nodes are then disconnected from the CFG and deallocated properly.

As an example consider Figure 4.9, where solid edges represent successor links, while dotted lines correspond to predecessor links. Figure 4.9(a) shows a CFG before the goto node with the number 1 is connected to its target. Figure 4.9(b) depicts the CFG after the `goto` was connected to its target (Node 7), with a subgraph, rooted at Node 2, that is only reachable by following the predecessor link $(7, 6)$. In Figure 4.9(c) one can see the the two sets that are computed: the one surrounded by a solid edge containing the nodes reachable through a DFS starting at the root node, and the set of nodes reachable from the root of the unreachable subgraph which is enclosed by a dashed line. The difference leads to nodes 2-6, which are then removed as shown in Figure 4.9(d).

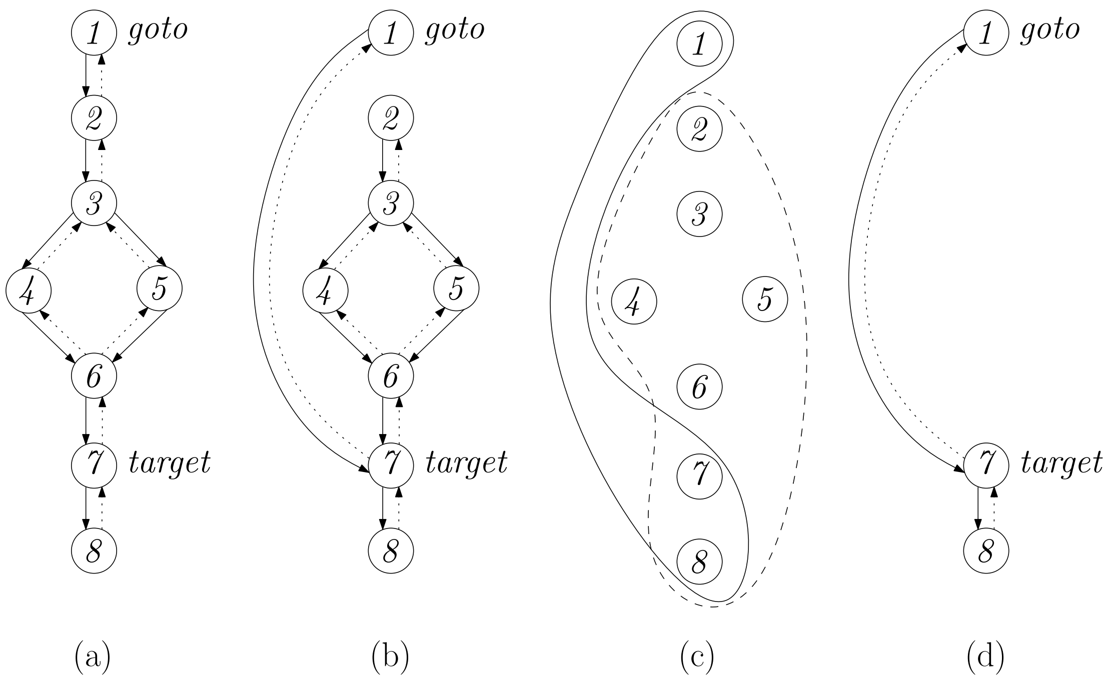


Figure 4.9: Dangling nodes being removed from the CFG.

4.5.5 Link Call Nodes

When a call node object (see Section 4.3.6) is first created during the transformation phase, normally it is not immediately linked to the CFG of the called entity since it may not yet exist at that point of time. Of course, statements that are on the one hand treated like parameterless procedure calls but on the other hand are created right after the artificial call node is inserted, like for instance block statements or accept statements (cf. Section 4.3.5), are an exception to this rule and the link is established right away. However, in the more common case of an ordinary call the ASIS element of the called entity is retrieved via an ASIS function and saved in the call node.

In the post transformation phase, the whole flow world is traversed in search for call nodes and parameter call nodes. This includes not only searching the individual nodes of a CFG but also the lists of variable declarations, formal declarations, renamings etc. that may occur in package and CFG objects. When a call node or parameter call node in the parameter tree of another node is encountered, the ASIS element of the called entity that was saved during the transformation phase is retrieved and every CFG object in the flow world is inspected for an equal ASIS element. In case the corresponding CFG object was found, the call node is linked to it, i.e., an access to the CFG object is saved in the call node.

4.5.6 Compress Parameter Trees

In the static context of this work it does not make sense to keep track of constant values, which is why parameter tree nodes are sometimes left empty rather that saving a constant value in them. At the end of the transformation phase, it can be feasible to compress parameter trees i.e. to remove unwanted empty nodes, subtrees or parameter trees that consist only of an empty root node without even a name.

However, due to the fact, that at higher compression levels information on the number and position of constant values gets lost, only the most basic one is implemented which is the removal of empty single node parameter trees.

Obviously the main task here is to traverse the whole flow world to examine every parameter tree at the various possible locations, because the removal of an unwanted parameter tree is trivial once it was found.

4.5.7 Computing Prefixes

Every node in the Pkg/CFG tree has a name, i.e. the name of the package or subprogram, and a full name which also contains, separated by dots, all names on the path to the node, starting at the root, but ignoring a possibly existent default package. The *prefix* is the full name without the name of the node itself. For instance, package `Baz` in Figure 4.10 has the prefix `Foo.Bar` and the full name `Foo.Bar.Baz`.

It is, again, most efficient to compute the prefixes in the post transformation phase. For each root package a procedure is called that saves an ordered list of the names previously encountered on the path in the current node, adds the current name to a copy of the list and then subsequently calls itself for every successor, passing the copy.

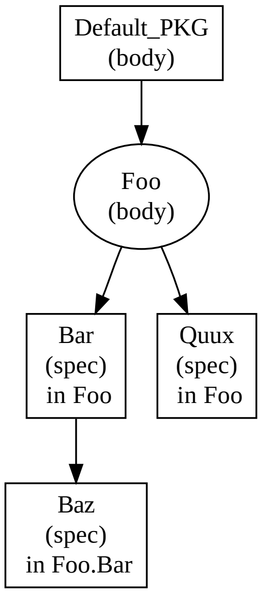


Figure 4.10: A Pkg/CFG tree with packages, subprograms and their prefixes.

As a result, the name, prefix and full name of a flow object may be constructed easily upon request by the application using Ast2Cfg.

Chapter 5
IMPLEMENTATION

5.1 Data Structures

This section describes some of the most important data structures used throughout the implementation of the Busy Wait Analyser for Ada (BWAA).

Although object orientation is supported since Ada 95 through tagged types, I decided to use simple record types in most cases. Since all the analysis algorithms are tightly coupled with a small set of data structures they operate on, encapsulation would have either led to a few huge objects or to loads of primitive get and set methods, which also would have led to bad design.

5.1.1 The Cfg Type

The `Cfg` type in `Bwaa.Graph` holds all information related to a CFG. While the `Cfg_Object` of Ast2Cfg (cf. Section 4.3.5) represents the flow of control by node objects which have pointers to their successors and predecessors, the `Cfg` type used in BWAA has two adjacency lists with DFS numbers: one for the successors and one for the predecessors. This representation using fixed size arrays instead of dynamic structures with pointers simply is more efficient as far as the algorithms used by BWAA are concerned, since information on a node with some DFS number can always be retrieved in constant time without the need to traverse the CFG.

As can be seen in Listing 5.1 the adjacency list actually is of type `Adjacency_Array`, which is an array of pointers to a type called `Vertex_Array` where a vertex array simply is an array of DFS numbers. The use of pointers is necessary here because the number of successors or predecessors is not known in advance.

However, sometimes it is necessary to get hold of a `Node_Object` as it is provided by Ast2Cfg. Therefore the array `Dfs_Sorted` saves pointers to the node objects sorted in DFS order. Both, the `Dfs_Sorted` array and the adjacency lists are built during initialisation of a `Cfg` variable with a `CFG_Object`.

During this initial DFS also the backedges are identified and saved as a `Vertex_Map` (see Section 5.1.3). Since there can be more that one backedge

```
type Cfg (Node_Count: Ast2Cfg.Long_Long_Natural) is limited
    record
        Dfs_Sorted: Node_Array(1 .. Node_Count) := (others => null);
        Succs     : Adjacency_Array(1 .. Node_Count) := (others => null);
        Preds     : Adjacency_Array(1 .. Node_Count) := (others => null);
        Id_Dfs    : Id_To_Dfs_Num.Map;
        Backedges : Vertex_Map.Map;
        Headers   : Vertex_Set.Set;
        L_Forest  : Loop_Forest_Ptr;
    end record;
```

Listing 5.1: The `Cfg` type

per loop, the key used for the vertex map is the source node of the backedge and the loop headers of a CFG are also saved as a `Vertex_Set` to allow for convenient access. In fact, because Ast2Cfg inserts end if and end case nodes after every `if` or `case` statement, which then link to the loop header with a single backedge, the only case where multiple backedges may point to a single loop header occurs when `goto` statements are employed like shown in Listing 5.2 and Figure 5.1.

```
procedure Backedges is
    X: Integer := 0;
begin

    <<START>>
    X := X + 1;
    if X < 10 then
        goto START;
    elsif X < 15 then
        goto START;
    end if;

end Backedges;
```

Listing 5.2: A loop with two backedges.

Another information gathered during the initial DFS is the mapping of id objects as provided by Ast2Cfg (see Section 4.3.3) to DFS numbers using a hashed map as described in Section 5.1.3. Finally, to create a variable of type `Cfg` the number of nodes has to be known, which is why I implemented a node counter in the Ast2Cfg project.

During a later phase of the analysis the loop forest is built and attached to the corresponding `Cfg`.

5.1.2 The Loop_Forest Type

Before a type for the loop forest can be declared in `Bwaa.Graph`, there has to be some representation for a single loop. Therefore I designed the type `L00p`, which has two zeroes in its name to avoid a conflict with the Ada keyword `Loop`.

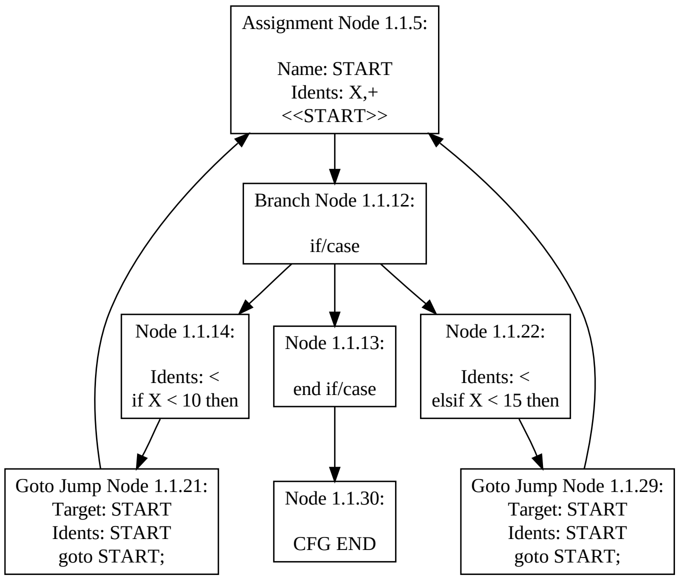


Figure 5.1: A loop with two backedges.

For every `L0Op` the DFS number of the header and a set of vertices which constitute the loop body is saved. Furthermore, a `L0Op` contains a reference to an array of variable size which contains the termination statements. The size does not really vary during execution, however, a pointer has to be used since the size is not known in advance. Finally, for every termination statement a hashed set with the busy wait candidate variables is saved. The `Variable` type contains the name of the variable, the name of the direct candidate if applicable and a boolean value indicating whether this variable is a wait variable.

A `Loop_Forest` has an array of `L0Op` variables, so that the index uniquely identifies a loop. Note that since the innermost loops are found first they are saved in reverse topological order. In addition, there is an array called `Preds` which records the nesting relation between the loops. If a loop is contained within another one, `Preds` holds the number of that loop, otherwise the value 0 is stored.

In order to instantiate a variable of type `Loop_Forest` the number of loops has to be known. However, this is no problem since the backedges are known before the loop forest is built and therefore also the number of loop headers, the targets of backedges, is known.

```
type L00p is
    record
        Header: Ast2Cfg.Long_Long_Natural;
        B0dy: Vertex_Set.Set;
        Terms: Vertex_Array_Ptr := null;
        Candidates: Variable_Set_Array_Ptr := null;
    end record;

type L00p_Array is array (Ast2Cfg.Long_Long_Natural range <>) of L00p;

type Loop_Forest (Loop_Count: Ast2Cfg.Long_Long_Natural) is
    record
        Preds: Vertex_Array(1 .. Loop_Count);
        Loops: L00p_Array(1 .. Loop_Count);
    end record;
```

Listing 5.3: The `L00p` and `Loop_Forest` type

5.1.3 Auxiliary Data Structures

In various packages I declared some auxiliary, more general data structures. `Id_To_Dfs_Num` is a generic instantiation of the `Hashed_Maps` package from the Ada 2005 container library. It is used to map an id of a node object as provided by Ast2Cfg back to the DFS number of the node. In order to minimise the lookup time I used the hashed version of the map container, and implemented `Ast2Cfg.Id.Hash`, which simply calls the predefined `Ada.Strings.Hash` on a string representation of the id.

A `Vertex_Map` maps the DFS number of a node to another one while a `Vertex_Set` simply is a set of DFS numbers. Both packages are used in many places throughout the BWAA project and use the hash function `Hash` (see Listing 5.4) which is implemented similarly to that for the `Ast2Cfg.Id`.

Finally, I declared a hashed set with unbounded strings as elements, where the hash function calls `Ada.Strings.Unbounded.Hash`. Throughout the project, I used this set mainly for storing variable names.

5.2 Initialisation

When a `Bwaa.Graph.Cfg` is initialised, first a DFS is performed on the corresponding `CFG_Object` from Ast2Cfg. Basically I implemented the DFS variant for finding the backedges as shown in Section 3.2. The implementation can be found in the `Bwaa.Traversal` package.

Throughout the DFS the array `Cfg.Dfs_Sorted` is filled with pointers to the node objects of the CFG as provided by Ast2Cfg. The position in the array is the DFS-number of the node. In order to be able to retrieve the DFS-number for a given node id, `Cfg.Id_Dfs`, a hashed map, is also constructed along the way.

```
package Id_To_Dfs_Num is new Hashed_Maps (Key_Type => Ast2Cfg.Id.Pointer,
                                          Element_Type => Long_Long_Natural,
                                          Hash => Ast2Cfg.Id.Hash,
                                          Equivalent_Keys => Ast2Cfg.Id."=");

function Hash (X: in Ast2Cfg.Long_Long_Natural) return Ada.Containers.Hash_Type;

package Vertex_Map is new Hashed_Maps (Key_Type => Long_Long_Natural,
                                       Element_Type => Long_Long_Natural,
                                       Hash => Hash,
                                       Equivalent_Keys => "=");

package Vertex_Set is new Hashed_Sets (Element_Type => Long_Long_Natural,
                                       Hash => Hash,
                                       Equivalent_Elements => "=");

function Hash (X: in Unbounded_String) return Ada.Containers.Hash_Type;

package Unbounded_String_Set is new Hashed_Sets (Element_Type => Unbounded_String,
                                                 Hash => Hash,
                                                 Equivalent_Elements => "=");
```

Listing 5.4: Auxiliary data structures from various packages.

After the initial DFS has finished, the `Cfg.Preds` and the `Cfg.Succs` array is filled, retrieving the predecessors and successors of every node in the `Cfg.Dfs_Sorted` array, using `Cfg.Id_Dfs` to get the corresponding DFS-numbers.

5.3 Constructing the Loop Forest

5.3.1 Tarjan's Algorithm

I implemented Tarjan's algorithm for constructing the loop forest (see Section 3.3) as a generic package called `Bwaa.Tarjan` . However, the only parameter that has to be supplied at instantiation is the size of the CFG. The size has to be known so that the `Union_Find` package (see Section 5.3.2) can be instantiated in the private part of the specification (see Listing 5.5). Therefore implementing `Bwaa.Tarjan` as a generic package is mainly a trick which allows the `Union_Find` package to be used conveniently.

`Bwaa.Tarjan` contains three procedures: `Build_Loop_Forest`, which is the only public subprogram, `Findloop` and `Collapse`.

One difference between my implementation and the algorithm, as it is presented in Section 3.3, is that since I did not need information provided by the loop parent array, I simply decided to skip its implementation. So when the procedure `Build_Loop_Forest` is called with a parameter of type `Cfg` (see Section 5.1.1), only the union find and loop forest data structure is initialised before `Findloop` is executed for every node in reverse-DFS-order. Note that the number of loops, which is needed to instantiate a `Loop_Forest`, is already known at this

```
generic
    Size: in Ast2Cfg.Long_Long_Natural;
package Bwaa.Tarjan is

    procedure Build_Loop_Forest (Cfg: in out Graph.Cfg);

private

    package Uf is new Union_Find(Ast2Cfg.Long_Long_Natural, 1, Size);

    Uf_Sets: Uf.Sets;
    Loop_Count: Ast2Cfg.Long_Long_Natural := 0;
    Header_To_Loop: Vertex_Map.Map;

end Bwaa.Tarjan;
```

Listing 5.5: The specification of `Bwaa.Tarjan`

point because it is equal to the number of backedges in the CFG. However, to program in a defensive manner, I decided to raise an exception in the impossible case that the number of backedges differs from the number of loops.

`Findloop` first checks whether the potential header that was given as a parameter is contained within the vertex map that contains the backedges. If so, the potential header is target of a backedge and in fact is a real header which has to be handled. Next, the canonical element of the set containing the node where the backedge originates is added to the worklist. I implemented the worklist and the loop body as vertex sets. Now elements are picked from the worklist until it is empty. Every element is first added to the loop body and then its predecessors are considered. If such a predecessor is not the origin of a backedge the canonical element of the set containing the predecessor is added to the worklist in case it is not already in the loop body and is different from the header.

In case the loop body is not empty, `Collapse` is called with the `Cfg`, the loop body, and the potential header as arguments. `Collapse`, which differs mostly from the collapse subprogram of the standard algorithm, first increases the `Loop_Count` which also serves as an id for the current loop. Then it adds the current loop to the loop forest and updates `Header_To_Loop` which is a vertex map that maps a loop header to the id of its loop.

Next, it iterates over the nodes in the loop body. After it collapses the current node on the header using the union find data structure, it checks whether this node is the header of a nested loop by testing if it is contained in the `Header_To_Loop` map. If so, the `Pred` array of the loop forest is updated to reflect this nesting relation.

The next action is an important addition because in Ada it is possible to exit an outer loop from within an inner loop. The nodes of the nested loop body are therefore also added to the outer loop body, so that the termination statements of a loop may be found correctly (see Sections 3.5 and 5.4).

5.3.2 The Set Union Algorithm

I implemented the union find algorithm presented in Section 3.4 as a generic package that allows the elements to be of any discrete type. This is because I used a simple array to hold the parent information, so that the value of the array at the position of a given element denotes its parent. In addition to the type of the elements the first and the last element have to be supplied at instantiation. If a parameter given to some subprogram is not within this range the program raises the `Out_Of_Range` exception.

In addition to the mandatory operations makeset, find and link I also implemented union and a procedure called `Init_Sets`, which performs makeset on every element. So after a call to `Init_Sets` every element is in a set on its own.

An important difference to the standard algorithm is the fact that the canonical element of a set must not change unexpectedly. This is because I used union find to implement Tarjan's algorithm for constructing the loop-nesting forest, where the canonical element always has to be the loop header. As a consequence, I was unable to implement union by rank, where the new canonical element, after a link or union operation, is chosen based on the rank. However, the worst-case bound of $\Theta(m\alpha(m,n)))$ is not affected by this change, since it also holds if only path compression is implemented [28].

Listing 5.6 shows the full specification for the union find package.

5.4 Termination Statements and Busy Wait Candidates

I implemented the search for termination statements as a single procedure in the `Bwaa.Analysis` package. The procedure iterates over all loops in the forest and checks for every header and node of the loop body whether one of the successors is not part of the loop. If this is the case it adds the current node to the vertex set that contains the termination statements for the current loop. Finally a vertex array of appropriate size is allocated and the nodes are copied into it. This is because I also save the busy wait candidates in an array, and need to access them with the same index as the corresponding termination node.

```
generic

    type Element is (<>);
    First: in Element;
    Last: in Element;

package Bwaa.Union_Find is

    type Sets is private;

    Out_Of_Range: exception;

    procedure Makeset (S: in out Sets; X: in Element);
    procedure Init_Sets (S: in out Sets);
    procedure Find (S: in out Sets;
                    X: in Element;
                    Canonical: out Element);
    procedure Link (S: in out Sets; X: in Element; Y: in Element);
    procedure Union (S: in out Sets; X: in Element; Y: in Element);

private

    type Sets is array (First .. Last) of Element;

end Bwaa.Union_Find;
```

Listing 5.6: The specification of the union find implementation.

The candidates for wait variables are collected by another procedure in `Bwaa.Analysis`. For each termination statement in a loop, it basically retrieves the right hand side parameter tree of the termination node and collects the variables, which are contained within the leaf nodes, by performing a DFS.

Note that the header node of a `for` loop (which is also one of the termination statements) never has a parameter tree, since the loop parameter and the components of the range expression are saved in a `Ast2Cfg.Flow_Types.Declaration`. This, however, is perfectly fine because a `for` loop cannot be a busy wait loop anyway, so the variables of a `for` loop statement need not be considered at all.

Another special case are loops that consist of `goto` statements in combination with branching statements such as `if` or `case`. In the current implementation of Bwaa and Ast2Cfg, the branch predicate does not need to be in the branching node itself. Consider the CFG in Figure 5.2 which was generated by Cfg2Dot (without parameter trees) from the code in Listing 5.7. The termination statement is in node 1.1.12 which is a branching node but contains no branching predicate. Instead there are multiple branching predicates which are already outside the loop: one in node 1.1.14 and one in node 1.1.22.

To solve this problem, the procedure which is responsible for finding the busy wait candidates first tests whether the termination node is of type `Branch_Node_Object`. If this is the case, it uses `Get_Idents` to check if there are any saved identifiers in this node. If there are identifiers, we can be sure that a

```
<<START>>
X := X + 1;
if X > 10 then
    goto ENDLOOP;
elsif Y > 1 then
    goto ENDLOOP;
else
    null;
end if;
goto START;
<<ENDLOOP>>
null;
```

Listing 5.7: Code snippet, Figure 5.2 was generated from.

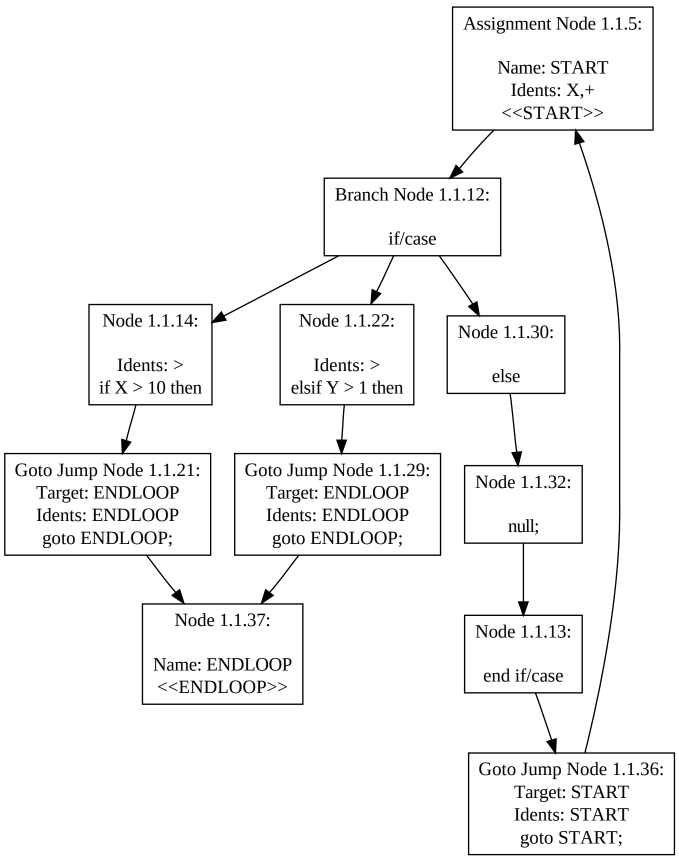


Figure 5.2: A loop where the branching predicate is not in the branching node.

case statement has been encountered because a branch node of an if statement is completely artificial and never has identifiers. Next the branch node is tested for a parameter tree, which, in case there is one, is searched for variables which can be added to the set of busy wait candidates. If there is no parameter tree, which can happen if only a single variable is used in the case statement, we can be sure that it has been added to the identifier list, so this time the identifiers of the node and not the parameter tree is used as a source for candidates. Note that the when branches of a case statement only contain constants which is why they are ignored.

If there are no identifiers in the branch node, there are two possible cases: First we may have encountered a case statement with a constant expression. Second we may have come across an if statement. While in the first case nothing special has to be done, in the second case every successor outside the loop has to be checked for candidate variables. So as candidate variables in the example in Figure 5.2 we get X and Y.

5.5 Indirect Busy Wait Candidates

In the Bwaa.Analysis package, I implemented the search for indirect busy wait candidates which is executed right after the procedure that is responsible for finding the ordinary candidate variables.

For every termination statement of a loop, first it initialises a worklist, which is implemented as a String_Set, with the direct candidates. While the worklist is not empty, it removes an element and checks every node in the loop, that is in the class wide type of an Assign_Node_Object, for indirect candidates induced by that element. For this purpose it first retrieves the left hand side parameter tree and collects its variables using the same procedure as described in Section 5.4. Since on the left hand side of an assignment statement there must be at most one variable, an exception is raised in case more than one are found. If the found variable is equal to the current element from the worklist, the variables from the right hand side parameter tree are collected. First the variables from the right hand side that are not already in the set of candidates or on the worklist (and therefore have not already been handled) are added to the worklist. Then the new variables are added to the set of candidates.

5.6 Aliasing

5.6.1 Specification Map

Although not directly related to the aliasing problem, it is, during the algorithms described in this section, of vital importance to be able to obtain the `Spec_Object` for a given `Body_Object`.

As can be seen in Figure 5.3, specifications and bodies declared in different enclosing units do not reside within the same Pkg/CFG tree. The bodies of the packages `Foo` and `Bar` declared in the file foo.ads shown in Listing 5.8 are specified in foo.adb, which is why they reside in the Pkg/CFG tree in Figure 5.3(b) and not together with their specifications in Figure 5.3(a). In contrast, since package `Baz` is completely specified within the declarative region of procedure `Qux`, specification and body are nodes of the Pkg/CFG tree in Figure 5.3(b).

```ada
package Foo is

    package Bar is
        procedure Qux;
    end Bar;

end Foo;
```

Listing 5.8: Contents of the file foo.ads used for Figure 5.3

```ada
package body Foo is

    package body Bar is

        procedure Qux is
            package Baz is
            end Baz;
            package body Baz is
            end Baz;
        begin
            null;
        end;
    end Bar;

end Foo;
```

Listing 5.9: Contents of the file foo.adb used for Figure 5.3

This is why I declared the `Specification_Map`, instantiated from the generic `Hashed_Maps` package of the Ada 2005 container library with an unbounded string as key and a flow class pointer as element type.

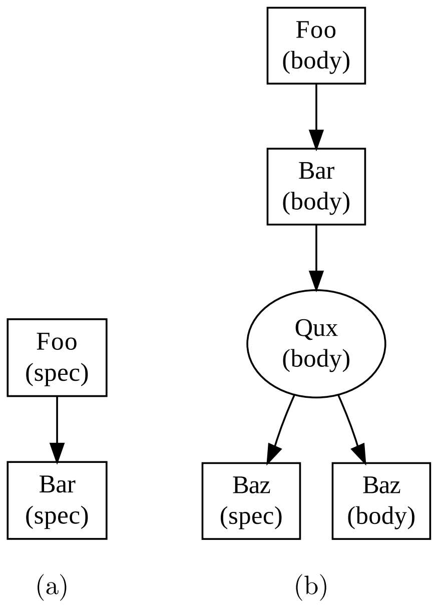


Figure 5.3: Specifications for a `Body_Object` can also be found in different Pkg/CFG trees.

Consequently, before the actual busy wait analysis starts, all Pkg/CFG trees are traversed recursively using a simple DFS, that whenever a package specification is encountered stores the flow class pointer in the `Specification_Map` using the full name of the `Spec_Object` as a key.

5.6.2 Visibility Map

The private type `Bwaa.Visibility.Declaration` holds the name of a declared object, its prefix and the declaration id object generated by Ast2Cfg. I defined several operators and subprograms, like for instance =, <, >, that allow for convenient analysis of the declaration order, not only limited to `Declaration` variables. Note that two variables of type `Declaration` are equal when their names are equal, regardless of their prefix and declaration id. This simply is the behaviour I need most during the analysis.

Since for every flow object a set of visible declarations has to be saved, I declared `Visible_Set`, a hashed set with `Declaration` as element type. In order to map a `Visible_Set` to its flow object I then make use use of a hashed map, `Visibility_Map`, with an unbounded string which holds the name of the flow object as key. Throughout the analysis four visibility maps are needed: two to record the visibility information for variables, divided into those declared in specifications and those declared in bodies and another two to save visibility information for packages in exactly the same way. To ensure convenient parameter passing, I embraced these four visibility maps in a single record, which I will refer to as *visibility info*.

Consequently, after the specification map is built and before the actual busy wait analysis, the visibility info data structure is populated. As a basis I once again used a DFS, however, when it comes to handling the corresponding specification of a flow object I had to deviate significantly from the usual DFS traversal order.

Starting at each root package, the visible variables and packages are collected in separate sets, recursively for each node in the Pkg/CFG tree. The declarations that should be visible in a node's successor are passed on as *inherited* variables and packages.

When the traversal reaches a specific flow object the first thing to do after checking whether this flow object has already been handled is to test whether it is a body or a spec object. In case of a body object the visibility info is searched for a spec object with the same name. When such a spec object is found, that is, when the body has a specification and it already has been visited, the declarations visible in the specification are added to the set of inherited variables and packages, respectively. Note that names from the specification may overwrite names inherited from the directly preceding flow object, which is why a set of inherited visible names after handling the specification I_{new} is the union of the names from the specification S and the difference of the directly inherited visible set I_{old} and S: $I_{new} = S \cup (I_{old} \setminus S)$. In this context it is important to remember that two `Declaration` variables are equal when they have equal names, so the elements of S added during the union may in fact be different from the ones removed from I_{old} right before, when all components are taken into account.

In case there is no visible set for the corresponding specification in the visibility info but a matching spec object is in the specification map, this means that the body has been reached by the traversal before the specification. Hence, normal DFS has to be interrupted and the visible set for the specification must be built. Figure 5.4 depicts the traversal of the Pkg/CFG trees from the example in Section 5.6.1 (omitting the retrieval of the visible set for the specification of `Foo`). The main traversal is outlined by the thick, dashed lines, starting at the body of `Foo`. When the body of `Bar` is reached, the flow class pointer to its spec object is retrieved from the specification map. Since it is not known at this point whether the preceding nodes that might affect the visible set of `Bar` (spec) have already been handled, the root node of the tree is determined by iteratively following the predecessor links. When the root node is reached, recursive traversal is started there, using the same subprogram that was called for the main traversal. Since already handled nodes are detected, this does not mean an additional effort but only that the processing of another Pkg/CFG tree is brought forward. When the traversal of the Pkg/CFG tree containing the desired spec object has finished, the visibility set of the specification is available and the inherited sets are adjusted as I just described above.

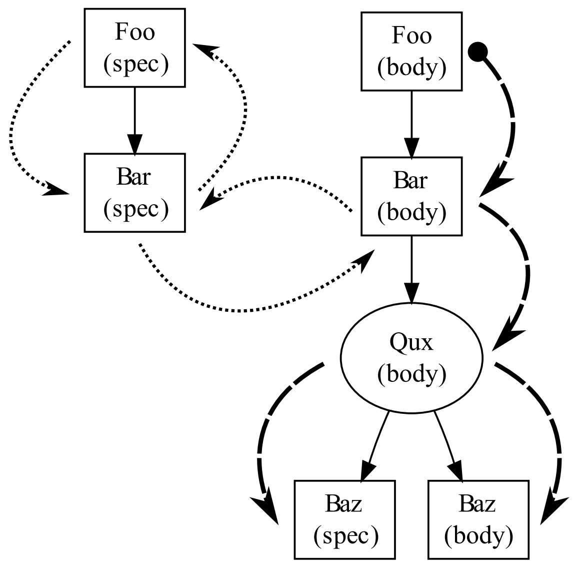


Figure 5.4: Creating the visible set for the body of `Bar`.

Finally, in the last possible case the current body object simply does not have a corresponding spec object, which is impossible for a package body but common among subprogram bodies.

Next, two sets are filled with variables, packages, etc. that are declared locally. In the case of subprogram parameters, variables and renamings I simply add a `Declaration` for each suitable element of the lists provided by the flow object to the appropriate set. Note that at this point only the new name of a renaming declaration is saved, because the visibility map is not responsible for recording aliases. Subsequently I also add a `Declaration` for every package declared within the current flow object, i.e. the successors that are of type `Pkg_Object`, and the current flow object itself in case it is a package.

At this point it is time to save the visible sets for the current flow object in the appropriate visibility maps of the visibility info. However, right before I ensure that the locally declared entities overwrite inherited ones by building a visible set V from inherited visible names I and local ones L according to $V = L \cup (I \setminus L)$, utilising the special behaviour of the equality operator for the `Declaration` type.

Finally, every successor of the current flow object is handled recursively. Obviously, only those entities declared before the declaration of a successor can be visible there or overwrite others. So for every successor every `Declaration` variable in the local set is examined and added to the set of variables to be inherited only in case it is declared before the successor. Of course, also the elements of the visible sets inherited by the current flow object itself are passed on, after the elements overwritten by the local declarations are dismissed.

For an example illustrating the issue of declaration order and inheritance of visible variables see Listing 5.10. Variable `C` does rename `Foo.Bar.A` since `Foo.Bar.A` is declared before `Baz` and therefore overwrites `Foo.A`. In contrast `D` renames `Foo.B`, because `Foo.Bar.B` is declared after `Baz` and therefore does not overwrite `Foo.B`.

```
package Foo is

    A: Integer;
    B: Integer;

    package Bar is
        A: Integer;

        package Baz is
            C: Integer renames A;
            D: Integer renames B;
        end Baz;

        B: Integer;
    end Bar;

end Foo;
```

Listing 5.10: Three packages to demonstrate the effect of the order of declarations.

5.6.3 Hashed Union Find

Obviously, for each variable a set of names, i.e. aliases, has to be maintained. The by far most used operation on these sets is to determine whether two names reside within the same set. For that purpose simple lists would be rather inefficient, since the alias lists for all variables would have to be searched with two comparisons per element and, in the worst case, with the first match being in the last set.

Another frequently used operation is the union of two sets, which occurs whenever a new pair of equal names is found, but both names were already added earlier. Therefore, I decided to use a special union find (see Section 3.4) implementation based on hashed maps and employing path compression and union by rank.

In order to find out whether two elements are within the same set, two find operations have to be performed and the returned canonical elements have to be tested for equality. The path compression, however, guarantees that after a sufficient number of find operations this can be done in constant time.

Since I specifically designed this union find implementation to handle elements of string types and the number of elements is not known in advance, this time a simple array for storing the parent information was not sufficient. A hashed map, as provided by `Ada.Containers.Hashed_Maps` in the Ada 2005 container library, however, not only solves the problem of the dynamic number of elements but also provides an efficient way to access a specific element. In order to keep track of the rank of an element, I use a second hashed map.

Listing 5.11 shows the public part of the `Hashed_Union_Find` specification. As generic formal parameters not only the type of the elements, a hash function, and a function for testing equality have to be provided, but also a `Null_Element`. This null element must not be in the set of possible element values during normal operation and is used for implementation of the cursor type that iterates over elements of the same set described below. For example, in case of a string type used to represent variable names, the empty string would qualify as the null element, since a variable name must at least contain one character.

Apart from the classic operations of the set union algorithm such as `Makeset`, `Find`, `Link` and `Union`, I implemented a series of new features. The most noticeable ones make it possible to iterate over all elements of all sets, or just over the elements of a specific set using a similar cursor based approach as the Ada 2005 container library.

Iterating over all elements regardless of the set they belong to is only needed for debugging purposes and introduces no runtime issues. In contrast, iteration over all elements of a specific set in a union find based structure is much more inefficient in comparison to an approach based on lists, since it is in $\Theta(n)$ where n is the number of all elements in all sets, regardless how many elements are in the set that should be iterated on. Anyhow, I use this feature very seldom and the considerable advantages of the union find approach concerning find and union operations are by far more important.

When the function `First` is called with a variable of type `Sets` as the only parameter, a cursor that iterates over all elements in all sets is returned. In case the procedure `First` is called with a `Sets` variable, an element and a cursor, the cursor is set to the first element in the same set as the given element and stays within that set.

The functionality of the subprograms `Next`, `Has_Element` and `Element` is quite obvious and corresponds to how their counterparts in the Ada 2005 container library work. The function `Parent` returns the parent of the current element in the union find tree and is intended for debugging purposes only.

The use of the subprograms `Length`, `Contains` and `Clear` is self-evident too, however, note that they operate on the elements of all sets.

```
generic

    type Element_Type is private;
    Null_Element: in Element_Type;
    with function Hash (Elem: Element_Type) return Ada.Containers.Hash_Type;
    with function "=" (Left, Right : Element_Type) return Boolean is <>;

package Bwaa.Hashed_Union_Find is

    type Sets is private;
    type Cursor is private;

    No_Element: constant Cursor;

    Hashed_Union_Find_Exception: exception;
    Not_In_Any_Set_Exception: exception;

    procedure Makeset (S: in out Sets; X: in Element_Type);
    procedure Find (S: in out Sets; X: in Element_Type; Canonical: out Element_Type);
    procedure Link (S: in out Sets; X: in Element_Type; Y: in Element_Type);
    procedure Union (S: in out Sets; X: in Element_Type; Y: in Element_Type);

    procedure First (S: in out Sets; X: in Element_Type; Cur: out Cursor);
    function  First (S: in Sets) return Cursor;
    function  Next (Position: in Cursor) return Cursor;
    procedure Next (Position: in out Cursor);
    procedure Next (S: in out Sets; Position: in out Cursor);
    function  Has_Element (Position: Cursor) return Boolean;
    function  Element (Position: in Cursor) return Element_Type;
    function  Parent (Position: in Cursor) return Element_Type;

    function  Length (S: in Sets) return Ada.Containers.Count_Type;
    function  Contains (S: in Sets; X: in Element_Type) return Boolean;
    procedure Clear (S: in out Sets);

private
    ...
```

Listing 5.11: The public part of the specification of the hashed union find package.

5.6.4 Alias Collection Overview

The alias info data structure holds information on names that refer to the same variable or package and is built for every CFG during the analysis phase. It enables convenient and efficient aliases lookups during the rest of the analysis.

The `Alias_Info` type is a record containing two `Alias_Sets.Sets`, one for variable aliases and one for package aliases. `Alias_Sets` is an instantiation of the generic `Hashed_Union_Find` package introduced in Section 5.6.3, using an `Unbounded_String`, the `Null_Unbounded_String` and the accompanying hash function as generic formal parameters.

The collection of aliases for the currently analysed CFG is done right before aliasing information is needed the first time, i.e. before the search for busy wait candidates and after the termination statements have been determined. The alias collection starts at the analysed CFG and basically continues upward until the root of the Pkg/CFG tree is reached. In case a flow object has a specification, aliases also are collected there.

The aliases of a flow object are collected in the following order:

1. parameter declarations of the current flow object (in case it is a subprogram)
2. variable declarations of the current flow object
3. package declarations of the current flow object
4. declarations/renamings of the packages contained in this flow
5. declarations/renamings of the packages included by a `with` clause
6. renamings of the current flow object

Adding a declaration (Steps 1–3) is, apart from the source of the declarations, similar in all three cases. It is done by first adding a *prefixed series* to the alias sets which I describe in detail in Section 5.6.5. In short, a prefixed series is composed of the name preceded by all possible prefixes. For instance, a variable `X` with the full name `Foo.Bar.Baz.X` might have the prefixed series `Foo.Bar.Baz.X`, `Bar.Baz.X`, `Baz.X`. Next, a new set for each name is created using the `Makeset` operation, and finally the sets with the name and the full name, that was added with the prefixed series, are united. The order of these operations is important and as a result, sets of names for this parameter that were added earlier are also united.

Next, the aliases from packages contained within the current flow are collected. This is necessary since, although not directly visible, variables, packages, or renamings from a lower level in the Pkg/CFG tree may be referenced using an appropriate prefix. In the simple example in Listing 5.12 variable `X` in package `Bar` may be referenced from within procedure `Proc` using the aliases `Bar.X` and `Foo.Bar.X`.

In order to collect such aliases, another recursive DFS traversal is started at every package in the list of successors. The example in Figure 5.5 and Listing

```
package Foo is
    package Bar is
        X: Integer;
    end Bar;

    procedure Proc;
end Foo;
```

Listing 5.12: Simple example for a visible variable in a subpackage.

5.13 shows such a traversal starting at the specification of `Bar` which contains `Baz` and `Qux`. The thick, dashed lines illustrate the main aliasing collection starting at `Proc` going upward until the root package is reached. The dotted lines mark deviations like the handling of specifications and subpackages.

```
procedure Foo is

    X: Integer;

    package Bar is

        package Baz is
            Y: Integer renames X;

            package Qux is
                Z: Integer renames Y;
            end Qux;
        end Baz;

        package Quux is
            procedure Proc;
        end Quux;

    end Bar;

    package body Bar is

        package body Quux is
            procedure Proc is
            begin
                null;
            end;
        end Quux;

    end Bar;

begin
    null;
end Foo;
```

Listing 5.13: The source code used for Figure 5.5.

When a node is reached during this traversal, first the variable and package declarations are collected, however, since they are not directly visible, this time only a prefixed series is added. Furthermore, this prefixed series must not contain names starting with a package below the root of the subtree. For instance, the prefixed series added for `Y` contains `Foo.Bar.Baz.Y` and `Bar.Baz.Y` but not `Baz.Y` or `Y`.

After the variable and package declarations, the renamings are collected. For

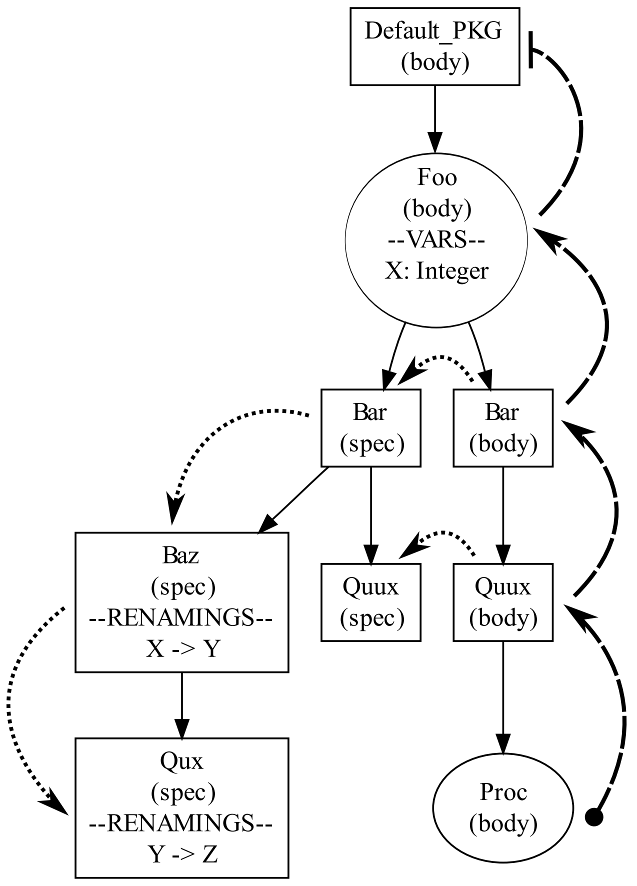


Figure 5.5: Collecting aliases from a subtree.

each element in the list of renamings of the current flow object it is tested whether it is a variable or package renaming. First, the old name of a renaming i.e. the name of the renamed entity is examined and the full name is retrieved. In case the old name does not contain a prefix, the full name is constructed using the method I outline in Section 5.6.6 based in the visibility info. Otherwise the potentially partial prefix is completed by searching the respective entity in the Pkg/CFG tree as I will explain in Section 5.6.7.

Once the full name of the renamed entity is found, it is, together with the new name added to the alias info using the `Makeset` operation and uniting the two resulting sets. In case a set containing one of the names already existed previously, no new set is created but the `Union` operation nevertheless merges both sets to reflect the renaming, possibly merging two sets already containing numerous aliases. Thereafter, an aliased series for the new name is added, following the restrictions concerning the first prefix element as described above.

However, in case of a package alias already collected aliases may be affected as they might have the renamed package name contained in their prefix. Therefore, variable and package aliases collected until this point have to be updated as I will outline in Section 5.6.8.

Finally, before the next node in the subtree is visited recursively, the packages included by with clauses are identified using the specification map and a collection is started there following the exact same algorithm as the collection from packages contained within a flow object. That is, a package included by a with clause is handled exactly the same way as when it was declared in the flow object the with clause is associated to.

When the collection of aliases of all packages contained within a flow object is done, it is time to collect aliases from the packages included by with clauses and finally collect the renamings.

After the full old name of each renaming has been found the same way I explained previously in this section, the aliases induced by the renaming are added to the alias info. The only difference in the algorithm between variable and package renamings is, that after adding a package alias, already existing aliases need to be updated.

First, it is tested whether the new name is already contained in the alias info. If this is the case, an alias for the full old name and the new name must not be added, since apparently another variable with the same name has been encountered on a lower level in the Pkg/CFG tree and overwrites the currently found new name.

As an example consider variable `Foo.C` and `Foo.Proc.C` in Listing 5.14 when procedure `Proc` is analysed. When the traversal has left `Proc` for `Foo` a set containing `Foo.Proc.C`, `Proc.C` and `C` already exists. Adding an alias mapping `Foo.A` to `C` would wrongly establish equality between `Foo.A` and `Foo.Proc.C`. Otherwise, an alias for the full old name and the new name is added. For instance the renaming in line 6 causes a set containing `Foo.B` and `D` to be created.

Next, it is tested whether the old name is not already present in the alias info (`B` in line 11) or was just added at the same level (`E` in line 12). If so, an alias for old and new name is added, for instance (`E`, `B`).

In any case, the full old name is mapped to the full new name, and a prefixed series is added for the new name. To conclude the current example, Table 5.1 reflects the alias info data structure for the analysis of `Proc` during traversal of the two levels of the Pkg/CFG tree.

5.6.5 Adding a Prefixed Series

As stated above, a prefixed series is a series of names for a variable or package composed using different variations of a prefix. When adding such a series the position of the name to be prefixed within the Pkg/CFG tree has to be taken into account in order to correctly reflect the visibility.

Furthermore it is important to keep track which of the segments separated by a dot belong to the prefix, and which to the base name. A component `X` of a record `Rec` with the full name `Foo.Bar.Rec.X`, for instance, has, assuming direct visibility, the prefixed series `Rec.X`, `Bar.Rec.X`, `Foo.Bar.Rec.X` where `X` must not be included.

Another problem that has to be taken care of when adding a prefixed series is caused by existing package aliases. Consider for instance Listing 5.15. The package alias mapping `Foo.Bar` to `Foo.Bar.Baz` has already been added when the renaming of `Foo.X` in line 7 is encountered.

Hence, whenever a new prefix is composed, the package alias set containing this prefix is determined and for every element within that set another alias is added after verifying that it is not a circular alias (see Section 5.6.9).

```
 1  procedure Foo is
 2      A: Integer;
 3      B: Integer;
 4      C: Integer renames A;
 5      D: Integer renames B;
 6
 7      procedure Proc is
 8          A: Integer;
 9          C: Integer;
10          E: Integer renames B;
11          F: Integer renames E;
12      begin
13          null;
14      end;
15
16  begin
17      null;
18  end Foo;
```

Listing 5.14: Example illustrating visibility issues when adding a simple renaming.

Variable	`After Proc`	`After Foo`
`Foo.A` (line 3)	-	`Foo.A`
		`Foo.C`
`Foo.B` (line 4)	`B`	`B`
	`E`	`D`
	`F`	`E`
	`Foo.B`	`F`
	`Proc.E`	`Foo.B`
	`Proc.F`	`Foo.D`
	`Foo.Proc.E`	`Proc.E`
	`Foo.Proc.F`	`Proc.F`
		`Foo.Proc.E`
		`Foo.Proc.F`
`Foo.Proc.A` (line 9)	`A`	`A`
	`Proc.A`	`Proc.A`
	`Foo.Proc.A`	`Foo.Proc.A`
`Foo.Proc.C` (line 10)	`C`	`C`
	`Proc.C`	`Proc.C`
	`Foo.Proc.C`	`Foo.Proc.C`

Table 5.1: Aliases for the variables in Listing 5.14 after the collection is finished for `Proc` and for `Foo`.

```
package Foo is

    X: Integer;

    package Bar is
        package Baz renames Bar;
        Y: Integer renames Foo.X;
    end Bar;

    procedure Proc;

end Foo;
```

Listing 5.15: Package renamings have to be taken into account when adding a prefixed series.

In the current example, the prefixed series `Bar.Y`, `Foo.Bar.Y` is therefore augmented with `Bar.Baz.Y` and `Foo.Bar.Baz.Y` while avoiding circular aliases like `Bar.Baz.Baz.Y`, `Bar.Baz.Baz.Baz.Y` and so forth.

When a prefixed series for a package alias is added, the existing aliases may be affected, which is why after adding every element of the series variable and package aliases are updated.

5.6.6 Discovering the Full Name Using Visibility Info

When a renaming does not specify a prefix for a variable or package to rename, the full name has to be constructed using visibility information. This is straightforward in most cases since the visibility info data structure already contains only those entities that are visible in each flow object and therefore not only helps to find the full name but also to distinguish between multiple entities with the same name but different visibility. So in the most basic case the visible set of the flow object containing the renaming is searched until a `Declaration` with a smaller declaration id than the renaming is found.

However, there is one special case that has to be taken into account for which Listing 5.16 shows an example. The declaration of `Foo.Bar.X` in line 7 overwrites variable `Foo.X` declared in line 3 for the renaming in line 8, but does not affect the renaming in line 6. So in fact variable `Foo.X` is renamed to `Foo.Bar.Y` and variable `Foo.Bar.X` to `Foo.Bar.Z`.

When there is one set for each flow object that holds all visible declarations it is clearly impossible to reflect visibility at each single point in the flow object. So when the visible set for `Foo.Bar` is constructed `Foo.X` is deleted from the set in favour of `Foo.Bar.X`. Hence, when there is no variable named `X` found that has a smaller declaration id than the renaming in line 6, the predecessor and

```
package Foo is

    X: Integer;

    package Bar is
        Y: Integer renames X;
        X: Integer;
        Z: Integer renames X;
    end Bar;

    procedure Proc;

end Foo;
```

Listing 5.16: Special case where the enclosing flow object has to be searched.

the specification of the predecessor are searched for the appropriate `X`. Note that this also works when the variable that is being searched for is not declared in the directly enclosing flow object but at a higher level in the Pkg/CFG tree, because it is inherited in all visible sets below the original point of declaration unless not overwritten.

5.6.7 Discovering the Full Name Using a Partial Prefix

When a entity is referenced using a prefix while being renamed, the prefix might have to be completed in order to obtain the full name. This is done starting at the flow object containing the renaming and then traversing the Pkg/CFG tree recursively upwards.

At each level, first the head of the prefix i.e. the segment before the first dot is compared to the name of the current flow, the names of the successors including the successors of its corresponding specification that are packages and the new names of package renamings contained within this flow. In case the strings match, the correct flow object has been found and the prefix is completed using the name or prefix of the flow object.

As an example for these three cases consider Listing 5.17. `Bar.A` (line 7) is completed to `Foo.Bar.A` because, when reaching `Foo.Bar` in the Pkg/CFG tree, `Bar` matches the name of the current flow. `Baz.B` (line 10) and `Baz` (line 12) are completed to `Foo.Bar.Baz.B` and `Foo.Bar.Baz` respectively, because `Baz` equals the name of a successor of `Foo.Bar`. Finally, `Qux.B` (line 14) is completed to `Foo.Bar.Qux.B`, because `Qux` is found in the list of package renamings in `Foo.Bar`.

```
package Foo is

    package Bar is
        A: Integer;

        package Baz is
            B: Integer renames Bar.A;
        end Baz;

        C: Integer renames Baz.B;

        package Qux renames Baz;

        D: Integer renames Qux.B;

    end Bar;

    procedure Proc;

end Foo;
```

Listing 5.17: Different situations for completing a prefix.

5.6.8 Updating Aliases

At various positions in the main aliasing algorithm it is necessary to update existing aliases, mainly when a new package alias is introduced. I basically use the same algorithm to update package and variable aliases. The only difference is that for variable aliases the prefix has to be separated from the name before the update is done. This has to be done in a way that ensures that component names are handled correctly.

Essentially, it is tested whether the old name of the newly added alias can be found at the beginning of the old name of an already existing alias, while taking correct prefix structure into account. Whenever such an alias is found, a new name is created where the part of the old name of the existing alias is substituted by the new name of the new alias. Next, a new alias mapping the old name of the existing alias to the newly constructed name is added, in case it is not circular (see Section 5.6.9).

Listing 5.18 shows a simple example where the renaming of package `Bar` in line 7 causes two updates to the existing aliases. When the alias mapping `Foo.Bar` to `Baz` is added, the variable aliases are updated by adding a mapping between `Foo.Bar.X` and `Baz.X`. Adding the alias that maps `Foo.Bar` to `Foo.Baz` results in the new name `Foo.Baz.X` for `Foo.Bar.X`. Note, that in this simple example the updates to the package aliases have no effect, since the package alias added in the first place already contains all names that have to be added because there is no other package that contains `Bar` in its prefix.

```
package Foo is

    package Bar is
        X: Integer;
    end Bar;

    package Baz renames Bar;

    procedure Proc;

end Foo;
```

Listing 5.18: A simple example to illustrate alias updating.

5.6.9 Circular Package Renamings

In Ada it is possible to rename a package wherever that package is visible, including the subtree of the Pkg/CFG tree that is rooted in the package that is actually renamed. This introduces circular dependencies between elements of a prefix for every entity declared within that package and leads to an infinite amount of aliases for that entity.

In Listing 5.19 the most elementary way to declare a circular package renaming is shown. `B` is located within `A` but in fact is just an alias for `A`. This leads to

```
package A is
    X: Integer;

    package B renames A;

    procedure Proc;
end A;
```

Listing 5.19: A simple circular package renaming.

the infinite set of aliases for variable `A.X` described by the regular expression `(A.)?(B.)*X`, that is, the set containing the following names:

```
X
A.X
B.X
B.B.X
B.B.B.X
...
A.B.X
A.B.B.X
A.B.B.B.X
...
```

Of course this is just a very simple example which reveals only a small part of the complexity of circular renamings. Consider for instance renamings of packages that are circular renamings themselves, circular renamings spanning multiple levels in the Pkg/CFG tree and so on.

Implementing an algorithm and data structures that allow for a complete handling of such circular renamings turned out to be not feasible for this project. First of all, the usage of such renamings in real world software can be expected to be very infrequent and would definitely be very bad programming practice since clarity not confusion should be the goal of every programmer regarding style. Furthermore, a missing alias leads to false positives and never to false negatives as far as the busy wait detection algorithm is concerned. Hence, the most reasonable decision was to detect circular renamings using the old and new name of an alias that should be added and the existing alias info data structure, add at the least the first alias of the infinite series to the alias info and ignore the others to prevent endless loops.

For the example in Listing 5.19 this means, that the following five aliases are added, three of them involving the package from the circular renaming:

```
X
A.X
B.X
A.B.X
B.B.X
```

Chapter 6
EXAMPLES

Finally, I present two simple examples for busy waiting and outline how the analysis algorithm reacts in these cases before I conclude each example with the corresponding output of BWAA.

6.1 Dekker's Mutual Exclusion Algorithm

Figure 6.1 shows the annotated CFG of Dekker's mutual exclusion algorithm as presented in Section 2.2. First, the two backedges are identified and the loop forest is built. In this case there are only two loops with loop 1 being contained in loop 2. Remember that inner loops are found first and that the nodes of a contained loop are added to its enclosing loop too.

Termination statements have at least one edge pointing to a node outside the loop, which is easy to see for nodes 1.4.22 and 1.4.7 in this example. The variables of these termination statements are then added to the busy wait candidate variables for each termination statement: `Turn` is a candidate in termination statement 1 and `T2` for termination statement 2.

Neither `Turn` nor `T2` are defined on a path in the corresponding loop that starts and ends at the termination statement. Hence, indirect candidates do not exist, aliasing has no effect, and both variables are wait variables. Listing 6.1 shows the verbose output of BWAA for `Task1`.

6.2 Advanced Example

I created the example in this section for demonstration purposes only. It shows a few, more advanced concepts of the busy wait analysis than Dekker's algorithm in the previous section.

The task `Counter` in Listing 6.2 iteratively increments a variable and prints its value until a maximum value is reached. The main task, procedure `Example` itself, waits until the count has reached the maximum value and also outputs a line whenever the counter has reached a multiple of ten.

In this case, two CFGs have to be analysed: the CFG for procedure `Example`

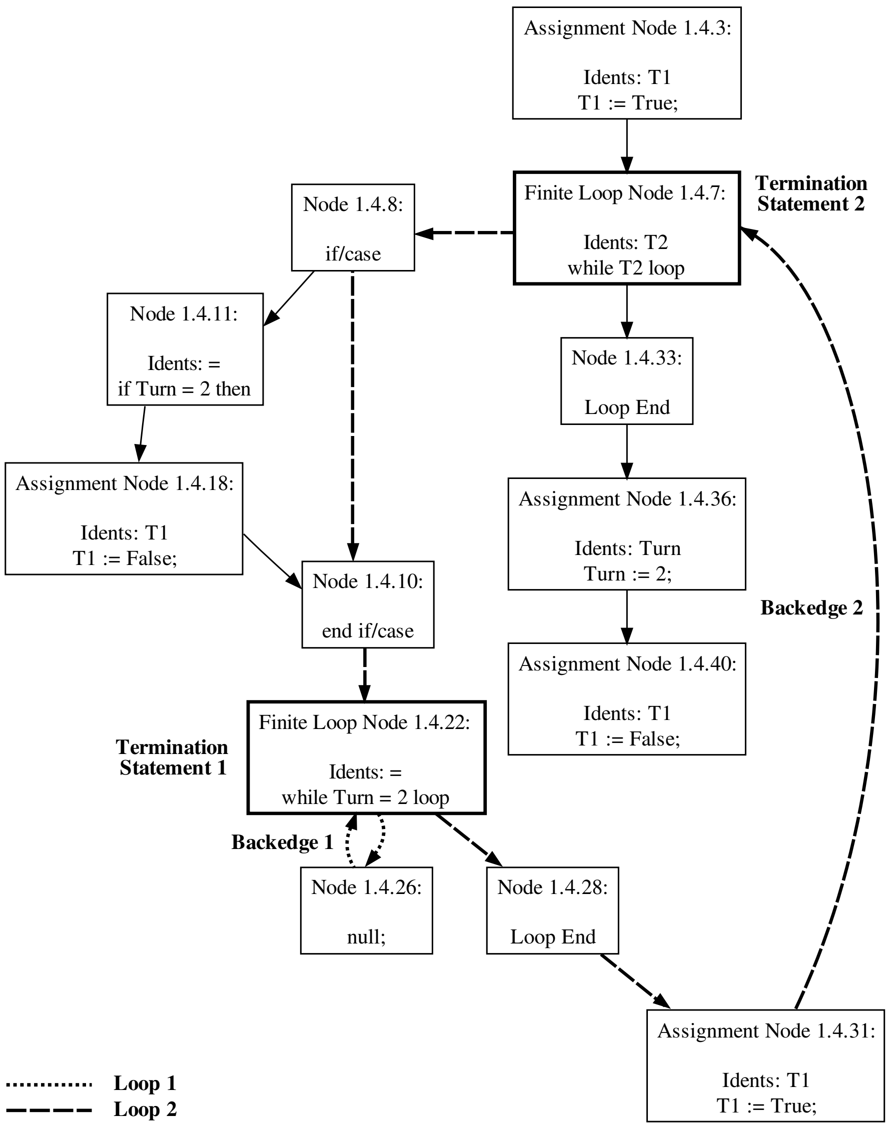


Figure 6.1: Backedges, loops and termination statements in the CFG of Dekker's mutual exclusion algorithm.

```
[Bwaa] Potential busy waiting with variable Turn in line 17 in loop starting at
line 17 in Dekker in /path/to/dekker.adb

 13:         while T2 loop
 14:            if Turn = 2 then
 15:               T1 := False;
 16:            end if;
 17:            while Turn = 2 loop
 18:               null;
 19:            end loop;
 20:            T1 := True;
 21:         end loop;

_______________________________________________________________________

[Bwaa] Potential busy waiting with variable T2 in line 13 in loop starting at
line 13 in Dekker in /path/to/dekker.adb

 10:      task body Task1 is
 11:      begin
 12:         T1 := True;
 13:         while T2 loop
 14:            if Turn = 2 then
 15:               T1 := False;
 16:            end if;
 17:            while Turn = 2 loop
 18:               null;
 19:            end loop;
```

Listing 6.1: Output of BWAA for `Task1` of the example implementation of Dekker's algorithm.

in Figure 6.3 and the one for task `Counter` depicted in Figure 6.2. `Counter` has only one loop with a single backedge and termination statement. Direct busy wait candidates are `Counting.Max` and `Counter.I`, however, thanks to the aliasing data structures, the assignment in line 23 of Listing 6.2 correctly adds `C.Gap` as an indirect busy wait candidate. Since `G` in line 22 is an alias for `C.Gap`, this variable definition is also taken into account. For this reason there is no path in the loop starting and ending at the termination statement that does not define `C.Gap` and `Counter.I`, which is why they are no wait variables.

Anyhow, `Counting.Max` is not defined and therefore reported as a wait variable, as can be seen in the BWAA output for this example in Listing 6.3. To avoid false positives like this, there is a command line option that only reports busy waiting when not a single candidate variable in a termination statement is properly defined. Note, however, with this option false negatives might be introduced.

The CFG for procedure `Example` can be found in Figure 6.3. It contains two loops that share the termination statement in node 2.2.49. For termination statement 1 there is only one busy wait candidate, `Cur`, and for termination statement 2 `Cur` and `C.Max` are candidates. Without processing of indirect candidates, the assignment in line 36 of Listing 6.2 would prevent `Cur` from being recognised as a wait variable. With indirect candidate processing enabled, however, `C.Current` is added as a candidate and `Cur` is reported as a wait variable for both termination statements. Of course, `C.Max` is a wait variable too and is therefore also reported in the BWAA output in Listing 6.3.

Note, that both loops in procedure `Example` are busy wait loops and the fact that they share a termination statement is also reflected in the BWAA output.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Example is

    package Counting is
        Current: Natural := 0;
        Gap: Natural := 1;
        Max: Natural := 30;
    end Counting;

    task Counter;

    package C renames Counting;

    task body Counter is
        I: Natural renames C.Current;
        N: Natural renames C.Max;
        G: Natural renames C.Gap;
    begin
        while I <= Counting.Max loop
            Put_Line(Natural'Image(Counting.Current));
            G := I + 1;
            Counting.Current := C.Gap + 1;
            delay 1.0;
        end loop;
    end Counter;

    Cur: Natural := C.Current;

begin

OUTER:
    loop

        loop
            Cur := C.Current;
            exit OUTER when Cur > C.Max;
            exit when Cur mod 10 = 0;
        end loop;

        Put_Line("Counter␣reached␣" & Natural'Image(Cur));
        delay 1.0;

    end loop OUTER;

    Put_Line("Counting␣stopped␣at␣" & Natural'Image(Cur));
end Example;
```

Listing 6.2: An advanced example for busy waiting.

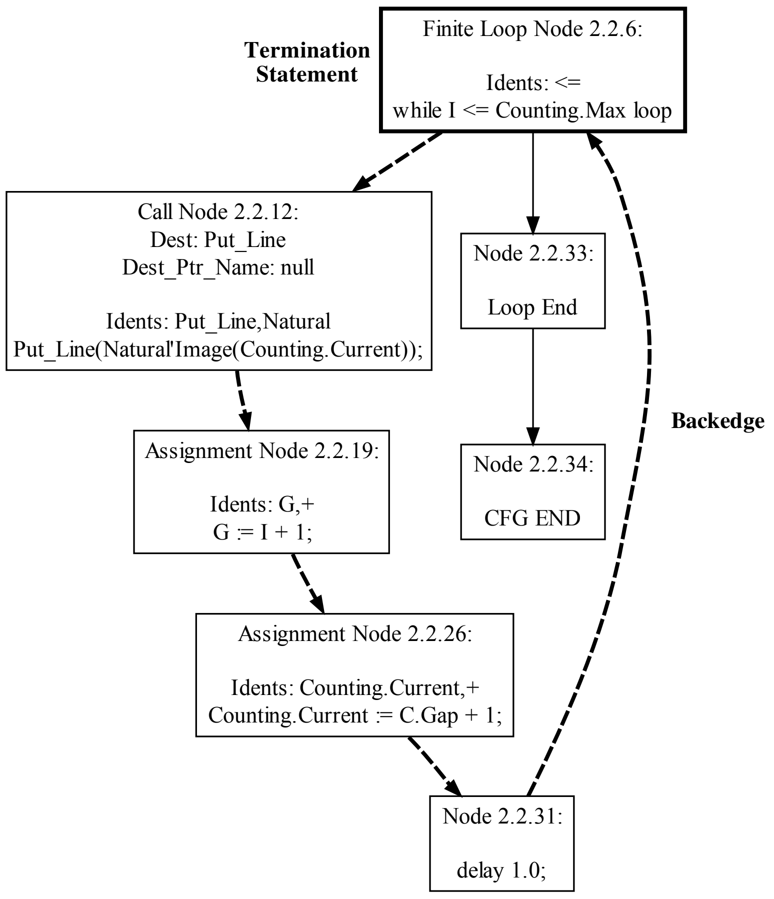


Figure 6.2: The CFG for task `Counter` in Listing 6.2.

```
[Bwaa] Potential busy waiting with variable C.Max in line 37 in loop starting at
line 35 in Example in /path/to/example.adb
[Bwaa] Potential indirect busy waiting with variable C.Current, direct candidate
Cur in line 37 in loop starting at line 35 in Example in /path/to/example.adb
[Bwaa] Potential indirect busy waiting with variable C.Current, direct candidate
Cur in line 38 in loop starting at line 35 in Example in /path/to/example.adb
[Bwaa] Potential busy waiting with variable C.Max in line 37 in loop starting at
line 32 in Example in /path/to/example.adb
[Bwaa] Potential indirect busy waiting with variable C.Current, direct candidate
Cur in line 37 in loop starting at line 32 in Example in /path/to/example.adb
[Bwaa] Potential busy waiting with variable Counting.Max in line 20 in loop
starting at line 20 in Example in /path/to/example.adb
```

Listing 6.3: Output of BWAA for the example in Listing 6.2

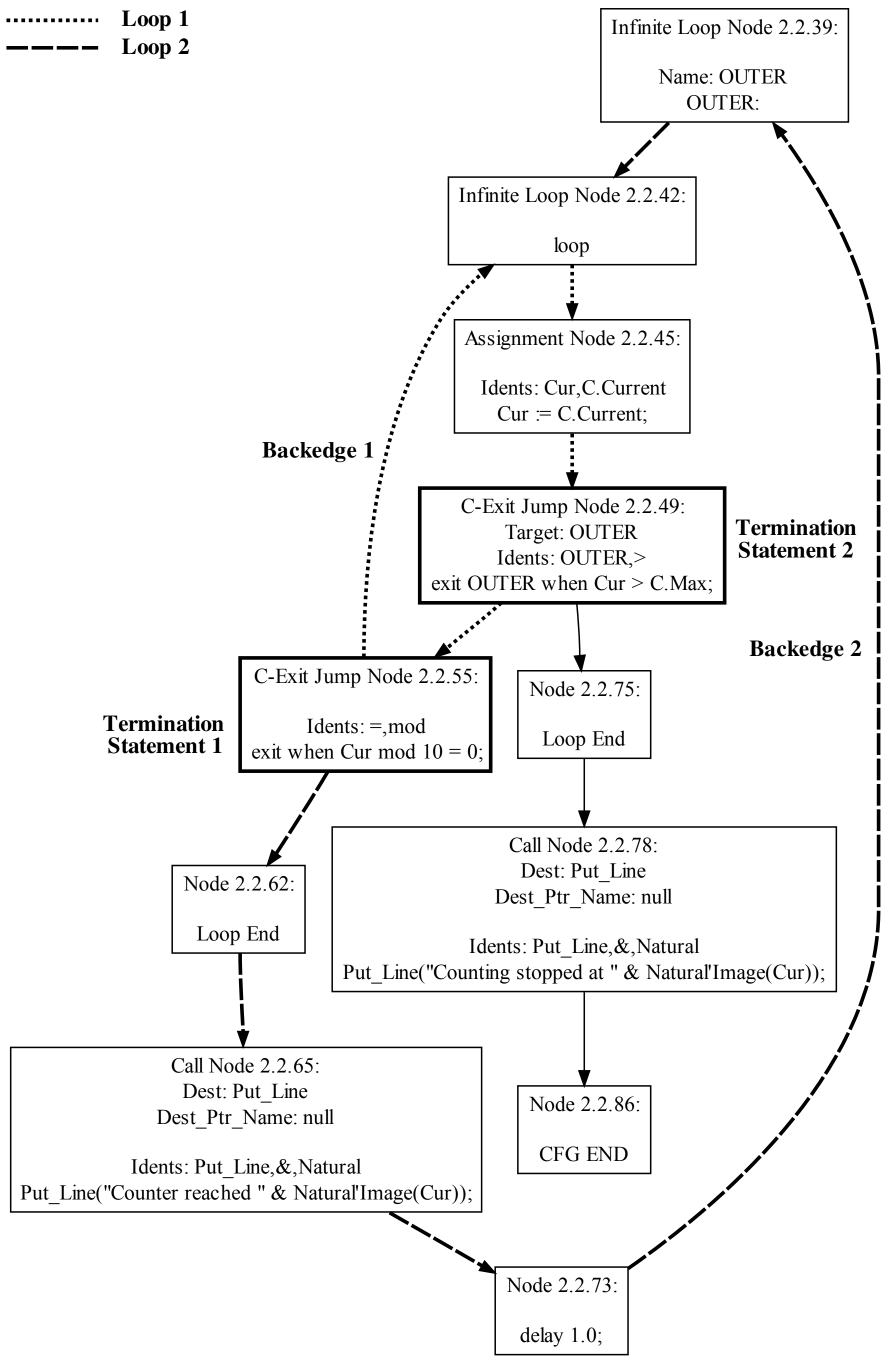


Figure 6.3: The CFG for procedure `Example` in Listing 6.2.

Chapter 7
SUMMARY

Busy waiting in general is a bad programming practice, that is a threat to quality, correctness and stability of a program. To eliminate occurrences of busy waiting in existing program code or to ensure its absence, a static analysis tool is needed.

I presented the busy wait detection algorithm introduced by Blieberger et al. [9] and improved it in terms of efficiency and accuracy. Next, I applied the analysis algorithm to the Ada 2005 programming language, where software quality assurance is of great importance, and developed various methods to handle the complex but powerful nature of this language.

The analysis methods I employed are located in the field of static control flow analysis, which is why I needed a control flow graph representation of the input programs for the analysis. Since for the Ada programming language a tool for generating such a representation was inexistent, I had to develop this software, together with a colleague.

This resulted in the development of a powerful framework for comprehensive general-purpose static control flow analysis, Ast2Cfg, and accompanying tools like Cfg2Dot and Ast2Dot.

Next, I implemented the busy wait detection algorithm as a separate application, named BWAA, using the comprehensive CFG-based data structures of the Ast2Cfg framework.

With the BWAA analysis software, it is possible to detect the usage of busy waiting within an arbitrary Ada input program. Although false alarms may occur, the output of BWAA, including line numbers and code snippets, is an excellent starting point for further manual investigation, enabling efficient discovery of busy waiting and therefore facilitating the quality assurance of Ada programs.

I published the software that was developed during this work, including the source code, under the terms of the GNU General Public License, and it currently is available for download at `http://cfg.w3x.org`.

The results of this work, however, are also a wonderful starting point for future work. On the one hand the busy wait analysis methods could be further improved using for instance the mentioned symbolic methods [10]. On the other hand, having a static control flow analysis framework like Ast2Cfg allows for the development of countless new analysis methods.

References

[1] AdaCore. *ASIS-for-GNAT Reference Manual*, January 2007. Revision 41867.

[2] AdaCore. *ASIS-for-GNAT User's Guide*, January 2007. Revision 41863.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers.* Addison-Wesley, Reading, Massachusetts, 1986.

[4] F. E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, 1970.

[5] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, 1976.

[6] G. R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[7] J. Barnes. *Programming in Ada 2005.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[8] J. Blieberger. Data-flow frameworks for worst-case execution time analysis. *Real-Time Syst.*, 22(3):183–227, 2002.

[9] J. Blieberger, B. Burgstaller, and B. Scholz. Busy Wait Analysis. In *Ada-Europe'2003 International Conference on Reliable Software Technologies, LNCS 2655*, pages 142–152, Toulouse, France, June 2003.

[10] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Trans. Softw. Eng.*, 5(4):402–417, 1979.

[11] T. T. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* MIT Press, Cambridge, MA, USA, 1990.

[12] E. W. Dijkstra. Co-operating sequential processes. *Programming Languages: NATO Advanced Study Institute*, pages 43–112, 1968.

[13] S. Even. *Algorithmic combinatorics.* Macmillan New York, 1973.

[14] T. Fahringer and B. Scholz. A unified symbolic evaluation framework for parallelizing compilers. *IEEE Trans. Parallel Distrib. Syst.*, 11(11):1105–1125, 2000.

[15] R. Fechete and G. Kienesberger. Generating control flow graphs for Ada programs. Technical Report 183/1-139, Department of Automation, TU Vienna, Treitlstr. 1-3, A-1040 Vienna, Austria, September 2007.

[16] R. Fechete, G. Kienesberger, and J. Blieberger. A Framework for CFG-based Static Program Analysis of Ada Programs. In *Ada-Europe'2008 International Conference on Reliable Software Technologies, LNCS 5026*, pages 130–143, Venice, Italy, June 2008.

[17] M. B. Feldman. Who's Using Ada? Real-World Projects Powered by the Ada Programming Language. Website, June 2008. Available online at `http://www.seas.gwu.edu/~mfeldman/ada-project-summary.html`; visited on July 17th 2009.

[18] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, NJ, USA, 1985.

[19] International Organization for Standardization. *ISO/IEC 15291:1999: Information technology — Programming languages — Ada Semantic Interface Specification (ASIS).* International Organization for Standardization, Geneva, Switzerland, 1999.

[20] G. Ramalingam. Identifying loops in almost linear time. *ACM Trans. Program. Lang. Syst.*, 21(2):175–188, 1999.

[21] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Comput. Surv.*, 18(3):277–316, 1986.

[22] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, MA, second edition, 1988.

[23] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. A new framework for elimination-based data flow analysis using DJ graphs. *ACM TOPLAS*, 20(2):388–435, 1998.

[24] W. Stallings. *Operating Systems: Internals and Design Principles.* Prentice Hall, Upper Saddle River, NJ, USA, fourth edition, 2001.

[25] R. M. Stallman. *Free Software, Free Society: Selected Essays of Richard M. Stallman.* GNU Press, Free Software Foundation, 2002.

[26] S. T. Taft, R. A. Duff, R. Brukardt, E. Plödereder, and P. Leroy. *Ada 2005 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/1995 (E) with Technical Corrigendum 1 and Amendment 1*, volume 4348 of *Lecture Notes in Computer Science.* Springer, 2006.

[27] R. E. Tarjan. Testing flow graph reducibility. *J. Comput. Syst. Sci.*, 9(3):355–365, 1974.

[28] R. E. Tarjan. *Data structures and network algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.