Designing a Program Multiviewer

Jüri Kiho and Asko Oja

Department of Computer Science, University of Tartu
Liivi str.2, Tartu, Estonia
e-mail: kiho@cs.ut.ee

Abstract. The idea of developing a program multiviewer (PMV) is introduced. PMV handles source texts of several types and provides multiple views for them. An important feature of PMV is the aid for automatic analysis and optimization of program structures. In the paper, a set of results about structure reductions for imperative languages is presented, and a general abstract structure for representing source texts is proposed.

1 Introduction

The need for different, multiple views to a program structure arises on many stages of software development process. Even a programmer "in-the-small" makes use of several representations of his/her program source code, at least after coming to a point where the usual textual representation becomes too messy and hardly understandable. For that purpose some more abstract and pictorial view is used, among which pseudocodes and flow-charts are the most traditional ones. The problem of getting a structural overview of a given piece of source code is inherent to software reuse and re-engineering activities. Also, it is of great importance for students and other novice programmers to have some graphical tool for representing the structure of the program under design. In this work, principles of a program multiviewer are outlined. In the following, the main characteristics of PMV are listed.

1. Transformation from text to picture (sketchifying).

2. Relative independence of the host language, in which elementary actions and declarations are written. The sketchy (keywordless) program representation [3] is suitable.

3. Possibility to represent and handle program flowgraphs. The quasi-structured program representation [6] is suitable.

4. Structure reduction [5], up to the theoretically minimal one.

5. Retranslation from picture to text (in original or some other language), or from picture to picture. It includes also output of pictures and texts into a typographic editor format (e.g. TEX). In particular, sketchifying and retranslation together provide a means for eliminating **goto**s [9].

6. Auxiliary tools for getting different display formats (suppress/reveal and compress/depress displayed program parts).

7. Tools for improving readability of texts (e.g. syntax highlighting, indentation)

8. Providing overall information (list of modules, procedures and functions, module dependency graphs).

Designing such PMV relates closely to more general problems of development heterogeneous systems [1], [7].

In the following, we focus on the topics related mainly to points 2-4 above.

2 Analysis of Quasi-Structured Programs

We consider a program representation method which enables to build programs, combining program primitives (assignments, procedure calls et al.) 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".

This approach, alternative to structured programming, allows to consider non-structured (real) software as the regular one and well-structured programs rather as exceptions. There exist natural graphical representation methods for quasi-structured programs, which enable to overcome inherent to textual languages incomprehensibility problem of non-structured programs.

In this section, some theoretical analysis of quasi-structured programs is presented (although, longer proofs are omitted). On this basis, sets of reducing rules can be derived for quasi-structured programs as well as for any other language, which semantics is described by means of quasi-structured constructs. Most of the results describe transformations of quasi-structured programs under very strong equivalence, i.e. the modifications exactly preserve not only the program (non-empty) primitives but also their static order. Possibilities of application of the theoretical results are illustrated by an example, where during the reduction of a given program, the maximum nesting depth changes from 5 to 2.

2.1 Quasi-Structured Programs

At present, most of the general-purpose programming languages as well as algorithmic languages (pseudocodes) are based on pure textual representation techniques. Although a number of efforts are made during the last decades [10], [11], no real breakthrough toward "industrial" graphical languages has not been detected. Among others, the following reasons of that should be mentioned: graphical tools are often too specific and different from ones used by programmers so far, restricted expression power (i.e. too strict requirements about structuredness), absence of theoretical reasoning for graphical representation techniques et al.

At the University of Tartu, a method for program visualization, called the sketchy programming [2], [3] was suggested in eighties. It has certain common features with Martin's action diagrams [8], suggested at about the same time. The main idea is to use elementary statements and declarations from a fixed (but arbitrary) textual language, and represent complex control and other constructs by certain simple graphical means. The practice of using corresponding experimental systems of sketchy programming (e.g. [4]) has proved the strength of this method. The efforts to implement this idea more widely have lead to the need for corresponding theoretical research. Lately the concept of quasi-structured program has been developed [6], which is a generalization of the notion of sketchy program.

In the following, some further research about quasi-structured programs is presented. Their semantics is defined more precisely, and a number of transformations of quasi-structured programs under very strong equivalence are described.

Quasi-structured programs have block structure, i.e. consist of clearly nested blocks. The formal syntax of block structure may be described as follows, given the set of predefined symbols {*empty*, *unsigned_integer*, *primitive*}.

$block ::= [segment]$

$segment ::= empty \mid member \mid segment\ member$

$member ::= primitive \mid arrow \mid block$

$arrow ::= strong\_arrow \mid weak\_arrow$

$strong\_arrow ::= \uparrow^{endpoint\_level}$

$weak\_arrow ::= \uparrow^{(endpoint\_level)}$

$endpoint\_level ::= unsigned\_integer$

The basic element of quasi-structured programs is the primitive: 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 corresponding 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

$$\left[\ \Pi\right.$$

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

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 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 brackets 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 bracket in P_1 may have a match either in P_1 or in P_2.

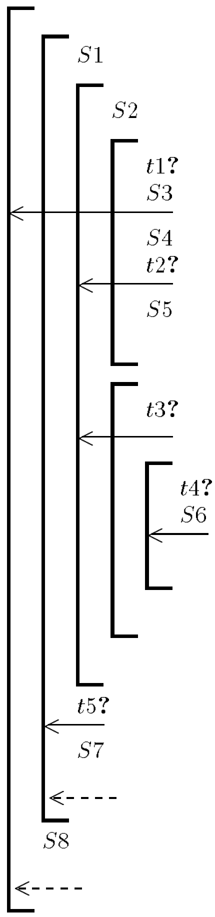
There are two kinds of arrows: $\uparrow^e$ is called the *strong arrow*, or simply *arrow*, while $\uparrow^{(e)}$ stands for the *weak arrow*. Every *arrow* has its *endpoint level* e - a natural number. The endpoint level of any arrow at level l must be less than l.

About the informal explanation of execution semantics of quasi-structured programs see [6]. In Fig. 1 the meaning of given quasi-structured program is explained by its Pascal equivalent.

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 Fig. 2). 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 analysis of quasi-structured programs.

2.2 Semantics of Quasi-Structured Programs

In this section we, first, introduce an idealized programming language called SKIP. SKIP programs have very simple syntax and intuitively clear semantics. Second, we define the procedure for translating any quasi-structured program




```
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 :;
```

Fig. 1. A quasi-structured program and corresponding to it Pascal code

into corresponding SKIP program. The further analysis of quasi-structured programs will be carried out from the point of view of corresponding SKIP programs: instead of the execution of quasi-structured programs the execution of corresponding SKIP programs is considered.

The *SKIP program* is a finite sequence of lines (commands), each of which has the form

a/s

where a is a primitive and s is an array of so-called *skip values*. If $a = x?(v)$ then the array s has v elements, $s = s_1, s_2, \ldots, s_v$.

The skip value in the i-th command is an integer number in the range $(1-i)\ldots(n-i)$, where n is program length (the number of commands).

The *execution* of a SKIP program starts from its first command. For the current (i-th) command

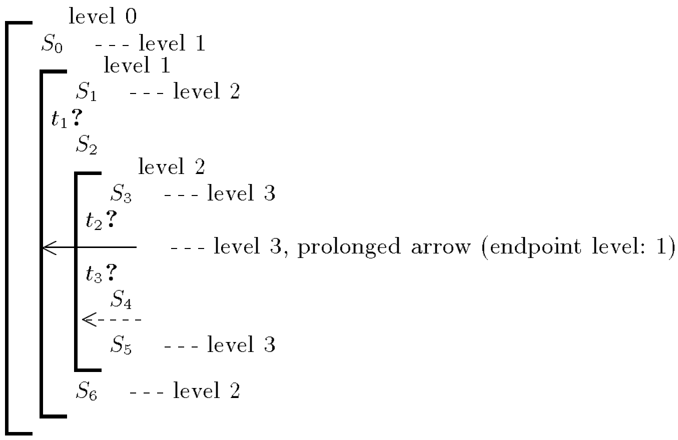


Fig. 2. The diagram for $[S_0[S_1\, t_1\, S_2[^2 S_3\, t_2 \uparrow^1\;\; t_3\, S_4 \uparrow^{(2)}\;\; S_5]S_6]]$

i)a/s

a is evaluated and the number j of the next current command is determined: $j = i + s_a$ (i.e. s_a lines will be skipped). The value $i + s_a$ is called the *scope* of the skip value s_a. The command

$\Lambda/1$

is called the *empty command*; it does not involve any operations and skips to the next line. The command a/s, where $a \neq \Lambda$, is the *non-empty-primitive command.*

After starting a SKIP program at an initial state (i.e. input), a sequence of primitives (including host language statements, transforming the program state) will be performed. The sequence of primitives performed by the SKIP program started at an initial state is called the *execution path* and the sequence of performed non-empty primitives is called the *computational history* of the program execution for that input.

Definition 2. We call two SKIP programs, SP_1 and SP_2 *equivalent*, if for any input their computational histories are identical.

The following two lemmas describe simple transformations of SKIP programs, obviously preserving their computational histories.

Lemma 3 (Remove/insert an empty command). *The SKIP program, which i-th line contains an empty command*

SP_1

i) $\Lambda/1$

SP_2

is equivalent to

SP_1'

SP_2'

where SP_1' is obtained from SP_1 by decreasing by 1 skip values of those commands, whose scope is greater than i and SP_2' is obtained from SP_2 by increasing by 1 skip values of those commands, whose scope is less than or equal to i.

Lemma 4 (Boolean negation). *The SKIP program of the form*

SP_1
t/m
Λ/n
SP_2
where t is a Boolean condition and $m \neq 1$, $n \neq 0$, is equivalent to
SP_1
$\neg t/(n+1)$
$\Lambda/(m-1)$
SP_2.

Definition 5. Two SKIP programs are called *strongly equivalent*, if they are equivalent and there exists one-to-one correspondence between their non-empty-primitive commands preserving primitives. If, in addition, the one-to-one correspondence preserves also linear order of commands, the two programs are called *very strongly equivalent.*

In this paper, mainly the transformations under very strong equivalence are considered. Therefore we use the term *equivalence* instead of *very strong equivalence.* Most of the transformations (in particular, those described in the lemmas above) obviously preserve linear order of non-empty-primitive commands, so it need not be proved specially.

Let P be a quasi-structured program. *Corresponding* to it SKIP program (denoted by $|P|$) is obtained from P by the following construction procedure.

1. P is written vertically in sequential lines so that each line contains either a bracket or a primitive or an arrow. Number of lines thus obtained is denoted by $\|P\|$.

Further, each line of P will be transformed into a SKIP command (steps 2 - 6).

2. Every bracket is replaced by an empty command $\Lambda/1$.

3. Every action a is replaced by the command $a/1$; in particular, the empty action Λ replaces by the empty command $\Lambda/1$.

4. Every arrow $\uparrow^l$ is replaced by the command
$\Lambda/(m-i)$
where i is the line number of the arrow and m is the line number of the end-bracket of the outer block of level l.

5. Every weak arrow $\uparrow^{(l)}$ is replaced by the command
$\Lambda/(i-m)$
where i is the line number of the arrow and m is the line number of the start-bracket of the outer block of level l.

6. Every condition $t?(v)$ at level l is replaced by the command
$t/s_1, s_2, \ldots, s_v$

where $s_1 = 1$ and the skip values $s_2, s_3, \ldots, s_v$ are defined as follows. Let i be the line number of the condition, m_j $(j > 0)$ – the line number of the j-th nearest arrow following the condition on the same level l ($m_j = +\infty$ if such arrow does not exists), and k – the line number of the end-bracket of the outer block of level $l-1$. Then $s_i = \min((m_{i-1} + 1 - i), k + 1 - i)$ $(i = 2, 3, \ldots, v)$.

Now we define the execution semantics of quasi-structured programs as follows: the *execution of a quasi-structured program* P means the execution of corresponding SKIP program $|P|$.

Therefore, two quasi-structured programs P_1 and P_2 are called *equivalent* (or *strongly* or *very strongly equivalent*) if corresponding SKIP programs $|P_1|$ and $|P_2|$ are equivalent (or *strongly* or *very strongly equivalent*).

Also, any segment Π in a quasi-structured program

P_1

Π

P_2,

where P_1 and P_2 are the initial and tail parts of the program, has corresponding to it group of commands $|\Pi|$ in the SKIP program. In formal expressions we will use mainly horizontal layout of the programs, using blanks as command delimiters. For instance, $P_1\, \Pi\, P_2$ instead of the program above.

Lemma 6 (Remove/insert empty statements). *Quasi-structured programs $P_1\, \Lambda\, P_2$ and $P_1\, P_2$ are equivalent.*

Proof. The corresponding SKIP programs, $SP_1\, \Lambda/1\, SP_2$
and

$SP_1'\, SP_2'$

satisfy the assumptions in Lemma 3 by the construction. □

The only necessary result we need about strong equivalence is the following theorem (not proved here):

Theorem 7 (Incorporate post-actions). *A quasi-structured program*
$P = P_1[^{l-1}\Pi_1[^l\, \Pi_2 \uparrow^l \Pi_3 \uparrow^f]\, \Pi_4\, \alpha]P_2$, *where α is an arrow, is strongly equivalent to* $P_1[^{l-1}\Pi_1[^l\, \Pi_2\, \Pi_4'\alpha\, \Pi_3 \uparrow^f]]P_2$ *where Π_4' is obtained from Π_4 by increasing by 2 endpoint levels of all arrows $\uparrow^e$ with $e > l$,*
if the following conditions hold simultaneously for P:

1^0. *Π_1 is condition-closed or Π_4 does not contain any member arrow.*

2^0. *$f \neq l$ and there is no arrows with endpoint level l in the segments Π_2 and Π_3.*

3^0. *The segment Π_3 is condition-closed.*

The *SKIP program flowgraph* is digraph which nodes are the commands in the SKIP program and there is an arc from node i)a/s to node j)a'/s' iff j is the scope for some skip value in s. A command in a SKIP program SP is called *unreachable* if it is unreachable from the node 1)a/s in the flowgraph. A segment Π in a quasi-structured program is called *unreachable* if all corresponding to it commands in the SKIP program are unreachable.

Theorem 8 (Remove an unreachable segment). *A quasi-structured program* $P = P_1[^l\, \Pi_1 \alpha\, \Pi_2]P_2$*, where* α *is an arrow, is equivalent to* $P' = P_1[^l\, \Pi_1\alpha]P_2$ *if* Π_1 *is condition-closed.*

Corollary 9. *A quasi-structured program* $P_1[\, \Pi_1\alpha_1\, \alpha_2\, \Pi_2]P_2$*, where* α_1 *and* α_2 *are arrows, is equivalent to* $P_1[\, \Pi_1\alpha_1\, \alpha_2]P_2$*, if* Π_1 *contains no other conditions rather than the Boolean ones.*

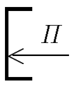
Proof. Follows from the fact that the segment $\Pi_1\alpha_1$ is condition-closed. □

2.3 Equivalent Transformations of Quasi-Structured Programs

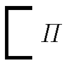
Theorem 10 (Remove/insert an unit-arrow at the end of a block). *Quasi-structured programs* $P = P_1[^l\, \Pi \uparrow^l]P_2$ *and* $P' = P_1[^l\, \Pi]P_2$ *are equivalent.*

Proof. By Lemma 6, P' is equivalent to $P'' = P_1[^l\, \Pi\Lambda]P_2$. By the construction, P and P'' have identical SKIP programs: $SP_1\, \Lambda/1\, |\Pi|\, \Lambda/1\, \Lambda/1\, SP_2$, where SP_1 corresponds to P_1 and SP_2 corresponds to P_2. □

In the graphical terms, Theorem 10 states that any block of the form



in a quasi-structured program may be replaced by



and vice verse.

Theorem 11 (Decrease/increase arrow length). *a) Quasi-structured programs* $P = P_1[\Pi_1[^l\, \Pi_2 \uparrow^l\, \Pi_3]\alpha\Pi_4]P_2$ *and* $P' = P_1[\Pi_1[^l\, \Pi_2\alpha\Pi_3]\alpha\Pi_4]P_2$ *where* α *is an arrow are equivalent.*

b) Quasi-structured programs $P = P_1[[^l\, \Pi_1 \uparrow^{(l)}\, \Pi_2]\Pi_3]P_2$ *and* $P' = P_1[[^l\, \Pi_1 \uparrow^{(l-1)}\, \Pi_2]\Pi_3]P_2$ *are equivalent.*

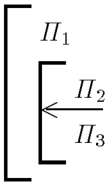
Corollary 12 (Prolong a unit arrow in tail block). *Quasi-structured programs* $P = P_1[\Pi_1[^l\, \Pi_2 \uparrow^l\, \Pi_3]]P_2$ *and* $P' = P_1[\Pi_1[^l\, \Pi_2 \uparrow^{l-1}\, \Pi_3]]P_2$ *are equivalent.*

Proof. By Theorem 10, P is equivalent to $P = P_1[\Pi_1[^l\, \Pi_2] \uparrow^{(l-1)}]P_2$.

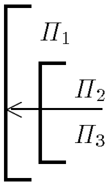
Corollary 13 (Prolong arrows in a single block). *In a quasi-structured program* $P_1[[^l\, \Pi]]P_2$ *any arrow with endpoint level* l *in the segment* Π *may be replaced by an arrow with endpoint level* $l - 1$*.*

Proof. Immediately from Theorem 11(b) and Corollary 12. □

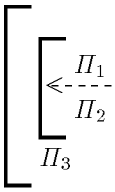
Graphically, Corollary 12 states that any block of the form



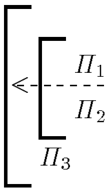
in a quasi-structured program may be replaced by



and vice verse; Theorem 11(b) states that any block of the form



in a quasi-structured program may be replaced by



and vice verse.

Let Π be a segment. By Π^- we denote the segment obtained form Π by omitting its last member; if Π is empty Π^- is also empty.

Theorem 14 (Close an open Boolean condition). *Quasi-structured programs* $P = P_1[^l\,\Pi_1\, t?\, \Pi_2]P_2$ *and* $P' = P_1[^l\,\Pi_1 \neg t?\ \uparrow^l\ \Pi_2]P_2$ *are equivalent, if* t *is a Boolean condition, and* Π_1 *is condition-closed and* Π_2^- *does not contain member arrows.*

Corollary 15 (Close all open Boolean conditions). *Quasi-structured programs*
$P = P_1\ [^l\,\Pi_0$
$t_1?\,S_1$
$t_2?\,S_2$
$\ldots$
$t_n?\,S_n$
$]P_2$
and

$P' = P_1\ [^l\, \Pi_0$
$\quad \neg t_1?\ \uparrow^l\ S_1$
$\quad \neg t_2?\ \uparrow^l\ S_2$
$\quad \ldots$
$\quad \neg t_n?\ \uparrow^l\ S_n$
$\quad]P_2$
are equivalent, if Π_0 is condition-closed and S_i $(i = 1 \ldots n-1)$, S_n^- do not contain member arrows and member conditions.

One way to simplify the structure of a quasi-structured program is to remove redundant pairs of brackets, i.e to decrease nesting depth when possible.

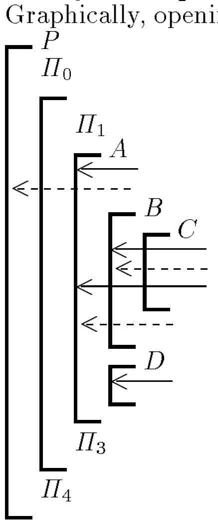
First, we introduce an operation of decreasing arrow endpoint levels in a segment. For a segment Π we denote $\Pi^{|l}$ the segment, obtained from Π by decreasing by 1 endpoint levels e of all arrows having $e \geq l$. The following lemma will be useful for further analysis.

Lemma 16. *Let Π be a segment and $l > 0$ an integer. If a segment Π' is obtained from the segment Π by decreasing by 1 endpoint levels l of some arrows in Π, then $\Pi'^{|l} = \Pi^{|l}$.*

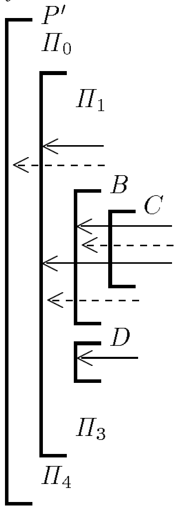
Proof. Immediately. $\square$

The general procedure of *opening a block* is defined as follows.

Let P be a quasi-structured program, which contains a block of level l, nested in another block of level $l-1$:
$\quad P = P_1[^{l-1}\Pi_1[^l\, \Pi_2]\, \Pi_3]P_2.$
To open the inner block means to construct the quasi-structured program
$\quad P' = P_1[^{l-1}\Pi_1 \Lambda\, \Pi_2^{|l} \Lambda\, \Pi_3]P_2.$
Note that all members of Π_2 become members (of level $l-1$) of the outer block. Also, by Lemma 6, P' is equivalent to
$\quad P_1[^{l-1}\Pi_1\, \Pi_2^{|l}\, \Pi_3]P_2.$
Graphically, opening the block A in



yields



In general, P is not equivalent to P'. There may be some differences in skip values of the corresponding SKIP programs SP and SP':

$SP = SP_1\, \Lambda/1\, |\Pi_1|\, \Lambda/1\, |\Pi_2|\, \Lambda/1\, |\Pi_3|\, \Lambda/1\, SP_2,$

$SP' = SP_1\, \Lambda/1\, |\Pi_1|' \Lambda/1\, |\Pi_2^{|l}|' \Lambda/1\, |\Pi_3|\, \Lambda/1\, SP_2.$

1) $|\Pi_1|$ and $|\Pi_1|'$ are different iff there is an opened condition in Π_1 (i.e Π_1 is not condition-closed) and Π_2 has an member arrow; in this case commands in SP and SP', corresponding to such condition have different skip values;
2) the skip values of commands in SP and SP', corresponding to an arrow with endpoint level e in Π_2, are different iff $e = l$;
3) $|\Pi_2|$ and $|\Pi_2^{|l}|$ are different also if there is an opened condition in Π_2 (i.e Π_2 is not condition-closed); in this case commands in SP and SP', corresponding to such condition have different skip values.

The above considerations serve as a proof of the following theorem.

Theorem 17 (Open a block). *A quasi-structured program*
$P = P_1[^{l-1}\Pi_1[^{l}\ \Pi_2]\ \Pi_3]P_2$ *is equivalent to* $P_1[^{l-1}\Pi_1\ \Pi_2^{|l}\ \Pi_3]P_2$
if the following conditions hold simultaneously for P*:*

1^0. *Π_1 is condition-closed or Π_2 has no member arrows.*

2^0. *There is no arrows with endpoint level l in the segment Π_2.*

3^0. *Π_2 is condition-closed.*

Corollary 18 (Open a tail block). *A quasi-structured program*
$P = P_1[^{l-1}\Pi_1[^{l}\ \Pi_2]]P_2$ *is equivalent to* $P_1[^{l-1}\Pi_1\ \Pi_2^{|l}]\,]P_2$
if P satisfies the conditions 1^0 and 2^{00}:

1^0. *Π_1 is condition-closed or Π_2^- has no member arrows.*

2^{00}. *In Π_2 there exists no arrow $\uparrow^{(l)}$ (i.e weak arrow with endpoint level l).*

Proof. Let $P = P_1[^{l-1}\Pi_1[^{l}\ \Pi_2]\,]P_2$.

If Π_2 does not contain any member arrow, Theorem 17 is applicable because all arrows $\uparrow^l$ in Π_2 may be replaced by $\uparrow^{l-1}$ (by Corollary 12). Let us consider the case $\Pi_2 = \Pi\alpha$, where α is an arrow.

Inserting into $P = P_1[^{l-1}\Pi_1[^l\ \Pi\alpha]]P_2$ a redundant (by Theorem 17) pair of brackets and a unit arrow at the end of inner block, we obtain an equivalent to P program $P' = P_1[^{l-1}\Pi_1[^l\ [\Pi]\alpha \uparrow^l]]P_2$. By Theorem 7, the latter is strongly equivalent to $P'' = P_1[^{l-1}\Pi_1[\uparrow^l\uparrow^l]\Pi^{|l}\alpha]]P_2$, where the inner block may be removed. Obviously, P'' is also strongly equivalent to P'. □

Corollary 19 (Open a head block). *A quasi-structured program*
$P = P_1[^{l-1}[^l\ \Pi_1]\ \Pi_2]P_2$ *is equivalent to* $P_1[^{l-1}\ \Pi_1^{|l}\ \Pi_2]P_2$.
if P *satisfies the conditions* 1^0 *and* 2^{00}*:*
1^{00}. Π_1 *is condition-closed.*
2^{00}. *In* Π_1 *there exists no arrow* $\uparrow^l$.

Proof. Let $P = P_1[^{l-1}[^l\ \Pi_1]\ \Pi_2]P_2$

By Lemma 6 and Theorem 11(b), P is equivalent to $P' = P_1[^{l-1}\Pi_0[^l\ \Pi_1']\ \Pi_2]P_2$ where Π_0 is empty and Π_1' is obtained from Π_1 by replacing all arrows $\uparrow^{(l)}$ by $\uparrow^{(l-1)}$. So, for Π' the conditions 1^0 - 3^0 of Theorem 4 hold:

1) Π_0 is condition-closed;
2) there is no arrows with endpoint level l in Π_1';
3) Π_1' is condition-closed.

Therefore P' is equivalent to
$P'' = P_1[^{l-1}\ \Pi_1'^{|l}\ \Pi_2]P_2$,
which by Lemma 16 is equal to $P_1[^{l-1}\ \Pi_1^{|l}\ \Pi_2]P_2$. □

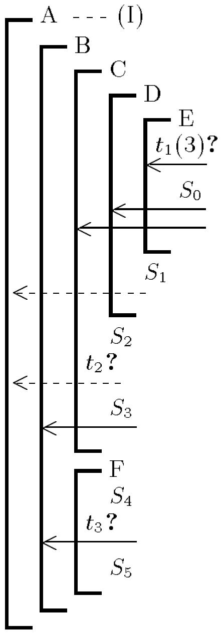
Corollary 20 (Open a single block). *A quasi-structured program*
$P_1[^{l-1}[^l\ \Pi]]P_2$ *is equivalent to* $P_1[^{l-1}\ \Pi^{|l}]P_2$.

Proof. Let $P = P_1[^{l-1}[^l\ \Pi]]P_2$.

Due to Lemma 6 and Corollary 12 P is equivalent to $P' = P_1[^{l-1}\Pi_0[^l\ \Pi']]P_2$ where Π_0 is empty and Π' is obtained from Π by replacing all arrows with endpoint level l by arrows with endpoint level $l-1$. P' satisfies the assumptions of Corollary 18, therefore it is equivalent to $P_1[^{l-1}\Pi_0\ \Pi'^{|l}]P_2$ and (by Lemma 6) also to $P_1[^{l-1}\ \Pi'^{|l}]P_2$. The latter is equal to $P_1[^{l-1}\ \Pi^{|l}]P_2$ by Lemma 16. □

2.4 Reduction by Transformations: an Example

Given a quasi-structured program in Fig. 3(I) we can reduce its structure by a sequence of equivalent transformations, obtaining much simpler program shown in Fig. 3(II). Note that the initial quasi-structured program may correspond to some piece of conventional textual code. Statement sequences are denoted by $S_1, \ldots, S_5$; t_2 and t_3 are some Boolean conditions, and $t_1(3)$ is a "three-fold" condition.



Block F is opened (Corollary 18), segment S_1 together with the following it weak arrow are incorporated into E (Theorem 7). Block E is opened (Theorem 17). The tail part of C following block D is incorporated into D, it is inserted just after segment S_0. Block D is opened. The unit-arrow at the end of block C is removed (Theorem 10), block C is opened (Theorem 17). Boolean condition t_3 is negated and the arrow next to it removed (Theorem 14). The arrow preceding S_4 is prolonged (Corollary 12) and block B is opened (Theorem 17).

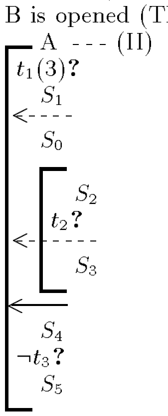


Fig. 3. Initial quasi-structured program (I), its final form after reduction (II).

2.5 Automatic Structure Reduction in PMV

As a result of theoretical analysis above a number of elementary transformations of quasi-structured programs under very strong equivalence has been established. Most of the results are rather simple, and have straightforward proofs. Nevertheless, for their application to practical reduction problems, appropriate component of PMV should be implemented.

If implemented, the quasi-structured representation method provides another extra view for source programs. Given a piece of software, the corresponding "sketchifier" automatically converts it into the quasi-structured format. The latter may be analyzed, modified, and retranslated into the original or some other language (or directly compiled). As for the real software, one of the main procedures to be applied to the sketchifier's output is control structure reduction. Usually, the source structure can be reduced drastically.

Note, that PMV should be able to display and print a source Pascal-code at least in three different modes; beside the textual and graphical form (as in Fig.1), the output in the scientific mode, for instance,
$[^0[^1 S1[^2 S2[^3 t1?\, S3 \uparrow^0 \;\; S4\ t2? \uparrow^2 \;\; S5][^3 t3? \uparrow^2 \; [^4 t4?\, S6 \uparrow^4]]]t5? \uparrow^1 \;\; S7 \uparrow^{(1)}]S8 \uparrow^{(0)}]$
is desirable.

3 Rules for Reducing Elementary Sketchy Programs

The *sketchy representation method* [2][3][4] for program control structures is quite similar to one, used in the case of quasi-structured programs. For the purpose of better comprehensiveness and closeness to the traditional programming paradigm, an additional specific representation for loops is introduced.

An acyclic block $[^l \Pi]$ where Π does not contain any weak arrow $\uparrow^{(l)}$ (at any level $\geq l$) is called the *simple sketch* and denoted by $(^l \Pi)$, or

Π

The weak arrows, continuing execution of such a block are forbidden: it is not allowed to modify $(^l \Pi)$ by insertion an arrow $\uparrow^{(l)}$ into its body. For blocks $[\Pi]$, which execution may be continued by at least one weak arrow $\uparrow^{(l)}$ in its body Π, another representation is used:
$[\!:^l \Pi :\!]$ (in textual form), or

Π

Such a block is called the *loop sketch* or the *free loop*. The semantics of $[\!: \Pi :\!]$ is slightly different from $[\Pi]$. It is assumed, that whenever the control reaches the end brace of the block, the execution continues from the first member of Π. More precisely, $[\!:^l \Pi :\!]$ denotes the block $[^l \Pi \uparrow^{(l)}]$.

Obviously, each *elementary sketchy program*, i.e. the program constructed from simple and loop sketches, is a quasi-structured program. The opposite is also true:

Theorem 21. *Any quasi-structured program can be represented by means of simple and loop sketches*

Proof. Let $P = P_1[^l\Pi]P_2$ be an arbitrary quasi-structured program and $[^l\Pi]$ a block in it. If there is no weak arrow $\uparrow^{(l)}$ in the block, it may be rewritten as (Π). Otherwise we insert three member arrows at the end of the block as follows: $P' = P_1[^l\Pi\ \uparrow^l\uparrow^l\uparrow^{(l)}]P_2$. Now the block $[^l\Pi\ \uparrow^l\uparrow^l\uparrow^{(l)}]$ in P' may be rewritten as a loop sketch: $[\!:\ \Pi\ \uparrow^l\uparrow^l :\!]$. On the other hand, P' is equivalent to P, on the basis of Theorem 8 and Theorem 10. $\square$

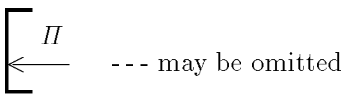
From the proof above, it is clear that even more strong proposition holds:

Theorem 22. *Any quasi-structured program can be represented by means of loop sketches*

Proof. Any block in a quasi-structured program may be complimented by the three arrows, as described in the proof of Theorem 21. $\square$

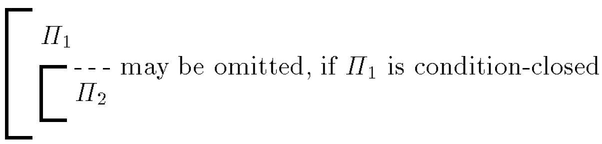
The following set of rules form the basis for the reducing algorithm for sketchy programs, where "$\Rightarrow$" stands for "is reducible to", α is an arrow and $\{\Pi\}$ denotes either $[\!:\ \Pi\ :\!]$ or (Π). The rules are illustrated by graphical sketches.

1^0 $(^l\Pi\ \uparrow^l) \Rightarrow (^l\Pi)$;



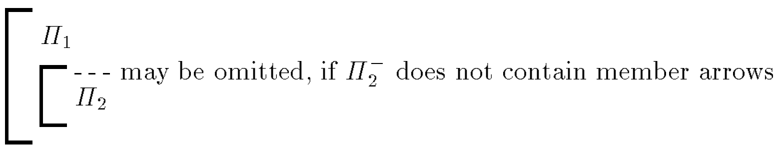
Immediately from Theorem 10.

2^0 $(^l\Pi_1(\Pi_2)) \Rightarrow (^l\Pi_1\Pi_2^{|l})$, if Π_1 is condition-closed;



This rule follows from Corollary 18. Note that condition 2^{00} in the corollary holds by the definition of simple sketch.

3^0 $(^l\Pi_1(\Pi_2)) \Rightarrow (^l\Pi_1\Pi_2^{|l})$, if Π_2^- does not contain member arrows;



Also from Corollary 18.

4^0 $P = [\!:^l\ \Pi_1(\Pi_2)\ :\!] \Rightarrow [\!:\ \Pi_1\Pi_2'^{|l}\ \uparrow^{(l)} :\!]$ where Π_2' is obtained from Π_2 by replacing all arrows $\uparrow^{l+1}$ by $\uparrow^{(l+1)}$, if Π_1 is condition-closed;

Π_1 - - - condition-closed

Π_2

may be reduced to

Π_1

Π'_2

<- - - -

By the definition of loop sketch and Theorem 10, P is equivalent to $P' = [^l \Pi_1(\Pi_2 \uparrow^{l+1}) \uparrow^{(l)}]$ and, by Theorem 11(a), to $P'' = [^l \Pi_1(\Pi''_2 \uparrow^{(l)}) \uparrow^{(l)}]$ where Π''_2 is obtained from Π_2 by replacing all arrows $\uparrow^{l+1}$ by $\uparrow^{(l)}$. By Theorem 17, P'' is equivalent to $[^l \Pi_1\, \Pi_2''^{|l} \uparrow^{(l)}\uparrow^{(l)}]$ which is $[\!:^l\ \Pi_1\, \Pi_2''^{|l} \uparrow^{(l)} :\!]$. By Lemma 16, $\Pi_2''^{|l} = \Pi_2'^{|l}$

5^0 $[\!:^l\ [\!:\ \Pi_1\ :\!]\Pi_2\ :\!] \Rightarrow [\!:^l\ (\Pi_1\ \uparrow^{(l)})\Pi_2\ :\!]$ where Π'_1 is obtained from Π_1 by replacing all arrows $\uparrow^{(l+1)}$ by $\uparrow^{(l)}$;

Π_1

Π_2

may be reduced to

Π'_1

<- - - - - -

Π_2

Immediately from Theorem 11(b).

Three following rules $(6^0, 7^0, 8^0)$ describe opening of an inner block:

- - - or a simple sketch

Π_1

- - - may be omitted, in some cases

Π_2

Π_3

Let S be a segment without member arrows and member conditions.

6^0 $\{^l \Pi_1(S)\alpha\Pi_2\} \Rightarrow \{^l \Pi_1\, S'^{|l}\ \alpha, \Pi_2\}$ where S' is obtained from S by replacing all arrows $\uparrow^{l+1}$ by α; in particular, $\{^l \Pi_1(S) \uparrow^l \Pi_2\} \Rightarrow \{^l \Pi_1\, S^{|l} \uparrow^l \Pi_2\}$.

Follows from Theorem 11(a) and Theorem 17.

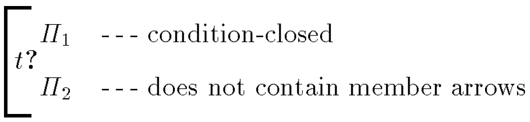
7^0 $\{^l \Pi_1(S)\Pi_2\} \Rightarrow \{^l \Pi_1\, S^{|l}\ \Pi_2\}$ if S does no contain any arrow $\uparrow^{l+1}$.

By Theorem 17.

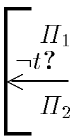
8^0 $\{^l \Pi_1 (\Pi_2) \Pi_3\} \Rightarrow \{^l \Pi_1 \Pi_2^{|l} \Pi_3\}$, if Π_1 and Π_2 are condition-closed and Π_2 does not contain any arrow $\uparrow^{l+1}$ (of any level);

Immediately from Theorem 17.

9^0 $\{\Pi\} \Rightarrow \{\Pi^c\}$, where Π is not condition-closed, Π^c is condition-closed and is obtained from Π by negating all open conditions and inserting arrows $\uparrow^l$ after them. For instance,



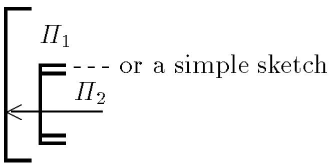
is equivalent to



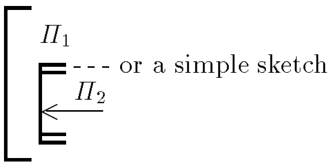
An application of Corollary 15.

Some additional rules for reducing arrow length in some special cases, detecting unreachable segments and infinity loops can be formulated. For instance,

10^0 $(\Pi_1\{\Pi_2 \uparrow^l\}) \Rightarrow (\Pi_1\{\Pi_2 \uparrow^{l+1}\})$;



may be reduced to



By Corollary 12.

4 The Abstract Structure of Sketchy Programs

Unlike ordinary text editors, instead of the program text PMV handles sketchy programs in a more abstract form. Corresponding to it abstract syntax is described briefly in the following. One of the main ideas is to keep the program concrete display (e. g. on the screen) apart from its full content (in the memory). Such approach enables to use easily multiple view-styles, according to the choice of screen-handler mode. In the general form, a sketch may have several branches, and the sketch, as well as any of its branches, may have a number of headings. So the elementary sketch is a special case with one branch and no headings.

General sketches can serve as representations of loops with headings, switches and parallel constructs.

The sketchy program is a sequence of sketches:

$S = \mathbf{root}(SL)$

$SL = \mathbf{sketch}(SI, SH, SB, SL)$

$\quad | \ \mathbf{empty}()$

The first component (SI) of a sketch defines its type and display mode(D) together with two comments (C), one for start-comment, the other for end-comment:

$SI = \mathbf{simple\_sketch}(D, C, C)$

$\quad | \ \mathbf{loop\_sketch}(D, C, C)$

$\quad | \ \mathbf{switch\_sketch}(D, C, C)$

$\quad | \ \mathbf{parallel\_sketch}(D, C, C)$

$C = \mathbf{comment}(text)$

$D = \mathbf{display\_mode}(text)$

Display mode is a string, encoding the display style options, such as level of suppression (fully or partly hidden), picture or line style, orientation etc.

Each sketch may have an arbitrary number of heads (headings), each of which has its type, display mode, comment and head text:

$SH = \mathbf{sketch\_head}(HI, HT, SH)$

$\quad | \ \mathbf{empty}()$

$HI = \mathbf{simple\_sketch\_head}(D, C)$

$\quad | \ \mathbf{loop\_sketch\_head}(D, C)$

$\quad | \ \mathbf{switch\_sketch\_head}(D, C)$

$\quad | \ \mathbf{parallel\_sketch\_head}(D, C)$

$HT = \mathbf{head\_text}(text)$

The main part of a sketch consists of a number of branches:

$SB = \mathbf{sketch\_branch}(BI, BH, BM, SB)$

$\quad | \ \mathbf{empty}()$

$BI = \mathbf{branch\_info}(D, C)$

A branch may also have several (or null) heads:

$BH = \mathbf{branch\_head}(BHI, BHT, BH)$

$\quad | \ \mathbf{empty}()$

$BHI = \mathbf{simple\_branch\_head}(D, C)$

$\quad | \ \mathbf{loop\_branch\_head}(D, C)$

$\quad | \ \mathbf{switch\_branch\_head}(D, C)$

$\quad | \ \mathbf{parallel\_branch\_head}(D, C)$

$BHT = \mathbf{branch\_head\_text}(text)$

The body of a branch is a list of elementary members and nested sketches:

$BM = \mathbf{elementary\_member}(EMI, EMT, BM)$

$\quad | \ \mathbf{sketchy\_member}(SL, BM)$

$\quad | \ \mathbf{empty}()$

$EMI = \mathbf{primitive}(D, C)$
$\mid \mathbf{condition}(D, C)$
$\mid \mathbf{strong\_arrow}(D, C)$
$\mid \mathbf{weak\_arrow}(D, C)$
$EMT = \mathbf{elementary\_member\_text}(text)$

Conditions and arrows are usually the means for representing control flow in an imperative programming host language.

References

1. Feather, M.,S.: Language support for the specification and development of composite systems. ACM Transactions on Programming Languages and Systems **9** 2 1987 198–234
2. Kiho, J.: Sketchy programming. Proc. Comp.Centre UT **50** 1984 52–68
3. Kiho, J.: Diagramming techniques and sketchy programming. Proc. 2nd Symposium on Programing Languages and Software Tools Pirkkala Finland 1991 78–82.
4. Kiho, J., Muts, M.: SKC: a C-based sketchy programming system. Proc. of Nordic Workshop on Program Environment Research 1992 Tampere Jan.8-10 1992 6p
5. Kiho, J.: Self-reducing control structures in the sketchy programming environment. Proc. Nordic Workshop on Programming Environment Research Lund 1994 333–334
6. Kiho, J. Quasi-structured programs. Proc. 4th Symposium on Programing Languages and Software Tools Visegrád Hungary 1995 June 8-14 67–78
7. Meyers, S.: Difficulties in integrating multiview development systems. IEEE Software **8** 1 1991 49–57
8. Martin, J., McClure, C.: Action diagrams: clearly structured program design Prentice Hall N.J. 1985 159 p
9. Ramshaw, L.: Eliminating goto's while preserving program structure. J.ACM **35** 4 1988 893–920
10. Tripp, L.L.: A survey of graphical notations for program design, an update ACM SIGSOFT Software Engineering Notes **13** 4 (1988) 39–44.
11. Tripp, L.L.: A bibliography on graphical program notations ACM SIGSOFT Software Engineering Notes **14** 6 (1989) 56–57