Well-formed Conversion of Unstructured One-in/one-out Schemes for Complexity Measurement and Program Maintenance*

G. CANTONE, A. CIMITILE AND U. DE CARLINI
Department of Informatics and Systems, University of Naples, 21 Claudio Street, 80125 Naples, Italy

This paper concerns the conversion from non-structured to equivalent structured programs. For this aim the conversion rules of one-in/one-out structures into the Dijkstra traditional structures are described. This approach allows both the application of complexity measurement methods for structured programs and an easier program maintenance. The suggested conversion rules have been implemented by a proper integrated system of software tools.

Received December 1984

1. INTRODUCTION

The paper deals with the conversion of the program control flow into a structured form. This subject has been widely addressed in the literature, from the method proposed by Boehm and Jacopini[1] to Williams' paper,[2] which clarifies some of the issues concerned with reducibility and structuredness in flowchart schemata; a wide bibliographic review has been reported by Oulsnam in a meaningful paper.[3]

The problem of transforming the control flow of a program into some standard form, other than theoretical interests, covers many important practical aspects such as the complexity measurement and maintenance of unstructured software.

For the complexity measurement it is well known[4] that reliable measurements are based mainly on the evaluation of both the quality and the nesting levels of structures, hence the complexity of non-structured programs cannot be measured by such methods. In this case it is only possible to use inadequate measurement methods based on the estimation of the elementary objects constituting a program (i.e. statements, branch number, operators, etc.).[5,6] Conversely, indirect methods of measurement employing the program control flow conversion can be conveniently used. Referring to maintenance two reasons, at least, suggest the use of the control flow conversion: (i) the improvement of program understanding during the first phase of the maintenance activity when the side and ripple effects probably introduced by a program change are to be revealed; (ii) the automatic reduction of the program's unstructuredness level.[7]

This paper presents a method for converting the program control flow into a structured equivalent control flow in TD_1 chart format.[8] For this aim, first, a Basic subset of one-in/one-out Unstructured Programming Structures (BUPS) not belonging to the TD_1 structures is defined. Then a set of rules is shown for the conversion from BUPS elements to equivalent TD_1 structures, the conversion being strong when possible, weak otherwise.[9] Finally an algorithm converting a generic unstructured programming structure (ups) into BUPS elements is shown.

* This paper was carried out as a contribution to the research 'Ingegneria del software: metodologie e strumenti per la produzione del software in ambiente mini e micro.' Supported by M.P.I. 40%.

Software tools able to recognize the program one-in/one-out structures and to convert BUPS structures into TD_1 structures have been implemented.

2. THE PROGRAM GRAPH MODEL

We shall refer to the program Graph Model (GM),[10] which describes a program by means of a direct graph (d-graph), named primitive d-graph (*pd-g*), almost strongly connected; a *pd-g* has a unique root $r \in pd\text{-}g$ and a unique anti-root $a \notin pd\text{-}g$ that represents the program external environment. Further, a *pd-g* has not: (i) nodes being the starting point of more than two edges; (ii) edges having the same node as both the starting and the ending point; (ii) edges having the same node as both the starting and the ending point; (iii) edges having the same starting node and the same ending node.

The GM can be defined as the quintuple

$$GM = \{N,B,Z,T,L\}$$

where: N is the *pd-g* node set; B is the *pd-g* edge set; Z is the program non-intersecting one-in/one-out structure set; T is the partial ordering on Z describing the Z structure nesting; $L = \{d\text{-}g_o, \ldots, d\text{-}g_m\}$ is a set of d-graphs describing the program according to different abstraction levels. L is obtained as follows. Let $z' \in Z$ be a structure directly preceding $z \in Z$ according to T and $d\text{-}g(z)$ be the z d-graph whose nodes either represent z' structures, if any, or belong to N. Assigning $d\text{-}g_o$ equal to the node representing the structure 'program', $d\text{-}g_i$ is obtained from $d\text{-}g_{i-1}$ by replacing each node representing a structure z by $d\text{-}g(z)$. Obviously $d\text{-}g_m$ describes the program at the lowest level having $d\text{-}g_m = pd\text{-}g$.

It should be noted that all the $d\text{-}g_i$s hold the abovementioned *pd-g* characteristics, and for each $d\text{-}g(z)$ there exists a unique $d\text{-}g_i$ of which that $d\text{-}g(z)$ is a subgraph. Such subgraph is connected and has only one input node $i \in d\text{-}g(z)$ and only one output node $o \notin d\text{-}g(z)$.

For the sake of simplicity, hereinafter $z \in Z$ denotes indifferently both the structure z and the d-graph $d\text{-}g(z)$. Moreover let us introduce the notations:

$N(z) = \{n_1, \ldots, n_k\}$: the set of nodes belonging to z;
$b(n_i,n_j)$: the direct edge from n_i to n_j;
prec (n_i): an n_j linked to n_i by b (n_j, n_i);
next (n_i): an n_j linked to n_i by b (n_i, n_j);

$E(z)$: the z exit nodes, i.e. the $N(z)$ subset of nodes linked to o.

Further, $z(i, o)$ and $z[n_1, \ldots, n_k]$ point out both a structure z and, respectively, the z input/output nodes or the z nodes.

Generally, a structure z has paths and circuits for which it is useful to introduce some definitions and notations used later on.

If a set of z edges can be ordered in the form $b(n_1, n_2)$, $b(n_2, n_3)$, $\ldots$, $b(n_{j1}, n_j)$ and all the nodes are distinct, then the sets of nodes and edges form an elementary ordered path μe having n_1 as input node and n_j as exit node. It is important to point that according to this definition, path nodes may not be revisited. Let us denote:

$N(\mu e)$: the set of nodes belonging to μe;

$\mu e[n_1, \ldots, n_j]$: the μe from the input node n_1 to the exit node n_j throughout $n_2, \ldots, n_{j-1}$;

$\mu e(n_i, n_j)$: a μe from n_i to prec (n_j).

A μe (i, o) is said to be a total μe; the notation μe (z) points out the set of z′ total μes.

If a set of z edges can be ordered in the form $b(n_1, n_2)$, $b(n_2, n_3)$, $\ldots$, $b(n_{j-1}, n_j)$ where the terminal nodes n_1, n_j coincide and the remaining nodes are distinct, then the sets of distinct nodes and edges form a circuit Γ. If a structure z contains a circuit Γ, then it is always possible to identify two non-empty subsets $H(\Gamma)$ and $E(\Gamma)$ of Γ nodes whose components are respectively the end-point and the start-point of edges not belonging to Γ. $H(\Gamma)$ and $E(\Gamma)$ are called Γ entry and, respectively, Γ exit node set; they need not be disjoint. In detail, we shall denote:

$N(\Gamma)$: the set of nodes belong to Γ;

$\Gamma[n_1, \ldots, n_j]$: the Γ from $n_i \in N(\Gamma)$ to n_i throughout $n_{i+1}, \ldots, n_j, n_1, \ldots, n_i$;

$H_i(\Gamma)$: an $H(\Gamma)$ element;

$E_i(\Gamma)$: an $E(\Gamma)$ element.

A Γ and a pair $H_i(\Gamma)$, $E_j(\Gamma)$ define an elementary ordered cycle γ. Particularly, we let:

$H(\gamma) = H_i(\Gamma)$, the γ entry node;

$E(\gamma) = E_j(\Gamma)$, the γ exit node;

$\Gamma(z)$ the set of z circuits;

$\gamma(z)$ the set of z elementary ordered cycles.

It should be noted that if h and k represent the order of $H(\Gamma)$ and $E(\Gamma)$, the triplet $\{\Gamma, H(\Gamma), E(\Gamma)\}$ identifies $h \times k$ elementary ordered cycles γs. We shall name *multi-entry* the one Γ with $h > 1$ and *multi-exit* the one Γ with $k > 1$. As a general rule, we will use $(h, k)\Gamma$ to highlight the number of the Γ entry and exit nodes, so that a multi-entry Γ is a $(> 1, \geqslant 1)\Gamma$ and a multi-exit Γ is a $(\geqslant 1, > 1)\Gamma$. Apparently, a $(1,1)\Gamma$ defines one and only one γ.

A GM is well formed when its elements z belong to one of the following six classes of d-graphs with input node i:

$\sigma[i, m_1, \ldots, m_k]$:	$E(\sigma) = \{m_k\}$;	$\mu e(\sigma) = \{\mu e[i, m_1, \ldots, m_k]\}$;	$\Gamma(\sigma) = \phi$
$\Omega[i, a]$:	$E(\Omega) = \{i\}$;	$\mu e(\Omega) = \{\mu e[i]\}$;	$\Gamma(\Omega) = \{\Gamma[i, a]\}$
$\Phi[i, a]$:	$E(\Phi) = \{a\}$;	$\mu e(\Phi) = \{\mu e[i, a]\}$;	$\Gamma(\Phi) = \{\Gamma[i, a]\}$
$\lambda[i, a, b]$:	$E(\lambda) = \{a\}$;	$\mu e(\lambda) = \{\mu e[i, a]\}$;	$\Gamma(\lambda) = \{\Gamma[i, a, b]\}$
$\Delta[i, a, b]$:	$E(\Delta) = \{a, b\}$;	$\mu e(\Delta) = \{\mu e[i, a], \mu e[i, b]\}$;	$\Gamma(\Delta) = \phi$
$\neg[i, a]$:	$E(\Gamma) = \{i, a\}$;	$\mu e(\neg) = \{\mu e[i, a], \mu e[i]\}$;	$\Gamma(\neg) = \phi$

σ, Ω, Φ, λ, Δ, $\neg$ are respectively the d-graphs of the structures 'sequence', 'while-do', 'repeat-until', '$n+\frac{1}{2}$ loop', 'if-then-else', 'if-then' constituting the set TD_1. In the following $\{\sigma, \Omega, \Phi, \lambda, \Delta, \neg\}$ will also denote the TD_1 corresponding structures. Moreover, $s[i, a, b]$ will denote a structure $\Delta[i, a, b]$ which, under some conditions, can degenerate into $\neg[i, a]$ or $\neg[i, b]$.

Let us name undefined program structure (ups) or undefined subgraph structure (uss) the **Z** elements not belonging to the set TD_1.

A software tool drawing the GM from a program and recognising the **Z** usses has been implemented.[11]

3. THE BASIC UNDEFINED SUBGRAPH STRUCTURES

To convert usses into TD_1 structures, at first, a proper classification is required which should allow usses to be subdivided according to classes which should:

- offer correspondence with the conversion rules used;
- include the most usses commonly found in practice;
- be limited in number and characterised in a simple and non-ambiguous manner allowing an easy automatic uss classification.

Consequently, classes have to be defined on the basis of specific properties of their uss cycles and paths rather than the number and type of unstructured elements[12, 3, 2] existing in the usses.

A first classification for usses can be related to the absence or the presence of at least one cycle in the graph. For this reason, two classes can be identified, the first which groups the acyclic usses indicated by Δ-uss, and the second which gathers the cyclic usses, indicated by Φ-uss. For the first class, it is easy to prove that:

Theorem 1

A $z(i, o)$ is a Δ-uss if and only if:

$$\nexists \Gamma \in z \wedge \exists \mu e_1 \in \mu e(z) \wedge \exists \mu e_2 \in \mu e(z): \mu e_1 \bigcap \mu e_2 \neq i$$

It should be noted that, as above defined, a Δ-uss is an acyclic structure whose total elementary paths share at least a node other than the input node.

For the second class, the following theorem does exist:

Theorem 2

A $z(i, o)$ is a Φ-uss if and only if:

$$\exists (h, k)\Gamma \in z: h > 1 \vee k > 1$$

It should be noted that, as above defined, a Φ-uss contains at least one multi-entry or multi-exit Γ, therefore a Φ-uss is a cyclic structure having more than one γ with their nodes belonging to the same Γ.

Referring to the set Φ-uss, let us consider the Φ-uss subsets containing only one Γ and characterised as follows.

Φ-men (multi-entry)

$$\Gamma: (> 1, 1)\Gamma \wedge E(\text{uss}) = E(\Gamma)$$

Φ-mex (multi-exit)

$$\Gamma:(1,>1)\,\Gamma \wedge H(\Gamma)=i \wedge \forall(n_1,n_2)\in E^2(\Gamma): n_1 \neq n_2,$$
$$\mu e(\text{next}(n_1)\notin\Gamma, o)\bigcap \mu e(\text{next}(n_2)\notin\Gamma, o)=\phi$$

The definitions above introduced imply that an uss $\in$ Φ-men has a unique multi-entry, non-multi-exit Γ, and the Γ exit node coincides with the one uss exit node. On the contrary, the unique Γ of an uss $\in\Phi$-mex is multi-exit and non-multi-entry and it contains the uss input node. Moreover, no common node exists among all the elementary paths starting from Γ nodes and ending at the Φ-mex output node without crossing any other Γ node.

Moreover, let us consider the Φ-uss subsets containing only two circuits, Γ_1, Γ_2 and characterised as follows:

Φ-nest (nesting)

$$\Gamma_1 \neq \Gamma_2 \wedge (\Gamma_1:(2,2)\,\Gamma) \wedge N(\text{uss}) = N(\Gamma_1)$$

Φ-int (intersecting)

$$(\Gamma_1:(>1,1)\,\Gamma) \wedge (\Gamma_2:(1,2)\,\Gamma) \wedge H(\Gamma_2)\in N(\Gamma_1) \wedge$$
$$E_1(\Gamma_2)=E(\Gamma_1) \wedge E_2(\Gamma_2)\notin N(\Gamma_1)$$

Φ-par (parallel)

$$(\Gamma_1:(1,>1)\,\Gamma) \wedge (\Gamma_2:(1,>1)\,\Gamma) \wedge$$
$$H(\Gamma_1)=H(\Gamma_2)\equiv N(\Gamma_1)\bigcap N(\Gamma_2)=i \wedge$$
$$\forall(n_1,n_2)\in(E(\Gamma_1)\bigcup E(\Gamma_2))^2: n_1\neq n_2,$$
$$\mu e(\text{next}(n_1)\notin\Gamma_1\bigcup\Gamma_2, o)\bigcap \mu e(\text{next}(n_2)\notin\Gamma_1\bigcup\Gamma_2, o)=\phi$$

It should be noted that, by such definition, an uss $\notin$ Φ-nest has one and only one exit node which coincides with one of the two Γ_1 exit nodes. Further, Γ_2 can be either a (1, 2) Γ or a (2, 1) Γ.

As far as the Φ-int and Φ-par d-graphs are concerned, the definitions state that an uss $\notin$ Φ-int offers a multi-entry, non-multi-exit circuit (Γ_1) and a two-exit, non-multi-entry circuit (Γ_2): these circuits are such that the entry node of Γ_2 belongs to Γ_1, while only one of the either exit nodes of Γ_2 belongs to Γ_1 and coincides with the Γ_1 exit node. Finally, an uss $\in$ Φ-par is obtained by putting together only the input and output nodes of two Φ-mex structures.

Examples of Δ-uss, Φ-men, Φ-nest, Φ-int, Φ-mex, Φ-par d-graphs are respectively shown in Figs. 1, 2, 3, 4, 5, 6 and 7.

We define BUSS (Basic Unstructured Subgraph Structure) as the d-graph set:

$$\text{BUSS} = \{\Delta\text{-uss}, \Phi\text{-men}, \Phi\text{-mex}, \Phi\text{-nest}, \Phi\text{-int}, \Phi\text{-par}\}$$

and BUPS (Basic Unstructured Program Structure) the corresponding program structure set.

It should be noted that all the seven basic unstructured one-in/one-out structures defined in the literature are included in BUSS. In particular, the four basic structures independently defined by McCabe[12] and Williams[13] belong to the subsets Δ-uss, Φ-men, Φ-mex, Φ-int respectively; the two structures defined by Oulsnam[3] belong to Φ-nest, and finally Φ-par includes the Williams' parallel loops.[13]

A software tool which is able to recognize and classify the uss $\in$ BUSS of the GM of a program has been implemented.

4. STRONGLY EQUIVALENT CONVERSION FROM BUPS TO TD$_1$ STRUCTURES

Let z_1 and z_2 be two structures. z_1 is intended to be strongly equivalent to z_2($z_1 \equiv z_2$) if their operative sequences are identical, i.e.

$$z_1 \equiv z_2 \Leftrightarrow \mu e(z_1)=\mu e(z_2) \wedge \gamma(z_1)=\gamma(z_2)$$

If $z_1 \equiv z_2$ and z_2 derives from the conversion of z_1 such a conversion is said to be strongly equivalent.

Referring to Δ-uss, Φ-men, Φ-nest, Φ-int subsets the following theorems do exist.

Theorem 3

An uss $\in$ Δ-uss can be strongly equivalent converted into a nesting of s-type structures.

In fact, let us assume:

D is the uss two-exit node set;

$\forall d_k \in$ D and for each path k_j (j = 1, 2) branching out from d_k:

(*a*) t_{kj} represents the first two-exit node subsequent to d_k on the path k_j if it exists, otherwise t_{kj} represents the uss output node o;

(*b*) σ_{kj} is the sequence, possibly empty, μe (next $(d_k)\in k_j$, t_{kj});

S is the d-graph obtained from the uss replacing the subgraph $z(d_k, o)$ by the structure $s[d_k, b_{k1}, b_{k2}]$ $\forall d_k \in$ D, where:

$$b_{kj}=\sigma[\sigma_{kj}, z(t_{kj}, o)] \quad \text{for} \quad t_{kj}\neq o$$
$$b_{kj}=\sigma_{kj} \quad \text{for} \quad t_{kj}=o \quad (t_{kj}=o \wedge \sigma_{kj}=\phi \Rightarrow b_{kj}=\phi)$$

On this basis, it should be easy to prove that uss $\equiv$ S.

An example of uss $\in$ Δ-uss conversion is shown in Fig. 1.

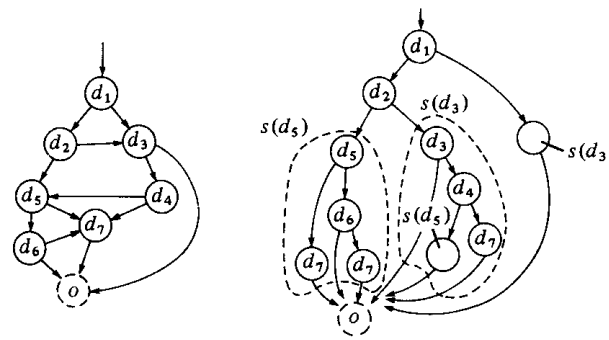

Figure 1. Example of uss $\in$ Δ-uss conversion.

Theorem 4

An uss $\in$ Φ-men can be strongly equivalent converted into a sequence of one s and one Ω structures.

In fact, having:

k = next $(E(\Gamma))$: $k \in N(\Gamma)$;

h the first entry node of Γ following $E(\Gamma)$;

g either $\mu e[k]$ for $k \neq h$ or the edge $b(E(\Gamma), h)$;

S_{21} the sequence coincident with the path μe $(k, E(\Gamma))$;

S_2 the structure Ω $[E(\Gamma), S_{21}]$;

S_1 the uss-connected component obtained from the uss suppressing g;

it should be easy to prove that:

S_1 is an s structure or it can be converted into an s structure by using Theorem 3;

the sequence S obtained from either S_1 or its conversion replacing $E(\Gamma)$ by S_2 is strongly equivalent to the uss.

An example of uss $\in \Phi$-men conversion is shown in Fig. 2.

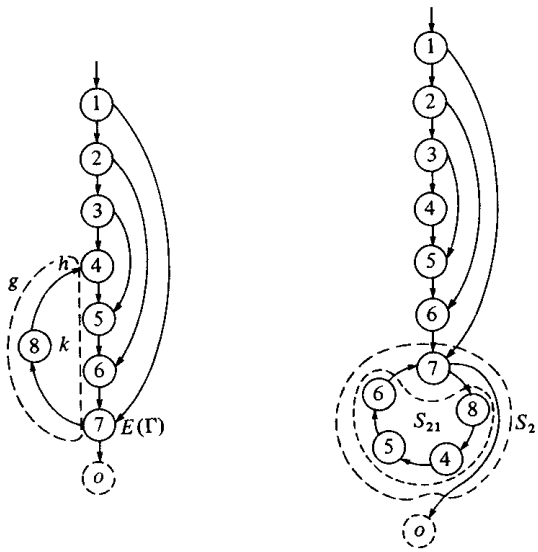

Figure 2. Example of uss $\in \Phi$-men conversion.

Theorem 5

An uss $\in \Phi$-nest can be strongly equivalent converted into a sequence of two structures. The first is σ or λ, the second is Ω with one s or Φ structure nested.

In fact having:

i, o respectively the uss input and output node;
$n \in E(\Gamma_1)$: next $(n) = o$;
$k =$ next (n): $k \in N(\Gamma_1)$;
h the first entry node of Γ_1 following n;
g either $\mu e[k]$ *for* $k \neq$ h or the edge $b(n, h)$;
S_{21} the uss one-in/one-out component $z(k, n)$ obtained suppressing the external entry edge on i;
S_2 the structure $\Omega[n, S_{21}]$;
S_1 the uss-connected component obtained suppressing g;

it should be easy to prove that

S_{21} is an s or Φ structure and S_1 is a simple sequence or a sequence with a nested loop respectively for Γ_2: (1, 2) Γ or Γ_2: (2, 1) Γ;
the sequence S obtained from S_1 replacing n by S_2 is strongly equivalent to the uss.

Examples of uss $\in \Phi$-nest conversions are shown in Figs. 3, 4.

Theorem 6

An uss $\in \Phi$-int can be strongly equivalent converted into a sequence of one s and two Ω structures.

In fact, let us assume:

i the uss input node;
$k =$ next $(E(\Gamma_1))$: $k \in N(\Gamma_1)$;
$w =$ next $(E(\Gamma_1))$: $w \in N(\Gamma_2)$;
$n \in E(\Gamma_2)$: $n \neq E(\Gamma_1)$;
$h =$ next (n): $h \in N(\Gamma_2)$;
μe_j *the* jth $\mu e(i, H(\Gamma_2))$;
S_1 the structure coincident with the μe_j path;
S_{21} the sequence coincident with $\mu e(H(\Gamma_2), E(\Gamma_1))$;
S_{221} the sequence coincident with $\mu e(k, E(\Gamma_1))$;
$S_{22} = \Omega\,[E(\Gamma_1), S_{221}]$;

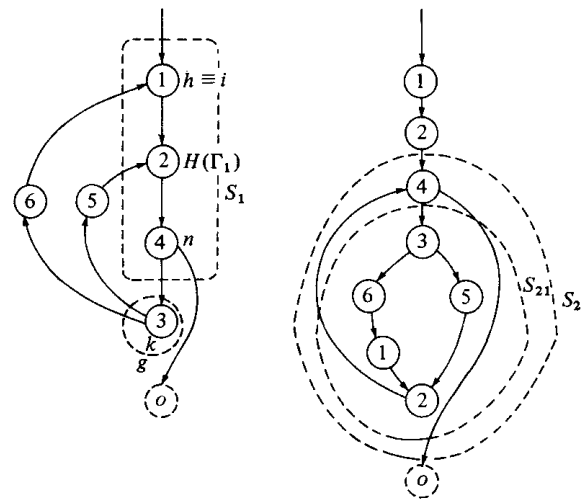

Figure 3. Example of uss $\in \Phi$-nest conversion (Γ_2: (1, 2)Γ)

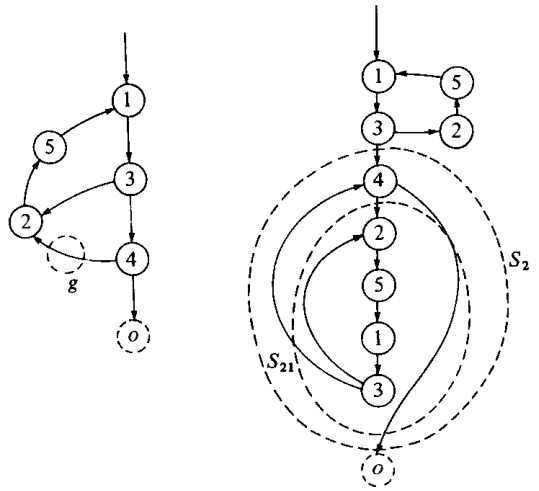

Figure 4. Example of uss $\in \Phi$-nest conversion (Γ_2: (2, 1) Γ)

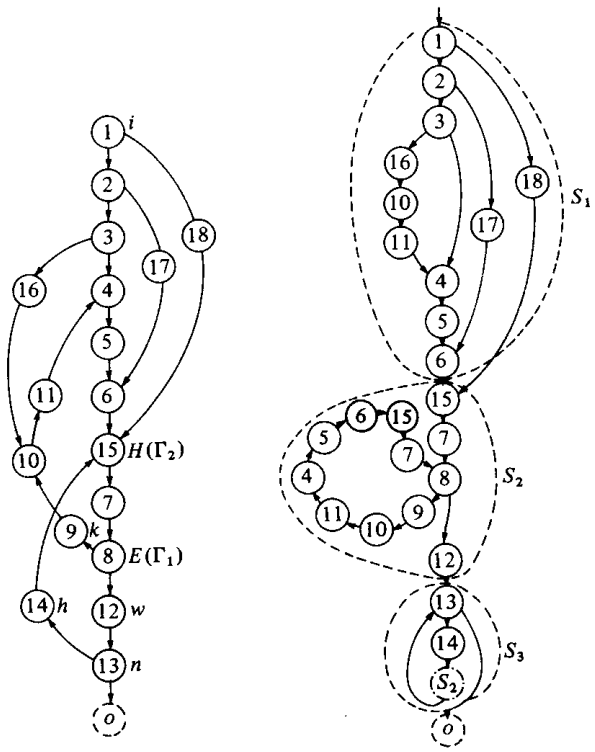

Figure 5. Example of uss $\in \Phi$-int conversion.

S_{23} the sequence $\mu e(w, n)$;
$S_2 = \sigma[S_{21}, S_{22}, S_{23}]$;
S_{311} the sequence of $\mu e(h, H(\Gamma_2))$;
$S_{31} = \sigma[S_{311}, S_2]$;
$S_3 = \Omega\ [n, S_{31}]$.

It should be easy to prove that:

S_1 is an s structure or it can be converted into an s structure by using Theorem 3;
the sequence $\sigma[S_1, S_2, S_3]$ is strongly equivalent to the uss.

An example of uss $\in \Phi$-int conversion is shown in Fig. 5. A software tool carrying out the abovementioned strongly equivalent conversions has been implemented.

5. WEAKLY EQUIVALENT CONVERSION FROM BUPS INTO TD$_1$ STRUCTURES

Let z_1 and z_2 be two structures. z_1 is intended to be weakly equivalent to $z_2 (z_1 \cong z_2)$ if their operative sequences only differ for: (i) assignments of logic constants to variables; (ii) tests on such variables. Therefore the z_1 and z_2 d-graphs are weakly equivalent if their elementary paths and cycles differ only for nodes of logic assertion (LT) or logic negation (LF) or predicative nodes (LP).

If $z_1 \cong z_2$ and z_2 derives from the conversion of z_1 then such a conversion is said to be weakly equivalent. Referring to the Φ-mex and Φ-par subsets the following theorem exists.

Theorem 7

An uss $\in \Phi$-mex $\bigcup$ Φ-par can be weakly equivalent converted into a repeat-until structure having the input node constituted by a sequence of one LF node and a nesting of s-type structures.

In fact let us assume:

D is the uss two-exit node set;
$\forall d_k \in D$ and for each path k_j $(j = 1, 2)$ branching out from d_k:
(*a*) t_{kj} represents the first two-exit node subsequent to d_k on the path k_j if it exists, otherwise t_{kj} represents the uss output node o;
(*b*) σ_{kj} is the sequence, possibly empty, $\mu e(\text{next}(d_k) \in k_j, t_{kj})$;
S_{11} is the nesting of s type structures obtained from uss replacing the subgraph $z(d_k, o)$ by the structure $s[d_k, b_{k1}, b_{k2}]\ \forall\ d_k \in D$, where:
$b_{kj} = \sigma[\sigma_{kj}, z(t_{kj}, o)]$ for $t_{kj} \neq o\ \wedge\ t_{kj} \neq i$
$b_{kj} = \sigma_{kj}$ for $t_{kj} = o\ \ (t_{kj} = o\ \wedge\ \sigma_{kj} = \phi \Rightarrow b_{kj} = \phi)$
$b_{kj} = \sigma[\sigma_{kj}, LT]$ for $t_{kj} = i$
S_1 is the sequence $\sigma[LF, S_{11}]$;
S is the repeat-until $\Phi[S_1, LP]$.

It should be easy to prove that: uss $\cong S$.

Examples of uss $\in \Phi$-mex and uss $\in \Phi$-par conversions are shown in Figs. 6, 7. A software tool carrying out the abovementioned weakly equivalent conversion has been implemented.

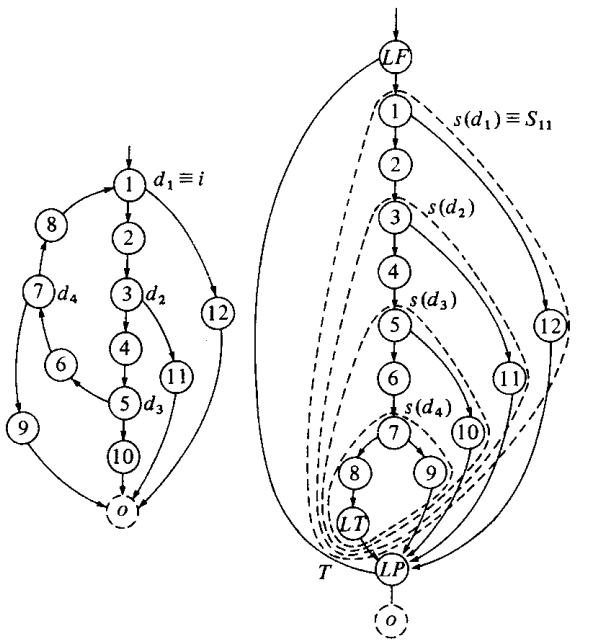

Figure 6. Example of uss $\in \Phi$-mex conversion.

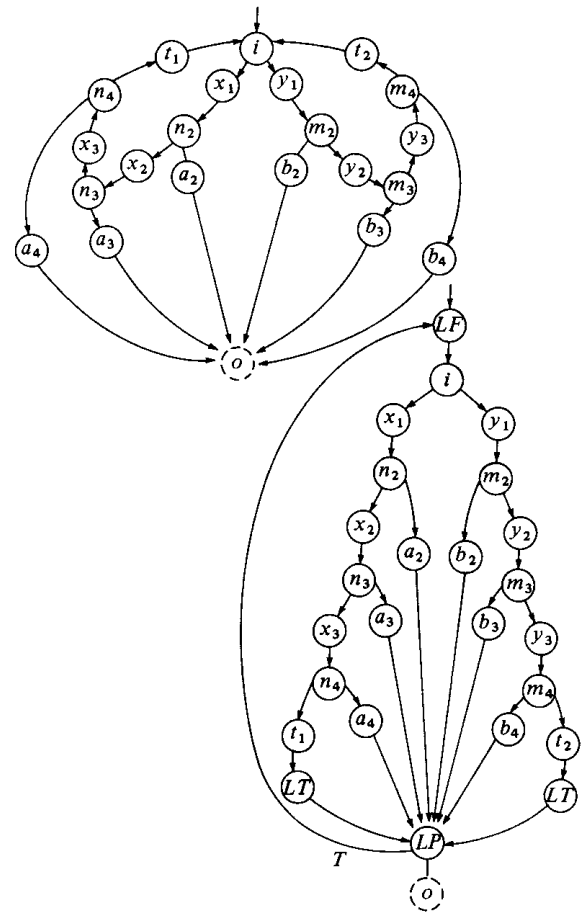

Figure 7. Example of uss $\in \Phi$-par conversion.

6. THE CONVERSION OF A GENERIC USS

Referring to subset Φ-uss the following theorem exists.

Theorem 8

An uss Φ-uss whose Γ_j are not multi-exit can be strongly equivalent converted into Ω structures nested into an s-type structure.

$\forall \Gamma_j \in$ uss; (let us assume):
$k_j = \text{next}\ (E(\Gamma_j))$: $k_j \in N(\Gamma_j)$;
h_j the first entry node of Γ_j following $E(\Gamma_j)$;
g_j either $\mu e[k_j]$ for $k_j \neq h_j$ or the edge $b(E(\Gamma_j), h_j)$;

S_{21j} the sequence coincident with the path $\mu e(k_j, E(\Gamma_j))$;
$S_{2j} = \Omega\,[E(\Gamma_j), S_{21j}]$.
S_1 the uss-connected component obtained by suppressing g_j $(\forall_j)$;

It should be easy to prove that:

S_1 is an s structure or it can be converted into an s structure by using Theorem 3; the structure S obtained from either S_1 or its conversion replacing the nodes $E(\Gamma_j)$ by the structures S_{2j} is strongly equivalent to the uss.

An example of cyclic and not multi-exit uss conversion is shown in Fig. 8. Obviously, Theorem 8 extends the set of usses which can be converted into TD_1 structures, yet the problem in its generality remains still unsolved. For this aim, let R be the total ordering on BUSS:

$\{\Phi\text{-par} > \Phi\text{-int} > \Phi\text{-nest} > \Phi\text{-mex} > \Phi\text{-men} > \Delta\text{-uss}\}$

and consider the procedures:

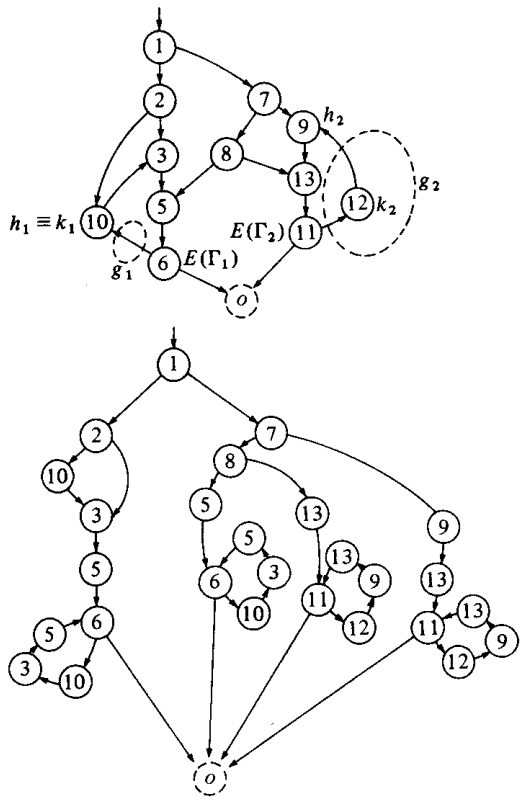

Figure 8. Example of cyclic and not multi-exit uss conversion.

extract (*uss*, *buss*, *g*)

if uss $\notin$ BUSS, one or more disconnections are performed on the graph uss in order to extract from the transformed graph a buss $\in$ BUSS structure. This structure is chosen among the maximum structures according to R. The procedure also defines the disconnected subgraph g drawn from uss by suppressing buss ($g = C^{\text{buss}}_{\text{uss}}$). On the other hand, if uss $\in$ BUSS, then the procedure returns buss $= \phi \wedge$ uss $= g$.

convert (*buss*, *ebuss*)

the procedure converts buss $\in$ BUSS into ebuss according to the above theorems.

merge (*ebuss*, *g*, *uss*)

the procedure draws the graph nuss by merging together ebuss and g graphs, under the hypothesis that g has been defined through *extract* (*uss*, *buss*, *g*) and ebuss through *convert* (*buss*, *ebuss*). In case of splitting in ebuss of buss nodes disconnected by *extract* the number of edges exiting from g is invariant with the merge process; conversely, the number of edges entering g increases. Hence, having:

$$\text{buss} \cong \text{ebuss}; \quad \text{g} = C^{\text{buss}}_{\text{uss}}$$

it leads to nuss $\cong$ uss.

Further the procedure searches the graph nuss for the presence of one-in/one-out BUSS or TD_1 subgraphs in order to replace them by a single node.

Then the procedure to convert an uss $\notin$ BUSS appears as:

begin
  extract (*uss*, *buss*, *g*)
  repeat
    convert (*buss*, *ebuss*)
    merge (*ebuss*, *g*, *nuss*)
    extract (*nuss*, *buss*, *g*)
  until *buss* $= \phi$
end

This procedure reaches convergence provided that the following conditions are satisfied: (i) the buss given by *extract* belong to BUSS subsets not increasing according to the ordering R and (ii) the number of graphs belonging to the same maximum subset is getting lower.

An example of uss $\notin$ BUSS conversion is shown in Fig. 9.

Conclusions

In this paper a set of unstructured one-in/one-out programming structures (BUPS) has been defined showing how BUPS elements can be converted into TD_1 structures. In addition, a procedure able to convert any one-in/one-out structure into BUPS or TD_1 structures has been proposed.

The BUPS structures include both the basic unstructured structures shown in literature and almost all the unstructured one-in/one-out structures commonly found in unstructured programs.

The conversion of BUPS elements into TD_1 structures is developed on the basis of a limited number of methods. In addition, it can be seen that methods proposed in theorems 4, 5 and 8 follow from a unique conversion criterium. Therefore, an ups $\in$ BUPS conversion can be reached by only four distinct criteria, one of them being able to convert any cyclic ups having no multi-exit cycle.

Naturally, the aim of defining conversion methods as generally as possible does not always lead to optimized structures. Indeed, it should be possible to define additional conversion rules of a particular BUPS subset leading to structures optimized in terms of memory used and/or executable speed and/or structure complexity.

The proposed methods outline the capability of strongly converting certain structures having multi-exit loops, i.e. Φ-nest and Φ-int structures. Therefore, some structures having unstructuredness due to a branch out of a loop exist which can be converted without the

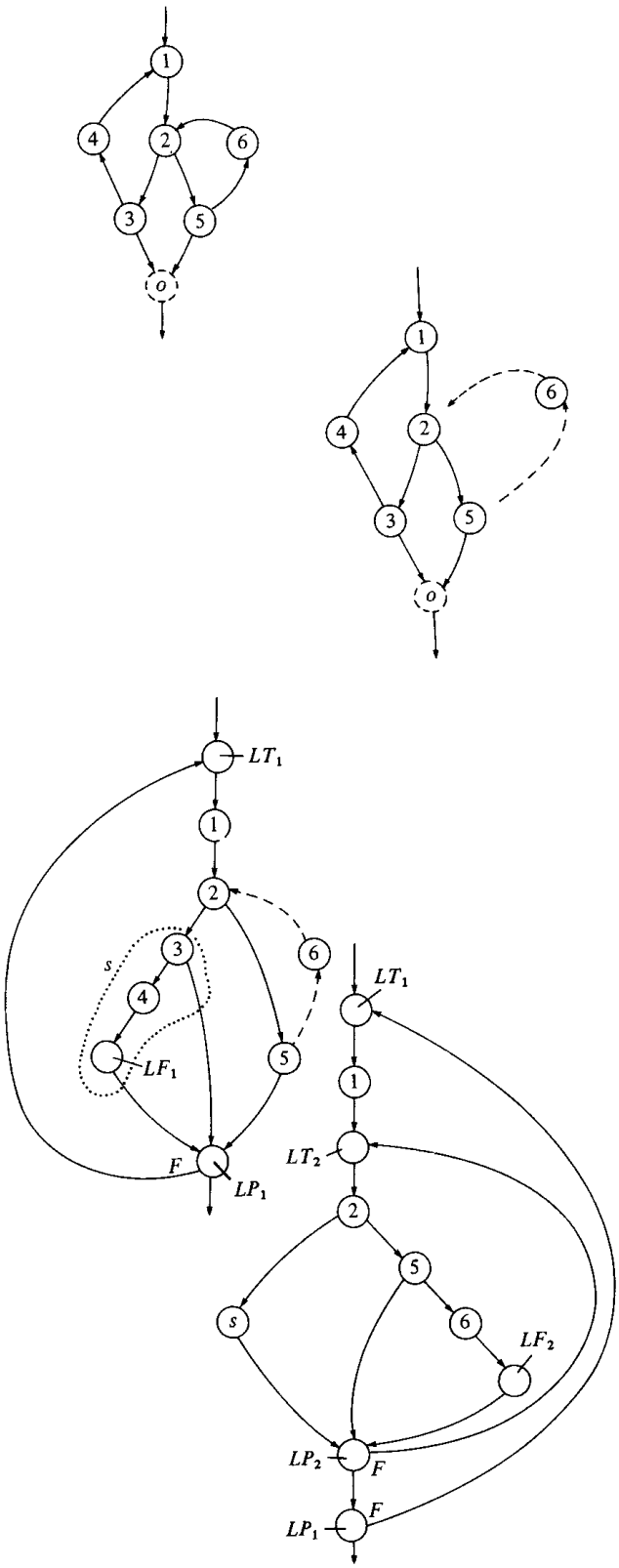

Figure 9. Example of uss ∉ BUSS conversion.

introduction of logical variables and predicates,[14] and without the predicate replication introducing auxiliary predicate variables.[3]

Conversion methods can be profitably used for (i) measuring the structural complexity of non-structured programs; (ii) program maintenance; (iii) software quality control.

Nowadays, the structured programming technique is widely diffused among software factories, therefore the presence of non-structured programs in that environment comes essentially from program maintenance operations. A tool implementing the conversion methods here described can be used to reveal and get rid of unstructuredness due to software manipulations and to measure its entropy increasing as well.[15]

The matter of non-structured software, indeed, becomes a major problem for those programs developed in mini/micro based environments, where the non-perfect compatibility of the structured programming with existing optimization criteria is the reason for a large amount of strongly non-structured programs. In this context, other than for maintenance, tools tailored according to the approach proposed in this paper seem to be well suited for software acquisition and certification, as a metering of the non-structuredness might constitute a valid quality control index.

As the authors have been outlining, the methods shown have been implemented in a fitting software system which has been largely employed both to examine the reasons of unstructuredness present in FORTRAN programs and to perform their conversion into structured equivalent programs. In particular, more than five hundred programs have been examined, revealing that for 98% of non-structured programs the reason was the presence of BUPS structures solely.

Acknowledgement

The authors wish to express their gratitude to Professor Bruno Fadini for the many stimulating discussions which gave a significant contribution to the development of this paper.

REFERENCES

1. C. Boehm and G. Jacopini, Flow diagrams, Turing machines, and languages with only two formation rules. *Communications of the ACM* **9** (5), 366–371 (1966).
2. M. H. Williams, Flowchart schemata and the problem of nomenclature. *The Computer Journal* **26** (3), 270–276 (1983).
3. G. Oulsnam, Unravelling unstructured programs. *The Computer Journal* **25** (3), 379–387 (1982).
4. S. B. Mohanty, Models and measurement for quality assessment of software. *ACM Computing Survey* **11** (3), 251–275 (1979).
5. P. Piwowarski, A nesting level complexity measures. *ACM Sigplan Notices* **17** (9) (1982).
6. G. Cantone, A. Cimitile and L. Sansone, Complexity in program schemes: the characteristic polynomial *ACM Sigplan Notices* **18** (3), 22–30 (1983).
7. G. Parikh, *Techniques of Program and System Maintenance*. Little, Brown, Boston (1982).
8. D. E. Knuth, Structured programming with goto statements. *ACM Computing Survey* **6** (4), 261–301 (1974).
9. H. Ledgard and N. Marcotty, A genealogy of control structures. *Communications of the ACM* **18** (11), 629–639 (1975).
10. G. Cantone, A. Cimitile and B. Fadini, Il grafo di un programma per problemi di analisi e ristrutturazione del software. *La Ricerca* **30** (2), 1–23 (1979).
11. G. Cantone, A. Cimitile and B. Fadini, Ristrutturazione ed analisi dei programmi. *Rivista di Informatica* **10** (1), 15–27 (1980).

12. T. J. McCabe, A complexity measure. *IEEE Transactions on Software Engineering* **SE-2** (4), 308–320 (1976).
13. M. H. Williams, Generating structured flow diagrams: the nature of unstructuredness. *The Computer Journal* **20** (1), 45–50 (1976).
14. M. H. Williams and H. L. Ossher, Conversion of unstructured flow diagrams to structured form. *The Computer Journal* **21** (2), 161–167 (1978).
15. E. C. Van Horn, Software must evolve. In *Software Engineering*, edited R. Freeman and R. Levis II, Academic Press, New York (1980). Pp. 209–226.