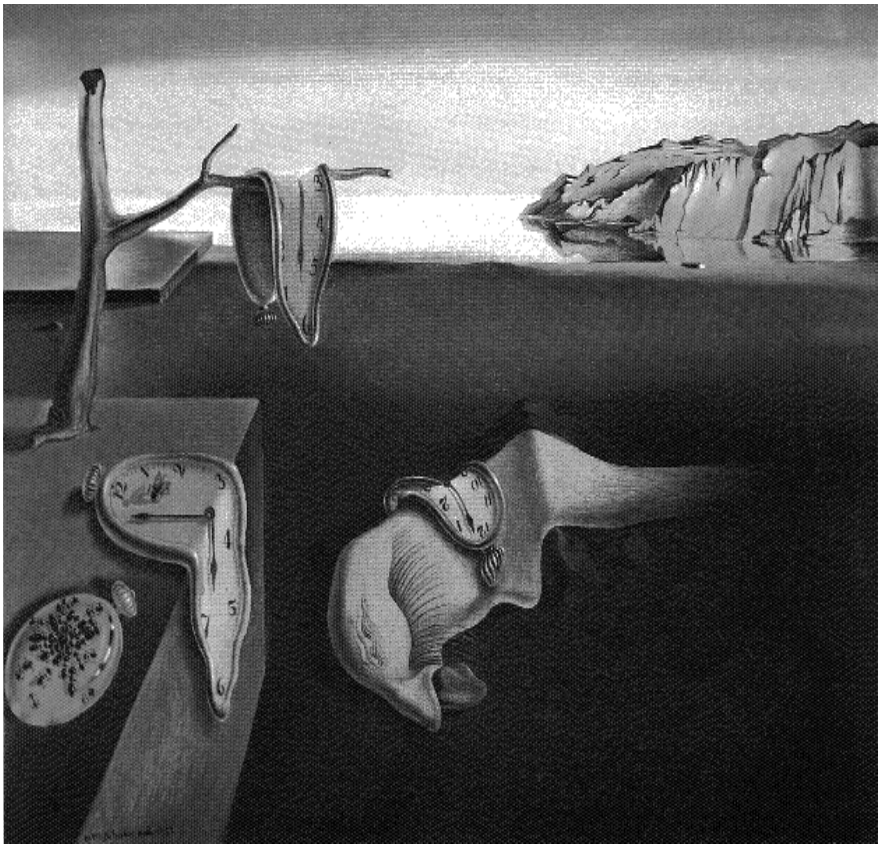
Institut für Automation
Abt. für Automatisierungssysteme

Technische Universität Wien

Projektbericht Nr. 183/1-139
September 2007

Generating Control Flow Graphs for Ada Programs

Raul Fechete *Georg Kienesberger*



Salvador Dali, "Die Beständigkeit der Erinnerung"

Abstract

Many code optimisation and analysis techniques depend on the representation of a program in form of a control flow graph. Nowadays, several tools that generate control flow graphs out of source code exist. Until we started our project, there were no such utilities for Ada.

The purpose was to develop a program that generates control flow graphs for arbitrary Ada programs, and lets other tools access this information.

As an interface to the Ada environment we used ASIS-for-GNAT. It provides us with the abstract syntax tree, that we use as input for our transformation. The control flow graph is generated during a single inorder traversal of the syntax tree. However, after that main transformation phase further refinement is done.

Contents

1 Introduction

Many code optimisation and analysis techniques depend on the representation of a program in form of a control flow graph. The purpose of this project was to develop a program that generates control flow graphs for arbitrary Ada programs, and lets other tools access this information. This program, *Ast2Cfg*, was designed as a library, and is already used by *Cfg2Dot*, a tool we developed in order to visualise the resulting control flow graphs.

In the remainder of this Section we will give an overview on the libraries and techniques (Section 1.1 to 1.3) used throughout this project and explain some of the key concepts of *Ast2Cfg* (Section 1.4). Section 2 contains detailed information on how to use *Ast2Cfg* and the data structures it provides, while Section 3 documents the implementation details of the underlying transformation process.

1.1 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. It 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 elements correspond to nodes of a tree representation of an Ada program and therefore represent Ada language constructs. [4]

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. During this project, which is aimed at supporting Ada 2005, eventually more and more features of Ada 2005 got implemented in GNAT and ASIS-for-GNAT. However, presently, none of them fully supports the Ada 2005 standard, which is why the current versions of our tools do not support the whole language set of Ada 2005 either.

1.2 Syntax Trees

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 subtree rooted at Node N7 in Figure 1, where the node representing the assignment statement has two children: one for each operand. However, Figure 1 also reveals that the syntax trees provided by ASIS are far more complex, because whole Ada programs have to be represented, not just simple expressions. So a predecessor successor relation may also have other meanings than the child node being an operator to its father. For example the root node in Figure 1 is split up into several parts represented by its children. The first child is the name of the procedure, the second the only variable declaration and the third child node corresponds to the only statement in the procedure body.

In general, syntax trees not only play an important role in compilers, but are also used by *CASE tools* and for program analysis that does not require control flow information.

In order to study the syntax trees provided by ASIS we also developed a small application named Ast2Dot (execute `ast2dot -h` for usage information). It transforms the trees into *dot* files, which may be converted into several graphics file formats using the `dot` command which is part of the *graphviz* package in most *GNU/Linux* distributions and is also available for other operating systems. Figure 1 shows an ASIS AST corresponding to the code in Listing 1 as output by Ast2Dot.

Listing 1: The code to the AST in Figure 1

```
procedure Test is
    X: Integer;
begin
    X := 1;
end Test;
```

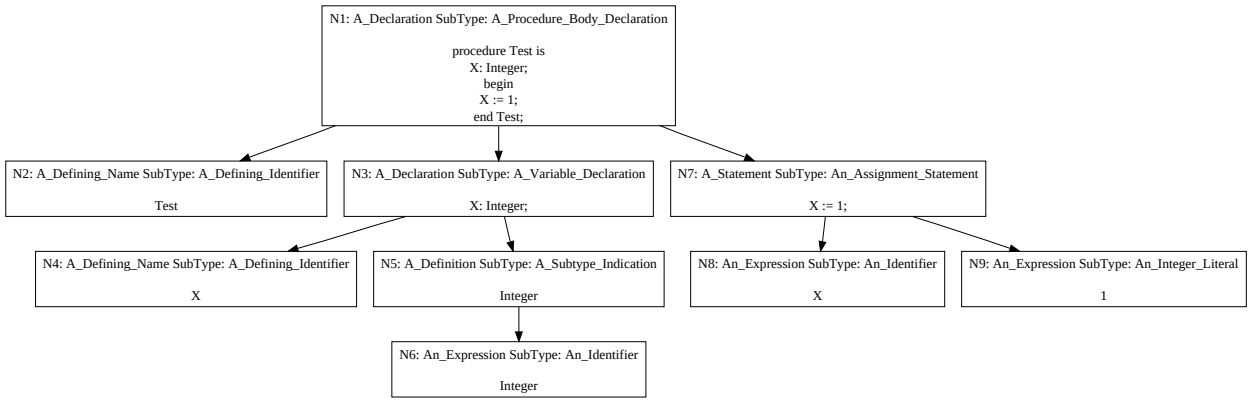


Figure 1: A simple ASIS AST as output by Ast2Dot.

1.3 Control Flow Graphs

A *Control Flow Graph* (CFG), or simply *Flow Graph* is a directed graph, where the nodes represent *basic blocks* which consist of a linear sequence of statements. There is a directed edge from Block B_1 to Block B_2 if B_2 immediately follows B_1 in some execution sequence. Then B_1 is a *predecessor* of B_2, and B_2 is a *successor* of B_1. The *initial* node is the block whose leader is the first statement.[3]

Figure 2 shows a simple CFG that corresponds to the program shown in Listing 2. The initial or *root* node is marked with grey background.

As a CFG represents the statically determinable control flow of a program in an easily accessible way, it is essential to many optimisation and static analysis techniques.

However, the control flow graph used throughout this project is in fact a more complex and richer data structure. First, in order to be able to link a node to its corresponding ASIS element in a convenient way, a linear sequence of statements is not compressed into a single

Listing 2: The code to the CFG in Figure 2

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

procedure Sum is
    I,S: Integer;
    A: array (1 .. 3) of Integer;
begin
    A := (1,2,3);
    S := 0;
    I := 1;
    loop
        if I <= A'Last then
            S := S + A(I);
            I := I + 1;
        else
            exit;
        end if;
    end loop;
    Put_Line(Integer'Image(S));
end Sum;
```

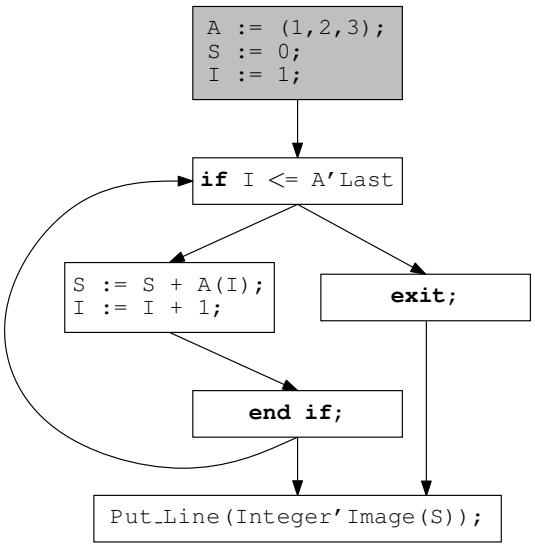


Figure 2: A simple control flow graph.

node (basic block) but appears as a linear sequence of nodes. Furthermore, *parameter trees* (see Section 2.2.2) which hold information on the structure of expressions are integrated into the CFG. Also information on used types has been integrated (see 2.2.7) and subprogram calls result in links to the corresponding CFGs. Finally, there are different types of nodes (see 2.2.5) and CFGs (see 2.2.4) which bear additional information.

1.4 Ast2Cfg

The main goal of this project was to develop a library that enables the user to retrieve the control flow graphs for a given Ada program. Since through ASIS basically the AST of a program can be accessed, and we generate the CFGs from it, we decided to call our library *Ast2Cfg*. This report documents version 0.1 of Ast2Cfg, version 0.6 of the related Cfg2Dot program, and version 0.9.1 of the already mentioned Ast2Dot. All programs are available under terms of the *GNU General Public License* from *http://cfg.w3x.org*.

Figure 3 gives an overview of the basic structure of the transformation process. First GNAT has to be used to generate the so-called *tree files* for the given program. 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 [2]. ASIS-for-GNAT uses these tree files as input, and provides access to the AST structure. Ast2Cfg traverses the ASTs 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 our transformation makes use of a procedure that is called whenever a successor (a *child*) of some node is *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.

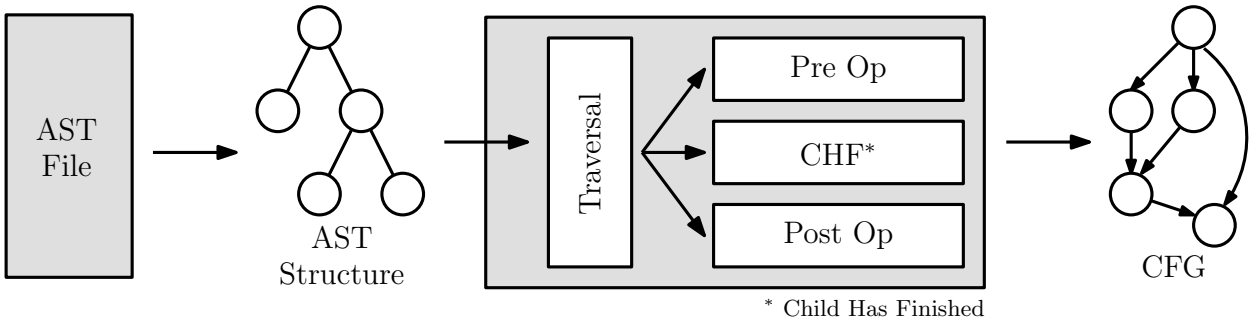


Figure 3: The basic structure of the transformation process.

The Cfg2Dot program we developed along with Ast2Cfg is a good example of how to use the Ast2Cfg library. It uses Ast2Cfg to generate the CFGs for a given program (execute `cfg2dot -h` for a list of command line options) and outputs them in *dot format*, which may then be compiled to several graphics file formats, including *postscript*.

2 Usage

2.1 Ast2Cfg Library

Ast2Cfg is designed as a library and therefore the directory containing its sources has to be included with the `-I` option when using *GNAT*. Furthermore there are three packages which will need to be included by **`with`** clauses almost always: `Ast2Cfg.Output`, `Ast2Cfg.Control` and `Ast2Cfg.Flow_World`.

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. All output Ast2Cfg produces will go 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 what tree files are considered. For example `-C1` considers a single tree file whose name is given as the next parameter [1]. The default is to consider every tree file in the current directory. For a complete list of ASIS context parameters see the ASIS-for-GNAT Reference Manual [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 2.2). 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.

To summarise, Listing 3 shows a minimal application that makes use of the Ast2Cfg library.

Listing 3: A minimal application using Ast2Cfg

```
with Ast2Cfg.Control;
with Ast2Cfg.Flow_World;
with Ast2Cfg.Output;

procedure Run is
    World: Ast2Cfg.Flow_World.World_Object_Ptr;
begin

    -- Initialisations
    Ast2Cfg.Output.Set_Level(Ast2Cfg.Output.Warning);
    Ast2Cfg.Control.Init;

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

    -- Utilise the Flow World

    -- Finalisation
    Ast2Cfg.Control.Final;

end Run;
```

2.2 Flow World

An object of type `World_Object` is the result of the transformation and therefore contains all control flow information derived from the tree files. `World_Object.Pkg_List` is a list (see 2.2.8) of all root packages of the analysed program, where a package object (`Pkg_Object`) corresponds to an Ada package of the transformed program. Control flow information that is not part of a package is contained within a default package.

A `Pkg_Object` is derived from the abstract `Flow_Object`, which is also the superclass of `CFG_Object` and `Node_Object`. A CFG object corresponds to an entity in Ada that contains control flow, which not only may be a subprogram but for instance also the initialisation sequence of a package, a block, etc. Finally, to reflect the control flow a CFG object makes use of node objects. As shown in Figure 4 these classes also have a couple of subclasses in order to allow a more fine grained classification.

2.2.1 Pkg/CFG tree

In Ada not only a subprogram may be declared in a package, but a package may also be declared within a subprogram. In this context we say that a flow object is a *successor* of another one if it is declared within the other one. Thus not only a node object but in fact every flow object has predecessors and successors which may be retrieved with `Get_Preds` and `Get_Succs`, respectively. This relationship between package and cfg objects imposes a tree structure on the flow world, the *Pkg/CFG tree*. In fact, there can be multiple Pkg/CFG trees in a world object, because there is usually more than one package in a flow world that serves as a root. Figure 5 shows the Pkg/CFG tree which was generated from the code in Listing 4 by *Cfg2Dot*.

Listing 4: The code from which Figure 5 was generated.

```
procedure Test is

    package Sub_Pkg is
        procedure Sub_Sub_CFG;
        package Sub_Sub_Pkg is
        end Sub_Sub_Pkg;
    end Sub_Pkg;

    package body Sub_Pkg is
        procedure Sub_Sub_CFG is
        begin
            null;
        end Sub_Sub_CFG;
    end Sub_Pkg;

    procedure Sub_CFG is
    begin
        null;
    end Sub_CFG;
begin
    null;
end Test;
```

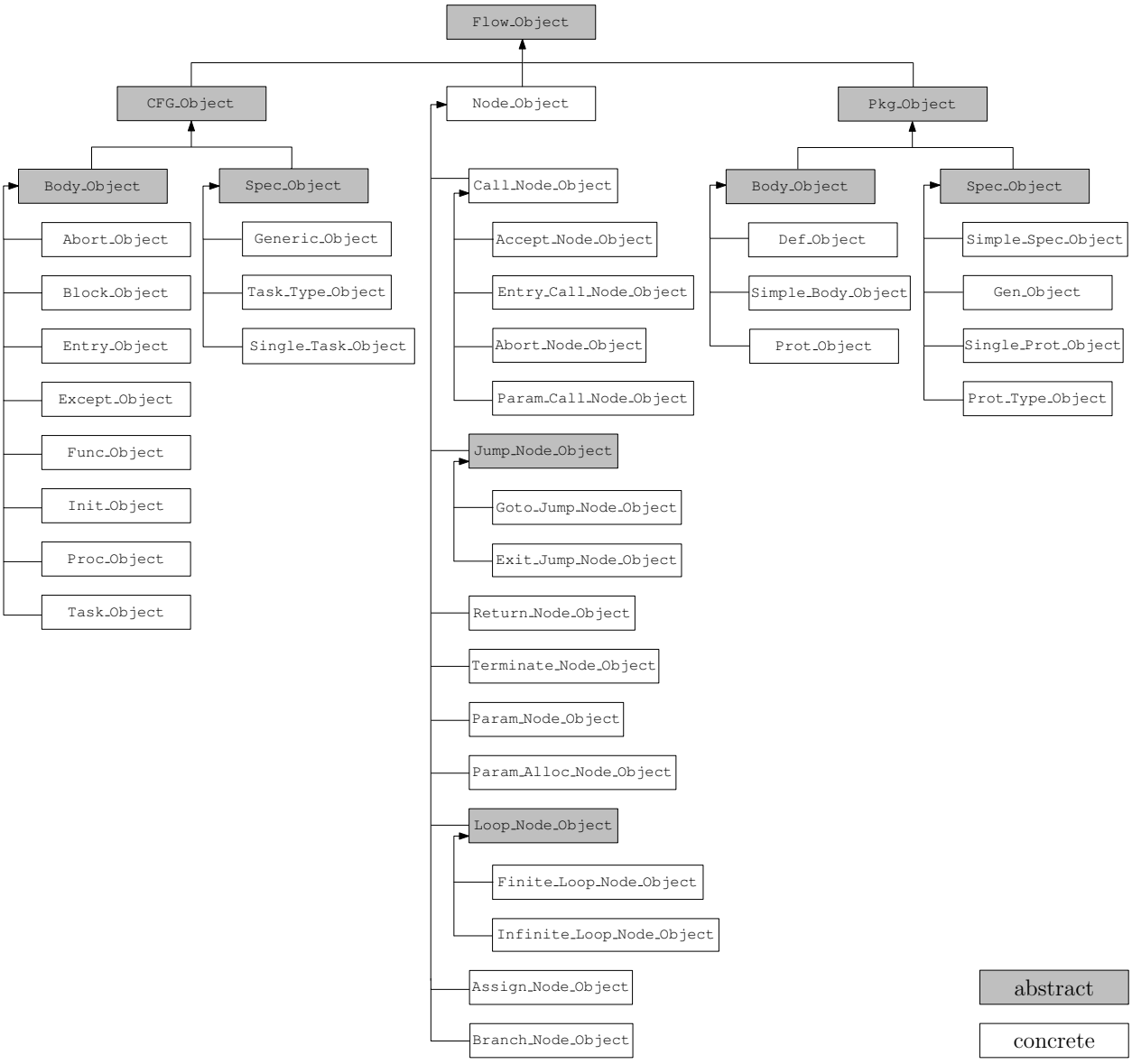


Figure 4: Class hierarchy of the flow types.

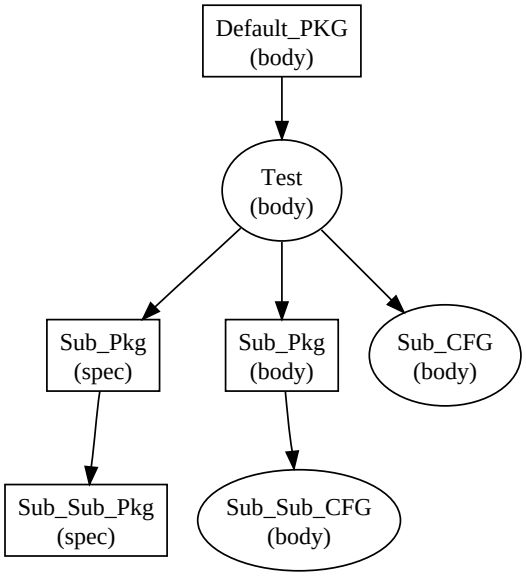


Figure 5: The *Pkg/CFG tree* generated from Listing 4.

2.2.2 Parameter Trees

Whenever an expression is encountered a *parameter tree* is generated which holds information on the used variables, functions, etc. and their nesting.

As a simple example consider the code in Listing 5 and the appropriate output from Cfg2Dot in Figure 6. The actual control flow comprises only a single assignment node (see 2.2.5) which has two parameter trees: one for the left hand side of the assignment, and one for the right hand side. As can be seen, a parameter tree basically consists of parameter call nodes which represent function calls and plain parameter nodes (see 2.2.5). Every level of nesting corresponds to a level in the parameter tree, so a child node contains a parameter to its predecessor. Therefore a subtree represents a subexpression and the leaf nodes contain either an indivisible expression or nothing. (A parameter node is empty if it represents a parameter that is a constant.)

2.2.3 Flow Object

Every flow object has a unique id which can be acquired with `Get_Id` and may be tested for equality. Each id consists of a number for the package, one for the cfg and one for the node, where a number is zero if it is not needed. For example `1.2.0` is the id of the second cfg from the first package. In addition package and cfg objects normally also have a name, saved as a `String`, which can be retrieved by calling `Get_Name`. The name of a node object can be the name of the label for that node, in case one has been defined, or the name of the parameter in case of a parameter node (see Section 2.2.5). Otherwise it is empty.

All flow objects have predecessors and successors, which can be obtained with `Get_Preds` and `Get_Succs`, respectively. While such relationships form a Pkg/CFG tree when used between package and CFG objects (see section 5), they build a control flow graph when used between node objects.

Listing 5: The code from which Figure 6 was generated.

```
procedure Test is

    function Func (I: Integer) return Integer is
    begin
        return 0;
    end;

    function Func2 (I: Integer; J: Integer) return Integer is
    begin
        return 0;
    end;

    X,Y: Integer := 0;
    A: array (1..10) of Integer := (others => 0);
begin

    A(1) := Func2(X, Func(I => Y));

end Test;
```

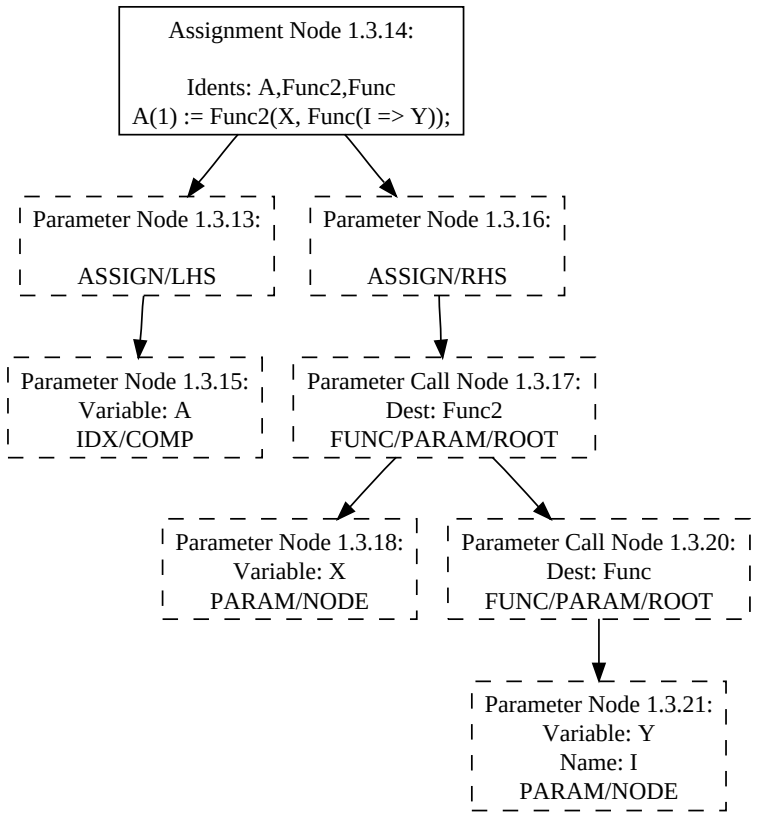


Figure 6: The CFG generated from Listing 5.

Calling Get_PTree on a flow object may return the root node of a parameter tree. There are some situations, where there simply is no other, more adequate place to save a parameter tree. One example is the parameter tree of an expression following **when** in an entry body of a protected object. It is saved as this so-called *extra parameter tree* in the corresponding Entry_Object (see Section 2.2.4).

Furthermore, a flow object has a list of variable declarations and a list of generic formal parameters. They are acquired with calls of Get_Decls and Get_FDecls, respectively. A list of renamings is returned by Get_Rens, whereas a list of types declared within this flow object is returned by Get_Types. Finally, a list of references, that is a list of **with**, **use** and **use type** clauses, is returned by Get_Refs. For a detailed description of the internal representation see Section 2.2.7.

2.2.4 CFG Types

As already mentioned, a CFG object represents the control flow information by using node objects. The root node of a CFG object can be retrieved with the Get_Head method, and the total number of nodes, excluding those in parameter trees, is returned by calling Get_Node_Sum. Since most CFGs represent some form of subprogram, the parameters are saved as a list of Declaration objects (see Section 2.2.7), and may be acquired by calling Get_Params.

As can be seen in Figure 4 CFG_Object itself is abstract, so all cfg objects have to be of a more specific, concrete subtype. There are subtypes which represent data gained from Ada bodies and those containing information on Ada specifications, which is reflected through derivation from either Body_Object or Spec_Object.

Proc_Object and Func_Object simply represent the CFG of a procedure or function, respectively. The member function Get_Decls returns a list of the variables which are declared in this procedure or function, and Get_Params returns a list of the parameters. A variable or parameter in one of those lists is represented as a Declaration (see Section 2.2.7).

A block within some other CFG object is transformed into a separate Block_Object and is handled like a parameterless function called at the position where the block is declared. So a call node (see 2.2.5) that points to the block object is inserted in place of the block. Again Get_Decls can be used to get the list of variables declared at the beginning of this block. Currently, only a string representation of the CFG object this block belongs to is available through Get_LName. However, in future versions of Ast2Cfg there will be an access type pointing directly to the CFG object.

The sequence of statements between **begin** and **end** of a package body is transformed into an Init_Object. For every exception handler between **exception** and **end** a separate Except_Object is generated.

A Task_Object corresponds to a task body, where an **accept** statement is transformed separately into a so-called Entry_Object. As with simple blocks, it is then linked to the enclosing task body like a parameterless procedure call, and the declared variables are available as usual. Furthermore the *protected entries* of a protected object (see also 2.2.6) are also mapped to entry objects. An Abort_Object represents the control flow of the abortable part of a **select - then abort** statement. It is, again, linked into the control flow of the select statement like a parameterless procedure call.

Finally, there are three CFG objects derived from `Spec_Object`, which means they cannot contain any actual control flow. The main purpose of including those objects in the Pkg/CFG tree is to keep track where they are defined, so that information on visibility may be gained later on. A task type declaration results in a `Task_Type_Object` while a simple task specification (without the **`type`** keyword) is mapped to a so-called `Single_Task_Object`. The discriminants of a task type declaration are handled like parameters of a subprogram, and therefore are returned by `Get_Params`.

A `Generic_Object` is constructed whenever a generic procedure or function specification is encountered. `Get_FDecls` returns a list of the generic formal parameters, which are represented as `Formal_Decl` objects (see Section 2.2.7). The list of the actual parameters of the subprogram is acquired as usual. The body of a generic subprogram is handled like a normal subprogram body.

2.2.5 Node Types

Unlike CFG or package objects, `Node_Object` is not declared abstract, so any node of the CFG that has no special properties as described below simply is of type `Node_Object`. Every node has a link to the CFG it belongs to, and a string which holds at least part of the code fragment that is represented by the node. Every node also has a parameter tree, which is returned by `Get_RHS_Tree`. This tree is called *right hand side tree* because every node other than an assignment node is considered to only have a right hand side. An `Assign_Node_Object` therefore is the only node object that also has a *left hand side parameter tree* for the part to the left of the assignment operator.

Furthermore every node has a list of the found identifiers for that node. Technically this is a list of lists of unbounded strings since every identifier name is saved, divided into the parts that are normally separated by a dot. Note, that this list may not be complete in the current version of Ast2Cfg and that the parameter tree has to be parsed in order to be sure to get all identifiers. Finally every node contains the `Asis.Element` that was the basis for this node. The element can be acquired by calling `Get_Element` and in fact is like a link back into the ASIS AST. So it is possible to gain additional information from ASIS by analysing the AST starting at some `Asis.Element` which was saved in an arbitrary node. In case there is no element that corresponds to the node `Get_Element` returns `Asis.Nil_Element`.

A `Call_Node_Object` is not only used to represent a subprogram call, but also in various situations that are treated like a subprogram call, as stated in Section 2.2.4. Therefore every call node saves a string representation of the destination CFG. Currently in some cases the CFG also holds a string representation of the call node. In future versions of Ast2Cfg every call node will also contain an access type pointing directly to the CFG. As can be seen in Figure 4 there are a few subtypes of `Call_Node_Object` that allow a more fine-grained classification. While an `Accept_Node_Object` is used to link an `Entry_Object` to its enclosing task body (see 2.2.4), an `Entry_Call_Node_Object` represents the call of such an entry. An `Abort_Node_Object` similarly is used to link the abortable part of a **`select - then abort`** statement into the control flow of the select statement. Finally a `Param_Call_Node_Object` simply is a call node within a parameter tree representing some function call.

There are two types derived from the abstract `Jump_Node_Object`, the `Goto_Jump_Node_Object` which is generated whenever a **`goto`** statement is encountered and

the Exit_Jump_Node_Object which represents an **exit** statement within a loop. In the current version of Ast2Cfg the target of the jump is already a successor of the jump node, however, Get_Target also returns the label as a string. A Return_Node_Object is generated whenever a return statement is encountered, while a Terminate_Node_Object represents a terminate statement.

A Loop_Node_Object indicates the header of a loop, and there are two concrete subtypes, the Finite_Loop_Node_Object which represents a **while** or **for** loop statement and the Infinite_Loop_Node_Object which corresponds to a simple **loop** statement. Note that a Infinite_Loop_Node_Object may in fact be a finite loop because for example there is an **exit** statement within the loop, and a Finite_Loop_Node_Object may represent an infinite loop because the loop condition always holds. The naming just reflects that an infinite loop node itself has no loop condition.

A parameter tree (see Section 2.2.2) has nodes of type Param_Node_Object, Param_Call_Node_Object or Param_Alloc_Node_Object. A Param_Node_Object not only saves the name of the variable that was supplied as a parameter, but also holds the name of the parameter in case it was specified. A parameter node with an empty variable denotes that a constant was given as a parameter. A Param_Alloc_Node_Object is used whenever a variable is allocated dynamically using the **new** keyword. The name of the instantiated type is returned by Get_Type_Name.

When an **if** or **case** statement is encountered, first an otherwise empty header node, where the branching actually happens, is created. So the successors of such a Branch_Node_Object contain the condition for each branch (or nothing, in case of an **else** node). Then the actual branches follow, until control flow is united again in an end node. Note that in case of an **if** statement without an else path, there is a direct link from the header node (the Branch_Node_Object) to the end node.

2.2.6 Package Types

Like the CFG_Object, the Pkg_Object is abstract too and all subtypes are derived either from Body_Object or Spec_Object like shown in Figure 4. The primary function of the package types is to help building the Pkg/CFG tree (see 2.2.1). So in order to see what is in a package Get_Succs will have to be used until the whole subtree is traversed. Every package object, except a Prot_Object and the Def_Object, also contains a list of declared variables which may be acquired by calling Get_Decls. However, some package objects contain additional information.

The default package is of type Def_Object. It is an artificial package that a CFG is added to in case there is no real enclosing package. The body of a protected object or type is transformed into a Prot_Object, which may contain subprograms or entries (see Section 2.2.4). Its specification is represented by a Prot_Type_Object in case of a protected type declaration or a Single_Prot_Object otherwise. The body of a generic package is mapped to a Simple_Body_Object like a normal body. The specification, however, is transformed into a Gen_Object which also contains the generic formal parameters. They can be acquired using the Get_FDecls function.

All other packages are transformed into a Simple_Spec_Object and, if there is a body, a Simple_Body_Object.

2.2.7 Auxiliary Types

Not every type used within Ast2Cfg is derived from `Flow_Object`, in fact there are many types that play an auxiliary role.

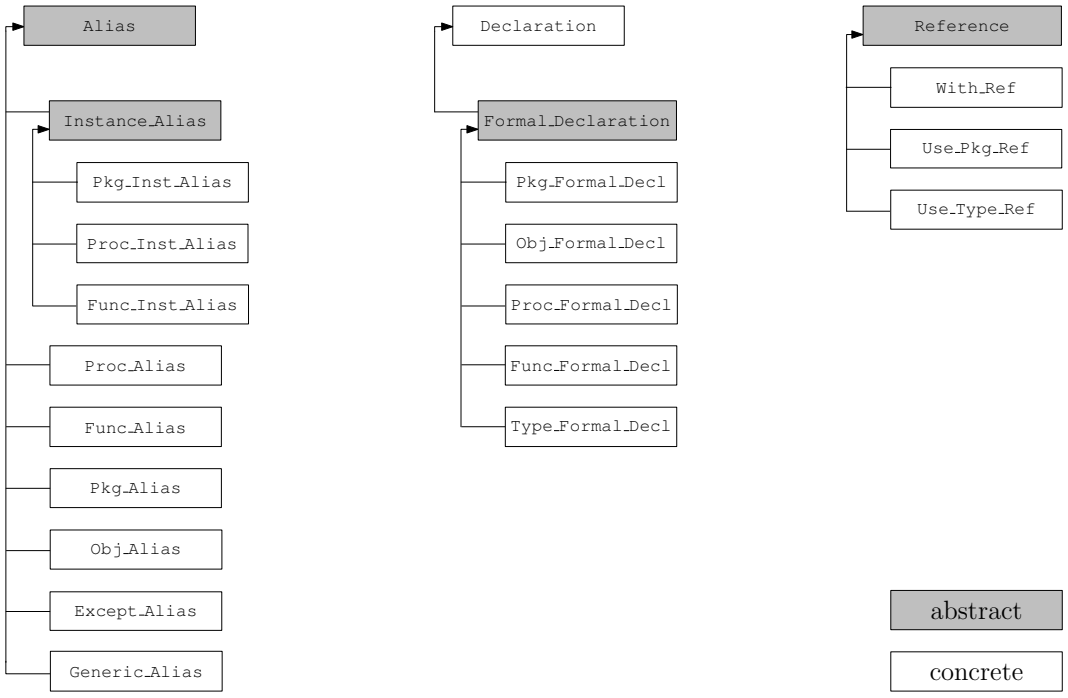


Figure 7: Class hierarchy of some auxiliary types.

A typical usage of the type `Alias` is to represent some kind of renaming. Its directly accessible member variables include the old name, the new name and an optional parameter tree. The parameter tree has to be saved, because an array component may be renamed too and therefore arbitrary expressions may be used to specify the index (e.g. `X` **`renames`** `A(Func(Y))` where `A` is an array and `Func` a function). Additionally, as shown in Figure 7, there are several subtypes of the abstract `Alias` which allow for a detailed classification.

A `Proc_Alias` or `Func_Alias` represents a renaming of a procedure or function, respectively. A `Pkg_Alias` is used for renamings of packages and a `Except_Alias` is generated whenever an exception is renamed. When a simple object of some type is renamed, an `Obj_Alias` is created.

Alias objects are also used to keep track from which generic unit a package, procedure or function was instantiated. This is done using `Pkg_Inst_Alias` objects for instantiated packages, `Proc_Inst_Alias` objects for procedures and `Func_Inst_Alias` objects for functions. These objects are derived from the abstract `Instance_Alias` (see Figure 7) and therefore also have a list of aliases as a member variable. This is because when an instance is created from a generic unit, the generic formal parameters can be given using the named notation (with =>). Such a relationship between a generic formal parameter name and the given parameter is saved using a `Generic_Alias`, however, in case the named notation is not used, the name of the generic formal parameter is empty. So every instance alias has a list of `Generic_Alias`

objects.

A `Type_Descr` object is used to hold information on a type declaration and is a simple record with several, directly accessible components. First there is the name of the type, followed by the supertype in case it is known like in the declaration `type T is new Integer`. In case of an array type, the type of the components is also recorded. Furthermore a list of identifiers found in the type declaration is available, and the discriminants and component types of a record are also saved as a list of `Declaration` objects (see below). Finally, a `boolean` value is used to determine if a type is declared as **`private`**. This is necessary, since for a private type a second declaration exists, containing more details.

A `Declaration` object represents a variable declaration. Its directly accessible member variables include a list of strings containing the variable names called `Vars`, a `Type_Descr` object named `VType` that holds information on the type of all the declared variables and the root node of the corresponding parameter tree named `Init`.

A `Formal_Decl` object is derived from `Declaration` and represents a declaration of a generic formal parameter. Since `Formal_Decl` itself is abstract one of the five concrete subtypes has to be used. Every object derived from `Formal_Decl` additionally has a member variable which holds a list of aliases. This is because for example in case of a package parameter, which is mapped to a `Pkg_Formal_Decl` object it is possible to specify parameters (e.g. `with package P is new Pkg(X => Y);`) which are saved as generic aliases in the same way as it is done for the `Instance_Alias` objects described above. A `Proc_Formal_Decl` object is used for procedure parameters, `Func_Formal_Decl` for function parameters and `Type_Formal_Decl` for type parameters. Finally, `Obj_Formal_Decl` objects are used for ordinary variables that are used as parameters for a generic unit.

A concrete subtype of `Reference` is instantiated whenever a **`with`** (`With_Ref`), **`use`** (`Use_Pkg_Ref`) or **`use type`** (`Use_Type_Ref`) clause is encountered. The name of the target of such a clause is saved in the component `Name`.

2.2.8 Lists

We implemented two types of generic lists, the `Generic_List` and the `Generic_Pointer_List`. All lists used throughout Ast2Cfg are instantiations of these two generic packages and their names end either with "_List" or "_Ptr_List". The difference between these two types of lists is that the `Generic_Pointer_List` not only takes the type of item as a generic formal parameter, but also has a subprogram parameter where a procedure that frees an item has to be provided. This is done to ensure correct deallocation of list elements.

Listing 6 shows how a list is accessed using the `Has_Next` and `Get_Next` methods. Note that the iterator has to be reset by calling `Reset`. Another approach is to access the list like a stack using the `Empty` and `Pop` methods which is shown in Listing 7. However, since `Pop` does remove an element from a list it is recommended to use this access method only on duplicated lists and not directly on a list in the *flow world*. Lists can be duplicated with the `Duplicate` method.

The `Clear` method of a list instantiated from `Generic_List` deallocates all elements, so the list is empty afterwards. If the list was allocated dynamically `Free` has to be called on the pointer to the list in order to deallocate the list header too. The semantics of `Clear` and `Free` differ a bit in the context of a list instantiated from `Generic_Pointer_List`. Here,

Listing 6: Iterating over a list

```
Succs := Get_Succs(Flow);
Reset(Succs);
while Has_Next(Succs) loop
    Get_Next(Succs, Succ);
    -- do something with Succ
end loop;
```

Listing 7: Accessing a list like a stack

```
Succs := Get_Succs(Flow);
while not Empty(Succs) loop
    Pop(Succs, Succ);
    -- do something with Succ
end loop;
```

a call of `Clear` still results in an empty list, however, without executing the supplied free procedure on every element. In contrast, `Free` executes the free procedure and, if called with a pointer, also deallocates the list header. Again it is not recommended to use this methods directly on lists in the flow world since deallocation is done by `Ast2Cfg.Control.Final` (see Section 2.1).

Other helpful methods may be `Get_Head` and `Get_Tail` which return the first and last element, respectively.

3 Implementation

This section describes the technical details of the Ast2Cfg implementation. In the beginning we offer a brief overview (3.1) of the transformation and present the general usage of the deployed data structures (3.2). We then proceed with an in-depth description of the transformation (3.3, 3.4) of each ASIS type handled by our program.

The presented structures are the pillars of the CFG data extraction and their description (3.2) should be studied before proceeding through the more advanced implementation details.

3.1 Overview

The abstract syntax tree to control flow graph transformation consists of two phases:

1. **AST Traversal**: As stated in the previous sections, the AST-to-CFG transformation is built on top of a depth-first-search traversal of the syntax tree. It suffices to walk the tree once to obtain the raw CFG structure. See 3.3 for further details.

2. **Post Transformation Processing**: To obtain the final CFG, further work is needed on the resulting structure. This, among others, includes compressing parameter trees, linking call and jump nodes to their targets and pruning unreachable CFG sections. See 3.4 for further details.

Both phases work on the flow world structure created upon initialisation of the transformation mechanism. The AST traversal adds new data to the flow world on-the-fly, during the AST-walk, whereas the post transformation processing modifies only the existing flow structures.

The processing done during the AST traversal is handled by three main methods that, together, build up a state machine. These methods are:

1. **PreOp**: `HandlePreOp(Node)` This event is triggered each time the in-order traversal reaches a new node in the AST, before any processing is done on the underlying subtree.

2. **PostOp**: `HandlePostOp(Node)` This event is triggered just after the traversal has left a node, after all processing of the underlying subtree has finished.

3. **CHF**: `HandleChildHasFinished(Parent,Child)` This event is triggered just after the traversal has left a node, and is basically similar to the PostOp event. The difference is, however, that this method is used to underline a father-child relationship, thus enabling the transformation logic to also handle context-induced information.

   The logical connection between PostOp and CHF is that after the former finishes it calls the latter to signal the termination of a child node to its parent.

Since all methods above are called for each node in the AST, every one of them must be able to handle all ASIS types that may occur. This leads to a symmetrical structure of the three subprograms, each consisting of almost exactly the same **`case`**-block.

The Ada syntactical constructs can be divided into groups, with each of these classes having a common abstract syntax subtree configuration for all its members. For example, let us consider the variable declarations, the component declarations (for aggregate types) and the parameter specifications (for subprograms). Each of these declarative constructs is described through the names of the declared entities, their type and possibly their initialisation expression. It is thereby easy to see why all subtrees induced by these declarations are identical.

To profit from such class-similarities, many of the **`case`**-branches have been pulled together, while keeping the transformation itself as context-free as possible. This allows us to use the same code to process similar subtrees, regardless of their point of origin.

In most cases, the flow data can be added to the flow world immediately upon reaching an AST node. As an example, let us consider the ASIS procedure call statement. This type stands for procedure calls in the original source code and the subtree rooted in this node describes the statement: labels, name of the called subprogram and parameters. The standard PreOp handling is to immediately add a new call node to the current CFG. Later on, upon processing the subtree, we will encounter a CHF event for a procedure call parent and an identifier child, in which case we will know we have reached the name of the called subprogram. This information will then also be saved in the call node we have created earlier.

In other cases, however, due mainly to the context-free nature of the transformation, an immediate update of the flow world is not possible. We return now to the previous entity declaration example. In order to save information about the subtree directly into the flow world, we would have to search each time for the point of origin of this subtree: e.g. the parameter specifications are not being saved together with the component declarations. By doing so, we would render our transformation context-sensitive again and would remove all advantages derived from the similarity of the subtrees. To avoid this, we use an interim storage (common to all these declarations) for the data gathered during the subtree traversal. Only when the processing has finished, when PostOp-ing the root node, we interrogate the structures for the declaration kind and move all the data, in one step, from the interim storage to its final location in the flow world.

There are several such cases in Ast2Cfg, where a temporary storage is required, and the complete description of the used structures can be found in Section 3.2.

3.2 Structures and Concepts

To keep track of the current transformation status, we employ a series of data structures declared in the Ast2Cfg transformation package, that are being updated along the way.

They can be roughly divided into two groups: positional structures (mainly stacks 3.2.1) indicating our current position in the AST or in the flow world and the interim storage structures (the transformation data container 3.2.2) that temporarily hold information extracted from the AST until it is ready to be transferred to the flow world.

Last but not least is the pointer to the flow world. The corresponding structure is created during the initialisation of the transformation and will be returned as result after the entire process finishes.

3.2.1 Stacks

Due to the nested nature of almost all processed structures we employ stacks to keep track of the nesting and order relationships between visited entities.

This section describes the four main stack categories and is best read before the study of the transformation implementation, since it also presents many of the concepts employed in the procedure.

Element Stack Our current position in the AST - induced by the DFS traversal - can be described as a path from the root of the tree to the current node. For several reasons, we must keep track of this path at all times. These include among others:

- Knowing the **children count** for a certain element[1]. Often, during a CHF-event we must consult the number of already visited children for a parent element before handling one of its sons.

- Respecting the **nesting relationship** of flow structures with a malformed PreOp-PostOp sequence. In the vast majority of the cases each flow package and CFG is opened/closed by a PreOp respectively PostOp event on the root node of the corresponding package-/CFG-subtree.

  In some cases, that we regard as ASIS inconsistencies, however, this sequence is deformed. Let us consider the initialisation block of a package. Statements are allowed, per default, only in CFG-structures (procedures, functions, etc.). The statements in the initialisation block are, in this case, direct children of the root (package) node.

  When walking the subtree, we receive a PreOp event for the new package and therefore create one in the flow world. Immediately afterwards, we receive PreOp events for statements, which theoretically should not happen. Without having previously received a similar event for a CFG, we have no proper structure to hold statements at this time and for this, we must force the creation of one.

  Upon PostOp-ing the package node we do, this time, not only have to close the open package but also force the closing of the subsequent initialisation CFG. The information on the element stack tells us in such cases - if everything went well - that the currently open initialisation CFG has been forcibly induced by the package element.

  This mechanism is also extremely useful as a transformation consistency check. The constraint that the currently open flow structure is only allowed to be induced by the current element must hold at all times. For this purpose we save, for each open flow structure, the inducing element (see 3.2.1). When closing the current flow object, its inducing element must be equal to the one on top of the element stack. If these structures evolve out of sync, it is a sure indicator that an error has occurred on the way.

Due to the previous considerations, we save the path from the root of the AST to the current node on the element stack as follows: for each PreOp we push the regarded element

[1] In respect to the ASIS naming convention, we will sometimes refer to the AST nodes as ASIS elements.

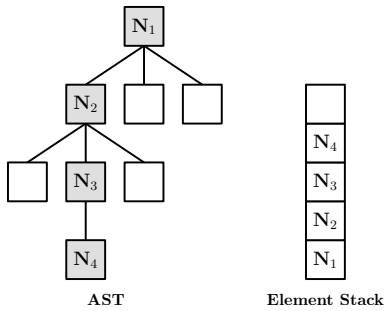


Figure 8: Element stack example

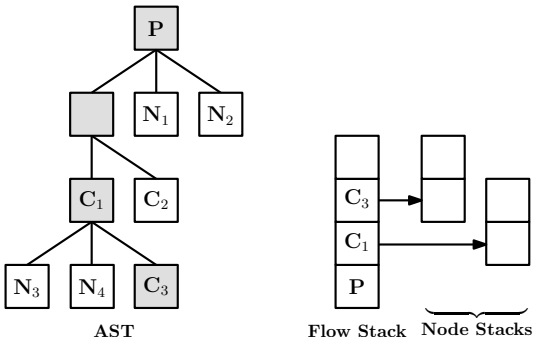


Figure 9: Flow stack example

onto the stack and for each PostOp we remove it. Together with each element we also save a child counter that we increment by a value of one for each of the element's visited children.

Figure 8 depicts a possible traversal scenario with a current path N_1 to N_4 in the given AST and the corresponding element stack configuration.

Flow Stack The location of a language construct in the source code of the analysed program is given in the AST implicitly by its relative position to the father structure, i.e. it must be located in the subtree describing it. Since Ada allows complex package- and CFG-nesting hierarchies, we must - at any point in time - be able to determine the parent flow structure for the currently active AST element.

To allow an error free processing of deep nesting relationships we employ a flow stack data structure. On top of this stack always lies the nesting-innermost flow structure (package or CFG) currently in processing. When PreOp-ing such a structure, we place a corresponding object on the flow stack and when PostOp-ing it, we remove it from the stack again.

Figure 9 depicts the AST for a source code implementing a package P. Inside P lie two CFGs (two subprograms) C_1 and C_2. A third CFG C_3 is nested inside C_1. We emphasise two of the advantages brought by the flow stack usage:

- The flow stack allows us to move up and down in the nesting hierarchy in order to construct the flow tree.
- Regardless of what node we are currently processing, its parent is always on top of the flow stack. This allows us in constant time, for example, to determine that the nodes

N_1 and N_2 are located in the package P and that the nodes N_3 and N_4 are situated in the CFG C_1.

The marked nodes in the AST show the current active path in the transformation process and alongside the tree we see the current status of the flow stack.

As stated in the previous section, for each open flow structure, we also save the inducing AST element. We will use this information together with the element stack to ensure the correctness of the nesting relationships.

Each open CFG also has a corresponding node stack (see 3.2.1) to allow the correct linking of the CFG nodes.

Node Stacks The abstract syntax tree nodes representing ASIS statements are transformed almost always into control flow graph nodes. Due to the sequential character of the statement execution, reaching a new statement in the AST triggers the addition of a new node to the currently open CFG. The new node is being appended at the end of the current flow structure.

In light of these considerations it would be enough to keep track of only the last node in the current CFG, in order to link it to the new one. In some cases, however, this is not enough:

- **Loop statements**: When appending nodes to a loop's body we must not lose trace of its head node. We need to link the last node in the body back to the beginning. Furthermore, in order to ensure the CFG correctness, the loop head must also be the last node in the CFG so far, even though we may have added other nodes since.

- **Decision statements**: When reconstructing the control flow for a decision statement, we must keep track of the decision statement's head node at all times. This head node must have a link to each of the statement's branches.[2]

In order to master situations where information about nodes handled in the past is needed, we employ node stack structures. Each open CFG (currently on the flow stack) has its own node stack.

Unlike the stacks presented so far, the new type maintains per default a height of only 1. It holds only the last node in the CFG it represents and this node is being replaced every time a new one is appended.

This stack only grows when information about previous nodes is explicitly needed and that is the case only for the mentioned loop and decision nodes.

Figure 10 depicts a schematic overview of a transformation example for each of the standard, loop and decision statements. Below each CFG is also its corresponding node stack.

[2]Since we are constructing a control-flow-based out of a syntax-based structure, the transformation itself may seem somewhat inconsistent at first. In the standard scenario, statement nodes from the AST that have the same father and depth are being linked sequentially in the CFG. On the other hand, however, statement nodes that have the same branch-father (**`if, case`**) and depth are being linked in parallel at their destination. As stressed earlier, this is only due to the syntactical nature of the AST.

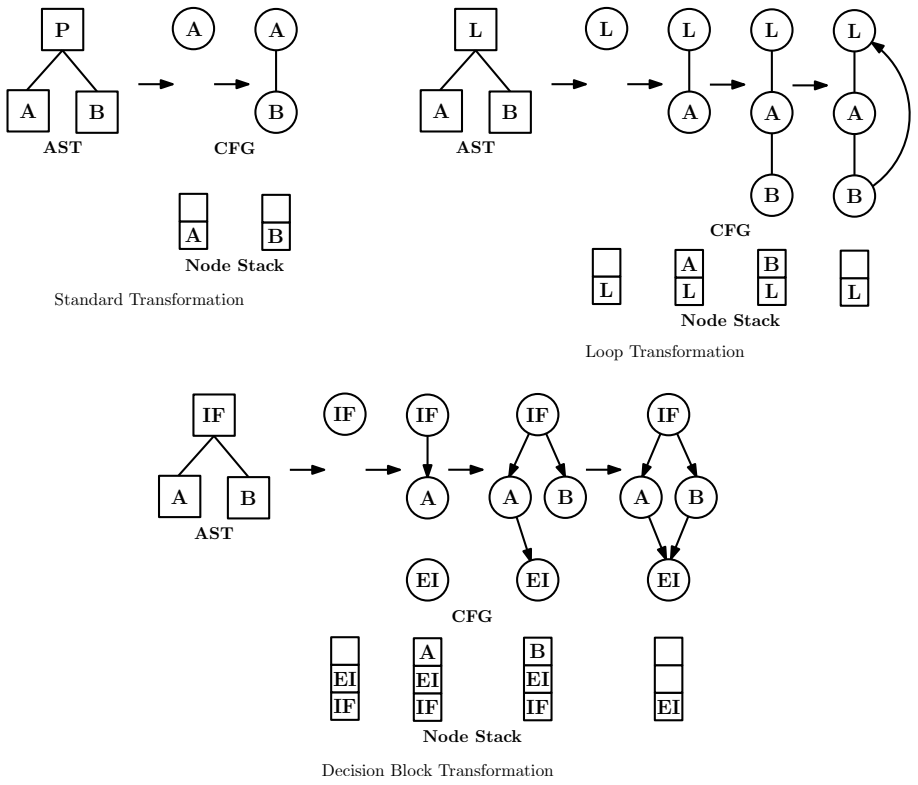


Figure 10: Node stack examples

Please note that figure 10 offers a slightly simplified view of the node stack, in order to better explain the principle of use.

- *Standard* transformation: The first of the three ASTs depicts the structure of a program consisting of a procedure P that in turn contains two statements A and B.

  The standard transformation procedure sequentialises the two statements, adding them one after the other to the CFG structure, while keeping track of only one of them at a time, on the node stack.

- *Loop* transformation: The second of the three ASTs depicts the structure of a program consisting of a loop L holding two statements A and B in its body.

  The loop handling is similar to the standard procedure, with one major difference: the head node does not leave the stack for the entire duration of the loop processing.

  When PreOp-ing the root of the loop subtree we place a corresponding node on the node stack. The next event will be the PreOp for the first of the loop's body statements. This node does not replace the head - as it would have done in the standard procedure - but increases the stack's height by 1. All the following statements are then submitted to the standard handling and do replace the top of the node stack.

  The procedure described above ensures that when PostOp-ing the last node in the loop's body, we also have access to the loop's head, enabling us to link it back. Also, the loop head is the one to point to the first node outside the loop.

To achieve these goals, once the loop finishes - when PostOp-ing its root node - we link the top of the node stack, i.e. the last node in the loop, back to the loop head. We then reset the stack and leave only the loop head on it for further usage.

- *Decision block* transformation: The third AST depicts the structure of a program consisting of a branch statement. The root IF node is followed by two statements A and B, each standing for one of two separate choices.

  In this case, we have to save two extra nodes on the stack. The first one is the IF node itself. We have to keep it, because each time we begin a new branch, we must save a pointer to it in the IF node. The second one is a pseudo-node that we will use to bring all branches back together after they finish. This will be the EI (end if) node.

  By using the EI node, we reduce the complexity of this transformation considerably. Since any of the branches could, at runtime, be part of the active control flow, we would have to link every of these branches, i.e. the last node from each, to the first node after the decision statement. By using this pseudo-node, we can do the linking on-the-fly, enabling us to maintain only a minimum of information about the process.

  When PreOp-ing the root of the decision subtree, we place the two IF and EI nodes, unlinked, on the stack.

  For each branch that begins, i.e. for each PreOp of a direct IF child, we link the decision head to the newly begun branch. For each branch that finishes, i.e. for each CHF event on the decision head we link the last node in the branch to the EI node.

  When the entire decision statement finishes, i.e. when PostOp-ing the root of the subtree, we reset the node stack and leave only the EI node on it for further usage.

By using the node stack this way, we manage to handle almost all statements in a context-free manner, thereby reducing the overall complexity of the transformation process.

Parameter Stack In ASIS, almost all control-flow-relevant information can be derived from statements. The only exception, however, are the function calls. As opposed to a procedure call, that is described by ASIS as a statement, a function call is deemed an expression and it can occur anywhere in the AST, even in declarative sections, as a variable initialisation or type constraint.

This makes it difficult to find a way to integrate them into the CFGs, yet, since for each function call, the control flow leaves the parent entity and passes over to the function's body, we had no choice but to develop a mechanism to keep track of all such jumps in the program.

Furthermore, function calls have parameters and they are themselves expressions. This means that complex hierarchies can be built by repeatedly nesting calls in other function's parameter values.

The easiest way to handle function calls, regardless of the call-nesting hierarchy, would be to save them in a list and scan each of their CFGs every time the control flow reaches the regarded section.

However, the above concept performs well only as long as no parameter aliasing is needed, i.e. only as long as no information about the values passed to each of the parameters of

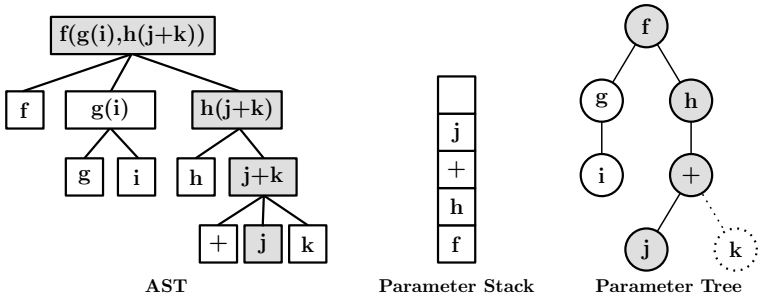


Figure 11: Parameter Stack Example

the called function is required. As soon as such information is needed, a mechanism to associate parameters to their passed values must be implemented, regardless if the values are immediates (constants), variables or themselves function calls.

To cover all possible scenarios, we have introduced the parameter trees (see 2.2.2) that describe a function call in tree form. The root of the structure is the top function in the call hierarchy. Each parameter of the call is described as an immediate successor of this node. The passed immediates and variables are saved as leaves - they have no successors -, whereas each passed function call will have its own children in a recursive manner. Please note that the child-order in the parameter trees is strict, so that parameter values cannot be switched by mistake.

The main advantages of the parameter trees are, among others:

- Storage support for non-statement CFG-relevant information
- Complete parameter aliasing support for call hierarchies of arbitrary depth

Due to the considerable possible depth of the call hierarchies, we must be able to move up and down in the parameter tree currently in construction, as the DFS traversal walks over the AST. The easiest way to achieve this goal is to implement the tree generation over a stack structure that will hold the current path in the tree in work. This will be the parameter stack.

The main purpose of the parameter stack is to describe calls. At times, however, it will also be used to hold complex array index hierarchies, that are described in the AST through function-call-similar subtrees. The partial call description held on the stack is actually the path from the root to the current node in the parameter tree.

The standard usage can be summarised as follows: for each PreOp of a function call respectively parameter specification, a new node is added to the current parameter tree and placed on the parameter stack, whereas for each PostOp of the same structure, the node is being removed from the stack again.

Figure 11 depicts the AST representation of a three-level deep call hierarchy and the associated parameter stack and tree. Please note that the AST in this diagram has been simplified for purposes of readability.

The marked nodes in the syntax tree show the current traversal path. We can clearly see here how each function called has been placed on the parameter stack, in this case finishing with the variable `j`.

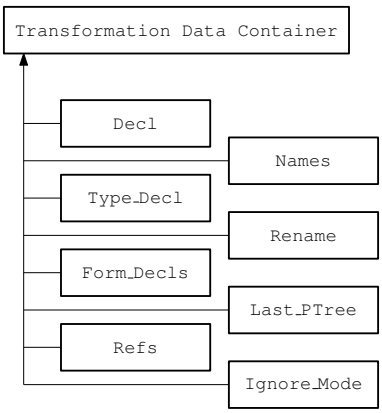


Figure 12: Transformation Data Container

The figure also depicts the corresponding parameter tree, the marked nodes in it showing the path to the current node. The dotted edge and node indicate the future position of the node `k`, that will be appended as soon as the DFS traversal of the AST reaches the corresponding element.

We summarise at this point the connection between the three structures again: while traversing the abstract syntax subtree of a subprogram call, we construct the corresponding parameter tree, with the path to the current node in the latter structure being held on the parameter stack.

3.2.2 Transformation Data Container

The state machine-like behaviour of the three event handlers PreOp, PostOp and CHF implies the existence of a global and throughout the procedure persistent data storage. Most of the gathered information will be saved directly into the flow world. Some, however, will need an interim storage, outside the flow world, until it can undergo more specialised processing.

This temporary storage is provided by the *transformation data container* (TDC).

Figure 12 depicts the component organisation of the transformation data container. We will now provide a description for each used substructure.

Declaration (`Decl`) As explained in the Overview (3.1) section, many Ada syntactical constructs can be grouped into classes, so that all structures in the regarded class are being described by ASIS using the same AST configuration.

This is especially well visible for variable-class entity declarations. These are, among others, the variables, the subprogram parameters and the named components of aggregate types (records).

Since each such declaration must be saved at a different location in the flow world, it is much easier for us to exploit their similarity and to use a common structure for all of them during the information gathering. This structure is provided by the data container, and it will be copied to the flow world, as a whole, once the subtree traversal has ended.

Upon receiving a PreOp event for one of the above declarations, we allocate a new `Flow_Types.Declaration` object and save a reference to it in the `TDC.Decl` variable. CHF events will fill the data structure with information and the PostOp will trigger its relocation to the flow world.

Type Declaration (`Type_Decl`) The exact same considerations and mechanisms used for the variable-class declarations are also employed for the types. Some of the type declarations sharing the same AST configuration are: the ordinary types, the subtypes, the private types and the private extensions.

A PreOp event for such an element triggers the allocation of a type description (`Flow_Types.Type_Descr`) object and the saving of its reference in the `TDC.Type_Decl` variable. CHF events will fill the data structure with information and the PostOp will trigger its relocation to the flow world.

Formal Declarations (`Form_Decls`) Formal declarations are used in the description of a generic entity (package, function or procedure) to mark the constructs that can be specified at a later point in time.

The formal entities can be: objects, functions, procedures, types and packages. For this purpose we will use class pointers in the TDC this time, instead of plain object pointers like in the previous two cases.

The variable- and type-class declarations described before, occur always in the context of their father flow object and for this reason, we only have to keep one pointer at a time in the TDC. Since formal declarations occur before the name of the generic entity, we have - during their processing - no father structure yet. Also, the number of the formal declarations is arbitrary. For these two reasons we must maintain a list of such declarations in the data container. Only when the generic entity has been created in the flow world do we relocate the entire formals list to its destination.

A PreOp event on a formal entity triggers the allocation of a corresponding object (`Flow_Types.Obj_Formal_Decl`, `Type_Formal_Decl`, ...) and its addition to the `TDC.Formal_Decls` list. The CHF events will fill the data structure with information.

At a later point in time, when the generic father structure is allocated, the formals list will be claimed automatically by it.

References (`Refs`) The inclusion clauses[3] **`with`** and **`use`** may carry control flow-relevant information and must, for this reason, be saved in the flow world. Let us assume an entity E refers (**`with`**) to a package B that has an initialisation block. Upon running E, the initialisation block of B will also be executed.

Since references occur out of context, i.e. without a parent structure, we must store them temporarily in the TDC until the tree traversal reaches the entity benefiting from these inclusion clauses.

A PreOp triggers the allocation of a reference object (`Flow_Types.With_Ref`, ...) and its addition to the `TDC.Refs` list. The CHF events will fill the data structure with information.

[3]We speak of inclusion clauses as references.

At a later point in time, when the benefiting entity is allocated in the flow world, the references list will be claimed automatically by it.

Names (`Names`) Almost every time a non-Ada entity, i.e. variable, subprogram, package, etc., is used in the AST, its name will be present as an ASIS element of type expression, subtype identifier.

The majority of these contexts is explicitly handled by Ast2Cfg and the names are used accordingly - for example, the identifier containing the name of a procedure called is saved correctly, as target CFG, in the newly created call node.

For all the cases that are not being handled explicitly, we have implemented a mechanism called *identifier harvesting*, that saves all the identifiers found on the way, that have not directly been used.

Let us consider the handling of a statement that contains variables which are not being used explicitly by the transformation logic. These variables will be saved on-the-fly in the `Unbounded_String` list `TDC.Names` and when PostOp-ing the statement, this list will be saved automatically in the corresponding CFG node. This way, no information will become lost during the process.

Rename (`Rename`) A renaming is an alias relationship between two entities, allowing the use of a pseudonym for an existing construct, in the context of this association. The AST of a renaming usually holds two children: the former being the surrogate and the latter the original. In complex renaming declarations, e.g. renamings of types with constraints, respectively renamings of function calls with parameters, several other child nodes may intercalate between the two.

The transformation data container provides a `Flow_Types.Renaming` structure to store the information during the gathering process. This record also offers the possibility to hold a parameter tree, should the need arise, i.e. should the renaming contain any function calls.

When PreOp-ing the root node of a renaming, the corresponding structure is allocated in the TDC. A PostOp triggers the data transfer to the flow world, and CHF events fill the record with information.

Last Parameter Tree (`Last_PTree`) In most of the cases, the need for a parameter tree is being announced by a special node in the AST. For example, let us consider the function parameters. The root of such a subtree is of ASIS type parameter specification. A Pre-/PostOp event on such a node triggers the creation or destruction of a corresponding parameter tree in the flow world.

For this reason, we process an AST node that needs an open tree, e.g. a function call, under the premises that the tree handling has already been done. We therefore perform no special check on the availability of the structure even if this may lead to inconsistencies in some cases.

Such problems arise when a parameter tree has not been properly announced. In this case the traversal constructs the tree without having saved its root in the flow world. After completion, the tree vanishes without leaving any reference behind, causing memory leaking.

This is the case when processing the guard expressions for protected entry declarations. They begin without an introduction node, as direct children of the subtree's root. Since more parameter trees may exist during the declaration, for example if the entry parameters have initialisation expressions, we can only look for the guardian expression once the subtree has been completely traversed. By this time, however, an eventual parameter tree will have already been lost.

To avoid the described inconsistencies, we save the root of each open parameter tree temporarily in the `TDC.Last_PTree` variable of the TDC, so that we can locate the last open tree even after it has been closed.

Ignore Mode (`Ignore_Mode`) We save, per default, each parameter specification we find during the traversal. In some cases this could lead to redundant information. The header of a subprogram is contained - in exactly the same form - both in the package specification and in the package body. Since a subprogram must be visible by the time it's being used and since we are working with successfully compiled code, we do not have to perform any visibility checks on our CFG call nodes, thus enabling us to ignore the spec-declaration. We will derive all our needed information from the package body.

To tell the transformation state machine to ignore the subprogram parameters present in a package specification, we use a boolean variable `TDC.Ignore_Mode` to instruct our three main methods to do so.

A Pre-/PostOp event on a, for example, function declaration (as opposed to a function body declaration) will set/reset the variable.

3.3 Transformation

This section describes the processing of each ASIS type explicitly handled by the AST-to-CFG transformation. Source code-insight may prove helpful during the reading.

Before we continue, however, we would like to offer a brief synopsis of the main ASIS types, to improve the reader's understanding of the transformation procedure.

ASIS employs an own, standardised, data structure hierarchy. Eight main classes exist:

- *Clauses*
  They describe inclusion and visibility relationships between Ada packages. References to packages containing initialisation blocks will trigger their execution, thereby adding control flow information to the current context. For this reason we must save any clauses we encounter into the flow world.

- *Associations*
  They are entity pairs describing a substitution relationship between an original and the surrogate to be used in its place. A typical such relation is the parameter association, that describes the value to be used as input for a parameter in a subprogram call. We need this kind of information to enable parameter aliasing, i.e. keeping track of a variable usage over a deep call hierarchy.

- *Definitions*
  ASIS uses definitions mainly to describe new types that appear in a program. Tasks

may also be used as type components. In order to allow a statical task analysis over our structures, we must provide means to locate any task references that may be hidden in type definitions.

- *Declarations*
  This is one of the most important ASIS type-classes. Every new entity, be it type, data structure or subprogram, will be introduced by a declaration. Aside from minor exceptions, we will always identify packages and CFGs over ASIS declarations.

- *Expressions*
  ASIS uses the type expression to refer mostly to the syntactical constructs that upon evaluation return a result. The main member of this category is the Function call. Since functions contain per definition control flow information, we must save them in our flow world.

- *Paths*
  Path-type nodes mark the beginning of each of a branch node's (**`if, case`**) possible alternatives. We will append them to the CFG only for reasons of readability.

- *Statements*
  With the statement being the basic control flow unit, it is by far the most important data structure in ASIS. Each imperative line of code in Ada will produce one statement node in the AST respectively one node in the CFG.

- *Exceptions*
  The exception handlers describe the actions to be performed should a certain exception be thrown. This information becomes especially important in a multi-tasking environment, where handler code may be run in parallel with other tasks.

Now that we have outlined the data types we can stress out the importance of the declarations and statements. The declarations tell us which entities exist and the statements present us with their content. The declarations will employ definitions to specify new types whereas the statements will use associations to call subprograms.

3.3.1 Clauses

Entity Visibility and Inclusion Clauses: *With, Use Package and Use Type Clauses*

These are the most common visibility clauses. They specify which entities are visible in the current context.

For each begun clause, we will add a new reference structure to the transformation data container. The name of the reference will be added by the CHF event, upon processing the corresponding child.

The list of visibility clauses will be claimed by the next flow structure to be created, e.g. when PreOp-ing the next package we will automatically save the list in it. The meaning of this transfer is: the structures named in the clauses list are hereby visible in the new package.

3.3.2 Associations

Parameter-class Associations: *Parameter, Discriminant, Array Component and Record Component Associations*

The expressions to be used as input for the parameters of a subprogram call are being described in the AST by using parameter association subtrees. Let us consider the function call `f(j)`. Its association subtree holds the node `j`. The parameter association subtrees may also hold the names of the parameters themselves, in calls like `f(i => j)`.

The task parameters are called discriminants and share the same structure with the former ones. The latter, may, however, also be used as initial input parameters for records.

Array component associations, as their name says, are used to initialise arrays, like `Var_Array := (Var1, Var2, [..])`. They use only a subset of the parameter-class association characteristics.

As described in the structures section (3.2), we employ parameter trees to represent the parameter hierarchies. The path to the current node in the active tree is saved on the parameter stack. For this purpose, when PreOp-ing such an element, we must place a new node (as root for this subtree) onto the stack. If this is an `i => j` association then `i` will be saved in `Node.Name` and `j` in `Node.Info`. This handling is especially useful if the surrogate is itself a function call. In that case, the parameter tree will continue to grow, with the parameters of this new function call as children of the one we have just created.

Generic Associations: *Generic Associations*

The generic associations are the parameters of either instantiations, i.e. when binding a generic construct and declaring a new entity of this type
(e.g. **package** `Real_Pkg` **is new** `Gen_Pkg(X => Y)`), or formal declarations, i.e. when specifying the formal parameters of a generic entity
(e.g. **with package** `Form_Pkg` **is new** `Gen_Pkg(X => Y)`). They have basically the same structure as the parameter associations. The generic ones, however, need special handling, since we must discern between them in order to store them correctly.

The former will be appended to the current renaming/instantiation structure, whereas the latter will be saved in current formal declarations list. Since they cannot be nested, we do not need stack handling.

A generic association subtree has one or two children, both expressions. The first one (if there are two) is always the name of the generic entity and last one is always the surrogate. The CHF will inspect the child count and ensure the appropriate storage of the data.

We are using `Alias` structures, with the `New_Name` holding the generic and the `Old_Name` holding the surrogate type.

3.3.3 Definitions

To better understand the usage of the ASIS definitions, we must first make a short digression towards the ASIS declarations (3.3.4). Every new entity (be it type, variable, subprogram, package or otherwise) to be used in an Ada program, has to be described by an ASIS declaration first. That is, for each entity used in a context, an abstract syntax subtree rooted in an ASIS declaration must be available at all times.

An ASIS definition describes the body of a type. It can be packed, together with a name, in a declaration to thereby create a new variable, type, task, etc. Each user-introduced type will have its own definition subtree in the AST.

Direct Type Definitions: *Type, Formal Type, Component, Private Type, Tagged Private Type and Private Extension Definitions*

The direct type definition class employs a specific abstract syntax tree configuration to describe new types. It holds the supertype (for variables), the base type (for arrays), the components (for records) and all the used identifiers. Our flow world type description structure accommodates the same fields.

This section is mandatory for each new type that occurs in the program. It's used both for named types (in ordinary type declarations, like `type X is new Y` and anonymous ones (in variable declarations, like `A: array() of Y`).

Since a type definition itself may hold function calls, we must also provide an adequate parameter tree.

The component definition describes either the base type of an array (anonymous items), or the type of a record component (named items). The type of a variable is described using a subtype indication. This fact is regarded as an inconsistency. Even if the record components have a semantical value closer to that of the anonymous array components, their syntax is still similar to that of the variables declarations. Despite this consideration, we will remain compliant with the ASIS standard and regard the type of a variable as supertype and the type of a component (named or not) as component type.

Most of definitions addressed in this sections are all situated near the top of the type description subtree, thus the handling done by PreOp, PostOp and CHF is minimal, mainly consisting of opening and closing the corresponding data structures. The real information gathering happens when processing the ASIS elements of finer granularity.

Subtype Indication: *Subtype Indication*

The subtype indication is one of the basic building blocks of the AST and has been referred to in the previous paragraph as container holding the type of a new variable. Its semantics are, however, more extensive, this element being used generally to mark an inheritance relationship between an existing type and a new entity, be it new type or static respectively dynamic variable. It can thereby be used in:

1. an inheritance relationship (type definition or variable declaration)
2. an allocation from subtype (e.g. `:= new type;`) in a type definition
3. an allocation from subtype in a variable-like declaration
4. an allocation from subtype in a statement

CHF will determine the context and save the name of the used type accordingly.

Record Definition: *Record Definitions*

A record definition marks the root of an AST containing the description of a new record-aggregate type. Since this element kind is always packed in one of the definitions presented in the previous paragraphs and since our type data structure can hold component declarations, we do not need any special handling for this node in any of the three functions.

Task Definition: *Task Definitions*

A special kind of definition, is the task definition. We mention it for the sake of completeness, the information derived from it, however, is negligible.

The tree rooted in such a node describes the specification of a task, i.e. which entries are available. We will deliberately skip this structure because for the task handling we employ the corresponding ASIS declaration type.

3.3.4 Declarations

We noted in the previous section, that each entity used in a program must be introduced by an ASIS declaration subtree. We will now describe the main declaration classes, the way Ast2Cfg perceives them.

Type-class Declarations: *Ordinary Type, Subtype, Private Type and Private Extension Declarations*

A type declaration in this category creates a new named type in the program. The subtree will have, among others, a name (always the first child) and a type definition (the body).

When beginning a type declaration (PreOp) we must create the appropriate data structure in the transformation data container. These structure will be closed and saved in the current flow object by PostOp.

Package-class Declarations: *Package, Package Body, Protected Type, Single Protected and Protected Body Declarations*

These are specifications, respectively bodies, of package-class flow structures. The package-class has, aside from the possible, yet not mandatory, initialisation blocks, no control flow semantics. The packages and the protected objects serve as containers for the CFG-class structures, i.e. the subprograms.

The flow world counterparts of these AST structures are the flow packages. As soon as we find the name of a new entity during the traversal, we can create it (CHF). Also, the new structure will now gather all data that occurred out of scope: the visibility and inclusion clauses, that are specified outside of the package and the formal declarations, that are also semi-external to the generic entity.

PostOp will close the package and any open initialisation blocks.

Variable-class Declarations *Variable, Constant, Deferred Constant and Component Declarations; Parameter, Entry Index, Discriminant and Loop Parameter Specifications*

These declarations are being described using some of the most complex AST configurations in ASIS, making them extremely powerful and versatile, yet equally difficult to handle.

This class contains variable and constant declarations on the one hand and parameter and entry index specifications on the other. All these types share a common grammar built up of:

1. entity names (variables, constants, components, parameters, etc.) of ASIS type defining name, subtype defining identifier
2. entity type of ASIS type definition or expression
3. initialisation expression of ASIS type expression

The relatively large size of this class is also an indication of the extensive synergies occurring between the ASIS types. To profit from them, we will use a common interim storage to hold the data during the gathering process.

When beginning such a declaration (PreOp), we must initialise a corresponding structure in the transformation data container. It will only hold the information, until this subtree has been finished. We also have to create a parameter tree to store a possible initialisation.

As noted above, using a common interim storage provided by the transformation data container enables us to profit from the declarations' similarities, allowing us to walk the tree and gather information regardless of the root type. Only after the processing finishes we will interrogate the node to decide upon the final destination of the data in the flow world.

As we have explained in the previous chapter, an active ignore mode indicates we are currently processing a subprogram specification header that we can skip.

When PostOp-ing this kind of declarations, we must transfer the gathered information to the current flow object. Due to the high complexity of the trees we are currently handling, this may, however, not always be a straight forward operation.

Let us consider the case of the discriminant specifications. They can occur not only as task parameters but also as input values in type declarations. As several similar cases exist, precedence rules had to be specified, to ensure the correct saving of the, please note, transparently gathered data. Concretely, we must query the existence of a type description structure. The latter is created only at the beginning of a type declaration subtree. Since it is possible to have a type inside of a task declaration, inspecting only the currently open flow object, would bear not enough information for a correct decision. The reverse of this case, i.e. a task inside of a type declaration, is not possible, hence enabling us to define the following priority rule: during the processing of a discriminant specification, the type description, if available, has precedence over the flow structure.

CHF will ensure the transformation logic switches correctly between the three base components: names, type and initialisation.

CFG-class Specification Declarations: *Procedure, Function and Entry Declarations; Procedure Body, Function Body and Task Body Stubs*

This class of declarations describe the header of a subprogram specification (function, procedure, entry, stub, etc.). Due to the fact that the body of the same subprogram must be visible when being used, we will ignore most of the above specifications.

PreOp sets the ignore mode flag and PostOp resets it again.

CFG-class Body Declarations: *Procedure Body, Function Body, Entry Body, Single Task, Task Type and Task Body Declarations*

The structures handled here are the main control flow information containers. Their Ast2Cfg counterparts are the control flow graphs. Similarly to the packages, we only create the appropriate flow structures once we find their names (CHF).

Entry body declarations also have a guard, i.e. an expression that decides whether the code will be executed or not. We will save the root of the generated parameter tree together with the current flow object.

Generic Structures Declarations: *Generic Package, Generic Procedure and Generic Function Declarations*

Although the generic types specified here are no bodies (but specifications) we still need to handle them like the former and create flow structures for them. The reason is that we need containers for their formal parameters.

Since by the time we reach them, we will have already processed their formal declarations list - the formals occur earlier in the AST - we only need to save their list into the newly created flow objects.

Formal Declarations: *Formal Object, Formal Type, Formal Procedure, Formal Function, Formal Package and Formal Package With Box Declarations*

Generic entities employ formal declarations to define which parameters (variables, types, subprograms, packages) are to be specified by the end-user of the construct.

Since we do not need the parameters of the declared subprograms we can enable the ignore mode. All the formal declarations will be stored in a list, until the generic flow object has been created. Please note that this object does not exist yet, since we do not have its name.

The formal declarations themselves are very similar to the variable-class ones. They also have three types of children:

1. entity names of ASIS type defining name, subtype defining identifier; these will be the generic parameters of the new object
2. entity base- or supertype of ASIS type expression
3. initialisation expression of ASIS type expression

The handling is done in CHF and is otherwise similar to the variable-class declarations, with special regard to the type, here not type definition but expression, and to the storage, here into the formal declarations list.

Renaming Declarations: *Procedure, Function, Generic Procedure, Generic Function, Package, Generic Package, Object and Exception Renaming Declarations*

Renaming declarations are rather straight forward aliasing operations, where an existing entity is assigned one more name. Some renamings may hold parameters to be passed to the existing entity.

The handling is simple. PreOp will create the appropriate records, CHF will save the surrogate name and PostOp will add the name of the existing structure. Eventual initialisation parameters will be stored in a parameter tree.

Generic Instantiation Declarations: *Package, Procedure and Function Instantiations*

Generic instantiations are used to bind a generic entity to given types, subprograms, etc. and thereby create a new one with certain characteristics.

Due to the fact that renamings may also be passed parameters, the handling of the generic instantiations is completely identical, with regard only to the new underlying data structure type.

3.3.5 Expressions

All syntactical constructs that upon evaluation return a value are deemed expressions. They range from simple array accesses to complicated function call hierarchies.

Aggregates References: *Positional Array, Record Aggregates*

This class of references includes array accesses like `A(i)` and record accesses like `A.i`. Since arrays, records and function calls can be combined to create complex accessing hierarchies, we act preemptively in PreOp by creating an extra node in the parameter tree. All subsequent function calls will be saved as children of this node.

Let us consider a configuration like `A(f(i),g(j))`. An item of the bidimensional array `A` is accessed using two functions `f` and `g`. By acting as described above, we will ensure that both function calls will be saved as children of the array access node, thereby enabling us to determine which call was used for which index.

Attribute References: *Attribute References*

An access to an entity's attributes is called an attribute reference. They are constructs like `Variable'Attribute`.

Per default, we gather all identifiers we encounter on the way, including attributes. Since the latter can only be read we will ignore them.

Allocations from Subtype: *Allocations from Subtype*

Each **`new`** dynamic memory allocation provided with a type, is called an allocation from subtype. Since they can be used both in the declarative and the imperative section of a program and since they resemble the function calls in their usage, we will treat them like the latter and spawn a new node on the current parameter tree for them.

Type Conversions: *Type Conversions*

A type conversion subtree describes the casting of a variable from its defined supertype to a second compatible or inherited one. The subtree has two children, the first describes the type we are converting to and the second describes the structure to be converted.

Since the target type is an ASIS expression/identifier, the variable harvesting process will save it into the identifier list, even though it is not a variable. For this, we will forcibly flush the list before proceeding with the converted structure.

Identifiers: *Identifiers*

The identifier is one of the most important basic building blocks of the AST. Each user-defined name occurring in the program will be found in the AST as identifier. Some minor exceptions exist, for example the labels that are of ASIS type defining name, subtype defining identifier.

Our identifier harvesting mechanism will gather all found identifiers in a special list. The names that are not being used directly by the transformation logic, will be saved automatically in the current CFG node, at the end of each statement.

Operator Symbols: *Operator Symbols*

Operator symbols are the predefined operations like +, −, etc. They would be ignored by the identifier harvesting since they are no identifiers, but for the sake of the CFG readability we will treat them as such.

Function Calls: *Function Calls*

As the name says, these are the genuine function calls. They possess their own parameters and can also be used as parameters for other subprograms, at the same time.

To enable parameter aliasing, i.e. the association of the input entities to the corresponding subprogram parameters in deep call hierarchies, we will construct a parameter tree. Each function call is represented by one node in the tree and the ordered set of a node's children are the specified input entities. This definition is recursive, allowing the tree to grow as the parameter call hierarchy becomes deeper. A call like `f(g(h(),i()))` will result in a three levels deep tree, with four nodes. `f` will be the root and will have one child: `g`. `g` will, in turn, have two children: `h` and `i`.

To construct such a tree, we must add for each function PreOp a new node to it and remove it in PostOp.

Selected Components: *Selected Components*

The selected components are Ada's way of addressing nested entities like `Pkg_Outer.Pkg_Inner.`**`Procedure`**. ASIS splits these names and constructs for each such composition a binary syntax subtree, with each token contained in one leaf. The root holds the complete name, the left child the first token and the right one the rest of the name. The construction process is recursive, leaving the last node at the bottom right of the tree with two leaves, each holding one token.

Since we need to know the complete name of an entity, we must recompose it from the tree. For this, we will gather the names bottom up, each CHF concatenating the name

constructed so far with the token in the left child, i.e. in exact reverse order as it was initially parsed.

Indexed-class Components: *Indexed Components, Slices*

Indexed components and slices are ASIS' way of addressing array aggregates.

Due to the extreme possible complexity of the array indexes we will only store the name of the leading variable. Also we will attempt to construct a parameter tree in a similar fashion to the function call hierarchies.

3.3.6 Paths

Branching-class statements, i.e. **`if, case`** are being described in the AST by providing a subtree for each possible decision. The root of each such subtree is of ASIS type path. To improve the CFG readability we will also add a path node at the beginning of each CFG branch.

3.3.7 Statements

The imperative constructs in an Ada program are being referred to as *statements*. They are by far the most important ASIS types, with the statement being the basic control flow semantic unit.

Each imperative line of code in Ada produces one statement node in the AST. Complex ones also have associated subtrees for each of their components. ASIS also generates for each possible branch in an **`if`** or **`case`** construct a so called path-node. One such node marks the beginning of a possible alternative.

The standard procedure when dealing with statements/paths (PreOp) is to create a general purpose node in the CFG and put it on the node stack of the current flow object. Special statements, however, require somewhat different handling, so, at a later point in time, we may decide to replace the general purpose node with a specialised one. In PostOp we will remove same node.

Statements occur mostly in CFGs and will thereby, most of the time, have a CFG parent structure. For those that do not fall in this category, i.e. the ones in the initialisation section of a package, we will create an Init-CFG and append it to the current flow object.

The statement handling is also the main user of the node stack. We have already outlined its usage in the data structures section: each new node replaces its top, with the stack only growing for branch- and loop-class statements. When describing these classes we will further examine this topic.

The identifier harvesting also plays an important role in the statement handling. At the end of each statement (PostOp), we save the list of gathered identifiers in it and then reset it. By doing so, we ensure that no variables will be lost on the way.

Statements may also contain expressions. To save them, we provide each statement with a right hand side parameter tree. The recursive nature of the trees makes them the ideal structures to represent nested expressions. In most of the cases, one tree per statement is enough. The only one that needs two is the assignment statement. For it, we will also provide a left hand side parameter tree.

Statements may also have names (labels). They are especially important since they can be targeted by the jump-class statements (**`goto`**, **`exit`**) adding extra information to the control flow.

All the statement types listed below replace the general purpose node with a specialised one.

Assignment Statements: *Assignment Statement*

The assignment statements, e.g. `A := B`, are the only ones that need a second parameter tree. This may be necessary since the left hand side of the assignment may also hold function calls.

Special attention must be paid to the written variable - it must be stored separately from the rest. There is no bound to the complexity of the expression used in the specification of this variable. This restrains the precision of the statical analysis. We will limit ourselves to keeping only the name of the leading variable.

Jump-class Statements: *Goto and Exit Statements*

The subtree holding the description of a statement in this category also holds the name of the jump's target. We will replace the generic node with a structure providing storage space for it. We will use this name in the post-transformation phase to make the correct links in the CFG.

Call-class Statements: *Procedure Call, Block, Raise, Entry Call, Requeue, Requeue With Abort, Abort and Accept Statements*

All the call-class statements named above represent jump points into foreign CFGs. The most common ones are the procedure calls. In a similar way we treat the task-related jumps, like entry call and requeues. A raise may not have the same semantics as the common call statement, yet it is still best treated here since it may be an inter-CFG jump. blocks and Aborts have no calling semantics, but for the sake of consistency we will pack all subsequent statements in an extra CFG that we will then call at their point of occurrence.

As previously suggested, the parameter tree of a call node (including function calls), describes the input entities provided to the referred subprogram. Its root represents the main subprogram and the children are its parameters. They may be function calls themselves and have own parameters. The definition is recursive, setting no bound to the call depth.

Branch-class Nodes: *If, Case, Selective Accept, Conditional Entry Call, Timed Entry Call and Asynchronous Select Statements*

This is the first of the two classes of nodes that, in their handling, modify the size of the node stack. In the appropriate section (3.2.1) we have already described the employed mechanism.

Similarly to the previous classes of statements, we replace the current top of the node stack with a new structure: this time with a branch node. If we were to leave it at that, the first node to be handled next will replace it and we would lose contact with the branch head. To enable a transparent handling of the rest of the nodes, we must increase the size

of the stack. Next we will place an EndIf node on it, that, in the end, will gather all split paths back together.

To not lose this node either, we will place a second, redundant copy of the branch head on top of the node stack. With this configuration we can now process the rest of the nodes as if the stack had only a height of one. At the end of each branch we restore the If-EndIf-If configuration. Once all branches have finished, we rearrange the stack leaving only the EndIf node behind.

If-statements have a default **`else`** branch, if the user does not explicitly provide one. The former, however, has no representation in the AST. To ensure the semantical correctness of the CFG, we will provide one automatically, should the AST not include one.

Loop-Class Nodes: *Loop, While Loop and For Loop Statements*

Similar to the branch-class, the loop-class also need a customised node stack. This time we must only link the last node in the loop back to the head, so a Loop-Loop configuration on the stack will suffice.

Once all statements in the have been processed, we clear the currently Loop-LastNode configured stack and link the nodes properly.

To ensure that no loop head lies at the end of a control flow, we will also add a pseudo post-loop node to the CFG.

Termination-class Nodes: *Return and Terminate Alternative Statements*

The handling of these statements only includes replacing the generic node with a termination one. Further processing will be done in the post-transformation phase.

This corresponds to the syntactical representation provided by the AST: the return node is followed by other statements. The semantics are, however, different: the nodes behind the return statement are not reachable over this path. In the post-transformation phase, we will correct the CFG by removing all the successors of the return statement.

3.3.8 Exception Handlers

We treat exception handlers in a similar way we treat block statements. The complete code of an exception handler will be packed in a separate CFG, leaving only one reference in the original code behind. Thrown exceptions can be caught up to a certain extent:

- exceptions thrown and caught in the same CFG can easily be put in relation to one another
- exceptions caught in foreign CFGs can be handled only if some information about the beginning of a call graph is available; this graph can then easily be constructed from the flow world

3.4 Post Transformation

3.4.1 Loop Refinement

As can be seen in Figure 13 a **`while`** loop without an **`exit`** statement is already represented correctly after the transformation process (The same is true for **`for`** loops without **`exit`** statements). However, when it comes to simple **`loop`** statements or **`exit`** statements some refinement has to be done in order to achieve a correct representation.

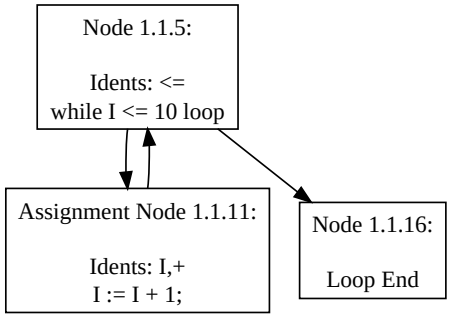


Figure 13: A **`while`** loop.

Figure 14 shows a simple loop with an **`exit`** statement. The edge from the exit jump node to the loop end is missing, and even though a simple loop has no condition in its header there is an edge from the header node to the loop end. In Figure 15 the same loop after the refinement is shown.

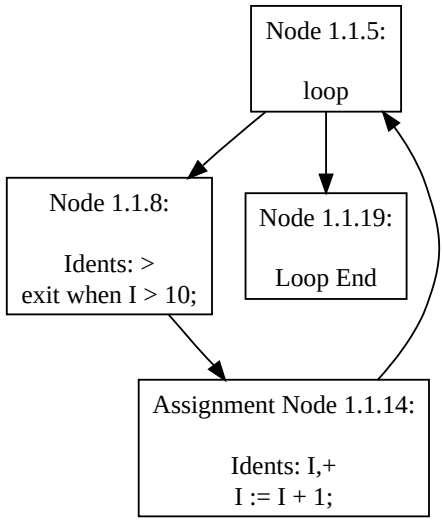


Figure 14: A simple **`loop`** statement before the refinement.

Since the loop refinement takes place after the CFG has been built, the first task is to find the loops again. As a basis Tarjan's algorithm for constructing the loop forest as presented by Ramalingam [5] is used. Since this algorithm needs the backedges, first a DFS with two different node marks is performed on the CFG. When a node is visited for the first

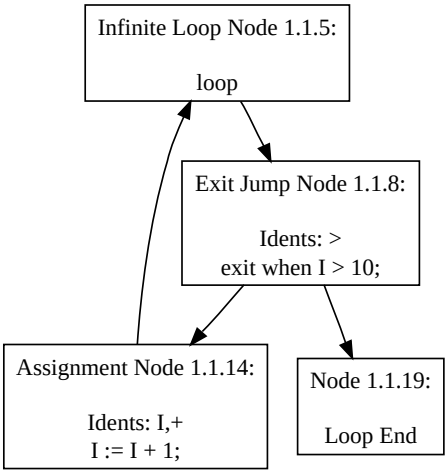


Figure 15: A simple **`loop`** statement after the refinement.

time it is marked as "grey", and when the recursion on that node is finished it is marked as "black". Any edge encountered during this algorithm that leads from a node marked as grey to another grey node is a backedge. This is because in a *reducible* CFG every retreating edge is a backedge [3].

There was no need to modify the *findloop* procedure of Tarjan's algorithm, however, in the *collapse* subprogram, the actual loop refinement is performed. `Collapse` is called for every loop with the loop header and the loop body as arguments. Each node of the body is *collapsed* on the header as it is in the standard algorithm.

Then exit jump nodes with an empty target, which means they exit the current loop, and exit jump nodes with a target loop are collected separately in two lists. Also for every node it is recorded if it has a successor that is not part of the loop. At the current state only one such edge may exist, namely the edge from the loop header to the loop end node (see Figure 13-14).

Now every exit jump node for the current loop and those that match the label of the current loop are connected to the loop end node. After that the list for the current loop exits is empty, however, the list for loop exits with a target may contain elements that are used in a later call to `Collapse`. Note that because of the nature of Tarjan's algorithm the inner loops are always found first, so the loop with the corresponding label is always found after the **`exit`** is found.

Finally, if the current loop is a simple **`loop`** statement, the edge from the header to the loop end is removed. In case there is no exit statement within the loop and therefore the loop is an endless loop, it is possible that now some nodes are not reachable any more, or at least are not reachable by only following the links to successors. However, they may still be reached by traversing the CFG backwards, using the links to the predecessors of each node. Therefore these nodes need to be completely removed and deallocated. So before removing the edge its target node is saved and at the end of the loop refinement `Strip_Dangling_Nodes` provided by `Post_Trans_Processing` is called (see Section 3.4.3) which removes the non-reachable subgraph in case there is one.

3.4.2 Connect Gotos

During the transformation the labels of a statement were saved separated by commas as a string and can now be retrieved by calling `Get_Name`. Similarly the target of a **`goto`** statement was stored. The only thing left to do is to connect every `Goto_Jump_Node_Object` to its target node, so that there is an edge in the CFG representing the jump. This is done by the `Connect_Gotos` subprogram.

First two lists are built during a depth first search: one containing references to the found goto jump node objects (`Sources`) and one for the nodes with labels (`Targets`). Note that a goto jump node may also have a label, and therefore may be on both lists simultaneously. Then for each item in the `Sources` list the list of targets is searched for the corresponding label. When the target is found it is connected to the source node using appropriate calls to `Add_Succ` and `Add_Pred`.

However, before the target is added as a successor of the source, the existing outgoing edges of the source node have to be deleted. This is to remove the auxiliary edge that connected the goto jump node to the node for the statement right after it. That auxiliary edge was necessary in order to keep the CFG connected until now. In case there is unreachable code following the goto statement, now there are nodes in the CFG which are unreachable by only following the links to successors (see 3.4.1). They have to be removed by calling `Strip_Dangling_Nodes` (see Section 3.4.3).

3.4.3 Removing Dangling Nodes

`Strip_Dangling_Nodes` is called by `Connect_Gotos` and the loop refinement subprograms in order to check for subgraphs that are not reachable by following the successor links any more. However, because a backwards traversal of the CFG which follows the predecessor links still reaches those subgraphs they need to be properly removed and deallocated.

`Strip_Dangling_Nodes` gets a set of nodes as parameter that are roots of such possibly unreachable subgraphs. The goal is to find all nodes that are not reachable through an ordinary DFS. Therefore a DFS is performed on the CFG and all reachable nodes are added to a set. Next, depth first searches are executed starting at every root node of a possibly unreachable subgraph, and the nodes that are reachable are added to another set. Note that such an unreachable subgraph may still be connected to the CFG somewhere else. So finally the set containing the nodes collected with an ordinary DFS is subtracted from the set containing the unreachable subgraphs so that only those nodes that are not reachable through DFS remain. These nodes are then disconnected from the CFG and deallocated properly.

As an example consider Figure 16. Solid edges represent successor links while dotted edges are predecessor links. Figure 16(a) shows a CFG before Goto Node 1 is connected to Target Node 7. In Figure 16(b) the CFG can be seen after the goto was connected. Node 2 is the root of an unreachable subgraph which consists of the nodes 2-6. Note that it still may be reached using the predecessor link (7,6). In Figure 16(c) the two sets built by `Strip_Dangling_Nodes` are depicted. The set surrounded by a solid edge is the set containing all nodes reachable through an ordinary DFS, while the set surrounded by a dashed line contains all nodes that are reachable from the root of the dangling subgraph. The difference

leads to nodes 2-6, which are removed as shown in 16(d).

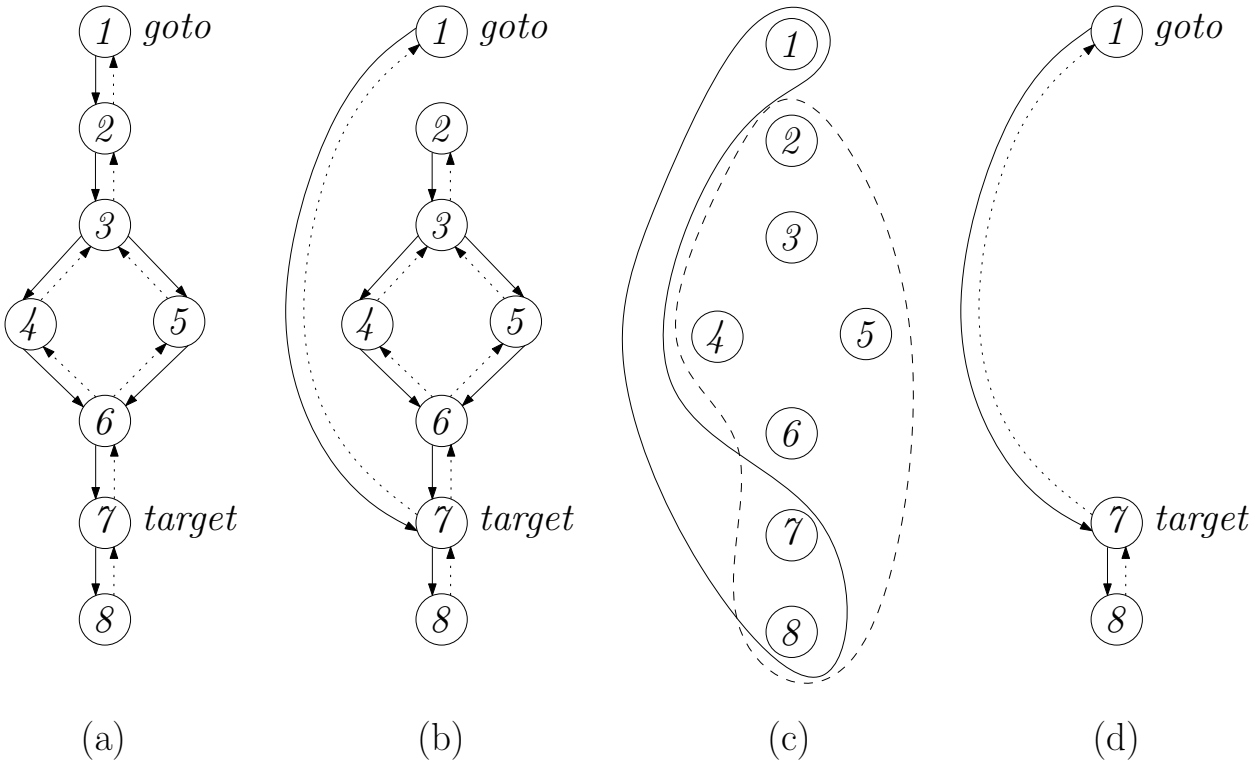


Figure 16: Dangling nodes being removed from the CFG.

4 Performance

To give an impression of the performance of Ast2Cfg, in this section we present the benchmark data for the execution of Cfg2Dot in two different cases. The execution times were measured with the Linux command *time* on a machine with an Athlon XP 2800+ processor and one gigabyte of RAM.

The first test was performed on the adb files located in the *adainclude* directory of a typical GNAT installation. Note that this directory also contained the ASIS-for-GNAT files. The 588 tree files with a total of 426.4 MB were generated with GNAT in approximately 91 seconds. Then Cfg2Dot generated 5472 separate CFGs and 189 Pkg/Cfg trees in 213 seconds.

The second test was executed using the adb files of Ast2Cfg itself. After 34 seconds of tree file generation, which led to 111 files with a total of 252.8 MB, we deleted all files that were not directly related to Ast2Cfg, such as tree files for Ada or ASIS includes. Since the latest ASIS-for-GNAT version does not already support some of the Ada 2005 constructs used in the post transformation code, one of the Ast2Cfg tree files also had to be deleted. Then Cfg2Dot was executed for the remaining 18 tree files with a total of 31 MB. After 35 seconds 293 CFGs with 53 Pkg/Cfg trees were generated.

5 Future Work

Due to the raw state of the gathered control flow data, further refinement is necessary in the post transformation phase.

Also, two projects based on Ast2Cfg are already in work. The first one aims at the detection of busy waiting while the second attempts to find access anomalies in Ada programs.

6 Acknowledgements

We would like to thank Johann Blieberger for his advice and guidance throughout this project.

List of Figures

Listings

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

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