A Framework for CFG-Based Static Program Analysis of Ada Programs

Raul Fechete, Georg Kienesberger, and Johann Blieberger

Institute for Computer-Aided Automation, TU Vienna
Treitlstr. 1-3, A-1040 Vienna, Austria
{fechete,kienes,blieb}@auto.tuwien.ac.at

Abstract. The control flow graph is the basis for many code optimisation and analysis techniques. We introduce a new framework for the construction of powerful CFG-based representations of arbitrary Ada programs. The generated data holds extensive information about the original Ada source, such as visibility, package structure and type definitions and provides means for complete interprocedural analysis. We use ASIS-for-GNAT as an interface to the Ada environment and extract the needed information in a single traversal of the provided syntax trees. In addition, further refinement of the resulting data structures is done.

1 Introduction

Many control and data flow analysis approaches [1,2,3,4,5,6] rely on the representation of a program in form of a *control flow graph* (CFG) [7].

We introduce a framework that generates CFG-based data structures holding comprehensive information about the original Ada source. The packages and CFGs of the input program are structured in trees according to their hierarchical organisation. We also use trees to represent complex expressions, allowing an in-depth analysis of the control flow. Types and variables are saved together with their complete definitions, thereby providing means to track them over several inheritance levels. Furthermore, we facilitate interprocedural analysis by referencing the target CFG in each subprogram call.

During the transformation, we extract the needed information from the *abstract syntax trees* (AST) [7] provided by ASIS-for-GNAT. Afterwards, in a post transformation phase, we further refine the resulting structures.

2 The Library

We designed *Ast2Cfg* as a library that uses ASIS [8] to get the information that is needed to build the CFG for a given Ada program. In fact we use the ASIS-for-GNAT implementation of the ASIS standard. Therefore, the so-called *tree files* for an Ada program, which are generated by GNAT [9], are used as input. Then ASIS provides us with the abstract syntax tree of the input program. During the traversal of this AST, which is done using *depth first search* (DFS)

F. Kordon and T. Vardanega (Eds.): Ada-Europe 2008, LNCS 5026, pp. 130–143, 2008.

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

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

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

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

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

Fig. 1. A small application using Ast2Cfg

[10], we simultaneously build the corresponding CFG. Next, right after some refinement, the control flow information is made available to the library user in form of a single object, the *flow world*. Figure 1 shows a small application that uses Ast2Cfg to output the name of all top-level packages of the adt-files in the current directory.

We already developed a simple program, called *Cfg2Dot*, that uses the Ast2Cfg library to output the CFG for a program in dot graphics format [11]. Ast2Cfg, Cfg2Dot and additional documentation [12] are available from http://cfg.w3x.org.

2.1 The World Object

All information gathered during the transformation phase is saved in an object of type `World_Object`. It contains a list of package objects (`Pkg_Object`) that correspond to the top-level packages of the analysed Ada program.

`Pkg_Object` is derived from the abstract `Flow_Object`. The same applies to `CFG_Object`, that represents a control flow, and `Node_Object`, which is used by `CFG_Object`. Also, as described in detail below, each of these types has a series of subclasses such that a more fine-grained classification is possible. Where necessary, the derived types are also grouped into specifications and bodies. Figure 2 shows an overview of the class hierarchy where rectangles with dashed lines represent abstract types and those with solid lines concrete ones.

Since in Ada subprograms and packages may be declared within each other we have to keep track of such nesting relationships. Every flow object has a list of predecessors and successors. While node objects use those lists to build up a CFG, package and CFG objects use them to represent the nesting structure we

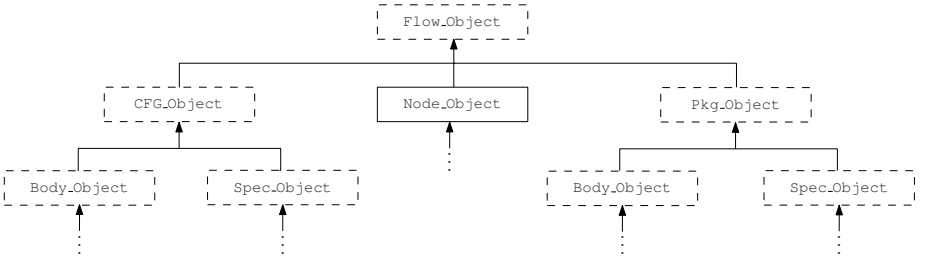

Fig. 2. An overview of the class hierarchy for the flow types

call *Pkg/CFG tree*. In other words, if B is declared within A, B is a successor of A and A a predecessor of B.

Another important tree structure is the *parameter tree* that we generate whenever an expression is encountered. A parameter tree is saved directly in the flow object that contains the expression and holds information on the used variables, functions, etc. and their nesting. As a result a parameter tree is a complete hierarchical representation of an expression, allowing the user to implement powerful static analysis algorithms.

Figure 3 shows the Cfg2Dot output for the single assignment statement `C(1) := Outer(A, Inner(X => B));` where `A` and `B` are variables, `C` is an array and `Inner` and `Outer` are functions. The CFG consists of a single assignment node with two parameter trees, highlighted by dashed lines: one for the part to the left of the assignment operator and one for the right hand side. Every level in the parameter tree represents a nesting level in the underlying expression.

Flow Object. Every flow object has a unique id which may be tested for equality. In addition, flow objects have names. However, for node objects which are derived from flow objects, the name is empty in most cases. Furthermore, lists of variable declarations, generic formal parameters and renamings are available. Finally, we also store a list of `with`, `use` and `use type` clauses in every flow object.

CFG Types. CFG objects use node objects to represent the control flow information of different Ada entities like subprograms, blocks, initialisation sequences etc. Therefore in every CFG object we save a reference to the root node, the total number of nodes (without those in parameter trees) and, since many CFG objects represent subprograms, a list of parameters.

`CFG_Object` itself is declared abstract, hence all actual CFG objects have to be of one of the more specific, concrete subtypes. In the simplest case a subprogram has to be represented, which is done by either creating a `Proc_Object` for a procedure and, in case of a function, a `Func_Object`. For a *block*, which is located within some other CFG object, we create a separate `Block_Object`. Next we insert a *call node*, which is used to represent a subprogram call, at the position where the block used to be within the enclosing CFG. So, in fact,

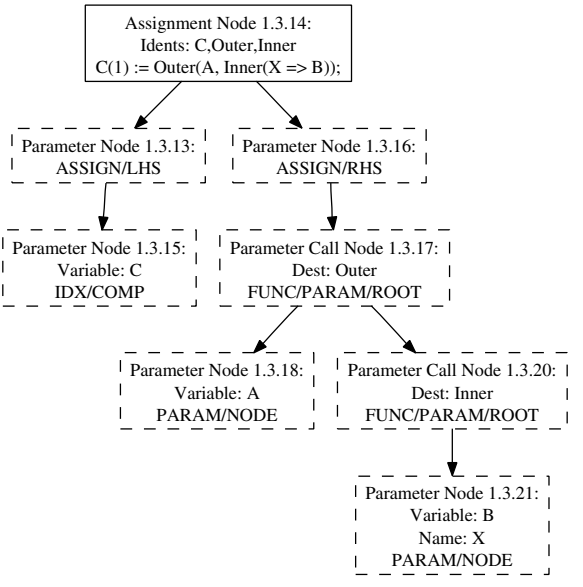

Fig. 3. A node with two parameter trees

a block is handled like a parameterless procedure, called when the block is declared. We transform an initialisation sequence of a package body into a so-called `Init_Object` and create an `Except_Object` for every exception handler.

In case we encounter a task body, a `Task_Object` is created where every `accept` block is represented separately by an `Entry_Object`. As it was the case with simple blocks, we link such an `Entry_Object` to its enclosing task by using a call node. Furthermore, we also map the protected entries of a protected object to entry objects. To represent the abortable part of a `select – then abort` statement we use an `Abort_Object`, which is, again, linked to the enclosing CFG using a call node.

Finally, there are three CFG objects which actually do not contain any control flow information. The main reason why we use them is that because of their position in the Pkg/CFG tree we are able to keep track where they are defined and can gain visibility information later on. So for a task type declaration we create a `Task_Type_Object`, while a simple task specification is represented by a `Single_Task_Object`. Finally, whenever we encounter a generic procedure or a generic function specification we create a `Generic_Object`.

Package Types. The `Pkg_Object`, like the `CFG_Object`, is abstract, which is why variables have to be of a more specific subtype. The main purpose of package objects is to help building the Pkg/CFG tree, and therefore the successor and predecessor lists are the most important components. Another component every package object except for the `Prot_Object` and the `Def_Object` has, is a list of the variables that are declared within.

For uniformity reasons we introduce an artificial default package of type `Def_Object` which contains CFGs that do not have an enclosing package such as library level procedures. A `Prot_Object` represents the body of a protected object or type, which may contain subprograms and entries. We map the specification that belongs to such a `Prot_Object` to a `Prot_Type_Object` in case of a protected type declaration and to a `Single_ Prot_Object` otherwise. The body of a generic package requires no special handling, however, we transform the corresponding specification into a `Gen_Object`.

Finally, for representing ordinary packages not mentioned above, we use a `Simple_Spec_Object` and, in case there is an accompanying body, a `Simple_Body_Object`.

Node Types. Any node that is not treated specifically as described below is of type `Node_Object`. If a statement has a label, then the label is saved as the name of the node representing this statement. Also, for every node we store a string that holds at least part of the code that this node is representing. Furthermore, all nodes have a *right hand side parameter tree*, and the `Assign_Node_Object`, which corresponds to an assignment statement, also has a *left hand side parameter tree* for the part to the left of the assignment operator. Finally, for every node we save the `Asis.Element` that is the source of this node. The `Asis.Element` can be seen as a link back into the ASIS AST. Consequently additional information can be acquired by analysing the AST starting at the element of an arbitrary node.

As already mentioned, we not only use a `Call_Node_Object` for the representation of a subprogram call, but also in several situations that are treated similarly. Clearly, the most important component of a call node is the reference to its destination CFG. We derived several subtypes from `Call_Node_Object` to convey additional information on the type of the call. So, for example we use an `Accept_Node_Object` to link an `Entry_Object` to its enclosing task body, while we represent a call of such an entry with an `Entry_Call_Node_Object`. Likewise an `Abort_Node_Object` links the abortable part of a `select – then abort` statement into its CFG. Finally, there is a subtype of `Call_Node_Object` that we exclusively use within parameter trees to represent a function call: the `Param_Call_Node_Object`.

Whenever we encounter a `goto` we use a `Goto_Jump_Node_Object` to point to the destination of the `goto`. Moreover we create an `Exit_Jump_Node_Object` for every exit statement within a loop. Note that the target of an exit jump node is empty in case it exits the innermost enclosing loop. For a `return` statement, we also create a special node, which is of type `Return_Node_Object`. A `Loop_Node_Object` marks the header of a loop, and the two concrete subtypes enable to distinguish between a `while` or `for` loop and a simple `loop` statement.

A parameter tree is built by nodes of type `Param_Node_Object`, `Param_Alloc_Node_Object` and the already mentioned parameter call nodes. We use the `Param_Node_Object` to save the name of the variable that was supplied as a parameter and the name of the parameter itself, in case it is known. The

`Param_Alloc_Node_Object`, however, represents a dynamic allocation using the `new` keyword.

Whenever we encounter an `if` or `case` statement, a special header node of type `Branch_Node_Object` is created so that its successors contain the branching conditions. After such a node with a branching condition the subgraph for the actual branch follows, until control flow is united again in an end node.

3 Transformation

We obtain the control flow data from the AST by using a two-phase mechanism. The first step, the *transformation*, includes extracting information from the tree and building the raw flow structure. The second one, the *post transformation*, further refines the output of the former.

The information retrieval is constructed on an inorder traversal skeleton provided by ASIS. The program walks the tree one node at a time, generating three types of events.

1. A *PreOp* event is triggered when the traversal reaches a node for the first time, before any other processing is done.
2. A *PostOp* event is triggered immediately after the traversal has left a node, as soon as all processing involving it has finished.
3. A *CHF* (child-has-finished) event provides us with a binary relation between a parent and a child node and is thereby context-sensitive. The previous two events, however, are context-insensitive, bearing no information of a node's relatives. CHF is triggered for each of a node's children, right after their processing has finished.

The event triggering traversal imposes a state-machine-like architecture on our transformation mechanism. We employ stacks to hold the current traversal state and three callback functions, one for each event named above. Since each method must be able to handle any of the ASIS node types, all three have a symmetrical structure.

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

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

As an example, let us consider the variable declarations, the component declarations of the aggregate types and the subprogram parameter specifications. A typical subtree for one of the declarations above holds the name of the new entity, its type and its initialisation expression. The only node that tells us what kind of tree this is, is the root. This information, however, is transparent to the handling mechanism of the tree. The complete information about the new entity

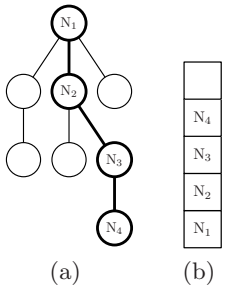

Fig. 4. AST path with associated element stack

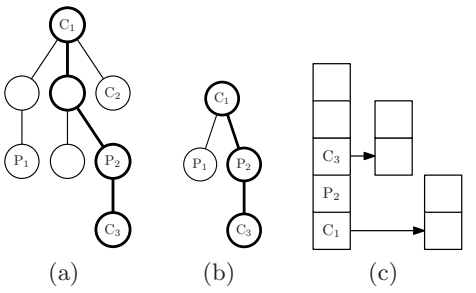

Fig. 5. AST path with associated flow tree, flow stack and node stacks

is gathered in an interim storage common to all the declarations of this class, and only the PostOp handling of the root node decides where the data should be transferred in the flow world.

The example above is also illustrative for the use of the interim storage, the *transformation data container* (TDC). Many ASIS types can, however, be added to the flow world immediately upon being reached. As an example, let us consider the handling of statements. Their flow world counterparts are the CFG nodes, and a statement PreOp triggers automatically the addition of a node to the current flow entity.

3.1 Element Stack

The *element stack* is the backbone of the AST traversal. It holds the path from the tree root to the current node. The stack grows with each PreOp event, as the search goes deeper, and diminishes with each PostOp event, as it returns.

Figure 4(a) shows an AST with the thicker edges and nodes indicating the current traversal state. The nodes in the active path are marked N_1 to N_4 with no regard to their syntactical value in the original program. Figure 4(b) displays the corresponding element stack state.

Keeping only a pointer to the current node is not enough, because for each node in the path we must be able to store additional information. We may need to access this information repeatedly, as the search keeps returning to this element. Such information is the count of already visited children, i.e. the number of CHF events, and the corresponding flow world structure for this node, e.g. a CFG node for a statement or a package pointer for the root of a package declaration subtree.

The element stack also provides us with an additional consistency check. The flow structure on top must also be the current one in the flow world.

3.2 Flow Stack

Ada allows both the nesting of subprograms in packages and vice versa. This fact leads to complex nesting hierarchies. We will represent these relationships

in the flow world using a tree structure, that we call the *flow tree*. Its root is the outermost package or CFG. The children of a node in the flow tree represent the structures immediately nested in the construct the parent node stands for. The tree only describes the nesting of packages and CFGs. No information about other nesting relationships, like that of loops, is saved in the flow tree.

Due to similar considerations as in the case of the AST, we will also employ a stack (the *flow stack*) to keep the active path in the current flow tree.

Figure 5(a) depicts an AST with the thicker edges and nodes indicating the current traversal state. The nodes P_1 and P_2 represent packages, whereas C_1 to C_3 represent CFGs. We can clearly see, that the AST describes an Ada program built of a subprogram C_1. Immediately nested in this CFG, are the packages P_1 and P_2 and the CFG C_2. Nested in the package P_2 is the CFG C_3. The purpose of the empty nodes in the same figure, is to underline the fact, that even though a package or CFG has to be situated in the AST subtree rooted in the node of its enclosing structure, it does not, however, from a syntactical point of view, have to be an immediate child of it.

Figure 5(b) displays the current flow tree and the active path in it. Please note that the tree does not hold the CFG C_2, since the AST traversal has not reached it yet.

Figure 5(c) shows the current state of the flow stack. Each CFG on the stack also holds a reference to a node stack (see Sect. 3.3).

3.3 Node Stack

In the vast majority of the cases, an AST statement node undergoes a one-to-one transformation to a CFG node. Each time the traversal reaches a new statement, we add a new node at the end of the presently open CFG. As explained earlier, the current flow structure can be found on top of the flow stack.

We now need a mechanism to keep track of the last node that has been appended. In the standard scenario, the next node will be saved as successor of the former. In some cases, on the other hand, we would need information about several previous nodes, so keeping only one pointer proves to be insufficient. We opt again for a stack structure, but this time, with slightly different semantics.

The *node stack* usually holds only one element, maintaining the pointer to the last node that has been appended. This pointer is replaced each time a new node is added to the CFG. The stack grows only when explicit information about nodes handled in the past is necessary. This need arises in two cases:

1. When processing loops, we must not lose track of the loop head. The last node in the block must be linked back to it.
2. When processing if statements, we must not lose track of the if head. This node must be linked to all the decision branches.

Figure 6(a) depicts the standard scenario: an AST describing a subprogram P with two statements A and B. The nodes are added one after the other to the graph. In each of the two transformation steps, we see the graph and the associated node stack. The stack remains one level high in both cases.

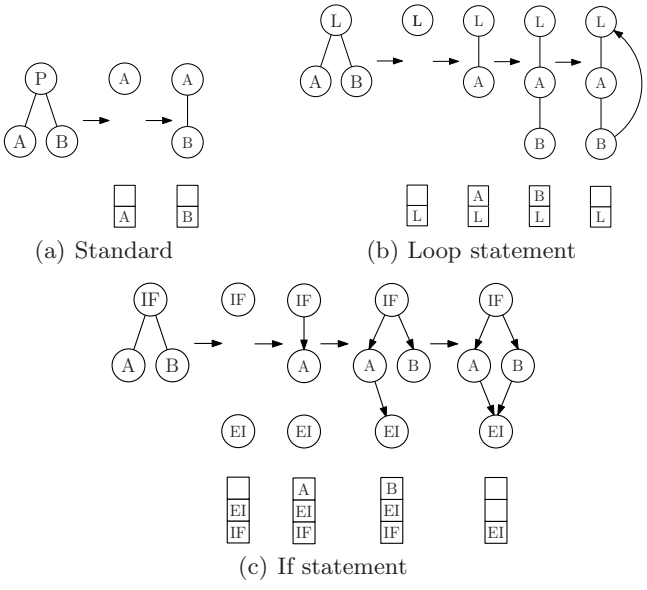

Fig. 6. CFG construction examples with node stack

Figure 6(b) shows a loop transformation: the AST is composed of a loop L holding two statements A and B. This time, the node stack holds the loop head at all times. The inner nodes are being saved on the stack one level higher, so when the loop finishes, we have the possibility to link B, in this case, back to L.

Figure 6(c) illustrates an if transformation: the AST describes a program built of an if statement with two alternatives A and B. This time, our stack must hold two extra nodes: the if head and the endif node. The former must be linked to each of the alternatives, while each branch must have a link to the latter. We use the endif node to pull all the branches back together, and thereby improve the CFG readability without adding alien control flow semantics to it. At the end of each alternative, we perform the described linking operations, and restore the if-endif stack structure. When the processing finishes we leave only the endif node behind. Semantically, this is the last node in the CFG so far.

The standard scenario works only with the stack's top and is thereby oblivious of the lower levels. This allows us to perform the special if and loop operations completely transparent to the rest of the transformation.

Please note that all three ASTs depicted in Fig. 6 have a similar configuration. Only their head nodes (P, L and IF) identify their type.

3.4 Parameter Stack

The statement is the basic control flow unit in ASIS while the CFG node is its basic counterpart in the flow world. ASIS has, however, other control flow relevant structures that cannot be represented as nodes. Such are the function calls, which are categorised as expressions. For each non-trivial call, i.e. other

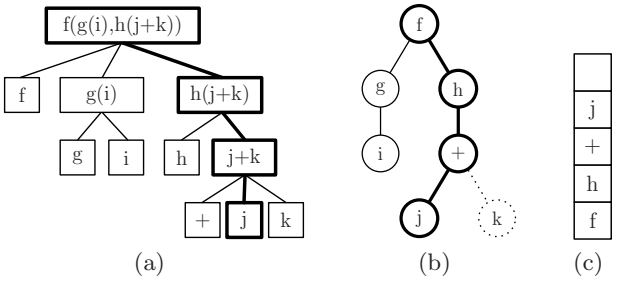

Fig. 7. AST path with associated parameter tree and stack

than an operation symbol like +, the execution leaves the original control flow temporarily and passes over to the function's body. This makes it imperative to save them in the CFG too.

A function call can also be nested inside another, as parameter of the former. These compositions can be structured into trees, with the primary called function in the root, and its parameters as children. The definition is recursive and the tree grows as the nesting hierarchy becomes deeper. Such parameter trees provide excellent means for static parameter aliasing, i.e. for determining the parameters used in function calls, regardless of the nesting depth.

When constructing parameter trees, we need to store the current path in them. Again, the best way to do so is to employ a stack, the *parameter stack*.

Figure 7 depicts a possible parameter tree construction scenario [12]. In Fig. 7(a) we can see an abstract syntax subtree describing a complex function call. The thick edges and nodes mark the current traversal state.

Figure 7(b) shows the corresponding parameter tree generated so far, with the thick edges standing for the current traversal state. The primary function `f` resides in the root. `g` and `h` are the functions used as parameters for `f`. The nesting, and therefore the tree, ends with the variables `i`, `j` and `k`. The edge and node for the variable `k` are dotted, displaying its future position. It has not been added so far, since the AST traversal has not reached it yet.

Figure 7(c) depicts the present state of the parameter stack. It is clearly visible that the current path in the tree is saved on the stack.

4 Post Transformation

4.1 Loop Refinement

After the transformation phase, `for` and `while` loops without `exit` or `return` statements are already represented correctly. However, simple loops and loops that contain `exit` or `return` statements need some refinement. For example consider the loop in Fig. 8(a). First, there should be no edge from the loop header to the loop end, since there is no condition in the header. Second, there should be an edge from the node containing the `exit` statement to the loop end. Figure 8(b) shows the refined, correct representation of this loop.

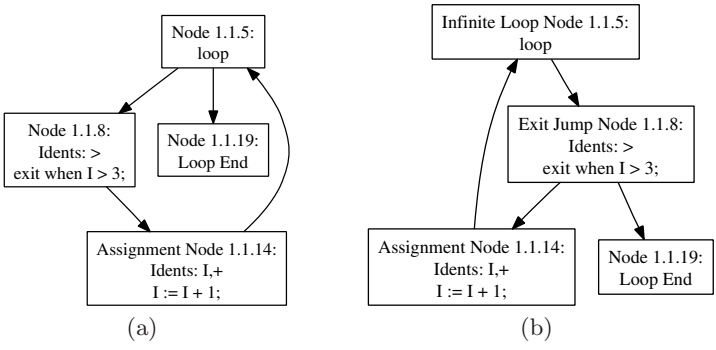

Fig. 8. Cfg2Dot output for a simple loop before (a) and after (b) loop refinement

Loop refinement is done after the main transformation phase, when the preliminary CFG has already been built. Consequently, we need to find the loops first. This has to be done since ASIS provides information only on the loop headers in a convenient way. Also, due to the considerable complexity of the traversal itself, it is easier to construct a raw version of the graph without extensive control flow semantics, and to gather this information in the post transformation phase. For that purpose, we employ Tarjan's algorithm for constructing the loop forest as it is presented by Ramalingam [13]. This algorithm needs the set of backedges of a CFG, which is why we first compute them using a simple DFS. This is possible since in a reducible CFG every retreating edge is a backedge [7].

The loop refinement is done in the `Collapse` procedure of Tarjan's algorithm. Every found loop results in a call of `Collapse` which takes the loop header and the body as parameters. After every node in the loop body is collapsed on its header, we collect the exit jump nodes for the current loop and those for outer loops in two different lists. The list with the exit jump nodes for outer loops is retained between different calls to `Collapse`.

Next, we determine the edge to the first statement after the loop which, at the current stage, is the only edge that points outside the loop. After that, we connect the exit jump nodes for the current loop and, in case the current loop has a label, also search the list with the exit jump nodes for outer loops. Note that Tarjan's algorithm always finds inner loops first, so that an exit jump node is always found before the corresponding loop. Finally, in case the current loop is a simple `loop` statement, we remove the edge from the loop header to the loop end. Return statements are handled in a similar way.

However, if the loop does not contain an `exit` statement, and therefore is an endless loop, there is the chance that some nodes, right after the loop, are not reachable any more by following only successor links. Apart from the problem of memory leakage, the fact that they still may be reached by traversing the CFG backwards, using only predecessor links, makes proper deallocation of those nodes necessary. So before the edge from loop header to loop end is removed, its target node is saved and handled later on (see Sect. 4.3).

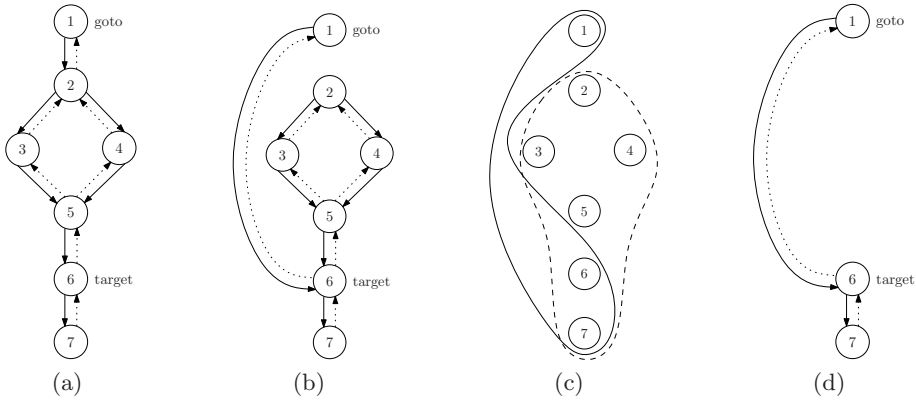

Fig. 9. Removing dangling nodes from the CFG

4.2 Connecting Gotos

During the transformation phase we only stored the target of a `goto` statement as a string but did not add an appropriate edge to the CFG. Instead, we connected the goto jump node to the node for the statement right after the `goto`. Now we have to remove that edge, and add one correctly representing the jump to the target of the `goto`.

Since the labels of a statement were also saved as strings, we basically build two lists during a DFS: a list of sources containing all found goto jump node objects and another one containing all nodes with at least one label, that is, all targets. Note that a goto statement itself may also be a target. Then we connect each node in the list of sources to the corresponding target and remove the existing, auxiliary edge.

However, as with endless loops, there may be unreachable statements following a `goto`. We will have to remove and correctly deallocate the nodes representing these statements later (see Sect. 4.3), which is why we store the target of the auxiliary edge.

4.3 Removing Dangling Nodes

As stated previously, we have to remove the subgraphs no longer reachable by only following their successor links. For example consider Fig. 9 where solid lines represent successor links and dotted lines correspond to predecessor links.

After we connect the `goto` node to its target as shown in Fig. 9(b) there now is a subgraph, rooted at Node 2, that is only reachable by following the predecessor link from Node 6 to Node 5.

We first perform a DFS on the CFG and add all reachable nodes to a set. In Fig. 9(c) this set is surrounded by a solid edge. Next, at each node that may be the root of an unreachable subgraph we start another DFS and create a second

set containing all visited nodes. In our example in Fig. 9(c) this set is surrounded by a dashed edge. Finally we subtract the set with the nodes that are reachable through normal DFS from the set containing the nodes of unreachable subgraphs and remove the resulting nodes as shown in Fig. 9(d).

5 Performance

To measure the performance of Ast2Cfg, we recorded the execution time of Cfg2Dot, which introduces only minimal overhead. The test was performed with the Linux command *time* on a machine with a single Athlon XP 2800+ processor and a gigabyte of RAM. We generated the tree files for the contents of the include directories of a typical GNAT and ASIS-for-GNAT installation. The generation of the 588 tree files with a total of 426.4 MB lasted approximately 91 seconds. Then we used Cfg2Dot to generate 5472 separate CFGs in 213 seconds.

6 Conclusions and Future Work

We developed a framework for static program analysis, which provides a CFG-based structure of Ada programs. Since currently, neither GNAT nor ASIS, fully support the Ada 2005 standard, it was not possible for us to fully implement it either. However, our work already covers most of the language specification. Ast2Cfg is therefore still in an early stage, and will be developed further.

Apart from the simple Cfg2Dot utility, there are already two projects in the field of static control flow analysis that use Ast2Cfg.

The first one aims at the detection of *busy waiting*. Busy waiting is a form of synchronisation [14] that is considered bad practice since it results in a severe overhead and even may be responsible for system failure because of race conditions [15]. Busy waiting occurs whenever a loop is only exited in case the value of a variable is changed from outside the loop. That is, the loop exit condition is not influenced from within the loop. To be able to statically detect such occurrences the algorithm proposed by Blieberger et al. [15] is implemented and extended in order to yield more accurate results.

The second project's goal is to detect *access anomalies*. They are an issue concerning multitasking environments employing non-protected shared memory areas. They occur when several concurrent execution threads access (write-write, or read-write) the same memory area without coordination. [16]

An implementation of the analysis framework proposed by Blieberger et al. [17] aims at detecting such access anomalies by means of static analysis.

References

1. Allen, F.E.: Control flow analysis. In: Proceedings of a symposium on Compiler optimization, pp. 1–19 (1970)
2. Ryder, B.G., Paull, M.C.: Elimination algorithms for data flow analysis. ACM Comput. Surv. 18(3), 277–316 (1986)

3. Fahringer, T., Scholz, B.: A unified symbolic evaluation framework for parallelizing compilers. IEEE Trans. Parallel Distrib. Syst. 11(11), 1105–1125 (2000)
4. Blieberger, J.: Data-flow frameworks for worst-case execution time analysis. Real-Time Syst. 22(3), 183–227 (2002)
5. Allen, F.E., Cocke, J.: A program data flow analysis procedure. Commun. ACM 19(3), 137 (1976)
6. Sreedhar, V.C., Gao, G.R., Lee, Y.F.: A new framework for elimination-based data flow analysis using dj graphs. ACM TOPLAS 20(2), 388–435 (1998)
7. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers. Addison-Wesley, Reading (1986)
8. International Organization for Standardization: ISO/IEC 15291:1999: Information technology — Programming languages — Ada Semantic Interface Specification (ASIS). ISO, Geneva, Switzerland (1999)
9. AdaCore: ASIS-for-GNAT User's Guide. Revision 41863 (January 2007)
10. Sedgewick, R.: Algorithms, 2nd edn. Addison-Wesley, Reading (1988)
11. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Software — Practice and Experience 30(11), 1203–1233 (2000)
12. Fechete, R., Kienesberger, G.: Generating control flow graphs for Ada programs. Technical Report 183/1-139, Institute for Computer-Aided Automation, TU Vienna, Treitlstr. 1-3, A-1040 Vienna, Austria (September 2007)
13. Ramalingam, G.: Identifying loops in almost linear time. ACM Trans. Program. Lang. Syst. 21(2), 175–188 (1999)
14. Andrews, G.R.: Concurrent programming: principles and practice. Benjamin-Cummings Publishing Co. Inc., Redwood City (1991)
15. Blieberger, J., Burgstaller, B., Scholz, B.: Busy wait analysis. In: Reliable Software Technologies - Ada-Europe, pp. 142–152 (2003)
16. Schonberg, D.: On-the-fly detection of access anomalies. In: PLDI 1989: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, pp. 285–297. ACM Press, New York (1989)
17. Burgstaller, B., Blieberger, J., Mittermayr, R.: Static Detection of Access Anomalies in Ada95. In: Pinho, L.M., González Harbour, M. (eds.) Ada-Europe 2006. LNCS, vol. 4006, pp. 40–55. Springer, Heidelberg (2006)