Chapter 4

4. Modelling Software for Structure Metrics [1]

In the traditional approach to structure software metrics, software is modelled by means of flowgraphs. A tacit assumption in this approach is that the structure of a program is reflected by the structure of the flowgraph. When only the flow of control between commands is considered this assumption is valid; it is no longer valid however when also the control flow inside expressions is considered. In this chapter, we introduce structure graphs for the modelling of software. Structure graphs can, just as flowgraphs, be uniquely decomposed into a hierarchy of indecomposable prime structures. We show how programs in an imperative language can be modelled by means of structure graphs in such a way that the structure of a program is always reflected by the structure of the corresponding structure graph.

4.1 Introduction

In the traditional approach to structure software metrics (Fenton, 1991; Fenton & Kaposi, 1987), a program in an imperative language is mapped onto an abstract program, whereby program parts without structure are replaced by atomic actions; the resulting abstract program is mapped onto a flowgraph, and this flowgraph is decomposed into a hierarchy of primes (i.e. irreducible flowgraphs), which results in a decomposition tree onto which metric functions are applied. Where these metric functions are defined inductively, the metrics are called structure metrics. Flowgraphs will be defined formally in section 4.2 of this chapter. First, we will consider the modelling of imperative program fragments by flowgraphs.

An atomic action in a program is modelled by a P_1-flowgraph, which consists of a start node, a stop node, and one edge between these nodes. The flow-

[1] This chapter is an adaptation of: P.M. van den Broek & K.G. van den Berg (1993). Modelling Software for Structure Metrics. *Memoranda Informatica 93-12.* Enschede: University of Twente.

graph for an abstract program is constructed by associating a node with each atomic action, adding a stop node, identifying the node for the first atomic action as the start node, and drawing arcs from each node to its possible successors. Here the stop node is a successor for all possible last atomic actions. For instance, consider the following abstract program fragment:

```
WHILE a DO b END
```

This program fragment is modelled by the flowgraph D_2 in Figure 4.1.

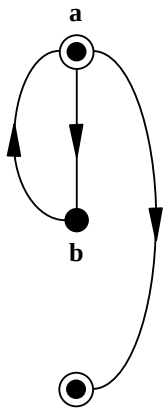

Figure 4.1 Flowgraph D_2 of WHILE a DO b END

Note that the node corresponding to `b` is a procedure node (has outdegree 1) and that the node corresponding to `a` is not a procedure node. This means that on node `b` another flowgraph can be nested, but on node `a` this is not possible. Consider the following program fragment:

```
IF c THEN d ELSE e END
```

This program fragment is modelled by the flowgraph D_1 in Figure 4.2.

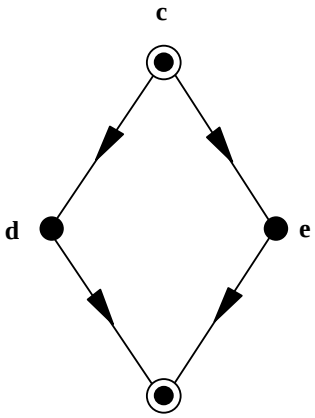

Figure 4.2 Flowgraph D_1 of IF c THEN d ELSE e END

Suppose we want to replace `b` in the program fragment `WHILE a DO b END` by `IF c THEN d ELSE e END`, resulting in the program fragment:

```
WHILE a DO
  IF c THEN d ELSE e END
END
```

Then, nesting the flowgraph D_1 (Figure 4.2) on node `b` of flowgraph D_2 (Figure 4.1) gives the flowgraph in Figure 4.3, denoted as $D_2(D_1)$.

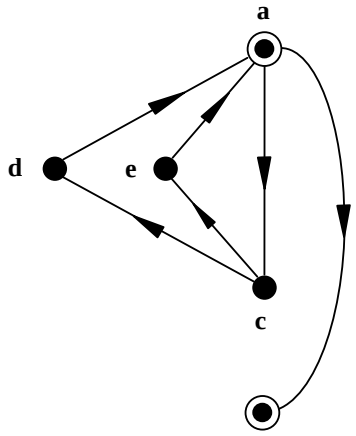

Figure 4.3 Flowgraph of WHILE a DO IF c THEN d ELSE e END END

Suppose we want to replace `a` in `WHILE a DO b END` by `c OR d`. Assuming a 'lazy' `OR`, the construct `c OR d` is modelled by the flowgraph D_0 which is given in Figure 4.4.

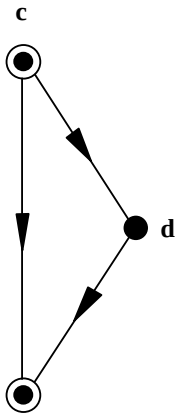

Figure 4.4 Flowgraph D_0 of c OR d

Now it is not possible to obtain the flowgraph of

```
WHILE c OR d DO b END
```

by nesting the flowgraph D_0 in Figure 4.4 on node a of the flowgraph D_2 in Figure 4.1. Instead, the flowgraph for this program, which is given in Figure 4.5, is a prime flowgraph to be called X_1.

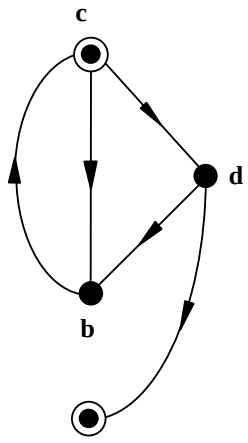

Figure 4.5 Flowgraph X_1 of WHILE c OR d DO b END

So, in this case the structure of the abstract program is not reflected by the structure of the corresponding flowgraph. In order to solve this problem, we will define the mapping from programs onto structure graphs, which are flowgraphs whose start node has outdegree 1.

This chapter is organised as follows. In section 4.2, we will recapitulate the theory of flowgraphs and flowgraph decomposition. In section 4.3 we will explain the notion of structure graph. Structure graphs can, analogous to flowgraphs, be uniquely decomposed into a hierarchy of prime structure graphs. Programs in an imperative language can be modelled by means of structure graphs in such a way that the structure of a program is always reflected by the structure of the corresponding structure graph. Structure metrics for these graphs are discussed in section 4.4. In the last section we consider two small example languages; we show how programs in these languages are mapped onto structure graphs.[2]

[2] This mapping has been implemented, and also the decomposition algorithm for the structure graphs, in the functional programming language Miranda.

4.2 Flowgraphs

In this section, we briefly recapitulate the theory of flowgraphs and their decomposition (Fenton, 1991; Fenton & Kaposi, 1987). We start with the definition of a flowgraph:

Definition A *flowgraph* is a 3-tuple (G,a,z) where G is a directed graph, and a and z are nodes of G, called *start node* and *stop node* respectively, such that:
- For each node x of G there is a path in G from a to z via x.
- The outdegree of z is 0.

The next two definitions specify operations on flowgraphs:

Definition If $F_1=(G_1,a_1,z_1)$ and $F_2=(G_2,a_2,z_2)$ are flowgraphs then the *sequence* $F_1;F_2$ of F_1 and F_2 is the flowgraph $(G_1;G_2,a_1,z_2)$ where $G_1;G_2$ is the directed graph which is obtained from the union of G_1 and G_2 by identifying the nodes z_1 and a_2.

Definition If $F_1=(G_1,a_1,z_1)$ and $F_2=(G_2,a_2,z_2)$ are flowgraphs and x is a node of G_1 with outdegree 1 (called a *procedure node*) then the *nesting* $F_1(F_2 \text{ on } x)$ is the flowgraph $(G_1(G_2 \text{ on } x),a_1,z_1)$ where $G_1(G_2 \text{ on } x)$ is the directed graph which is obtained from the union of G_1 and G_2 by deleting the edge whose source is x, identifying x and a_2, and identifying z_2 and the successor of x.

Definition The flowgraph $F_2=(G_2,a_2,z_2)$ is a *subflowgraph* of the flowgraph $F_1=(G_1,a_1,z_1)$ if G_2 is a subgraph of G_1 and z_2 is the source of all edges from G_2 to $G_1 \setminus G_2$.

Definition The subflowgraph $F_2=(G_2,a_2,z_2)$ of the flowgraph $F_1=(G_1,a_1,z_1)$ is a *one-entry subflowgraph* if
- the target of each edge from $(G_1 \setminus G_2) \cup \{z_2\}$ to G_2 is either a_2 or z_2, and
- if a_1 belongs to G_2 then $a_1=a_2$ or $a_1=z_2$

Definition The subflowgraph $F_2=(G_2,a_2,z_2)$ of the flowgraph $F_1=(G_1,a_1,z_1)$ is a *proper subflowgraph* if $G_1 \neq G_2$ and F_2 is not one of the two trivial flowgraphs. Here, the two *trivial flowgraphs* are the flowgraph consisting of one node only (P_0), and the flowgraph consisting of two nodes and one edge (P_1).

Definition The proper one-entry subflowgraph F_2 of the flowgraph F_1 is a *maximal one-entry subflowgraph* of F_1 if there exists no proper one-entry subflowgraph F_3 of F_1 such that F_2 is a proper one-entry subflowgraph of F_3.

The next two theorems show a way in which each flowgraph can be decomposed uniquely into a hierarchy of indecomposable flowgraphs (*primes*).

Theorem Each flowgraph F can be written uniquely as a *sequence* of nonsequential flowgraphs $F_1;F_2;..;F_n$.

Theorem Each nonsequential flowgraph F can be written uniquely as a simultaneous *nesting* $F_0(F_1 \text{ on } x_1,F_2 \text{ on } x_2,...,F_n \text{ on } x_n)$, where $F_1,F_2,...,F_n$ are the maximal proper one-entry subflowgraphs of F_0. Moreover, F_0 is a prime.

The nesting - in the last theorem - is usually denoted as $F_0(F_1, F_2,...,F_n)$, in which is abstracted from the nodes onto which the flowgraphs are nested.

An algorithm for the decomposition of flowgraphs is given in Bache & Wilson (1988). Note that, according to our definition of one-entry subflowgraphs, a requirement for (G_2,a_2,z_2) to be a one-entry subflowgraph of (G_1,a_1,z_1) is the absence of edges in G_1 from z_2 to nodes in G_2 other than a_2. This requirement is absent in the definitions in Fenton (1991), Fenton & Kaposi (1987) and Bache & Wilson (1988). Without this requirement, testing for the one-entry property in the algorithms above is not sufficient.

4.3 Structure graphs

As shown in the introduction, a drawback of the traditional way of modelling software by means of flowgraphs is that no refinement of conditions can be modelled, since conditions do not correspond to procedure nodes, and the operation of nesting is defined for procedure nodes only. The solution of this problem will lead to the introduction of a subset of flowgraphs, called structure graphs, as will be explained below.

As a first step towards a solution of this problem, we propose to model software by flowgraphs in such a way that each atomic action corresponds to a procedure node. For instance, `IF c THEN d ELSE e END` is modelled by the flowgraph of Figure 4.6, to be called structure graph D_1' .

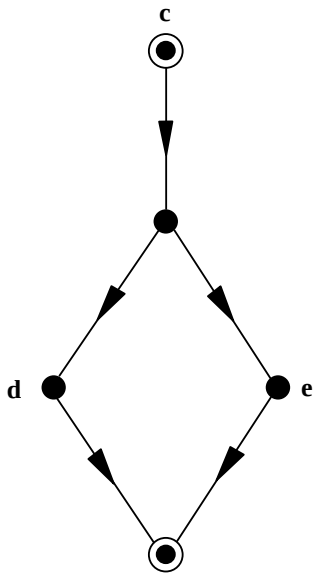

Figure 4.6 Structure graph D_1' of IF c THEN d ELSE e END

Similarly, `WHILE a DO b END` is modelled by the flowgraph of Figure 4.7, to be called structure graph D_2' .

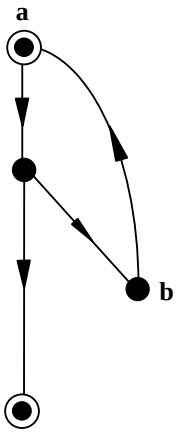

Figure 4.7 Structure graph D_2' of WHILE a DO b END

Note that there are nodes with outdegree > 1 which do not correspond to atomic actions. After Whitty (1988), we call these nodes *select* nodes.

For each program, such a new representation as a flowgraph can be obtained by the old one by replacing in the former each node with outdegree > 1 by a procedure node with an edge to a new selection node. Unfortunately, this does not solve our problem. The program `WHILE c OR d DO b END` would correspond to the graph of Figure 4.8:

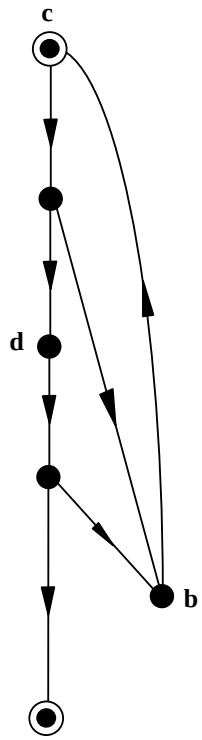

Figure 4.8 Tentative structure graph of WHILE c OR d DO b END

This is not what we want. Let us consider the flowgraph for `c OR d` in our new model, to be called structure graph D_0' ,which is given in Figure 4.9:

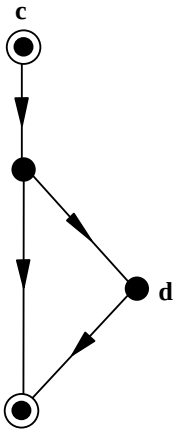

Figure 4.9 Structure graph D_0' of c OR d

We want the structure graph for `WHILE c OR d DO b END` to be the structure graph D_0' for `c OR d` (Figure 4.9) nested on node `a` of the structure graph D_2' for `WHILE a DO b END` (Figure 4.7). This structure graph is shown in Figure 4.10.

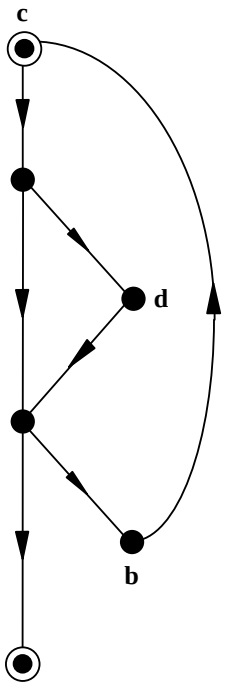

Figure 4.10 Structure graph of WHILE c OR d DO b END

In general, as the second step to the solution of our problem, we propose to assign graphs which can be interpreted as control flowgraphs only to 'basic' programs, and to assign to other programs graphs which are obtained from the graphs of their subprograms, using sequencing and nesting. In section 4.5 we illustrate this for two example languages.

We are left with one major problem. The graphs assigned to 'basic' programs should be primes. However, the graph for `IF a THEN b ELSE c END`, which is given in Figure 4.6, is not a prime; it is a sequence $P_1;D_1$ of the prime flowgraphs P_1 and D_1. The same is true for the graph in Figure 4.9, which corresponds to the basic program `c OR d` , which is the sequence $P_1;D_0$. The third (and final) step to the solution of our problem is therefore to consider only a subset of the flowgraphs, called structure graphs. A *structure graph* is a flowgraph whose start node is a procedure node.

This choice is justified by the observation that programs should start with an action, not with a selection. Note that the flowgraphs in Figure 4.6 and Figure 4.9 are structure graphs which cannot be decomposed into smaller structure graphs, i.e. they are prime structure graphs, respectively D_1' and D_0'.

The theory of structure graphs is, fortunately, analogous to the theory of flowgraphs. The operations of sequencing and nesting are well-defined on structure graphs, and structure graphs can be decomposed uniquely into a hi-

erarchy of prime structure graphs. An algorithm for the decomposition of structure graphs may be obtained from an algorithm for the decomposition of flowgraphs in a straightforward way. However, the result of the decomposition of a structure graph into prime structure graphs can be quite different from the result of the decomposition of the same graph into the traditional prime flowgraphs. Consider for instance the graph in Figure 4.11.

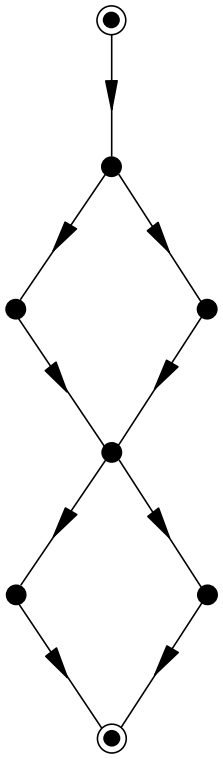

Figure 4.11 Example graph

As a traditional flowgraph, its decomposition is a sequence of three prime flowgraphs: P1; D_1'; D_1'. As a structure graph, its decomposition is a nesting of the prime structure graph D_1' on the prime structure graph D_1': i.e., $D_1'(D_1')$.

4.4 Structure metrics

The prime decomposition of flowgraphs has been used for the definition of the important class of structure metrics (Fenton, 1991). These metrics can be described completely in terms of the primes and the operations of sequencing and nesting. A *structure metric* m is determined uniquely by the following three characteristics:

1. m(F) for each prime F
2. a function g_n such that $m(F_1;...;F_n) = g_n(m(F_1),...,m(F_n))$
3. a function h_F such that $m(F(F_1,...,F_n)) = h_F(m(F_1),...,m(F_n))$ for each prime F.

A structure metric with these properties is called a hierarchical metric. Moreover, if the nesting function *h* is independent of F then the metric is called a recursive metric. So, the class of recursive metrics is contained in the class of hierarchical metrics.

For example, the structure metric *depth of nesting* m_d is defined as follows (Fenton, 1991):

1. $m_d(P_1) = 0$
   for each prime $F \neq P_1$: $m_d(F) = 1$
2. $m_d(F_1;...;F_n) = \max(m_d(F_1),...,m_d(F_n))$
3. $m_d(F_0(F_1,...,F_n)) = 1 + \max(m_d(F_1),...,m_d(F_n))$

For structure graphs, structure metrics can be defined in the same way. It should be kept in mind that the decomposition for structure graphs, differs from flowgraphs as used in the structure metrics given above, as discussed in the previous section. The metric values need not be the same in both approaches. E.g., the depth of nesting for `WHILE c OR d DO b END` in the traditional modelling (see Figure 4.5) is 1 and in the new model (see Figure 4.10) the depth of nesting is 2, i.e. the depth of $D_2'(D_0')$.

4.5 Two small languages

In this section we consider the mapping from programs of some small languages to structure graphs. We do not consider the mapping from programs to abstract programs; our languages themselves consist of abstract programs. Our first language is given in Table 4.1:

```
<program>     = PROGRAM <name> ; BEGIN <body> END <name> .
<body>        = <expression> | <expression> ; <body>
<expression>  = S | WHILE <expression> DO <body> END |
                IF <expression> THEN <body> ELSE <body> END
<name>        = <letter> | <letter> <name>
<letter>      = 'a' | ..... | 'z'
```

Table 4.1 Sample programming language

A program consists of a body, which equals a sequence of expressions. There are three kinds of expressions: the `WHILE` expression, the `IF` expression, and the expression `S`. This last expression corresponds to atomic actions.

The mapping from programs to structure graphs is defined by induction. The structure graph of a program is the structure graph of its body. The structure graph of a body is the sequence of the structure graphs of its expressions.

The structure graph of the expression S is the structure graph with two nodes and one edge. The structure graphs of a IF expression and an WHILE expression are the structure graphs of Figure 4.6 and Figure 4.7 respectively, on which the structure graphs of their subexpressions are properly nested.

It is easily shown that each expression is mapped onto a nonsequential structure graph. From this it follows that the decomposition tree of the structure graph of each program can be obtained from the parse tree of the program, and vice versa. So, for this language there is no need to construct and decompose a structure graph in order to obtain the structure of a program; the structure is completely determined by the syntax of the program. This remains true when we add more 'structured' expressions, like REPEAT-loops, and OR and AND expressions.

Our second example language is obtained from the first one by adding labelled expressions and a GOTO expression (see Table 4.2):

```
<program>     = PROGRAM <name> ; BEGIN <body> END <name> .
<body>        = <expression> | <expression> ; <body> |
                <label> : <expression> |
                <label> : <expression> ; <body>
<expression>  = S | WHILE <expression> DO <body> END |
                IF <expression> THEN <body> ELSE <body> END |
                GOTO <label>
<name>        = <letter> | <letter> <name>
<label>       = <digit>  | <digit>  <label>
<letter>      = 'a' | ..... | 'z'
<digit>       = '0' | ..... | '9'
```

Table 4.2 Extended sample programming language

Since we will not assign a structure graph to a GOTO expression, the mapping from programs to structure graphs cannot be defined by induction in this case. Informally, the structure graph corresponding to a program in this language is obtained by first replacing the GOTO expressions by S and constructing the structure graph as in the first example language (forgetting about the labels) and then removing the GOTO-nodes by redirecting the incoming arcs for each GOTO-node to the start node of the structure graph corresponding to the expression with the appropriate label. More formally we proceed as follows.

Definition A *generalised structure graph* is a 7-tuple consisting of
- a set N, the elements of which are called nodes,
- a node a, called the start node,
- a node z, called the stop node,

- a set E of ordered pairs of nodes, the elements of which are called edges,
- a set L, the elements of which are called labels,
- a set B of ordered pairs of a label and a node, the elements of which are called bindings,
- a set D of ordered pairs of a node and a label, the elements of which are called dangling edges

A structure graph can be seen as a generalised structure graph for which the sets L, B and D are empty. Sequencing and nesting are defined for generalised structure graphs just as for structure graphs (if dangling edges are treated as 'real' edges).

A mapping from programs to generalised structure graphs can be defined by induction as follows.

The generalised structure graph of a program is the generalised structure graph of its body. The generalised structure graph of a body is the sequence of the generalised structure graphs of its (labelled) expressions. The generalised structure graph of a labelled expression is the generalised structure graph of the expression to which a binding is added consisting of the label and the start node. The generalised structure graph of the expression `S`, of a `WHILE` expression and of an `IF` expression are the structure graphs as in the previous example, on which the generalised structure graphs of their subexpressions are properly nested. Finally, the generalised structure graph of a `GOTO` expression consists of two nodes, the start node and the stop node, and a dangling edge consisting of the start node and the label.

Having obtained a generalised structure graph for a program, its dangling edges can be replaced by 'real' edges; their target nodes can be obtained from the set of bindings. If no binding for a label is found, the program is not well-formed. If for all labels bindings are found, either the result is a structure graph or the program contains unreachable code. So we have defined a mapping from well-formed programs to structure graphs, where the well-formed programs are programs without missing labels and without unreachable code. The procedure nodes of the structure graph correspond to basic actions and to `GOTO` statements. The nodes corresponding to `GOTO` statements are indirection nodes, and can be removed. Note that the syntax of our language allows a `GOTO` expression as the conditional of a `WHILE` or an `IF` expression, but that this is impossible for a well-formed program.

As an example, consider the rather unusual[3] program given in Table 4.3:

[3] For example in Pascal, such a program is not allowed according to the ISO standard

```
PROGRAM example;
BEGIN
   IF S1 THEN GOTO 1 ELSE S2 END;
   IF S3 THEN 1:S4 ELSE S5 END
END example.
```

Table 4.3 Example program

The structure graph and the (traditional) flowgraph of this program is given in Figure 4.12.

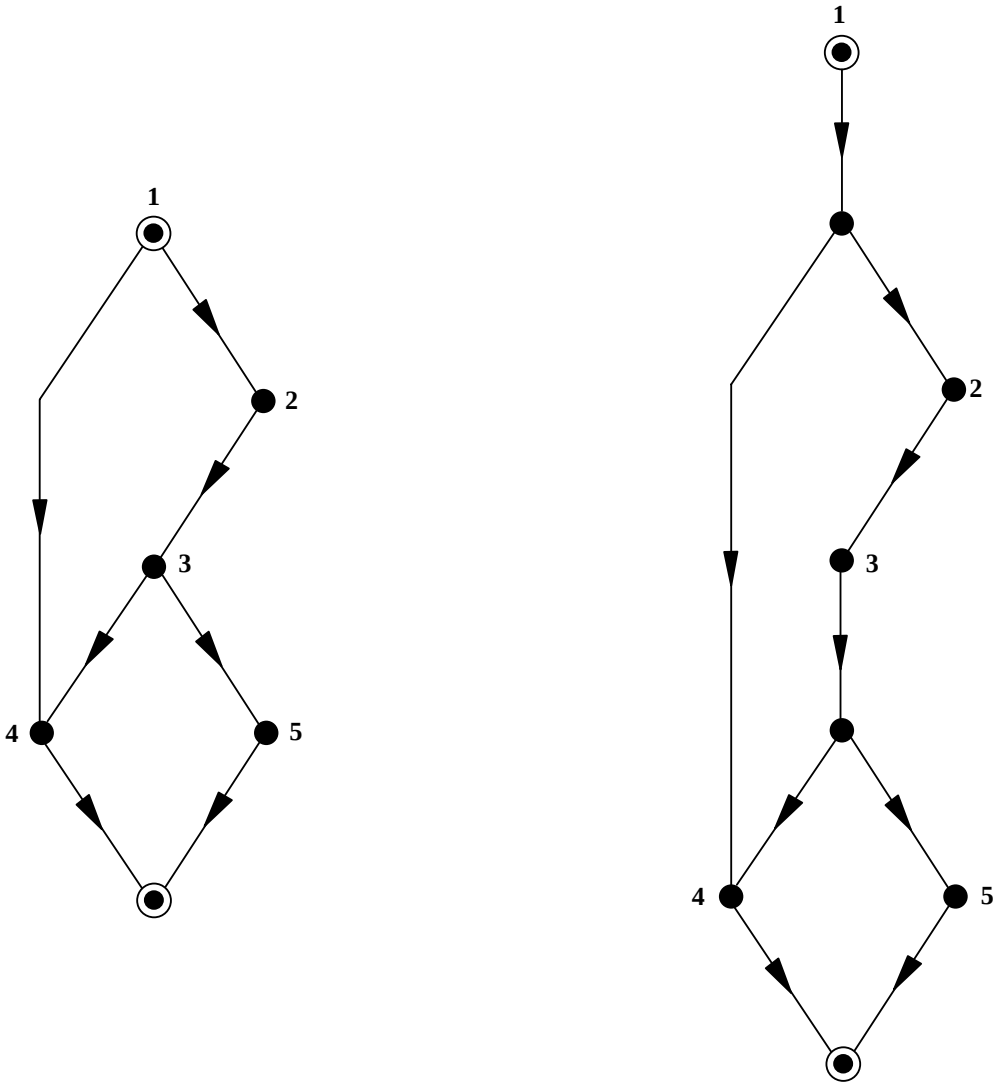

Figure 4.12 Flowgraph (left) and structure graph (right) of the example program in Table 4.3

The occurrences of `S` in the example program have been given indices; these indices are used in the graphs to show the correspondence between atomic actions and graph nodes. The flowgraph of Figure 4.12 is a prime flowgraph; the structure graph of Figure 4.12 however is not a prime structure graph: it contains as substructure the sequence of S_2 and S_3. This is an example where our structure graph approach reveals a substructure which remained unnoticed in the traditional flowgraph approach. It is also interesting to note that this sequential substructure was not explicitly denoted as a sequence in the program.

4.6 Conclusion

We have introduced the modelling of programs in terms of structure graphs, which are flowgraphs whose start node is a procedure node. Structure graphs can, just as flowgraphs, be uniquely decomposed into a hierarchy of indecomposable prime structures. It has been shown that structure graphs are better suited than flowgraphs to model the structure of programs in an imperative language. We have given explicitly the mapping from programs of small example languages to structure graphs.