Quasi-Structured Programs

Jüri Kiho[1]

[1] Department of Computer Science, University of Tartu
e-mail: kiho@cs.ut.ee

Abstract

In order to increase productivity of program development, several diagramming techniques have been developed to compliment conventional pure-textual program representation methods. To handle real software, it is important to have tools for processing also non-structured programs. In this paper, the concept of quasi-structured programs and the corresponding formal notation are introduced. It is proved constructively that any reducible flowgraph can be represented by a quasi-structured program under the strong equivalence. On the other hand, there exists a quite natural way to represent quasi-structured programs in the form of most comprehensible diagrams. Therefore the concept may serve as a basis for further theoretical research as well as for developing effective tools for quasi-structured programming and software engineering.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs - *control primitives*;

Key Words and Phrases: Structured programming, reducible flowgraphs, diagramming techniques

1 Introduction

The concept of well-structured programming techniques, having roots in sixties and intensively developed in seventies, is still of great importance in the field of software engineering. This approach has proved especially fruitful as the basis for higher level design methods and tools. Decomposition of given complex problems into clearly nested one-entry-one-exit subproblems which, in turn, are decomposed in the same way, facilitates developing comprehensible and reliable software. However, it should be noted that attempts to extend this approach to lower level programming techniques have failed. The famous call for gotoless, i.e. well-structured programming style was not too inspiring. In fact, the proposed self-restriction to pure **if-then-else**, **while**, etc. control structures was never fully accepted by the programmers' community, nor by programming languages designers. All conventional and widely used imperative languages provide means (**goto** and alike) for jumping from a statement out to some outer level. Similar possibilities are found in many higher level (data base) systems too. Of course, source texts where such jumps are heavily used tend to be unclear and less reliable (see Figure 1). Under the pressure of well-structured programming ideology, such programs are often called "unstructured" or even "ugly". For this reason, even a skilled programmer who has developed a piece of highly effective and correct code using **goto**s may feel some embarrassment about the result. On the other hand, there may not exist a natural way to express the idea (i.e. flowgraph) of his/her algorithm in the well-structured manner. It has been proved [4] that an arbitrary flowgraph can be represented by means of one-entry-one-exit constructs as an algorithm which, in general, is only weakly equivalent to the original one. It means that the programmer should use some artificial tricks, such as duplicating some parts of code etc., which are not inherent to his idea of solving the task.

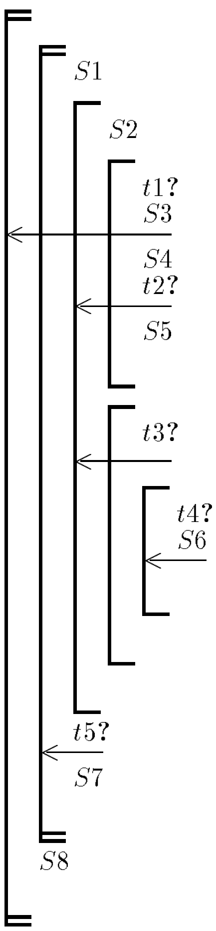
Actually, in most cases the incomprehensibility of the "unstructured" program is just due to the restricted expression power of textual representation form. It may be often overcome by choosing one of the appropriate graphical representation methods. A lot of such techniques have been developed [5], however, only a few enable to represent more than simply well-structured algorithms and, at the same time, serve as real programming tools. One example of the suitable diagramming techniques is the so-called sketchy programming method developed in Tartu [2]. In Figure 1 the sketch corresponding to the given piece of "ugly" Pascal-code is shown. Such a diagram is obviously much more comprehensible and there is no reason to call it unstructured. It should be mentioned that the sketch may be obtained automatically from the Pascal-code, and also automatically reduced to an even more simple (and strongly equivalent) sketch.

We shall use the sketchy notation also for algorithms in this paper.

Beside inherently "unstructured" algorithms leading to "unstructured" programs, another essential source of such programs come from software reverse engineering, when existing software (written in some lower level language) needs rewriting and restructuring for the purpose of reuse.

Another possible approach, as opposed to structured programming, is to consider non-structured (real) software as the "regular" one and well-structured programs rather as exceptions. It involves the need for the corresponding research and development of appropriate (apparently non-textual) tools. In this paper the concept of quasi-structured programs is introduced. It has been developed as a generalization of the sketchy representation form and is more suitable for theoretical analysis. Instead of several sketch types only one type of "building" blocks is used – the simple sketch where also weak arrows are allowed. Accordingly, the syntax of quasi-structured programs is extremely simple, and the semantics can be easily formalized. On the basis of this concept and corresponding theoretical results, several applied systems may be derived, for instance, by defining specific control constructs as special cases of quasi-structured blocks.

A quasi-structured program consists of clearly nested blocks, but there is no restriction for jumping to any outer level block. Even more, such jumps (i.e. multilevel **exits**) are among the basic control features, together with some conditional jumps inside the blocks. Thus, the well-structured programs form the subclass of quasi-structured programs (where only the closest outer blocks are exited). As for expression power, it is possible to construct a quasi-structured program, strongly equivalent to a given reducible flowgraph, i.e flowgraph which does not contain loops with multiple entries [1]. The main theoretical result of this paper is the constructive proof of the fact.




```
repeat
   repeat
      S1;
      begin
         S2;
         begin
            if t1 then begin S3; goto L1 end;

            S4;
            if t2 then goto L2;

            S5;
         end;

         begin
            if t3 then goto L2
                else
            begin
               if t4 then begin S6; goto L3 end

            end; L3 :;

         end;

      end; L2 :;

      if t5 then break else S7

   until false;

   S8;
until false; L1 :;
```

Figure 1: An "ugly" Pascal-code (on the right) and its sketchy diagram (on the left)

2 Quasi-structured program representation

In this section we introduce a program representation method which enables to build programs, combining program primitives into the specific control structures. Each building construct as well as the whole program has a clearly structured nested-block form. However, jumps to outer level blocks, violating structuredness in the strict sense, are here present among regular control features. Hence, the programs are called "quasi-structured".

Focusing on the program control structures we suppose that other program elements (such as types, constants, variables, statements etc.) come from a certain *host language*. The basic element of quasi-structured programs is a host language expression together with an associated decision domain. The *decision domain* is a finite sequence of host language expressions, called the *choices*. We denote $x?\backslash c_1, c_2, \ldots, c_m\backslash$ a *primitive* where

x is a host language expression having the decision domain $(c_1, c_2, \ldots, c_m)$, $m \geq 0$. For better understanding, one can imagine that choices are just the labels at arrows leaving correspondig to x box in the program flowchart.

We assume that any primitive can be evaluated. Evaluation of $x?\backslash c_1, c_2, \ldots, c_m\backslash$ means making (possibly) some changes in the program state (according to x), inspection the decision domain and returning a choice number in $\{1, 2, \ldots, m\}$. The primitive with a single choice, $x?\backslash c_1\backslash$ returns always the value 1; we call it an *action* and denote simply x. The primitive with no choices (if $m = 0$), $x?\backslash\backslash$, cannot return any value, and is called a *stop action*. Other primitives, which have the number of choices $m > 1$ may return different values, so making real decisions, and are called the *conditions*. A condition $x?\backslash \mathbf{true}, \mathbf{false}\backslash$ is called a *Boolean condition*, and denoted $x?$, if x is a host language Boolean expression and the choice is 1 iff x. There exist, of course, conditions with more rich decision domains, for instance, $x?\backslash' A',' G',' X - W', others\backslash$, or $y?\backslash a < 10, a = 11 \vee a = 13, a > 25, b = 1\backslash$. Such conditions usually correspond to headers of **case** or **switch** statements, and the decision making mechanism depends much on the host language. In most cases, we omit the decision domain and explicitly (in parenthesis) give only the number of choices at a primitive, so $t?(m)$ denotes a condition which has m choices in its decision domain.

A *program segment* is a sequence of 0 or more *members*. The *control member* is either an arrow or a condition. The *executive member* is either an action (or empty statement Λ) or a block.

The *program block* or simply *block* is a program segment surrounded by special brackets. We shall use two forms of program representation: textual and graphical. Let Π be a program segment. Then corresponding block is textually represented by $[\Pi]$ and graphically by

Π

In the last case, Π is represented by a vertical sequence of its members.

A block B *is nested* in a block B', if B is a member of B' or there exists a block B'' such that B is nested in B'' and B'' is nested in B'. If a block B is nested in a block B' then B is called also the *inner block* of B', and B' is called the *outer block* of B.

Let $B = [b_1\, b_2\, \ldots\, b_n]$ be a block. By the definition, the member $b_i (1 \leq i \leq n)$ is either an action, or an arrow, or a condition, or a block. Quite naturally, we say that members of B, as well as members of its nested blocks "occur in" (or "belong to" or "are in") B; also, block B "contains" (or "has") b if b occurs in B. For instance, "block B contains no arrows" means that the block is arrow-free as well as any of its nested blocks. To emphasize that a certain action (or arrow or condition or block) occurs in B as its member, we call it "member action" ("member arrow", "member condition", "member block"). For instance, if "block B has no member arrow", B may contain arrows in some of its nested blocks anyway.

Definition 2.1 *A* **quasi-structured program** *is a block which is not included in any segment.*

The *levels* in a quasi-structured program are defined as follows: the program itself is at level 0 (has level number 0); if a block has level number l, then all its members are at level $l + 1$. When appropriate, the level number of a block may be explicitly shown as a superscript at a block bracket (we use it mainly at the opening bracket):

$P_1[^l\Pi]P_2$ denotes a quasi-structured program including a block $[\Pi]$ at level l. Note that in a general form of a quasi-structured program, the matching brakcets are either both hidden or both explicit. So, for instance, in the latter example any bracket in Π can have a match only in Π, while a brakcet in P_1 may have a match either in P_1 or in P_2.

There are two kinds of arrows: $\uparrow^e$ is called the *strong arrow*, or simply *arrow*, while $\uparrow^{(e)}$ stands for the *weak arrow*. Every *arrow* has its *endpoint level* e - a natural number. The endpoint level of any arrow at level l must be less than l. The formal syntax of block structure may be described as follows, given the set of predefined symbols $\{$*empty*, *unsigned_integer*, *primitive*$\}$.

$block ::= [segment]$
$segment ::= empty \mid member \mid segment\ member$
$member ::= primitive \mid arrow \mid block$
$arrow ::= strong\_arrow \mid weak\_arrow$
$strong\_arrow ::= \uparrow^{endpoint\_level}$
$weak\_arrow ::= \uparrow^{(endpoint\_level)}$
$endpoint\_level ::= unsigned\_integer$

In general, the quasi-structured program, as well as any of its nested blocks is supposed to be executed sequentially, member by member. This order can be changed only by control members (i.e. arrows and conditions), describing unconditional and conditional jumps. Their semantics will be briefly considered later on.

Execution of a (nonempty) block always starts with the execution of its first member. If it, in turn, is a block, execution of the first member of the latter starts etc., until a primitive occurs as a next "first member". In particular, it means that the first primitive to be executed in a block $[Ly\ldots]$ where L stands for 0 or more left brackets '[', is certainly the element y. When started (and not stopped by an stop action), the execution of a block B *leads* either (a) to the end or (b) to the beginning of block B or any of its outer blocks. In the case (a), i.e. if execution leads to the end of a block B we say that the block B *is finished.* Note that when started, execution of a block not necessarily leads to the end of this block, so the block can remain unfinished. It is assumed that an (non stop) action will be finished (when started). After finishing a member, execution of the next member at the same level starts. After finishing the last member of a block B, execution leads to the end of block B, i.e. B is also finished. There are other possibilities to finish a block, due to conditions and arrows. Program halts when it has been finished or a stop action in it has been executed.

Let $\uparrow^e$ be a member arrow of a block B in a quasi-structured program. *Execution of the strong arrow* $\uparrow^e$ finishes the outer block at level e.

Let $\uparrow^{(e)}$ be a weak member arrow of a block B at level l in a quasi-structured program. *Execution of the weak arrow* $\uparrow^{(e)}$ leads to the beginning of the outer block B' of level e, i.e. the execution continues from the first member of B'. In particular, $\uparrow^{(l)}$ continues from the beginning of B.

While (strong) arrows represent unconditional forward jumps (multilevel **exit**s), weak arrows correspond to unconditional backward jumps (multilevel **continue**s to form loops).

In the graphical form, the arrow with endpoint level e is represented by a left arrow, reaching the bracket of the outer block at level e. The weak arrow is shown by a similar, but dotted left arrow (see Figure 2).

Conditions allow to program conditional forward jumps. Let $t?(m)$ be a condition in a block $B = [\Pi_1\, t?(m)\, \Pi_2]$ where Π_1, Π_2 are some segments. *Execution of the condition*

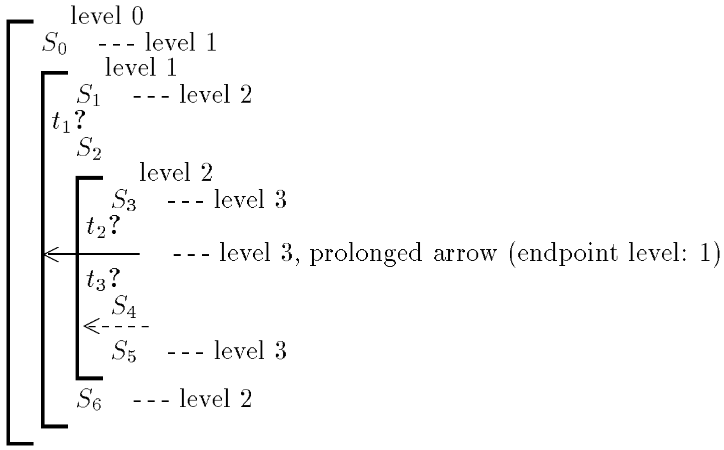


Figure 2: A quasi-structured diagram

$t?(m)$ is followed by a jump iff the choice number $i > 1$ has been returned. In that case, if there are less than $i-1$ member arrows in the block tail segment Π_2, following the condition, block B is finished, otherwise the execution continues after the $i-1$-th member arrow in its segment part Π_2.

In particular, this semantics is valid for actions as well. An action $x = x?(1)$ never causes a jump because the choice is always 1. In the special case of a Boolean condition $t?$ where t is a host language Boolean expression, the following simpler rule is obtained.

Evaluation of a Boolean condition to **false** causes a jump: if there are no member arrows in the tail segment of the block, the block is finished, otherwise the execution continues after the first member arrow in the tail segment of the block.

The meaning of conditions in quasi-structured programs is clearly specific. It may be formulated more precisely as follows.

Let $B = [\Pi_1 X\, \Pi_2]$ be a block where Π_1, Π_2 are some segments, X – the currently finished block or action, i – the choice value returned by X ($i = 0$ if X is a block). Let n be the number of member arrows in Π_2; in the case $n > 0$ we assume that that the member arrows in Π_2 are numbered $1, 2, \ldots, n$. The segment Π_2 is considered consisting of two subsegments, $\Pi_2 = \Pi_2'\Pi_2''$, where Π_2' is the initial part of Π_2 until the i-th member arrow (incl.); if $i = 0$ then Π_2' is empty and $\Pi_2'' = \Pi_2$, if $i > n$ then $\Pi_2' = \Pi_2$ and Π_2'' is empty. The control activity performed after finishing X depends on the length of Π_2: if Π_2 is empty then block B is finished, otherwise the first member of Π_2'' is executed.

However, as we shall see in the next section, the quasi-structured control methods are quite universal.

The condition $t?(m)$ in a segment $\Pi = \Pi_1 t?(m)\Pi_2$ is called *open*, if there is less than $m-1$ member arrows in the segment tail part Π_2, following the condition. Otherwise the condition is called *closed*. A segment Π is called *condition-closed*, if it does not contain any open member conditions. The concept of condition-closedness has an important role in theoretical reasoning about rules for automatic reducing of quasi-structured programs [3].

3 Block structure of reducible flowgraphs

The *block structure* of a given set X is a hierarchical partition of the set: a *block on* X is a finite linearly ordered set of members, where the *member* is either an element of X or a block. A block B is called a *block structure of* X, or simply *block of* X, if each element of X occurs in the block exactly once.

Elements of X which occur in a block B we call also elements of B, and the fact that $x \in X$ occurs in B denote by $x \in B$ too.

A block uniquely defines a linear order of its elements: it is the order in which the elements occur in the block. The first element of a block B is said to be *on top of* B.

To keep similarity to the program blocks from the previous section we represent a block as a list (string) of its members, included between block delimiters [/].

For instance, $B = [x[[y]t]z]$ and $B' = [[xy][zt]]$ are just two sample blocks of $X = \{x, y, z, t\}$. As we shall see later on, blocks of program primitives can be easily converted into corresponding program blocks by inserting some arrows (which implement jumps to the beginning or end of some outer blocks). Therefore it is reasonable to define *accessibility* in blocks as follows.

Definition 3.1 *Let B be a block of X. An element $y \in X$* **is accessible** *from an element $x \in X$ in B (denoted: $x \cdots\succ y$ in B), if y follows directly (except any number of brackets '[') after the opening or closing bracket of a block containing x, i.e B has the form*

$\ldots[\ldots x \ldots]Ly\ldots$

or

$\ldots[Ly\ldots x\ldots]\ldots,$

where L stands for null or more brackets '['.

For instance, in the example block B' above, x is accessible from every element, z is accessible from x and y, y and t are not accessible at all.

Lemma 3.1 *The first element in a block is accessible from every element in the block.*

Proof. Obvious.

Note that $x \cdots\succ x$ iff x is the first element in a block.

Definition 3.2 *Let X be a block structure of nodes in a graph (N, A) and A' the accessibility relation on N defined by block B. The condition $A \subseteq A'$ is called the* **conformity condition** *for the block structure of the graph nodes. A block B of nodes N is called a* **conformity block structure** *for graph (N, A) if the conformity condition holds.* In the following, we consider also blocks of a special form.

Definition 3.3 *Given a sequence of blocks $B_1, B_2, \ldots, B_k$ $(k \geq 0)$, the composed block*

$[[\ldots[[B_1]B_2]\ldots]B_k]$

is called the **topoblock** *and denoted by $tb(B_1, B_2, \ldots, B_k)$.*

Lemma 3.2 *In any topoblock $tb(B_1, B_2, \ldots, B_n)$ the first element in the block B_i $(1 \leq i \leq n)$ is accessible from every element in any previous block $B_j (1 \leq j \leq i)$.*

Proof. Obvious.

Lemma 3.3 *Let $G = (N, A)$ be an acyclic graph of blocks and $B = tb(B_1, B_2, \ldots, B_n)$ a topoblock of its nodes, where $(B_1, B_2, \ldots, B_n)$ is a topological order of N. For every arc $(B_i, B_j) \in A$, the first element in B_j is accessible from every element in B_i.*

Proof. Follows from Lemma 3.2, because if $(B_i, B_j) \in A$ then $i < j$.

Theorem 3.1 *For any acyclic graph (N, A) there exists a block of its nodes N such that the conformity condition holds.*

Proof. Let $(x_1, x_2, \ldots, x_n)$ be the sequence of nodes of G in a topological order. Then topoblock $[[[\ldots[[x_1]x_2]\ldots]x_{n-1}]x_n]$ satisfies the conformity condition.

Definition 3.4 *A* **flowgraph** *is a triple $G = (N, A, s)$, where (N, A) is a finite directed graph, and there is a path from the* **start node**, *$s \in N$, to every node.*

Definition 3.5 *Let $G = (N, A, s)$ be a flowgraph and let h be a node of G. The* **interval with header h**, *denoted by $I(h)$, is the block on nodes of G constructed as follows:*

$I(h) := [h]$

find a node x such that
$x \notin I(h)$ and $x \neq s$ and all arcs entering x leave nodes in $I(h)$;

x found **?**
$I(h) := [I(h)m]$

Lemma 3.4 *Interval $I(h)$ is a topoblock of its elements, and h is on top of it.*

Proof. Obvious from the interval construction.

Conventionally, interval is considered simply as a subset of nodes which, in our case, is just the set of nodes occurring in our "block interval". In spite of having an additional block structure for intervals, all the results obtained for "conventional" intervals are valid for the node sets of our "block intervals". In particular, we make use of the following lemma and theorem, which are from [1].

Lemma 3.5 *Let $G = (N, A, s)$ be a flowgraph, and let $I(h)$ be an interval of G. Every arc entering a node of the interval $I(h)$ from the outside enters the header h; that is, an interval is single-entry.*

Theorem 3.2 *There exists an algorithm which for any flowgraph constructs a set of disjoint intervals whose union is all the nodes in the flowgraph.*

In the following we consider derived flowgraphs, which nodes are blocks composed from the nodes of an (original) flowgraph.

Definition 3.6 *Let $G = (N, A, s)$ be a flowgraph. The sequence of* **derived flowgraphs** *$G_0 = (N_0, A_0, s_0), G_1 = (N_1, A_1, s_1), \ldots$ is called the* **derived sequence** *for the original flowgraph G if*

a) N_0 is the set of single-node blocks of elements N, i.e. $[x] \in N_0$ iff $x \in N$, and A_0 is corresponding set of arcs: $([x], [y]) \in A_0$ iff $(x, y) \in A$, and $s_0 = [s]$;

b) for $0 \leq i$, N_{i+1} is the disjoint set of intervals in G_i, and $(X, Y) \in A_{i+1}$ iff for each $X \neq Y$ and for some $x \in X$, $y \in Y$ there is an arc $(x, y) \in A$;

c) $s_{i+1} = I(s_i)$ *.*

We denote nodes of an original flowgraph $x, y, \ldots$ and nodes of derived flowgraphs $X, Y, \ldots$, i.e $X, Y, \ldots$ are blocks on the node set of the original flowgraph.

Lemma 3.6 *Let* $G_0 = (N_0, A_0, s_0), G_1 = (N_1, A_1, s_1), \ldots$ *be a derived sequence for* $G = (N, A, s)$.

a)The set of nodes N_i $(0 \leq i)$ *of any derived flowgraph defines a partition of the set* N*: every element of* N *occurs exactly once and exactly in one node of* N_i.

b) s *is on top of* s_i $(0 \leq i)$.

Proof. Inductive hypothesis: For the flowgraph G_i every element of N occurs exactly once and exactly in one node in N_i, and s is on top of s_i.

Basis: $(i = 0)$. Obvious for G_0.

Inductive step: $(i > 0)$. Assume the inductive hypothesis is true for $i - 1$, and consider the case for i.

a) Let x be an arbitrary node in N and X a node in N_{i-1} where x uniquely occurs (such a node X exists by the inductive assumption above). By Theorem 3.2, during the construction of the derived flowgraph G_i, the set of nodes of the previous derived flowgraph G_{i-1} is partitioned. In paticular, it means that the node X occurs exactly once and exactly in one block in N_i, and, consequetly, so does the element x.

b) By construction of interval, $s_i = I(s_{i-1})$ and s_{i-1} is included in s_i first of all. So the first element in s_{i-1} (which by the induction hypothesis is s) becomes the first element in s_i.

Lemma 3.7 *Let* $G_0 = (N_0, A_0, s_0), G_1 = (N_1, A_1, s_1), \ldots$ *be a derived sequence for* $G = (N, A, s)$. *For every derived flowgraph* G_i $(0 \leq i)$ *and every pair of its distinct nodes* $X, Y \in N_i$ *holds: every arc in G which leaves (a node belonging to)* X *and enters (a node belonging to)* Y*, enters the first node of* Y.

Proof. Inductive hypothesis: For the flowgraph G_i and every pair of its distinct nodes $X, Y \in N_i$ holds: any arc in G which leaves X and enters Y, enters the first node of Y.

Basis: $(i = 0)$. True, since in G_0 any node Y contains only a single node of G, which is also the first element of Y.

Inductive step: $(i > 0)$. Assume the inductive hypothesis is true for $i - 1$, and consider the case for i. Let X and Y be two distinct nodes in G_i, i.e intervals in G_{i-1}, and (x, y) be an arc in G which leaves X and enters Y, i.e. $x \in X, y \in Y$. We have to prove that the node y is the first element of Y. Since intervals are disjoint, in G_{i-1} there exist two distinct nodes, say X', Y' such that $x \in X', y \in Y'$. By the inductive hypothesis, the arc $(x, y) \in A$ enters the first node of Y'. In other words, y is on top of Y'. Due to the arc (x, y), there is also the arc (X', Y') in the derived flowgraph G_{i-1}. For the interval Y' it is an arc "from outside" and, therefore (by Lemma 3.5), Y' is the header of the interval Y. So, by the construction of interval, Y' is included into the block Y first of all, and its first node y becomes the first node in the block Y as well.

Definition 3.7
A flowgraph G *is called reducible if for some* $k \geq 0$ *the k-th flowgraph* G_k *in the derived sequence is a single node with no arc.*

Theorem 3.3 *For every reducible flowgraph* $G = (N, A, s)$ *there exists a conformity block structure of its node set* N.

Proof. Let $G_0 = (N_0, A_0, s_0), G_1 = (N_1, A_1, s_1), \ldots$ be a derived sequence for $G = (N, A, s)$. First, we prove that for every derived flowgraph $G_i (0 \leq i)$ holds: if X is a node in graph G_i and a pair of nodes $x, y \in X$ form an arc in G, $(x, y) \in A$, then $x \cdots\succ y$ in the block X.

Inductive hypothesis: for every node X of graph G_i, if a pair of nodes in X, $x, y \in X$, form an arc in G, $(x, y) \in A$, then $x \cdots\succ y$ in the block X.

Basis: $(i = 0)$. Every node of G_0 is a block $[x]$ of a single node where $x \in N$, and x is accessible from itself as the first element of a block. Therefore, if $(x, x) \in A$ (the only possible arc in the case), $x \cdots\succ x$.

Inductive step: $(i > 0)$. Assume the inductive hypothesis is true for G_{i-1}, and consider the case for G_i. Let Z be a node in G_i, x, y - any two elements of the block Z forming an arc in G (i.e. $x, y \in Z$, $(x, y) \in A$), and X, Y - nodes of G_{i-1} containing x, y correspondingly $(x \in X, y \in Y)$. Consider the following two cases:
a) $X = Y$. The elements x, y belong to the same node of G_{i-1}. By the inductive hypothesis, $x \cdots\succ y$ in the block X. Block X is included as a whole, i.e as a member into the interval Z, therefore $x \cdots\succ y$ in the block Z as well.
b) $X \neq Y$. The elements x, y belong to different nodes of G_{i-1}. Since $(x, y) \in A$, there exists an arc (X, Y) in G_{i-1}. By Lemma 3.7, y is on top of Y. Consider two subcases.

b1) Node Y is the header for the interval Z. Then Y is included into Z as the first member block. Therefore y is also the first element in Z and, by Lemma 3.1, $x \cdots\succ y$ in Z.

b2) Node Y is not the header of Z. By Lemma 3.2, Z is a topoblock. Due to the arc (X, Y) in G_{i-1}, Y follows X in any topological order of nodes in G_{i-1}, so $Z = tp(\ldots X, \ldots, Y \ldots)$. By Lemma 3.2, $x \cdots\succ y$ in Z, and this ends the induction.

Finally, since $G = (N, A, s)$ is reducible, there exists $k \geq 0$ such that G_k in the derived sequence has only one node, $G_k = (\{X\}, \emptyset, \{X\})$ where X is a block of N. Applying the inductive hypothesis for $i = k$ proves the theorem.

Usually flowgraphs are control flow models of real programs. It means that each node in a flowgraph corresponds to a primitive action or test and arcs define the execution order: nodes with outdegree 0 are the stop nodes, for any other node x one of its successors is to be choosen after the execution of x. So a flowgraph itself may also be considered as an abstract executable program. The execution starts from the initial node, and ends after a stop node has been executed.

We assume that nodes of such flowgraphs come from a set $\mathcal{P}$ of program primitives such as described in the previous section. Each element in $\mathcal{P}$ is a string of the form $t?(m)$. Of course, we can have (as much as needed) semantically equivalent elements in $\mathcal{P}$ which differ only by some negligible syntactic details, by the number of tail blanks, for instance. Thus, we need not to introduce graphs with labeled nodes.

Definition 3.8 *A flowgraph $G = (N, A, s)$ is called a* **program flowgraph** *if*

a) $N \subseteq \mathcal{P}$;

b) the outdegree of any node x in G is equal to the number m of choices in $x = t?(m)$;

c) each subset of arcs in A leaving the same node is linearly ordered, i.e. for any node $x = t?(m)$ the arcs leaving x are numbered $1, 2, \ldots, m$.

Definition 3.9 *Let P_1 and P_2 be two programs constructed from some primitives in $\mathcal{P}$. The programs P_1 and P_2 are called* **strongly equivalent** *if the set of primitives occurring in P_1 is the same as the set of primitives occurring in P_2 and the execution*

Algorithm $\mathcal{A}$.
Given: a program flowgraph $G = (N, A, s)$
Assumption: G is reducible
Result: corresponding to G quasi-structured program X
I. *Find the conformity block structure X for G*

II.
$\star\ \forall x, x \in N$

\- - - Let $x = t?(m)$ and $(y_1, y_2, \ldots, y_m)$ be the list of successors for x

III.
$\star\ \forall i, i \in \{1, 2, \ldots, m\}$

\- - - Since X is the conformity block structure there exists $l \geq 0$ such that
\- - - (a) $X = \ldots [^l \ldots [x] \ldots] Ly \ldots$
\- - - or
\- - - (b) $X = \ldots [^l Ly \ldots [x] \ldots] \ldots$

$case(a)$?
$\alpha_i :=\uparrow^l$ - - - in the case (a)
<- - - -
$\alpha_i :=\uparrow^{(l)}$ - - - in the case (b)

IV. *Replace $[x]$ in X by $[x\alpha_1\alpha_2 \ldots \alpha_m]$*

Figure 3: Algorithm for translating a flowgraph

paths (the sequences of executed primitives) of the programs are identical for every input.

Theorem 3.4 *Algorithm $\mathcal{A}$ constructs from a reducible flowgraph $G = (N, A, s)$ a quasi-structured program X which is strongly equivalent to G.* (See Figure 3)

Proof. First, $\mathcal{A}$ constructs a quasi-structured program. By Theorem 3.3, step I is executable and yields a quasi-structured program X (without any arrow in it). Any futher changes of X – insertions some member arrows after an element x in an inner block $[x]$, do not change the accessibility relation for primitives in X, nor violates the construction rules for quasi-structured programs. Hence, $\mathcal{A}$ constructs a quasi-structured program X. Also, after step I, block X contains all elements of N and only those. Since during the further construction no primitives are added to X or removed from it, the result X contains all elements of N and no other primitives.

Second, consider the execution of programs G and X for an arbitrary input. In G the execution starts from the initial node s. Block X is the single node of a derived flowgraph $(\{X\}, \emptyset, \{X\})$, therefore, by Lemma 3.6(b), s is on top of X, and is to be executed as the first primitive in X. So both programs start from the initial primitive s. Let $(x_1, x_2, \ldots, x_k)$ be a common (so far) execution path where $x_1 = s$ and $x_k (1 \leq k)$ is the currently executed element. If x_k is a stop action, executions of both programs stop at identical execution paths. Otherwise, $x_k = t?(m), m > 0$, and the choice $i \in \{1, 2, \ldots, m\}$, has been made. In the case of G it means that the next node to be executed (and added to the execution path) is node y_i from the successors list $(y_1, y_2, \ldots, y_m)$ of node x_k. In the case of X, the first element of a block $B = [x_k\alpha_1\alpha_2 \ldots \alpha_m]$ was executed last. After that, by definition of execution semantics for quasi-structured programs, the arrow α_i is executed. If $\alpha_i = \uparrow^l$ is a strong arrow then (case a in $\mathcal{A}$) $X = \ldots [^l \ldots [x\alpha_1\alpha_2 \ldots \alpha_m] \ldots] Ly_i \ldots$ and execution of the block of level l finishes, and following it next member (beginning with $Ly_i \ldots$) will be executed. So the next primitive executed (and added to the execution path) is y_i. If $\alpha_i = \uparrow^{(l)}$ is a weak arrow then (case b in $\mathcal{A}$) $X = \ldots [^l Ly_i \ldots [x\alpha_1\alpha_2 \ldots \alpha_m] \ldots] \ldots$ and execution continues from the first member (beginning with $Ly_i \ldots$) of the block of level l. So the next primitive executed (and added to the execution path) is y_i. Hence, the two programs continue always at the same execution path.

4 Conclusion

It has been shown that quasi-structured programs are rather universal, despite their extremely simple syntax and semantics. From practical point of view, the possibilities to represent program structure naturally by suitable graphical means are of great importance. For further theoretical research, formal notations introduced in this paper as well as results of analysis may be useful. Among the issues are: rules and algorithms for structural simplification, verification, parallel control structures etc.

Several instrumental tools may be developed to support quasi-structured approach. Many such tools are independent from host-languages and can be integrated into corresponding multi-language software systems. Also, since quasi-structured programs are closely related to reducible flowgraphs, a number of problems involving flowgraphs may be redefined in terms of quasi-structured programs. For instance, the problem of eliminating **goto**s from a code is reducible to the task of recompiling the code via quasi-structured representation.

References

[1] Hecht, M.S.: *Flow Analysis of Computer Programs*, Elsevier North-Holland, 1977.

[2] Kiho, J.: Diagramming techniques and sketchy programming, in *Proc. 2nd Symposium on Programing Languages and Software Tools.* Pirkkala, Finland. 1991, 78-82.

[3] Kiho, J.: Self-reducing control structures in the sketchy programming environment, in *Proc. Nordic Workshop on Programming Environment Research*, Lund, 1994. 333-334.

[4] Kosaraju, S.R.: Analysis of structured programs, *J. Comput. System Sci*, Vol. 9, Num. 3 (1974), 232-255.

[5] Tripp, L.L.: A bibliography on graphical program notations, *ACM SIGSOFT, Software Engineering Notes*, Vol. 14, Num. 6 (1989), 56-57.

Postal Addresses

Jüri Kiho
Department of Computer Science
University of Tartu
2, J.Liivi Street, Tartu, EE2400
Estonia