Dependency Diagrams

by Mark Ray

A method of illustrating program structure by showing how sections depend on each other is presented. This suggests an intuitive metric for program partitioning, which is developed with supporting theory.

Introduction

Many methods exist for the visualization and measurement of structure during the design of a programming solution. Such methods include structure charts and coupling [1],[2]. In contrast, few methods exist for the measurement of the structure of the finished program. As most programming courses now center on the concept of structured programming, measurements of structure for the program are desirable.

Also, the lack of a way of visualizing the program structure has lead to many programmers (myself included) becoming lost in the final solution as they converted it into an actual program. I started to draw diagrams of the program that I was working on, adding, moving and deleting pieces as the program developed and problems with the translation from design through pseudo-code to high-level language code were overcome. Over time, I decided that it was most helpful to make the diagram show the relationships between sections of the program, so that if one of the fundamental units of the program required alteration (for some unforeseen reason), it was obvious which other units this would affect.

I noticed that these diagrams and their construction could be formalized, as presented here.

Independent and dependent units

An integral part of structured programming style in any imperative is the splitting of the program into smaller units. These units are variously called functions (C), procedures (Modula-2), subroutines (BASIC) and a host of other names, but the idea is always the same. Here, they shall all be referred to as **units**.

All units can be classified as either dependent or independent. An **independent unit** does not invoke any other program unit. That is, it is written using only the language's built-in keywords and library routines supplied to the program. A **dependent unit** invokes at least one other program unit.

Either of the above definitions could equally well have been stated as ``a unit is dependent if and only if it is not independent." Being able to classify each unit as either dependent or independent, by itself, is not very useful in helping to assess how well the problem has been separated, but it is an essential step towards this. More useful, is the generalization of dependence, provided by the following definition:

> A unit, x, is said to be **dependent on another unit**, y, if x contains a call to y. Equivalently, x is called a dependent of y.

This is the basic relationship that I will work with. Notice that the unit y does not need to be called whenever x executes, only that x may need to call upon y. This definition of dependency was arrived at by the original need for documenting these relationships -- to verify that unit x would still work correctly if unit y is changed, regardless of the fact that x may not always call y (and so errors may exhibit themselves in some tests, but not others).

One way of keeping track of dependency relationships is through the use of a **dependency table**, which lists each unit, followed by its dependent units. For example, such a table could look like this:

Unit	Dependent Units
NewList	Copy, Sort
Length	Copy, Search
Copy	--
Search	Sort
Sort	--

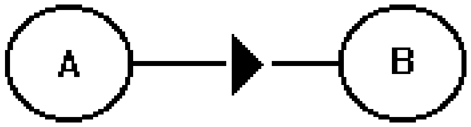
The table shows that Copy includes calls to Length and NewList; Search includes a call to Length; Sort includes calls to NewList and Search.

A dependency table is a useful tool for tracing the effects of changes through small programs, but searching through a table of any length (even as little as 15 units) is rather tedious. Therefore the dependency diagrams were developed.

Creation of Dependency Diagrams

Diagrammatic representations allow the visualization of the structure of a program far more clearly than a considerable quantity of documentation ever can. With a relationship between units defined, it is possible to represent the dependencies as a directed graph (digraph), with each node representing a program unit and each directed edge representing a dependency.

So, a unit *A* dependent on a unit *B* would be represented on a dependency diagram like so:

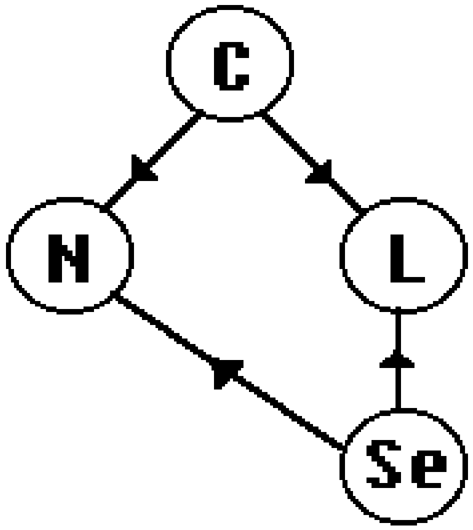


A dependency diagram may be built up by using the following algorithm:

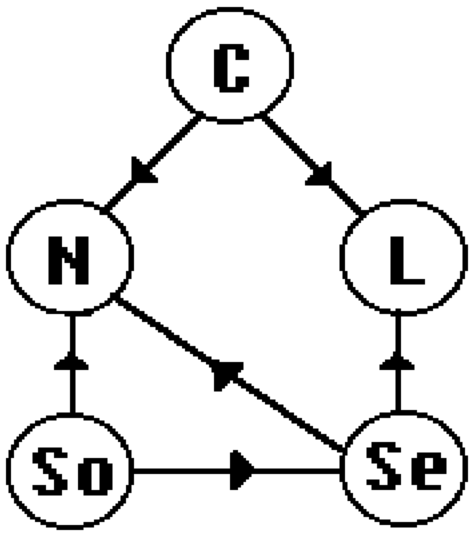
- Place any independent units on the graph.
- While there are units which are not represented on the graph:
    - Add any units that only link to the units already on the graph.
    - If there is no such unit, add the unit with the fewest links to non-existent nodes and note the missing links.

To create a dependency diagram for the example shown in the dependency table, we first place the independent units NewList and Length on the graph:

Then, Copy and Search link only to NewList and Length:

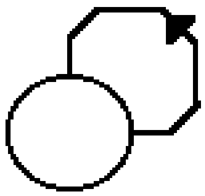


Finally, Sort links to Search and NewList:

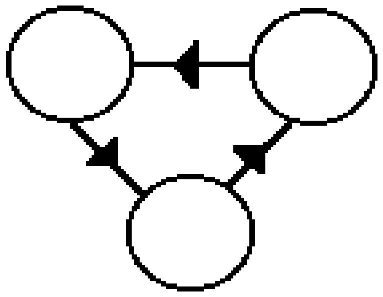


A few features of dependency diagrams deserve special mention:

- Recursion can be represented on a dependency diagram with a unit which links to itself:

- A cyclic definition of some units (where, for example, a calls b, b calls c and c calls a) exhibits itself on the dependency diagram as allowing a *tour*:

  If the compiler being used does not support such a situation, it may be identified and eliminated before compilation.

- If the dependency diagram is *disjoint* or there is at most one node with paths to both of a given pair of nodes, then the system probably has poor cohesion (see [1]) and the factoring (division of the problem into modules) should be re-assessed.

- Changes to the specification of a unit, for whatever reason, often leads to a ``ripple effect,'' eventually affecting many other units. With a dependency diagram, it is easy to identify which modules will need modifying or re-testing. Only units which depend directly on the altered one should need changing (although if this results in a changed specification for the dependent unit...), but all units with a path to the altered one should be re-tested.

Dependency Levels

It can be argued that as the program is required to perform more functions, it should be divided into more units, with each of the lower level units performing a repeatedly used sequence of the related basic operations. This is one of the basic ideas of structured programming.

So, in a program that has been divided up well, each of the ``top-level" units will call a few units from the levels just ``beneath." This can be quantified by the **dependency level**:

- An independent unit has **level zero**.
- A dependent unit has a **unit level** equal to the length of the longest path on the dependency diagram from it to a level zero unit.
- The **dependency level** of a dependent unit is the average level of the units that it is dependent upon.

A well structured program will have units with dependency levels only a little below their unit levels. In the example, NewList and Length are independent units, so their dependency level is undefined (number of units called is zero). The level one units, Copy and Search, both have a dependency level of zero.

In fact, all level one units have a dependency level of zero. By definition, the only units that level one units may call are level zero units, so the average level of the units that level one units are dependent upon is always zero.

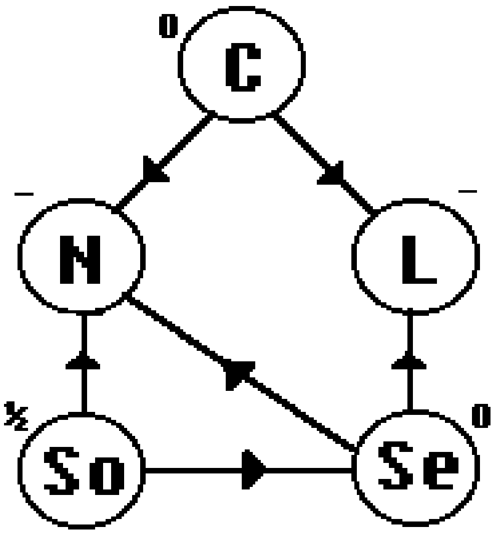
For the Sort unit in the example, the dependency level is:

```
          DL(Sort) = (Level of NewList + Level of Search) / 2
                   = ( 0 + 1 ) /2
                   = 1/2
```

So, the level two unit Sort has a dependency level of 1/2.

By annotating a dependency diagram with the dependency level of each node we can create a **dependency level diagram**.

So for the above example, the dependency level diagram is:



As can be seen, the dependency level provides a measurement of program partitioning. However, this alone does not allow judgments to be made about whether a program is sufficiently well partitioned. Generally, aiming for a dependency level of at least one quarter of the unit level for all units that have level two or above seems to be a good rule of thumb. It should be noted, however, that this is an entirely empirical result.

Also, it should be noted that there are ways of ``fiddling'' the dependency levels. But these should not be part of the structured programmer's habits, so they do not really pose a significant problem.

Conclusion

If used properly and developed in parallel with the program, dependency level diagrams are a valuable method with which to visualize the structure of the program, a useful aid in checking for ``knock-on effects'' of any changes in lower level units, and a method of measuring how well structured a program is. I would not recommend using dependency level as the sole measurement, but it should be used in conjunction with established qualitative judgments such as cohesion.

References

1

Page-Jones, Meilir: *The Practical Guide to Structured Systems Design.* Yourdon Press. 1980

2

Yourdon, E., and Constantine, L.: *Structured Design.* Prentice-Hall 1979.

Acknowledgments

Thanks to Sarah Bown (School of Modern Languages and European Studies, UEA) and Martyn Drake

(School of Information Systems, UEA) for reading and suggesting significant improvements to the original text and to the editors at *Crossroads* for advice on the redraft.