University of Illinois at Chicago - Politecnico di Milano
CS Master Program
CS569 - High Performance Processors and Systems course

Student Project: System Dependency Graphs in Earendil

June 22, 2004
student: Matteo Giani - PdM matricola 667487
instructor : prof. Donatella Sciuto
tutor : ing. Marco D. Santambrogio

Contents

List of Figures

1 Introduction

This project focuses on the analysis of a language-independent program representation, the *System Dependency Graph (SDG)*, and of some of the algorithms involved in its construction, especially those regarding computation of control dependencies. This kind of intermediate representation is used by *Astinus*, one of the modules of the *Earendil* methodology for dynamic reconfiguration, as presented in [1].

1.1 Earendil : overall view

As is shown in figure 1, the Earendil methodology is composed of different modules which work in series: the output of one stage is the input to the next one.

The Earendil methodology starts by taking as input a SystemC specification, which is first analyzed in order to partition it into modules and parallelize the execution as much as possible. This step is carried out by the first stage of the methodology, *Astinus*. The original SystemC code has now been partitioned into modules, and the output of Astinus is an intermediate representation called *Thread Dependency Graph*. The following stage in the Earendil methodology is *Salomone*, which starts from the Thread Dependency Graph in order to solve the static scheduling problem for the modules identified by Astinus. Once Salomone has completed its job, it produces SystemC modules specifying the behavior for the overall system that, once compiled to `VHDL` and synthesized, will be loaded into the Caronte architecture, which implements the actual Dynamic Reconfiguration.

1.2 Astinus

As stated above, the Astinus module starts from the original code and produces as output a Thread Dependency Graph. It is internally further structured in different steps:

- parse the original SystemC code to obtain the Abstract Syntax Tree (AST)
- obtain a System Dependency Graph (SDG) in order to expose parallelizable parts of the code
- produce a Thread Dependency Graph (TDG) by merging parallelizable nodes of the SDG into basic blocks.

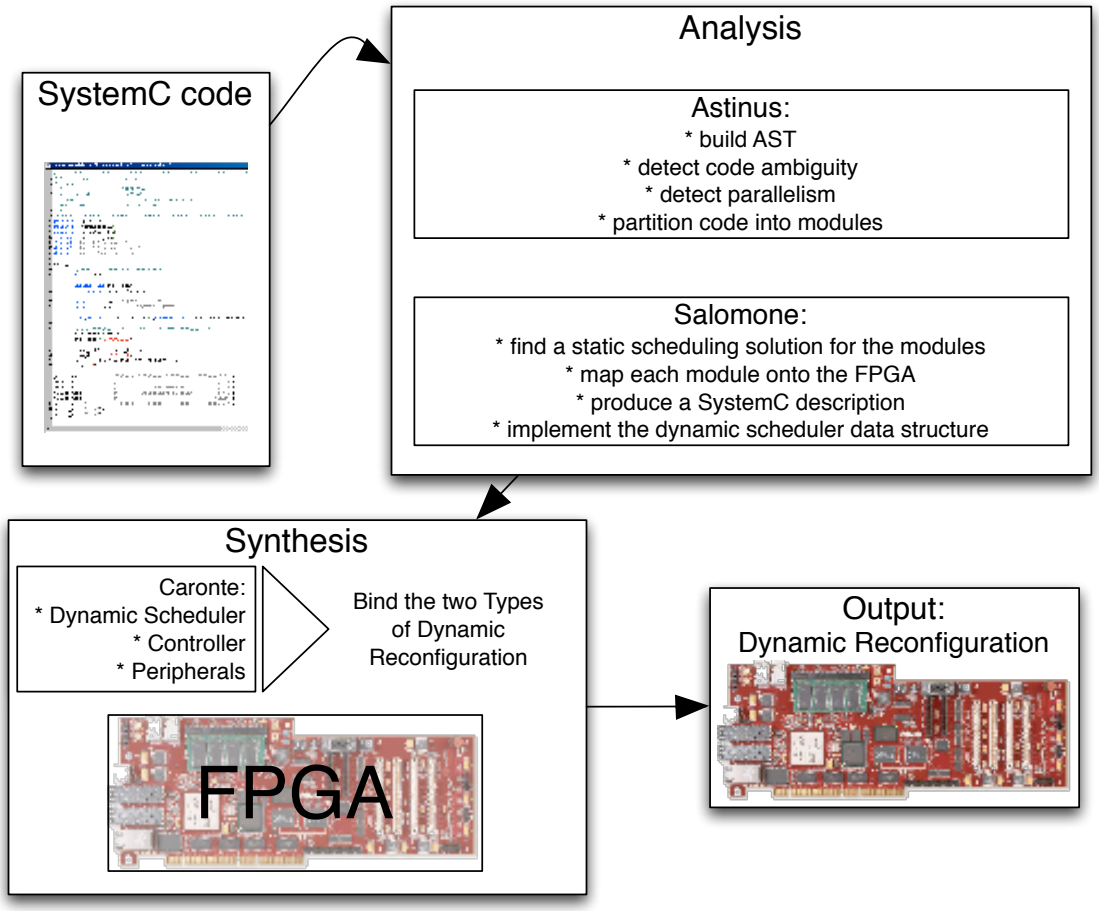

Figure 1: Earendil flow

The SDG representation was chosen ([1]) because it stresses all, and only, the essential dependencies between statements in the source. Most importantly, while it does not explicitly contain control flow information, the way control dependencies are summarized in a SDG is very concise and exposes parallelizable blocks quite clearly.

This project went through the following phases:

- analysis of literature in order to obtain a clear definition of the structure of a SDG
- formalisation of some of the algorithms involved in the construction of a SDG

Moreover, an algorithm for computing the postdominator tree starting from a Control-Flow Graph (see 3.1.1) was implemented in `C++` using the `boost` graph library ([2]).

2 Definitions

This section aims at showing the features of a SDG. Its contents illustrate the results of the work carried out in collaboration with Marco Magnone, who is also exploiting the SDG representation for his Laurea thesis work at Politecnico. One first observation is that a *System Dependency Graph* (SDG) represents a system made of different procedures. The basic structure from which a SDG is built is a *Program* (or *Procedure*) *Dependency Graph*, which is the representation of a single procedure. Let us then start by analysing this latter kind of representation.

2.1 Program Dependency Graph

It is worth noting that many different variations over the original definition of PDG have been developed, especially considering that several different uses have been found for such a representation. Take as examples:

- applications of the PDG to the *slicing* problem [3, 4]
- extensions of the PDG for supporting object-oriented programming languages [5]
- uses of the PDG to detect similar code [6]
- the representation of the original control flow of the program in addition to the usual features of the PDG [7]

Since all these different interpretations of the PDG introduce *ad-hoc* extensions and modifications of its structure it was decided, in order to stick to a clear and commonly used definition, to first adopt the original definition found in [8].

2.1.1 The original PDG

The PDG [8] represents a program (or, better, a procedure) as a **directed graph** in which the following kinds of **nodes** are present:

- **statement** nodes, which represent single executable statements in the code;
- **predicate** nodes, which represent control flow conditions, such as loop conditions or if-statement conditions;

- an **entry** node, which represents the external condition that causes the procedure to start running. As we will see, when taking into account multiprocedure systems, this node will be uses to connect procedures by means of call edges;
- **region** nodes, used to "summarize" common control dependency predecessors between statements. More information on this is given in 3.1.2.

Before considering the edges that we can find in a PDG, we have to properly understand the meaning of control dependency, since this is the most novel feature of a PDG. Let us consider at first one intuitive definition: one node B is control dependent on A if A can control whether or not B will be executed. As an example, we will have control dependencies between nodes representing statements in the "if" branch of a conditional expression and the node representing the predicate that controls the "if" construct. A formal definition [8] is as follows:

A node B is control dependent on A if, and only if,

- A is not post-dominated by B in the CFG, and
- there exists a directed path from A to B in the CFG such that every node other than A on the path is postdominated by B.

For a definition of postdominance see 3.1.1.

The following kinds of **edges** can be found in a PDG:

- **control dependency** edges: they exist between nodes that are control dependent on one another according to the definition above, and between region nodes inserted afterwards (3.1.2). They are further characterizable as follows:
  - labeled control dependency edges : they are outgoing from a predicate node, and the label (T/F) allows us to distinguish which one is in effect depending on the result of the predicate evaluation
  - unlabeled control dependency edges : they typically originate when representing control dependencies in which the origin is a region node.

- **data dependency** edges: they are the representation in a PDG of the corresponding edges in a "traditional" Data Dependency Graph, and can be further subdivided into four main types:
  - data-flow dependencies (or true dependencies): they are another way to see `RAW` dependencies: they exist between a definition of a data item and a use of it if no other definition occurs in the middle
  - output dependencies: they represent the same concept as `WAW` dependencies, so they exist between two consecutive definitions of the same data item, with no intervening definition between the two
  - anti-dependencies : anoter way of calling `WAR` dependencies, they exist between one use of a variable and the first subsequent definition of the same data item.
  - def-order dependencies: this last kind of data dependency is a specialisation of the output dependency edge, that exists between different definitions of the same data item that could both reach one use (see [9] for an example of its rationale).

Moreover, a distinction must be made between data dependencies that arise from loop constructs, which are called *loop-carried* data dependencies, from the ones that do not, which are called *loop independent*, in order to correctly distinguish situations like these two [9]:

```
- FRAGMENT 1
x:=0
while P do
    y:=x
    if Q then x:=1 fi
od

- FRAGMENT 2
x:=0
while P do
    if Q then x:=1 fi
    y:=x
od
```

Without treating loop-carried dependencies in a separate way, the two code fragments would have the exact same PDG representation.

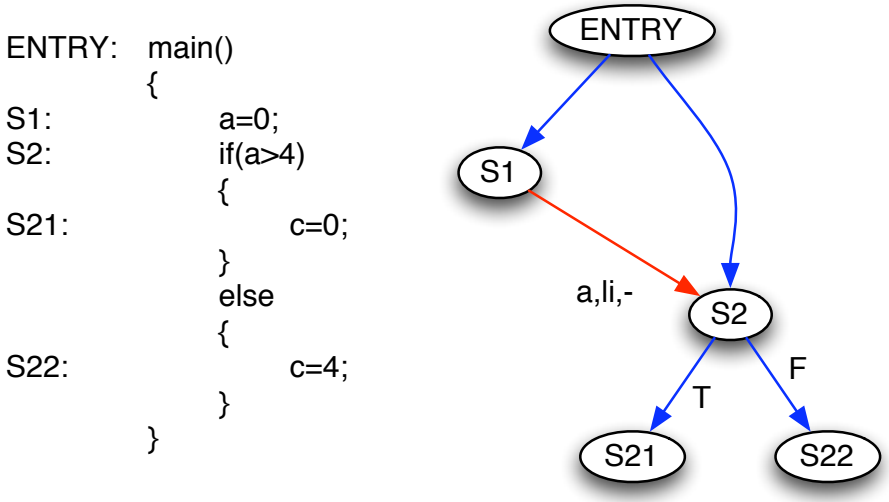

Figure 2: Example of PDG for a simple code fragment

To summarize, it was decided that, in our example graphs, data dependencies will be represented as edges labeled with a triple $item, loop, type$ (see e.g. figure 2), in which:

- *item* stands for the name of the data item which the dependency takes into account,
- *loop* is either `lc`, for loop-carried, or `li`, for loop-independent dependencies,
- *type* is
  - "`-`" for true data dependencies,
  - "`do`" for def-order dependencies,
  - "`o`" for output dependencies,
  - "`a`" for anti-dependencies.

Figure 2 also shows our graphical convention for representing dependency edges: control dependency edges are drawn in blue, while data dependency edges are drawn in red. This convention will be adopted throughout this section.

2.1.2 Fine-Grained Program Dependency Graph

In the "traditional" PDG defined above each statement in the program maps to a single node. There might be situations in which a finer degree of detail is desirable. The *Fine-Grained PDG* allows us to achieve a more detailed representation by introducing more specialisation in the kind of nodes used. The structure obtained is closer in terms of detail to that of an Abstract Syntax Tree, and each node is now labeled using different fields:

- a **label** that relates each node to its corresponding statement in the original source
- a **class/operator** field, which specifies the kind of node we are dealing with. As we will see in some of the examples, common values are:
  - **entry**, if the node is the entry node of a procedure, with a meaning like the one exposed for "traditional" PDGs
  - **binary**, if the node is a binary expression, i.e. one with a left-hand side and a right-hand side, such as a comparison
  - **assign**, if the node represents an assignment operation
  - **constant**, if the node expresses a constant value
  - **statement** in order to represent uses of data items
  - **reference** in order to represent definitions of data items
- a **value** field, for some of the node types it is used to specify the actual contents of the node, such as the numerical value for constant nodes, the actual operator for binary expressions or the name of the data item for reference nodes.

Region nodes are kept exactly the same, since they do not represent any actual program feature but are introduced in order to better represent control dependencies.

The Fine-Grained representation must also introduce new kinds of edges in order to take into account *intra-statement* dependencies, i.e. the ones that occur between nodes belonging to a single statement in the program. There are substantially three new kinds of edges, which will be represented in our examples as dashed lines to distinguish them from the ones defined in the original PDG:

- **immediate control dependency** edges: they exist to represent the need of evaluating their target node before executing the operation

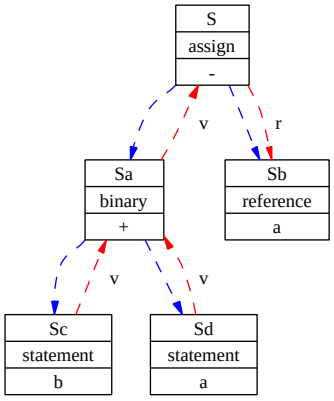

Figure 3: Fine-Grained PDG representation of an assignment statement

represented by the source node. They will be present, e.g., between the **binary** node representing a comparison and the nodes representing its right and left-hand side expressions, which will need to be evaluated before computing the outcome of a comparison;

- **value dependency edges** are a new kind of data dependency edge that takes into account the data flow occurring inside a given statement: they mean that a value computed in the start node is needed to compute the result of the target node. They will be represented as dashed red lines labeled "v";
- **reference dependency edges** express a concept similar to the one of value dependency edges, but they are differentiated in order to represent the fact that the value computed by the start node is going to be saved in a data item represented by the target node. Their target will typically be a **reference** node. They will be represented as dashed red lines labeled "r".

For an example of some of these features, see figure 3, which represents a single assignment statement, `a = b + c`. In a "traditional" PDG it would have been represented by a single node. Nodes have been labeled appending letters in alphabetical order to nodes generating from one single statement, in this case `S` followed by lowercase letters.

Moreover, the same example seen in figure 2 can be seen represented in a Fine-Grained PDG in figure 2.1.2. For more examples of different control structures represented as PDGs, the reader can refer to the related chapter in Marco Magnone's thesis (work in progress at the time of writing).

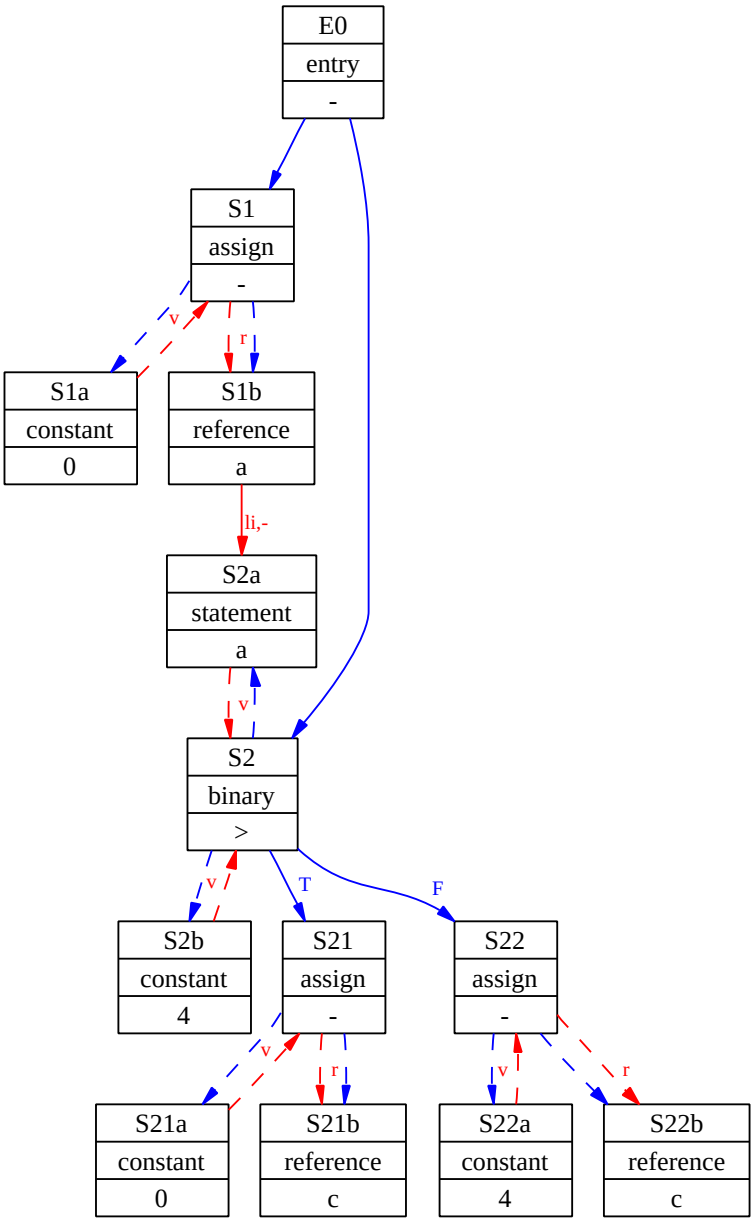

Figure 4: Fine-Grained PDG representation of the code fragment in figure 2

2.2 System Dependency Graph

PDGs represent single procedures. In order to represent complex systems, we must consider multiple PDGs and add information about procedure calls. The System Dependency Graph (SDG) representation allows us to represent this kind of program feature by introducing additional node and edge specialisations. Parameter passing is modeled upon passing by reference, i.e. both input and output parameters are represented. It is then necessary to create, for each procedure, two sets of nodes:

- **formal-in** nodes that represent formal parameters used as input values, and
- **formal-out** nodes that represent formal parameters that are potentially modified by the procedure.

Every callsite for the procedure (i.e. statement from which the procedure is called) will have to match the formal parameter nodes with a set of actual parameter nodes, called **actual-in** and **actual-out**. A further refinement of the System Dependency Graphs is in [3] that allows to represent potentially non-returning procedure calls.

The SDG also has specialised edges that allow us to connect callsites and procedure entry points, both from control dependency and data flow points of view:

- **call edges** which connect call nodes to entry nodes of the called procedure. They represent control dependencies that arise due to the call operation, we will label such edges with a "c";
- **parameter passing** edges, that take into account the data flow dependencies that arise due to parameter passing. They are subdivided into:
  - **parameter-in** edges, that connect actual-in nodes to the corresponding formal-in nodes in the procedure's PDG, labeled as "p-IN", and
  - **parameter-out** edges, that connect formal-out nodes to the corresponding actual-out nodes in the caller's PDG, labeled as "p-OUT".

The idea of how different PDGs are interconnected in order to take into account procedure call is presented in figure 5. It is worth noting that in the figure the procedure body is left represented as a generic single node,

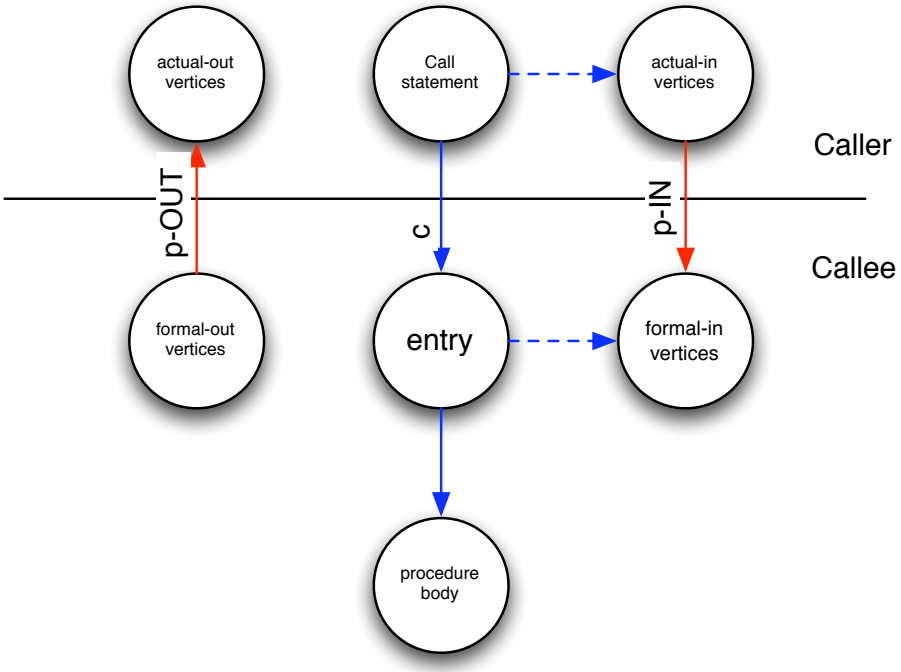

Figure 5: Summary of the representation of a procedure call in a SDG

and this makes it a bit unclear how the code of the procedure is connected to output and input parameters. The basic idea is that parameters can be treated as variables, with input parameters being defined at the beginning of the procedure and output parameters having a use of the variable at the end of the procedure body. This way, data operations inside the procedure can be related to input and output parameters just like it happens for any other variable. The parameter in and out edges also expose data dependencies that could arise between actual parameters because of the procedure's behaviour.

3 Algorithms involved SDG construction

As has been outlined, a SDG is a collection of PDGs connected by means of call edges and parameter passing edges. Thus, we can build a SDG incrementally by first constructing PDGs for the single procedures involved in our system, completing them with parameter passing nodes, and then linking them together with call and parameter passing edges. Moreover, if needed, a PDG could be turned into a fine-grained PDG by expanding each node into the corresponding set of fine-grained vertices, properly linked by the needed intra-statement dependency edges.

The first step in building a System Dependency Graph is to produce Program Dependency Graphs for the individual procedures involved in the overall system. Each Program (or *Procedure*) Dependency Graph contains, as has already been outlined, information about:

- data dependencies, and
- control dependencies.

The entire PDG can be seen as the union of two different subgraphs, each representing one kind of dependency. The subgraph representing data dependencies is actually a "traditional" Data Dependency Graph (with some additional detail, as exposed in 2.1.1). The problem of building the Data Dependency Graph is actually a recurrent problem in program analysis, and was not a central subject of this project. The rest of this section will, then, focus on construction of the Control Dependency Subgraph.

Most of the work that has been done around PDG construction revolves around concepts that were already present in the original definition [8]. An algorithm for computation of control dependencies was also outlined therein, and since it is, in several aspects, directly based on the definition it seemed like a good starting point in order to get a thorough understanding of the concepts involved in PDG construction.

3.1 The original approach to PDG construction

The paper in which Program Dependency Graphs were originally defined [8] contained the description of an algorithm for computing exact control dependencies, which took as input the Control-Flow Graph of the program. From there, the first step is to compute the postdominance relation between nodes in the Control-Flow Graph, in form of the postdominator tree. This is why postdominance is an important subject for the analysis of Program Dependency Graphs.

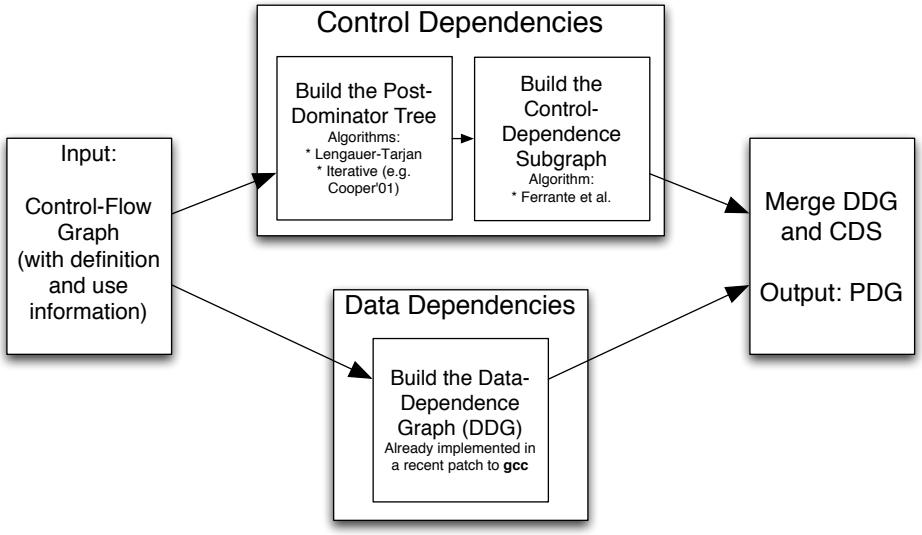

Figure 6: Overview of a possible workflow for building a PDG

Once the postdominator tree had been built, the algorithm used information both from this intermediate graph and from the original Control-Flow Graph in order to first compute exact control dependencies, and then simplify their representation by introducing *Region Nodes* in the graph. The rationale of region nodes is that every region node represents a *set of control dependence predecessors*, so that two statements, if they share a subset of control dependence predecessors, are made dependent on the same region node instead of having separate edges coming from all the predecessors in the subset.

This is better understood by looking at the pseudocode that summarizes the steps carried out by the algorithm (3.1.2). Let us now take into consideration the problem of computing the postdominator tree, which as has been stressed is the first step towards computing control dependencies.

3.1.1 Postdominance

Postdominance is a relation between nodes in a directed graph with one distinguished (*end*) node. Let us start from the definition of *dominance*, which is basically the opposite of postdominance, and is the relation that was originally defined.

Consider a directed graph with a distinguished node, call it `start`. A node `V` in the graph is *dominated* by another node `W` if, and only if, every

directed path from `start` to `V` contains `W`.

In other words, a node is postdominated by another one if every possible path we have to reach it from the start node passes through the latter node.

Postdominance is, dually, the relation that is defined as follows: in a directed graph with a distinguished node `end`, a node `V` is *postdominated* by another node `W` if, and only if, every directed path from `V` to `end` contains `W`.

This last definition appears clearly related to the definition of control dependency in a PDG: if all possible control paths that can originate from a single statement `V` contain another statement `W` then there can be no control dependency from `V` to `W`, since `V` cannot affect whether or not `W` will be executed. This is also why the algorithm to compute control dependencies initially considers the postdominance relation in order to discard such pairs of nodes in the CFG.

The postdominance relation, as can be shown, is an irreflexive, asymmetric, transitive relation. As such, it is most naturally represented as a tree, the *postdominator tree*, which is the representation of this relation that is used by the algorithm shown in 3.1.2. It is worth noting that most of the algorithms that have been defined to compute the relation produce directly the tree representation, from which of course the transitive closure is extractable trivially in linear time.

The problem of (post) dominance is actually quite a common one in many approaches to program analysis and optimization. A good summary of the various algorithms proposed over time to solve it can be found in [10]. The early algorithms adopted an "iterative" approach, achieving (roughly speaking) quadratic time complexities in the number of edges in the CFG. An important turning point in the development of algorithms for solving such a problem is [11]: it significantly lowered the time bound of the algorithm, achieving near-linear complexity. Moreover, many of the later algorithms were based on modifications of the one proposed in [11].

However these algorithms, while displaying better asymptotical time complexity, are significantly more complex in terms of implementation effort and more importantly, as outlined in [10], they need some significant time to setup their data structures, so that this overhead does not justify the adoption of the more refined algorithms with respect to the simpler iterative ones in most practical cases.

Therefore, it was chosen to implement, as a starting point, the algorithm proposed in [10], which is basically an iterative algorithm adopting some refinements in the data structures used. At a later stage it could be substituted with a more efficient one, like the one in [11]. A pseudocode representation of the algorithm follows:

```
BEGIN dominators_cooper
    (do a depth-first search on the CFG starting from the start node)
    (number the nodes in reverse postorder)

    FOREACH(node b in the CFG)
    {
        doms[b] := undefined
    }

    doms[start_node] := start_node;

    Changed = true;

    WHILE(Changed)
    {
        Changed = false;
        FOREACH(node b in the CFG, in reverse postorder, except start_node)
        {
            // pick one, the choice should not matter
            new_idom := (first processed predecessor of b in the CFG)
            FOREACH(predecessor p of b in the cfg other than the one chosen above)
            {
                IF doms[p] != undefined
                THEN
                {
                    new_idom := intersect(p,new_idom)
                }
            }
            IF doms[b] != new_idom
            THEN
            {
                doms[b] := new_idom
                Changed := true
            }
        }
    }
END dominators_cooper

FUNCTION intersect(b1,b2) RETURNS node
BEGIN
    finger1 := b1
```

```
    finger2 := b2
    WHILE( finger1 != finger2 )
    {
        WHILE( finger1 < finger2 )
            finger1 := doms[finger1]
        WHILE( finger2 < finger1 )
            finger2 := doms[finger2]
    }
    return finger1
END intersect
```

The algorithm computes the dominance relation by means of the immediate dominators for each node. The immediate dominator of a node is actually its parent in the dominance tree. Of course, the algorithm can be trivially adapted to computing postdominance instead of dominance by just feeding it the reversed control-flow graph. As can be easily seen from the pseudocode above, the algorithm iterates to a fixed point. Its temporal complexity is $O(N + E * D)$ per each iteration, where N is the number of nodes in the CFG, E the number of edges, and D the cardinality of the biggest dominators set. In [10] it is shown that the algorithm terminates in $d(G) + 3$ iterations, where $d(G)$ is the loop connectedness of the CFG. Loop connectedness is a structural property of a directed graph G and a depth-first spanning tree (DFST) of G, such as the one constructed as a first step in the above outlined algorithm. The DFST partitions the edges in G into three classes: edges used in the DFST, called tree edges or forward edges; edges that run from a node back to an ancestor in the DFST, called back edges; and edges that run between nodes in disjoint subtrees of the DFST, called cross edges. With this partitioning, $d(G)$ is the maximum number of back edges that can occur on any acyclic path through G. In practical cases, [10] suggests that $d(G)$ is bounded to low values, typically lower than 3.

Figures 7 and 8 show a Control-Flow Graph (taken from [8]) and its corresponding postdominator tree, computed by the implementation of the postdominance algorithm that was developed for this project.

3.1.2 Control Dependency Computation

Now that we have outlined the problem of postdominance and exposed an algorithm to solve it, we can look into the actual algorithm for computation of control dependencies presented in [8]. The algorithm consists of two phases:

- computation of exact control dependencies,

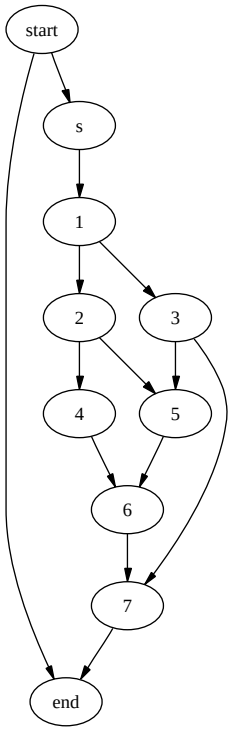

Figure 7: Example: Control Flow Graph with ”start” and ”end” nodes

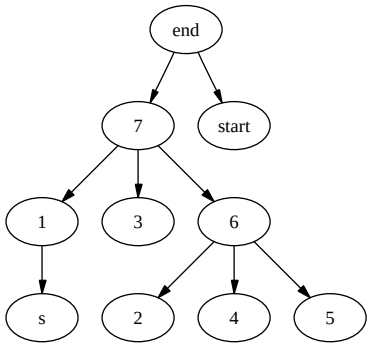

Figure 8: Postdominator tree for the CFG of figure 7

- insertion of region nodes in order to factorize common control dependencies between statements.

Both phases rely on the information provided by the postdominator tree. The second phase, in which the algorithm tries to factor common control dependencies between nodes by substituting them with dependencies on region nodes that summarize sets of control dependence predecessors, is further structured in two steps:

- insert region nodes "from the bottom", which is summarize common control dependence predecessors
- insert region nodes "from the top", which is create a region node each time a node has more than outgoing control dependence edge with a given label

This yields a complete factoring of common control dependencies, but is quite heavy both from the point of view of computational complexity and from the point of view of implementation difficulty. It is worth noting that, in later works, this phase has been somewhat simplified [12] since it is not clear whether there is need for such a complete factorization of control dependencies.

A commented pseudocode description of the algorithm follows:

```
1.1 - computation of exact control dependencies
######## START 1.1 #

    compute the postdominator tree
     (see the appropriate section for further info)

    FOREACH( edge (A,B) in the CFG )
    {
        if (A is not an ancestor of B in the postdominator tree)
        {
               current_node := B
               WHILE(current_node != parent(A))
               {
                    mark current_node as control dependent from A
                     with the same label as the (A,B) edge in the CFG
                    current_node := parent(current_node)
               }
```

```
        }
    }
######## END 1.1 #


- 1.2 - insertion of region nodes to factorize common control dependencies

####### START 1.2 #

    FOREACH ( node N in the post-dominator tree, in postorder traversal )
    {
        CD := set_of_control_dependence_predecessors(N)
        if ( there is no region node associated to set CD )
            // i.e. no other node has already been visited with
            // the same set CD of control dep. predecessors
            // compare labels as well
        {
            // the CD parameter summarizes the operation
            // of updating the data structures
            // with the new association between the set of control
            // dependence predecessors and the newly created region node.
            R := new_region_node(CD)
            create_edges ( from every node in CD to R )
        }
        else
        {
            // the data structure (hash table as suggested in Ferrante et al.)
            // is looked up for an association from set CD to a region node.
            R := region_node(CD)
        }
        // we now have a region node accounting for the
        // control dependencies of the node we are visiting.
        make R the only control dependence predecessor of N

        FOREACH( immediate child C of N in the post-dominator tree )
        {

            // the set used to compute the intersection is the original
            // set of CD predecessors, not the one taking into account
            // potential inserted region nodes. The CD sets for the children
            // have already been computed since we are traversing the tree
```

```
        // in postorder.
        INT := intersection (CD,set_of_control_dependence_predecessors(C))

        if( INT equals CD ) // CD(N) is contained in CD(child)
        {
            delete_edges ( from nodes in INT to region_node(child) )
            create_edge ( from R to region_node(child) with no label )
        }

        if ( set_of_control_dependence_predecessors(C) equals INT )
         // i.e. CD(child) is contained in CD(N)
            // Ferrante et al. says "if every control dependence
            // of the child is contained in INT"
            // but INT cannot have any member that is not in CD(C).
        {
            delete_edges ( from nodes in INT to R )
                // notice that the following statement relies on
                // the fact that the child statement has a single
                // control dependence predecessor, i.e. a single
                // region node has already been created for it, which
                // is guaranteed thanks to the postorder traversal.
            create_edge ( from region_node(child) to R with no label )
        }
    }
}


// we should now make sure that no predicate node in
// the Control Dependence Subgraph has more than one successor
// with the same truth value.
FOREACH ( predicate node P in the control dependence subgraph )
{
    if ( there is more than one control dependence successor with label L )
    {
        succ := set_of_control_dependence_successors( node P, label L )
        R := new_region_node({P})
        delete_edges ( from P to succ with label L )
        create_edge ( from P to R with label L )
        create_edges ( from R to nodes in succ with null label )
    }
}
```

```
####### END 1.2 #
```

3.2 A different approach starting from the AST

A different approach is to construct the PDG starting directly from the Abstract Syntax Tree of the program [7]. Since the AST is built incrementally as the source code is parsed, this kind of approach allows to avoid computing intermediate auxiliary graphs such as the CFG and the PostDominator tree.

This algorithm is rather straightforward and definitely more intuitive than the original one proposed in [8]. The drawback is that it is highly language-dependent since extracting information from the AST implies knowing in advance all the control structures present in the original language, while the more traditional approach exploits information from the CFG, which is an abstract representation, and as such is more general.

References

[1] M.D.Santambrogio, "A methodology for dynamic reconfigurability in embedded system design," 2004, m.Sc. Thesis.

[2] "Boost c++ libraries." [Online]. Available: www.boost.org

[3] S. Sinha, M. J. Harrold, and G. Rothermel, "System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow," in *Proceedings of the 21st international conference on Software engineering*. IEEE Computer Society Press, 1999, pp. 432–441.

[4] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 1, pp. 26–60, 1990.

[5] D. Liang and M. J. Harrold, "Slicing objects using system dependence graphs," in *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 1998, p. 358.

[6] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings. Eighth Working Conference on Reverse Engineering, 2001.*, 2001, pp. 301–309.

[7] M. J. Harrold, B. Malloy, and G. Rothermel, "Efficient construction of program dependence graphs," in *Proceedings of the 1993 international symposium on Software testing and analysis*. ACM Press, 1993, pp. 160–170.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[9] S. Horwitz, J. Prins, and T. Reps, "On the adequacy of program dependence graphs for representing programs," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 1988, pp. 146–157.

[10] K. D. Cooper, T. J. Harvey, and K. Kennedy, "A simple, fast dominance algorithm," 2001. [Online]. Available: citeseer.ist.psu.edu/cooper01simple.html

[11] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 1, pp. 121–141, 1979.

[12] K. J. Ottenstein and S. J. Ellcey, "Experience compiling fortran to program dependence graphs," *Softw. Pract. Exper.*, vol. 22, no. 1, pp. 41–62, 1992.