A Non-Deterministic Approach to Restructuring Flow Graphs

Toni A. Bünter

Technical Report CUCS-019-93
COLUMBIA UNIVERSITY

Abstract

The history of programming is filled with works about the properties of program flow graphs. There are many approaches to defining the quality of such graphs, and to improving a given flow graph by restructuring the underlying source code. We present here a new, twofold approach to restructuring the control flow of arbitrary source code. The first part of the method is a classical deterministic algorithm; the second part is non-deterministic and involves user interaction. The method is based on node splitting, enabling it to satisfy the definition of the extended Nassi-Shneiderman diagrams.

Contents

1 Introduction 1

2 Related Work and History 1

3 Structure and Transformation of Flow Graphs 2

4 Incomplete D-Structure Transformation 3

5 Unfolding : A Non-Deterministic Restructuring 5

6 An Application of Unfolding 6

7 Conclusion and Future Work 7

8 Figures 10

1 Introduction

The quality of the control flow graphs of procedural programming languages has concerned programmers and theorists for a long time. Investigation into flow graph structures has led to new programming paradigms. The *structured programming* appoach [7], dealing with D-structures which are well known to Pascal programmers, is considered to be of high quality. Therefore, countless attempts have been made to find theorems and algorithms for updating the flow graphs of given source code to structures similar to D-structures. It turned out, however, that most of these algorithms involve a trade-off between the readability of the restructured source code and its closeness to the desired D-stucture.

We present here a new compromised method that is concerned with both the readability and the quality of the produced structure. The method is twofold. A deterministic, preliminary transformation prepares the flow graph for the second stage of the method. The second stage entails presenting some basic operations which allow the programmer to decide how far he wants to unfold a nested structure. Such subjective unfolding is quite useful and may even be necessary, especially for software enhancement.

The method employs a process called *node splitting* [17]. Node splitting consists of copying and inserting single sequential blocks. There is no change of predicates and no addition of boolean variables. The continuity of the method preserves the context of the original program blocks. This process both decreases unnecessery complexity, and improves maintainability and readability.

As a contribution to graphic illustrations of source code, we present an extension of the NS diagram, the *extended NS diagram*. By applying the first stage of our restructuring method, the source code meets the requirements for the eNS diagram. Examples are shown in section four, five and six.

2 Related Work and History

The first results about flow graphs, and particulary about control flow graphs, go back to the 1970's. Numerous works were published, which can be roughly classified into three cathegories, according to topic:

- classification of flow graph structures,
- complexity and metrics definitions about flow graphs,
- transformations of flow graphs.

The results of these earlier works were often controversial. A broad discussion arose about the harmfulness of the *goto* statement [8], [14]. One group supports a liberal, but responsible programmer, who produces understandable code, while the opposing view advocates the use of structures that also cover design and specification areas (as in the structured programming approach). One of the most popular outcomes of this debate is the programming language, Pascal [12]. Pascal integrates the so called *block structure* as its basic component of control flow [12].

For better or for worse, huge amounts of source code have already been and still are being written with goto statements. In spite of the proclaimed attitude of disciplined programmers, much of the source code

produced is hardly understandable because of the complex control structure. This has encouraged research about transformation of source code. The results of various approaches show the trade-off between the quality of the achieved structures, and the often harmful variety of the changes which have to be made. The two extreme postitions of control flow graphs concerned transformations are (1) the detection of block structure-like patterns and their replacement through loop structures such as `while`, and braching structures such as `if-then-else` [10], and (2) the transformation of arbitrary source code into block structures [4]. The latter approach introduces additional boolean variables that are necessary to uphold the semantics.

Despite the large quantity of varied results, relatively few software maintenance tools are currently in use. One of the main reasons for this is the aforementioned trade-off. Strong restructuring often makes the semantic structure as data-flow or local meaning of variables more confusing than before, while "soft" restructuring, such as [10], is often merely cosmetic. Our approach circumvents this disadvantage by defining two stages of transformation: (1) a deterministic transformation and (2) a user-driven, non-deterministic transformation, which involves the programmer's responsibility.

3 Structure and Transformation of Flow Graphs

The structure of a flow graph can be expressed by a *directed, labeled graph* where the *nodes* represent sequential program blocks, ending, in the event of branching, with a conditional predicate. The branching *edges* are labeled by predicate values which direct the control flow. The graph is fully connected. One node is the *start node*. From the start node, every node is accessible along the edges.

Marcotty and Ledgard give an overview of the structure and transformation of control flow graphs [15]. Below are a few definitons which are important for this paper.
The most restricted and probably most maintainable structure is the *D-structure* (in [15] originally called D-structure, 'D' in honor of Dijkstra's work as [9]). Programming that exclusively applies D-structures is called *structured programming*. A D-structure consists of a sequence of `if-then-else`, `case`, `while`, `repeat-until` and `for` with a single entry and a single exit point. These can be parsed by a context-free graph grammer.

Other definitions of control flow graphs only restrict the loop structure. The *repeat-exit-cycle structure* defines the loop as a single-entry/single-exit structure, with the freedom of exiting and continuing the loop at various places (figure 8).

In our own approach the focus will be on the *single-entry structure*. The single-entry structure allows exiting loops to different nodes (figure 8).

In a generic flow graph, a loop is a *multiple-entry/multiple-exit structure*. This makes it possible to enter a loop at various nodes. According to [2], such a structure is considered harmful.

Transformation of Flow Graphs and Source Code

Our main concerns in flow graph transformation are control flow graphs with underlying source code. The atomic operation which we apply on the flow graph in our apporach is node splitting [17]. In [5] we developed a mathematical model for node splitting using the cartesian product and supersets.

The fundamental restriction of program transformation is the functional equivalence, which has to be preserved. Applied to the node splitting operations , the following constraint preserves the functionality.

- *For each node and each label the successor node must be either the same as it was before the transformation, or an exact copy.*

We call a transformation that fullfills this condition a *sequential block preserving* transformation.

4 Incomplete D-Structure Transformation

In this section we show the deterministic part of our appoach; we will define the term *incomplete D-structure*. Next to the theoretical transformation theorems, we introduce the concept of the extended Nassi-Shneiderman diagrams, which are adaptable for the incomplete D-structure.

The first basic theorem is about the transformation to single-entry flow graphs. This is the main theorem belonging to the deterministic part of the method. The following proof shows the basic definitions and operations.

Theorem 1. Given a generic flow graph f, it is possible to achieve a single-entry flow graph by applying a sequential block preserving transformation.

Instead of giving the full detailed mathematical proof, we explain the proof idea which shows also the algorithm that can be deduced (details in [5]).

Proof.

(1) Let us partition the set of nodes of a given, generic flow graph into *maximum loops*. A maximum loop is a subset of all nodes which forms a loop. None of these nodes is a member of another loop. If there is only one maximum loop or all maximum loops are already single-entry loops, we can go to (4).

(2) We focus now on one freely choosen maximum loop. We determine one entry node of the loop as the main entry h. Because of (1) there is another loop entry e different from h. Let E be the set of nodes not in L that lead directly to e. Now we make copies of e and of all nodes in L that are on the direct line back to h. We redirect all edges pointing to e so that they lead now to the copy e', and arrange the other copies P' outside of L in the appropriate way. ("Appropriate" in this context means that every node has the same successors or copies of them. Figure 8 makes this clear.) We repeat this procedure for all other nodes in L that are additional entry points to h, in order to finally get a single entry loop.

(3) If there are still other multiple-entry maximum loops, we apply (2) to them. The termination of this process can be shown by complete induction (details in [5]).

(4) If all maximum loops are single-entry loops, we go one *maximum loop level* deeper. We look at the nodes of each maximum loop. We define the *flow graph of a maximum loop* as (a) the nodes of the maximum loop with the h node as the start node and (b) the edges that connect only members of the maximum loop, and are not back links to h. We now apply recursively steps one through four to each of these flow graphs. We proceed until no multiple-entry loop exists. Termination is guaranteed by [5], page 85.
Finally we have to make sure that the sequential block order is preserved. Because every operation of the transformation preserves the sequential block order, the order of single action inside a block remains untouched, unless there is a branching address to be changed.
QED.

A further outcome of this transformation is that the flow graph is now *reducible* [5]. The reducibility of a flow graph is important for data-flow equations, which enable data-flow scrutiny for optimization or maintenance purposes [1] [13].

Before we describe the non-deterministic part of our method we take a look at the loop-free part of a flow graph. By building equivalent sets of the maximum loops we get a *super-structure* in which nodes are sets of sequential blocks. Edges of the super-structure represent the existence of an edge between at least two nodes in the nodes of the super-structure. This flow graph can then be restructured due to the following theorem.

Theorem 2. If f is a loop-free flow graph, there exists a *sequential block preserving transformation* transforming f into a D-structure.

Proof idea.

This proof operates much like the proof of theorem 1. Copying sequential blocks disconnects links into D-structures. Figure 8 shows fragmental flow graph and its transformation. Details of the proof can be found in [5].

We can summarize the two given theorems, which signify the deterministic part of the method, with the following corollary.

Corollary 1. Through the application of a sequential block transformation, every generic flow graph can be transformed into a flow graph with two structural properties:

1. All maximum loops on any level (as defined in Theorem 1 (4)) are single-entry loops.
2. All loop-free structures and super-structures are D-structures.

From now on, we will call a flow graph that fullfills conditions 1 and 2 of corollary 1 an *incomplete D-structure*.

Extended Nassi-Shneiderman Diagrams

Nassi-Shneiderman diagrams are well known for structured design of the control flow and standard programming [16]. They are closely related to the D-structure and can be used in a pre-coding phase with programming languages such as Pascal or MODULA-2 [3]. To provide a similar illustration we developed an extended NS diagram for incomplete D-structures.
An extended NS diagram consists of the following items:

- Sequential program blocks: squares containing source code which, in the event of branching, have a triangle at the bottom.
- Forward edges: a polygon, usually a square.
- Loop edges: polygons connecting a sequential block with a loop entry, a back link. Loop edges are different from forward edges by the keywords `loop` or `while`.
- Flow semantic: an ENS diagram is entered at the top and ends at the bottom. Each sequential block is entered at the top and exited through the bottom line or a side of the triangle in case of branching. The forward edges and loop edges lead from one block to the next, or to one previous, respectivly.

The following example illustrates the ENS diagram definition. The incomplete D-structure flow graph of figure 8 can be drawn as a ENS diagram, as shown in figure 8. The graph and the ENS diagram still have unpleasent back links; the node b is part of two loops. In the next section, we will see that with non-deterministic unlinking (called unfolding) we can simplify this structure.

5 Unfolding : A Non-Deterministic Restructuring

In this section we show the non-deterministic part of our method, called *unfolding*. In case of an incomplete D-structure, further restructuring will preserve the D-structure property.

The examplified flow graph of figure 8 still has a tangled structure. The maximum loop $\{a, b, c\}$ consists of two subloops $\{a, b\}$ and $\{b, c\}$ which share node b. By applying a similar mechanism as displayed in the proof of theorem 1, we can unfold this nestedness. The resulting flow graph is shown in figure 8.

The algorithmic description of unfolding

The unfolding algorithm consists of three major steps. Step one includes the non-deterministic or user-dependable part of the method. Steps two and three are deducible from step one.

- (1) Select an edge e that represents an unpleasant entry to a node v of a maximum loop L.
- (2) Make copies of all nodes of L between e and the loop-entry node of L.
- (3) Add the edges and the labels, so that the set of successor nodes and their labels are the original ones or copies of them.

There are still some crucial points to mention. Our experience has shown that the edge in (1) has to be chosen carefully. If the set of nodes and their edges are very big, the structure of the unfolded graph can be very different from the original, in size as well as in shape. This can impair readability and maintainability. In example 8 there was only one node to copy, so the result is adequate.

The variability compells us to provide a user interface, that supports an *undo* and a *redo* functionality of single operations, enabling the user to estimate his changes.

However, destroying an incomplete D-structure by unfolding would also be undesirable. Fortunately, we can prove that additional unfolding can not destroy a previously gained D-structures. In [5] we proved the following theorem.

Theorem 3. Let f be a D-structure flow graph. Let L be a maximum loop with the entry nodes e_L. Let a be an element of L, different from e_L and with one or more entries. With a sequential block preserving transformation it is possible to free a from all edges pointing to it except one. The gained graph is still a D-structure.

In case of loop-free D-structures, unfolding produces the same copying mechanisms as used in theorem 2, and therefore all resulting structures are going to be D-structures (details in [5]).

6 An Application of Unfolding

In this section we illustrate unfolding applied to a source code fragment, and then we show how to enhance this fragment.

Unfolding can be applied for two different purposes.

- To Ease a flow graph from a tangling structure, for analysis and reading purposes, without physically restructuring the source code. (For defining data-flow equations, it is only necessary to know about the restructured graph and the information of the sequential blocks. In an advanced source code representation, such as we proposed in [6], it is not necessary to make copies of the whole sequential block. In our relational database approach, a simple entity can represent the copied nodes.)
- To physically unfold the flow graph and the source code in order to maintain and enhance the copied and the original block independently.

For the second point, we will give here an illustrative example. The following small fragment, part of a parser, has the control flow graph depicted in 8. The vertix a consists of the line `100`, the vertix b consists of the lines `110` and `120`, and the vertix c consists of the lines `130` to `150`.

```
10 count :  integer
20 token :  enumerate ...
...
100 write "newline"
```

```
110 read token
120 if token = nl then 100
130 count = count + 1
140 if token != end then 110
150 ...
```

We unfold the flow graph, choosing the edge (c, b) and gain the structure shown in figure 8. Figure 8 depicts the unfolded source code fragment represented as an ENS diagram. We emphasize the copied sequential block with a dashed square.

The unfolding enables us to change the copy block independently of the original one. This is particularly interesting because the precondition in the copy is different than in the original block. In the original block the value of the variable `token` is `nl` or an initial value. In the copy the state of the variable `token` is not `end` and not `nl`. This enables changing the behaviour of the program in case of a "regular" token (neither `nl` nor `end`) to insert the token into a token table.

Finally the source code can be printed in a pretty print manner and with changing syntactical keywords (as shown in [11]).

```
10 count :  integer
20 token :  enumerate ...
...
100 loop
110 ..repeat
120 ....write "newline"
130 ....read token
140 ..until token = nl
150 ..loop
160 ....count = count + 1
170 ....if token != end then exit 250
180 ....set_in_table(token)
190 ....read token
200 ....if not (token = nl) then exit
210 ..endloop
230 endloop
250 ...
```

7 Conclusion and Future Work

The restructuring method presented in this paper consists of a two-part flow graph transformation. The first part transforms a flow graph into a so called *incomplete D-structure*, which is somewhere between a reducible flow graph and a D-structure. The non-deterministic part unfolds remaining unpleasant nestedness in flow graphs. This has to be accomplished through user-interaction. The result of this restructuring

depends on the skill and experience of the user, and his or her intended goal. In this paper we have only applied this technique to control flow graphs. But the generality of the method enables application with the same transformation constraints to any kind of directed graphs such as data-flow graphs, semantic networks or workstation computer network topologies.

The theoretical part of the method has already been completed. This paper can thus be regarded as providing closure to previous works in this field.

The results and experience it has yielded will doubtless influence the fledgling project developing fine grain source code database (c.f. [6]). This database is thought to support the development and the maintenance of large software systems.

References

[1] Alfred A. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.

[2] E. Ashcroft and Z. Manna. The translation of 'goto' programs to 'while' programs. *Inform. Proc.71*, pages 147–152, 1971.

[3] G. Blaschek, G. Pomberger, and F. Ritzinger. *Einführung in die Programmierung mit Modula-2.* Studienreihe Informatik. Springer-Verlag, 1986.

[4] C. Böhm and G. Jacopini. Flow diagrams, Turing maschines and languages with only two formation rules. *Communications of the ACM*, 9(5), 1966.

[5] Toni A. Bünter. *Eine Architektur eines Software-Wartungssystems.* PhD thesis, Universität Zürich, 1992.

[6] Toni A. Bünter. A repository for a care environment. *CASE'93 Sixth International Workshop on Computer-Aided Software Engineering*, 1993. (submitted in January).

[7] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic-Press (London,New-York), 1972.

[8] E. W. Dijkstra. Goto statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[9] E. W. Dijkstra. Notes on structured programming. In Academic Press New York, editor, *Structered Programming*, 1972.

[10] Paul Eisner. Strukturierte Wartung von Cobol-Software. *Output*, (6), 1986.

[11] Paul Eisner. *Strukturierte Software-Wartung.* PhD thesis, Universität Zürich, 1988.

[12] C. A. R. Hoare and Niklaus Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1975.

[13] Ken Kennedy. *A Survey of Data Flow Analysis Techniques.* Prentice-Hall software series, 1981.

[14] D. E. Knuth and R. W. Floyd. Notes on avoiding 'go to' statements. *Inform. Processing Letters 1*, pages 23–31, 1971.

[15] Henry F. Ledgard and Michael Marcotty. A genealogy of control structures. *Communications of the ACM*, 18(11), 1975.

[16] I. Nassi and B. Shneiderman. Flowchart techniques for standard programming. *Sigplan Notices*, 8(8):12–26, 1973.

[17] W. W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat, and exit statements. *Communications of the ACM*, 16(8):503–512, April 1973.

8 Figures

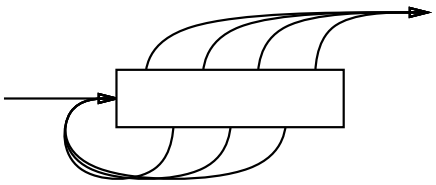

Figure 1: Single-entry/single-exit flow graph.

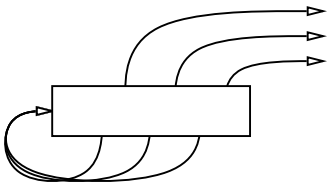

Figure 2: Single-entry/multiple-exit flow graph.

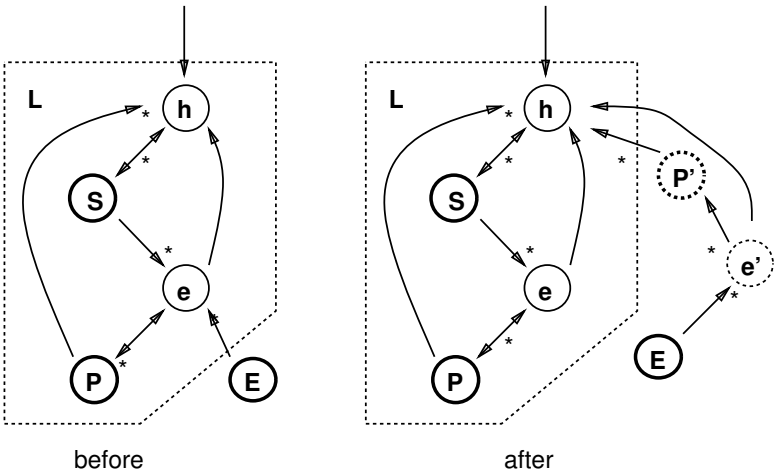

Figure 3: Display of a maximum loop **L** with a second entry from the nodes **E**. The maximal loop consists of **h**, the main entrance of the loop e_{main}, the second entrances from **E** ,and the other nodes collected in the bold verices **S** and **P**. The arrows represent zero or more directed edges and only the affected edges are displayed. After the transformation **P** and **e** are doubled and the necessary edges are built.

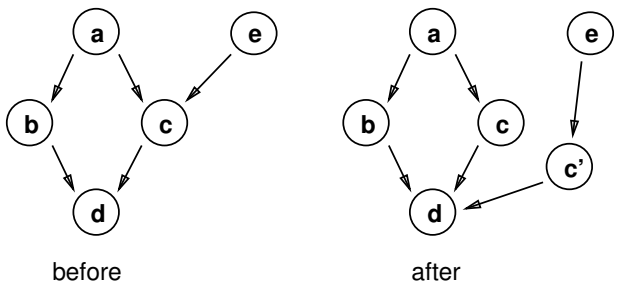

Figure 4: Transformation of loop-free flow graphs to D-structures. The "before" graph show a entrance of **e** disturbing the D-structure **a**, **b**, **c**, and **d**. By copying **c** and redirecting **e** to **c'** the D-structure can be rescued.

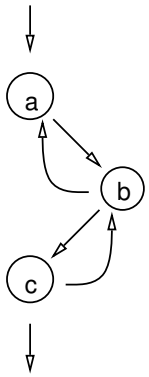

Figure 5: Incomplete D-structure with two subloops { **a**, **b** } and {**b**, **c**} and the shared node **b**.

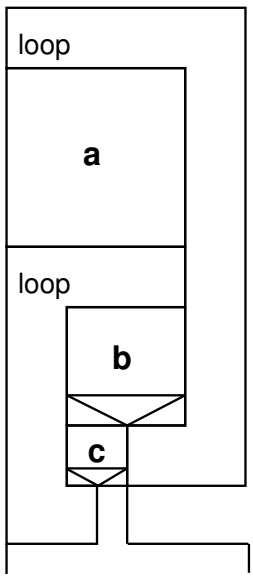

Figure 6: Example of a ENS diagram.

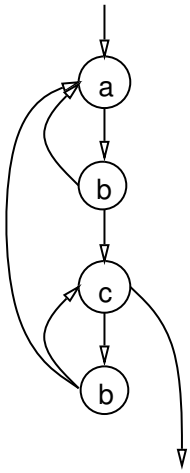

Figure 7: Incomplete D-structure after unfolding of the double used node **b**.

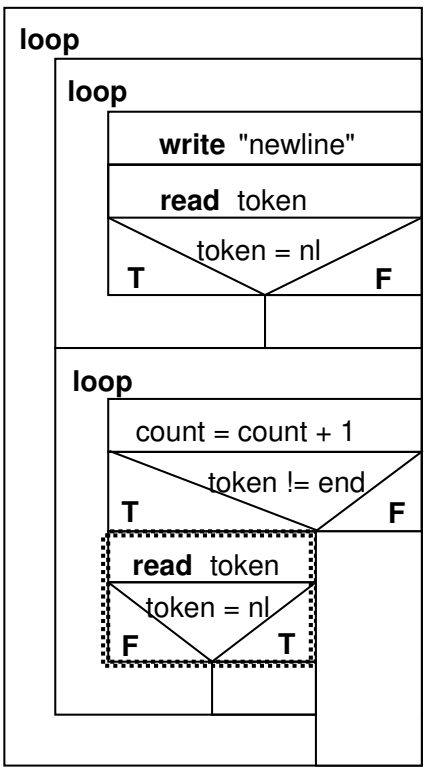

Figure 8: Exemple source code fragment depicted as an ENS diagram. The dashed square signifies the copied sequential block.