Science of Computer Programming 10 (1988) 247-280
North-Holland

A 'DIVISION' TRANSFORMATION FOR PROGRAM AND DATA STRUCTURES AND THE STRUCTURE CLASH PROBLEM

Beat MICHEL
Charles Veillon SA, P.O. Box 1486, CH-1001 Lausanne, Switzerland

Communicated M. Sintzoff
Received February 1986
Revised December 1987

Abstract. Motivated by the structure clash problem, this study examines certain formal transformations of data and program structures and relates them to the structure clash problem. It defines a *division* and a *decomposition* transformation of program and data structures with respect to one of its structure blocks. The latter allows a formal derivation of a new structure where this block appears at the beginning. The transformation is intuitively introduced for program and data structures. At the same time, it is mathematically treated in terms of regular algebra and is shown to be reflexive, symmetric and transitive. Thus, equivalence classes of regular expressions that are *decompositions* of each other may be defined. A formal realization of these is constructed as a type of circuitless graph.

1. Introduction

1.1. Data oriented program design methods

In 1966, Jacopini [4] showed that each program may be built with only three basic control structures: sequence, selection and iteration. Since then, the technique of nesting these base structures is known as structured programming. In the following discussion, we will speak of 'structured programs' in this exact sense.

The same three base structures may be used for the description of a data structure. A data element corresponds to the "operation" box of program structure. Sequence, selection and iteration define possible valid data element series, as read or written by a program.

J.-D. Warnier [13] and M. Jackson [11] have each developed a data oriented program design methodology which derives the logical structure of a program from the data structures it processes (see also Cohen [5] and Orr [12]). The two methodologies differ on one essential point:

- Warnier considers one consolidated input data structure (the "logical input file"), from which the program structure is derived.
- Jackson takes into account all input and output data structures and then tries to match the different structures on each level. The program structure is established according to this correspondence. For example, if we suppose that each page of

the list of Fig. 1(a) corresponds to one 'employee' record in the input file of Fig. 1(b), we can derive the program structure of Fig. 1(c). A graphical notation close to that of Cohen [5] is used. Figure 2 shows this notation of base structures and their correspondence to Jackson's notation.

1.2. The structure clash problem

Both approaches tend to produce similar results as long as all data structures can be brought into correspondence. However, there are situations where this is not possible. Jackson describes this as "structure clash". It is illustrated by the two structures of Figs. 1(a) and (d) which correspond on the highest ("report") and the

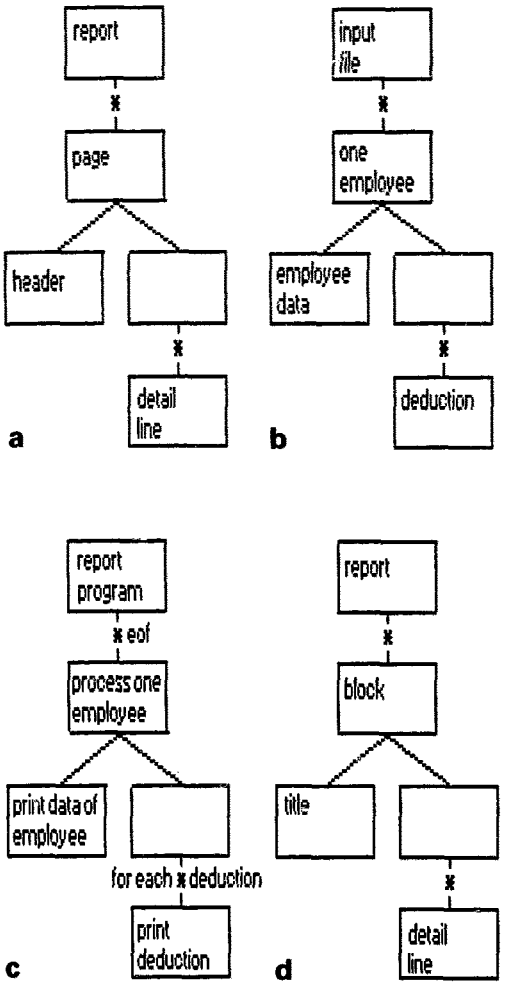


Fig. 1. (a) Page report structure, (b) logical input file structure, (c) corresponding program structure, (d) block report structure.

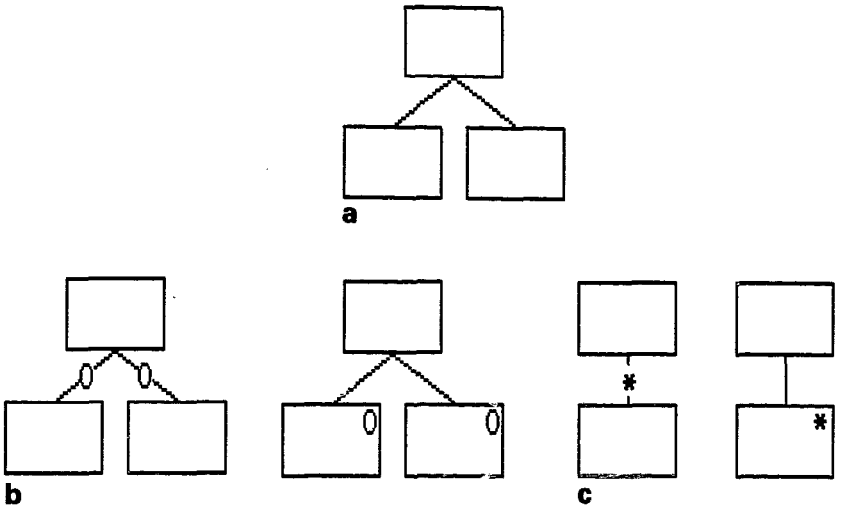


Fig. 2. Graphical notation for program and data structures (left: Cohen, right: Jackson): (a) sequence, (b) selection, (c) iteration.

lowest ("line") level, but on the intermediate level, page and logical block do not correspond to each other.

Jackson proposes the following solution to the structure clash problem:

(1) As there is no unique data structure, two programs must be written, one for each conflicting structure. Program A writes an intermediate file with block titles and detail lines. This file is read by program B, which does only physical page lay-out.

(2) To avoid an intermediate file, program A and B can be written as co-routines. At each place where the original program A would write to the intermediate file, co-routine A will suspend its operation and activate co-routine B.

The trouble with the co-routine mechanism is that the most commonly used programming languages and operating systems do not support it: SIMULA67 does, but COBOL, PL1, Pascal do not. UNIX supports this mechanism by the 'pipe' concept. Thus, where no such mechanism exists, a further step is needed:

(3) One program must be written as the main program, say A, while B is written as the subroutine that simulates a co-routine. To do this, the read instructions of program B must be replaced by a suspend point, using the following procedure:

- store a state variable identifying the suspend point,
- GO TO end of program,
- At program entry, GO TO the suspend point CORRESPONDING to the state variable.

Jackson calls this technique, using state variables and GOTO's to reach suspend points, "program inversion".

A mathematical formalization of Jackson's method and a discussion of its limits have been presented by Hughes [13]. Hughes describes input and output data structures by regular expressions and the correspondence between them by functions. The work of Durieux [7, 8], which is based on the same formalism of regular expressions, uses rational transductions to formalize the program. He shows that

constraints on input-output correspondence may be expressed by inference rules in the style of Gentzen. This makes it possible to formalize not only the result but also the procedure of Jackson's method.

1.3. Inversion vs. structured subtroutine

Arguments for and against program inversion are discussed in Cohen [5].

Some of the arguments in favor of the program inversion technique are:

- The subroutine still reflects the underlying data structure.
- It leads to the right solution by a strictly deterministic procedure.

Some of the main problems are:

- The use of GOTO's to leave and reach suspend points is not very satisfactory.
- If the suspend point resides within selection or iteration structures, these must be 'hand coded' by GOTO's even for a programming language with explicit structure statements (Pascal, PL/1).

The alternative is to write a structured subroutine which is functionally equivalent to an inverted program. Figure 3(a) shows the obvious solution for the problem given by the conflicting structures of Fig. 1(a) and 1(d). This subroutine is called for each detail line and prints a header after the end of a page. In fact, this second solution may be derived from a consolidated data structure as shown in Fig. 4. One might ask whether there are formal techniques to get the structured subroutine from the co-routine or the consolidated data structure from the two conflicting structures. This question is the major concern of this paper.

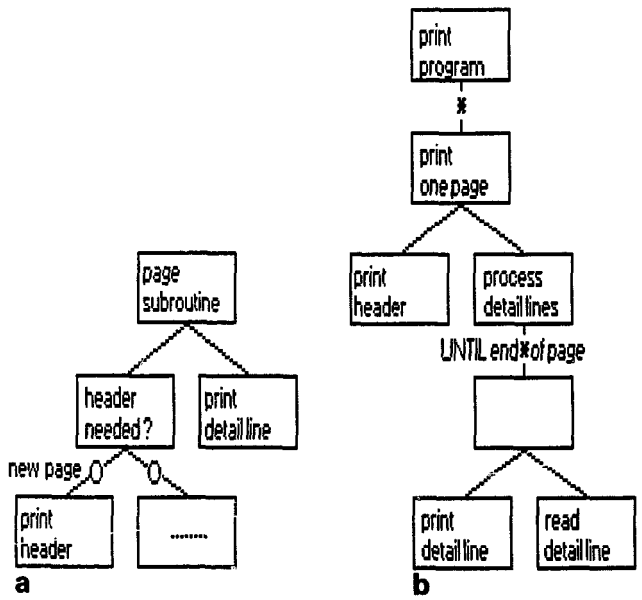


Fig. 3. (a) Page layout subroutine, (b) page layout program for intermediate file.

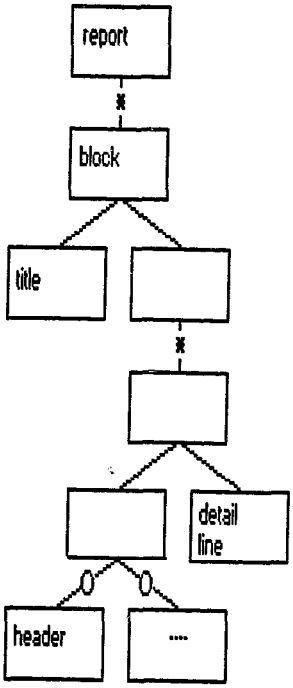


Fig. 4. Consolidated report structure.

1.4. *Problem statement*

This study is motivated by the following questions:

– Given an algorithm with a read statement of a sequential file, may it be transformed into a subroutine which, when called, is equivalent to the input and processing of one record? Formulated differently: Suppose a program is activated and suspended from a SUSPEND point nested somewhere in its logical structure and its outermost structure is an infinite iteration, then what is the corresponding 'normal' program, called at its beginning and stopping at its end? I.e., may the structure of Fig. 3(a) be formally derived from that of Fig. 3(b)?

– Is there a systematic way to transform conflicting data structures so as to produce a consolidated structure?

1.5. *Approach of this paper*

The paper is divided into two parts:

– In Section 2, the concepts of *division* and *decomposition* transformation and that of *generalized structure* are defined for program and data structures, and their properties are discussed in an intuitive way.

- Hierarchical data structures as used by Jackson may be formalized by regular expressions as has been shown by Hughes [10]. In Section 3, we define the same concepts as in Section 2, but for regular expressions. Their properties are formulated

as theorems and mathematically proved. The formalism of regular algebra allows a precise mathematical development of the transformations defined in Section 2 (see also Acknowledgement). It appears that the transformation rules of Section 2 may be founded on the base of a few fundamental equations on regular expressions. An essential step is the extension of regular expressions by the *binary iteration* operation. The concept of *regular structures* will be defined as a circuitless graph.

The results in terms of regular algebra may be seen as a rigorous foundation of the rules and properties stated for data structures. Their application to program structures remains on a more intuitive level, because the concept of conditions on some internal state variables is absent in regular algebra.

– The final section suggests possible further applications and outlines some open questions.

2. Transformation of program and data structures

2.1. Division transformation of programs structures

We consider program structures built from the following binary base structures:

sequence: BEGIN p q END
selection: SELECT c THEN p ELSE q END
iteration: ITER p WHEN c EXIT q END
(where c is a condition)

The binary iteration structure is a loop which can be exited only between the two blocks p and q. One of the blocks p or q may be null. The familiar iteration structures correspond to:

ITER WHEN c EXIT q END (WHILE NOT c DO q END)
ITER p WHEN c EXIT END (REPEAT p UNTIL c END)

A binary iteration ITER p WHEN c EXIT q END is equivalent to:

BEGIN p WHILE NOT c DO BEGIN q p END END

The notion of binary iteration is essential for the transformation that we will introduce. In fact, it will appear that, in a precise sense, the binary iteration structure is symmetric to the binary selection IF THEN ELSE.

We call a program structure *closed* if its outermost structure is an infinite iteration: ITER p NEVER EXIT q END. (i.e. ITER p WHEN c EXIT q, where c is always false). We will focus our interest on closed program structures, because the transformations that shall be defined apply to them. In fact, any program may be written as a 'closed' structure by a construct such as: ITER program NEVER EXIT suspend execution END

The central concept of *division transformation* is defined as follows: Let $P(X)$ be a closed program structure and X a structure block, perhaps a single operation,

within it. Then the *division of P by X* is a program structure such that an execution of the *division* is the same as the part of the execution of the original program that occurs between two executions of the block X.

The division represents "whatever happens" between two executions of X. The division of $P(X)$ by X shall be noted $N_X P(X)$. The most interesting property of this transformation is that its result may be formally derived for any closed program structure with respect to one of its structure blocks. This is done by a set of rules that give the result of $N_X P(X)$ in terms of the division of P by the block Y into which X is nested. (i.e., Y is one of the base structures

BEGIN X q END, BEGIN q X END,

SELECT c THEN X ELSE q END, SELECT c THEN q ELSE X END,

ITER X WHEN c EXIT q END, ITER q WHEN c EXIT X END)

Then the result of the division transformation is:

(1) $N_X P$(BEGIN X q END) = BEGIN q; $N_Y P(Y)$ END
(2) $N_X P$(BEGIN q X END) = BEGIN $N_Y P(Y)$; q END
(3) $N_X P$(SELECT c THEN X ELSE q END) = ITER $N_Y P(Y)$ WHEN c EXIT q END
(4) $N_X P$(ITER X WHEN c EXIT q END) = SELECT c THEN $N_Y P(Y)$ ELSE q END
(5) $N_X P$(ITER q WHEN c EXIT X END) = ITER q WHEN $\neg c$ EXIT $N_Y P(Y)$ END
(6) N_X ITER q NEVER EXIT X END = q
(7) N_X ITER X NEVER EXIT q END = q

The division of a closed program structure is derived by a recursive application of these rules. This means that the result of the transformation is constructed top down while the original structure is inspected bottom up.

Example 2.1. P is the closed program structure:

```
(P) ITER
    (Z)  BEGIN
           open
    (Y)    ITER
    (X)      read
           WHEN eof EXIT
             process
    (Y)    END
    (Z)  END
    NEVER EXIT
      suspend
(P) END
```

Then division by the "read" statement is:

$N_X P$ = SELECT eof THEN $N_Y P$ ELSE process END

```
= SELECT eof
  THEN BEGIN N_Z P ; open END
  ELSE process
  END

= SELECT eof
  THEN BEGIN suspend ; open END
  ELSE process
  END
```

We will show the validity of the above rules by intuitive arguments:

(1) After X, q is executed before the division N_Y (i.e. the 'outside') and then X again.

(2) After X, the division N_Y is executed before q and then X again.

(3) After X, the division N_Y of the selection block will always be executed; as long as c is false it may be followed by several executions of q followed by the division N_Y.

(4) If c is true, the iteration is exited and its division is executed once; otherwise the iteration continues and q is executed.

(5) q is always executed after X. As long as c is true the iteration will be exited and its division followed by q will also be executed.

(6) and (7) are obvious.

Analogous identities are mathematically proved for regular expressions in Section 3.

Example 2.2. A typical situation of division transformation may be seen in the context of a transaction monitor. The monitor invokes the application program for a screen input and the program returns control to the monitor after the next output to the screen. This may be seen as a division transformation of a program that resides in memory while sending to and receiving from the screen. The following program continuously receives data from a screen, processes and sends to the screen. The outermost iteration and the "suspend" operation have been added to make it a closed program.

```
(F) ITER
      suspend
    NEVER EXIT
     ITER
       BEGIN
         prepare next screen
         ITER
           send/receive
           check input
         WHEN ok EXIT
```

```
      END
    END
  WHEN enough EXIT
    process
  END
(P) END
```

When applying the above rules, the division of this program by the "send/receive" operation produces the following result:

```
BEGIN
  check input
  SELECT ok THEN
    BEGIN
      SELECT enough THEN suspend
      ELSE process
      END
      prepare next screen
    END
  END
END
```

This is the equivalent program as it would be written for execution under a transaction monitor. I.e., it is called after a screen input and stops before a screen output. However, the "suspend" operation should be replaced by an operation that informs the monitor to stop execution.

2.2. Decomposition of program structures

The division transformation removes the structure block X by which the structure was divided. We may combine X and the division N_X to get a program structure that is in a certain sense equivalent to the original one. To do so, we define a new transformation:

The *decomposition* of a closed program structure $P(X)$ by a block X is the closed structure: ITER X NEVER EXIT $N_X P(X)$ END. The decomposition of $P(X)$ by X will be noted M_X. (We also define a *right* decomposition M'_X = ITER $N_X P(X)$ NEVER EXIT X END.)

Executions of the decomposition are the same as those of the original program but the structure block X appears at the beginning of the decomposition. The decomposition is the program as seen from block X.

Example 2.3. For example 2.2 the decomposition by "send/receive" is:

```
(Q) ITER send/receive
    NEVER EXIT
```

```
      BEGIN
        check input
        SELECT ok THEN
          BEGIN
            SELECT enough THEN suspend
            ELSE process
            END
            prepare next screen
          END
        END
      END
(Q) END
```

The decomposition transformation is *symmetric*: it is possible to get back the original program structure from its decomposition by performing another decomposition. This is seen from the example. In fact, the decomposition of Q (Example 2.3) by "suspend" reconstitutes the original program P of Example 2.2.

If Q is the decomposition of P by X, then P can always be retrieved from Q by another decomposition transformation. To do so, Q is decomposed by the block nested in the outermost (NEVER EXIT) iteration of P that does *not* contain the block X by which the first decomposition was done.

On the other hand, one may be interested in seeing the result when a first decomposition is again decomposed by another block. The following example illustrates that the result is the same as that of directly decomposing the original structure by this block.

Example 2.4. The decomposition of program P by "prepare next screen" is

```
ITER
  prepare next screen
NEVER EXIT
  BEGIN
    ITER BEGIN
           send/receive
           check input
         END
    WHEN ok EXIT
    END
    SELECT enough THEN suspend
                  ELSE process
    END
  END
END
```

The decomposition of this program by "send/receive" gives the same result as the decomposition Q of the original program by the same statement. This is easily verified.

In fact, the division transformation is *transitive*: a decomposition transformation by the same block applied to two closed structures P and Q, where Q is a decomposition of P, gives the same result.

The properties of symmetry and transitivity can be intuitively understood from the concepts that will be introduced in the next section. A mathematical proof for regular expressions is given in Section 3.

The decomposition transformation is also *reflexive*, i.e., any closed program structure is a decomposition of itself. In fact ITER P NEVER EXIT Q END decomposed by P remains unchanged. The properties of reflexivity, symmetry and transitivity make it possible to define equivalence classes on closed program structures, where structures of the same class are all decompositions of each other. One may ask whether there is an adequate representation for such an equivalence class as a whole. Such a representation is the subject of the next section.

2.3. Generalized program structures

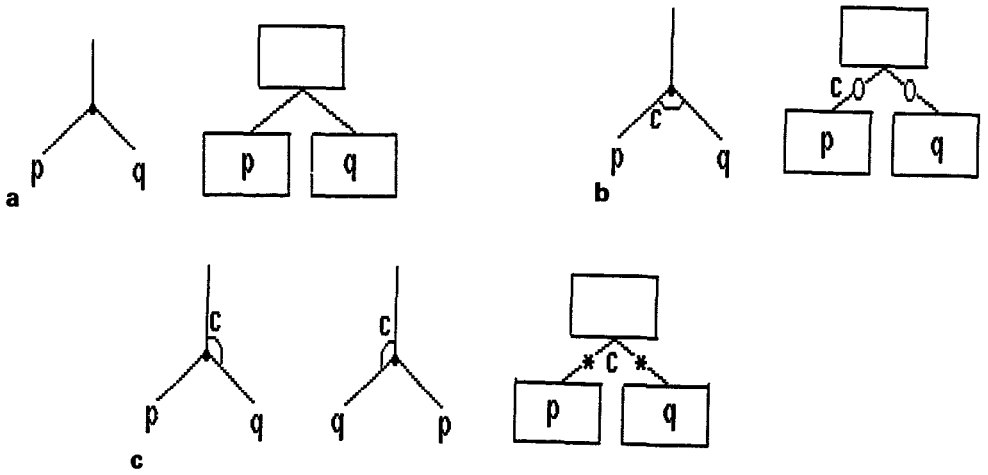
All decompositions of a closed program structure may be summarized by a graph. This will be called the *generalized program structure*. This is developed, starting from Jackson's (in fact Cohen's) graphical representation of a program structure as an arborescence. A new graphical representation is introduced which meets two conditions: first, the meaning of the graph must be unambiguous when it is turned in any direction, and second, when the graph is read as a hierarchical structure from any point (e.g. turning the graph upside down), the result is a division of the original structure. This is achieved by a graph where a vertex corresponds to each box of the arborescence.

Figure 5 shows the correspondence between boxes of hierarchical program structure notation and vertices of the generalized structure graph. With respect to the graphical notation used in section 1, the unary iteration is replaced by a binary iteration box with two descendants.

– The sequence corresponds to a *sequential* vertex (Fig. 5(a)). The order of execution is always counter-clockwise.

– The selection structure corresponds to a *branching* vertex (Fig. 5(b)). A branching type vertex has one *selection* type incident edge and two *iteration* type incident edges. On the graphical representation, iteration type edges are linked together by a sector of a circle. The branching vertex represents a selection structure when entered from the selection edge. The condition of the selection structure is associated with the corresponding iteration edge.

– The iteration structure corresponds to a branching vertex as well, but this time it is entered from one of the iteration type edges (Fig. 5(c)). The first (mandatory)



descendant of an iteration structure always corresponds to the selection type edge. The two representations of Fig. 5(c) are equivalent. The exit condition is associated with the edge from which the vertex is entered.

– Finally, an elementary operation (leaf of the program structure) corresponds to a terminal vertex.

Given a hierarchical (closed) program structure, the corresponding generalized structure is constructed like this:

For each one of the two descendants of the root, a graph is built according to the above correspondences. The outermost (NEVER EXIT) iteration is *not* represented by a vertex but the free edges of the two graphs are simply connected together. This is shown by Fig. 6, where the hierarchical structure and the generalized structure graph corresponding to Example 2.1 are shown together.

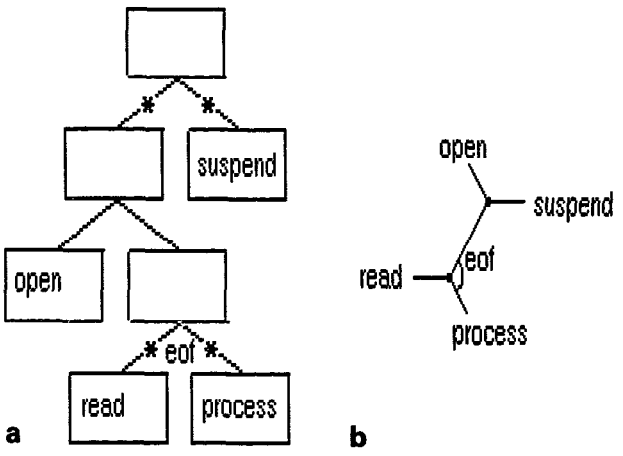


Fig. 6. (a) closed program structure of Example 2.1, (b) corresponding generalized program structure.

In the generalized structure the root of the program structure has been lost. The interest of the generalized structure concept lies in the fact that *any* edge may be considered as the origin of a hierarchical program structure. In fact, starting from an edge in one direction we get a hierarchical program structure by following the above correspondences. We call this structure the *partial view* from a directed edge. We may do the same thing for the same edge in the opposite direction. Combining both by an infinite iteration gives a closed program structure. Such a structure from one of the edges of a structure graph will be called a *local view.* More precisely, if p is the partial view from the directed edge $u \to v$ and q the partial view from $v \to u$, then the local view from $u \to v$ is "ITER p NEVER EXIT q END" and the local view from $v \to u$ is "ITER q NEVER EXIT p END". Local views are a kind of projection of the generalized structure. The set of local views is exactly the set of all possible decompositions of the original structure.

Example 2.5. Figure 7 shows the structure graph of the program of Example 2.2. The local view from $y \to$ "suspend" is the original program structure P of example 2.2. The partial view from "send/receive" $\to u$ is the division by send/receive of the same example. Example 2.4 is the local view from $w \to$ "prepare next screen".

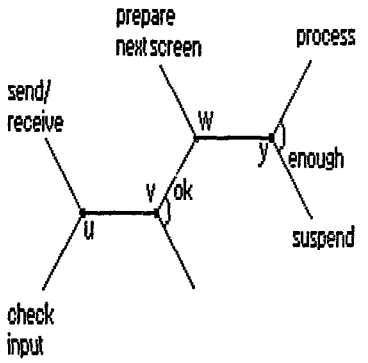


Fig. 7. Generalized program structure of Example 2.2.

Any two local views of the same generalized structure are decompositions of each other. In fact, the partial view from one direction of an edge is the division by the partial view from its opposite direction of any of the local views located in the part of the graph that the first partial view points to.

Example 2.6. For the same generalized structure of Fig. 7: "BEGIN send/receive; check input END" is the partial view from $v \to u$. The division of any of the local views $v \to w$, $w \to v$, $w \to y$, $y \to w$, $w \to$ "prepare next screen" etc. by this, produces the partial view from $u \to v$:

```
SELECT ok THEN BEGIN
                SELECT enough THEN suspend
                ELSE process
                END
                prepare next screen
              END
END
```

The reflexivity, symmetry and transitivity properties of the decomposition transformation are now easily understood because the result of a decomposition transformation is always the local view of the corresponding edge of the underlying generalized structure.

2.4. Data structures

In this section we consider data structures in the sense of Jackson. More precisely, we use *data structures* described by nesting the following base structures

sequence: BEGIN p q END
selection: SELECT p ELSE q END
iteration: ITER p EXIT q END

The binary iteration, ITER p EXIT q END means that a mandatory occurrence of p may be followed by an optional repetition of the sequence qp. It represents a repetition of p, where occurrences of p are always separated by q. One of the data p or q may be null and thus left out. In particular the structure ITER EXIT q END is equivalent to the usual optional iteration as used by Jackson.

Then we introduce the same concepts for data structures that we have used for program structure:
- A *closed data structure* is a data structure whose outermost structure is an iteration.
- The definition of the division transformation is analogous to the definition of the transformation for program structures. I.e., the *division* of closed data structure by one of its structure blocks X is the data structure that represents a series of data elements occurring between two occurrences of X.

The division of a data structure may be formally derived by a series of rules equivalent to those for data structures:

$$N_X S(\text{BEGIN } X\, q \text{ END}) = \text{BEGIN } q\, N_Y S(Y) \text{ END}$$

$$N_X S(\text{BEGIN } q\, X \text{ END}) = \text{BEGIN } N_Y S(Y)\, q \text{ END}$$

$$N_X S(\text{SELECT } X \text{ ELSE } q \text{ END}) = \text{ITER } N_Y S(Y) \text{ EXIT } q \text{ END}$$

$$N_X S(\text{ITER } X \text{ EXIT } q \text{ END}) = \text{SELECT } N_Y S(Y) \text{ ELSE } q \text{ END}$$

$$N_X S(\text{ITER } q \text{ EXIT } X \text{ END}) = \text{ITER } q \text{ EXIT } N_Y S(Y) \text{ END}$$

$$N_X \text{ ITER } q \text{ EXIT } X \text{ END} = q$$

$$N_X \text{ ITER } X \text{ EXIT } q \text{ END} \quad = q$$

Example 2.7.

```
(S) ITER
      BEGIN
        form feed
        header
      END
    EXIT
(y)   ITER
(x)     line
      EXIT
        line feed
(y)   END
(S) END
```

The division by "line" is:

$$N_x S(x) = \text{SELECT } N_y S(y) \text{ ELSE line feed END}$$

$$= \text{SELECT BEGIN form feed; header END ELSE line feed END}$$

– The *decomposition* of a closed structure by a structure block x is the closed structure:

$$M_x S(x) = \text{ITER } x \text{ EXIT } N_x S(x) \text{ END}$$

Example 2.8. The decomposition of the structure of example 2.7 by "line" is:

```
ITER line
EXIT
  SELECT BEGIN form feed; header END ELSE line feed END
END
```

The result of a decomposition transformation is the data structure as seen from the selected sub-structure. I.e., when thinking of the data structure as an infinite data stream, the decomposition is its structure when starting from an occurrence of x.

Reflexivity, symmetry and transitivity properties are the same as for program structures, and the generalized data structure is defined in the same way but without conditions.

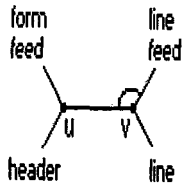


Fig. 8. Generalized data structure of Example 2.7.

Example. Figure 8 is the generalized structure of Example 2.7.

The concepts of data structure transformations and their properties have their mathematical foundations in regular algebra, as will be demonstrated in Section 3.

2.5. Application to the structure clash problem

We now return to the structure clash problem. More precisely, we will treat the "boundary clash" problem [10] stated in section 1.2: Consider a program which produces a report with the logical structure of Fig. 1(d), whereas the physical structure of the listing is that of Fig. 1(a). The two structures correspond on the highest and the lowest level, but not on the intermediate ("title" vs. "header") level. We admit that the example is trivial and easily resolved by any "report writer" type of software. However, it has the advantage of being simple and well understood and it allows us to show the principles of the solution.

Following Jackson's method (section 1.2), we will find ourselves with two co-routines. P_1 produces the logical report and invokes its partner for each detail or title line. P_2 when invoked continues to print the physical page layout. P_2 corresponds to the program of Fig. 3(b) that prints the report from an intermediate file. It may written as:

Example 2.9.

```
ITER
  print header
NEVER EXIT
  ITER
    print one line
    suspend
  WHEN end of page EXIT
  END
END
```

This co-routine shall always be invoked at the "suspend" point. The division of this co-routine by the "suspend" operation gives the following result (corresponding to Fig. 3(a)):

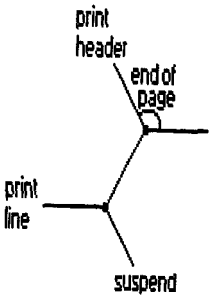


Fig. 9. Generalized program structure of Example 2.9.

```
BEGIN
  SELECT end of page
  THEN print head
  ELSE nothing
  END
  print one line
END
```

The generalized structure that summarizes subroutine and co-routine is shown in Fig. 9.

There is, however, an unresolved problem: the first invocation will not produce a header. More generally, a subroutine obtained from a division transformation should always be completed by an "open" (eventually a "close") routine. The open routine and the division subroutine should be encapsulated into a module. A formal derivation of open and close routines seems possible by the concept of "header" and "trailer" (section 3.6). However, it does not have the simplicity of division transformation, despite the fact that we tend to think that these open routines are easily found by intuition.

The same problem may be tackled by transformation of the data structures. The conflicting structures are:

Example 2.10.

```
ITER                        ITER
  header                      title
EXIT                        EXIT
  ITER detail line            ITER detail line
  EXIT                        EXIT
  END                         END
END                         END
```

Corresponding generalized structures are shown in figures 10a and 10b.

Both structures may be decomposed by "detail line" to give:

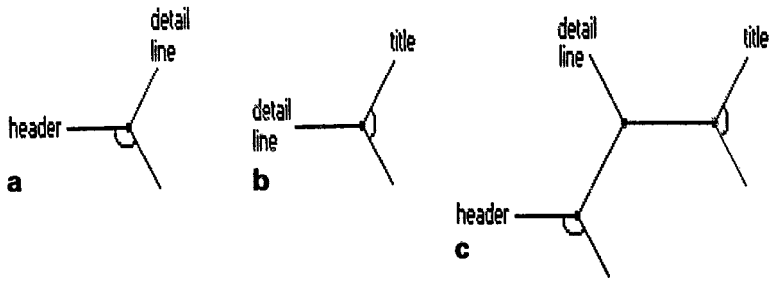


Fig. 10. Generalized data structure of Example 2.10. (a) page structure, (b) block structure, (c) consolidated structure.

```
ITER                          ITER
  detail line                   detail line
EXIT                          EXIT
  SELECT                        SELECT
    header                        title
  ELSE                          ELSE
  END                           END
END                           END
```

These structures may be combined into one that shows what occurs between two detail lines:

```
ITER
  detail line
EXIT
  BEGIN
    SELECT header
    ELSE
    END
    SELECT title
    ELSE
    END
  END
END
```

Decomposed by "header" we find the familiar structure:

```
ITER
  title
EXIT
  ITER
    BEGIN
      SELECT
        header
```

```
      ELSE
      END
      detail line
    END
  EXIT
  END
END
```

This structure is equivalent to that of Fig. 4.

The corresponding generalized structure is shown in Fig. 10(c).

2.6. Division algorithm

The division (and thus the decomposition) of a program may be derived by an algorithm following the above rules. We give an algorithm in a Pascal-like pseudo-code. The program structure is assumed to be given in form of an arborescence structure. Nodes are linked upwards (ancestor pointer) and downwards (descendant pointers).

The function argument is a structure block within the arborescence of a program structure. The program structure must be *closed.* "Division" is a recursive function that returns the result of the division transformation according to the rules of section 2.1.

```
TYPE
  structure_block:
    RECORD
      ancestor: POINTER TO structure_block;
      left_descendant, right_descendant: POINTER TO structure_block;
      structure_type: (sequence, selection, iteration, terminal)
      CASE structure-type OF
        selection, iteration: condition;
        terminal:             operation;
      END
    END

FUNCTION division (branch: structure_block): structure_block;
BEGIN
father = ancestor OF branch;
IF branch = left_descendant OF father THEN
  brother := right_descendant OF father;
ELSE
  brother := left_descendant OF father;
```

```
CASE structure_type OF
sequence:
  BEGIN
  division.structure_type := sequence;
  IF branch = left_descendant OF father THEN
    BEGIN
    division.left_descendant := brother;
    division.right_descendant := division(father);
    END
  ELSE
    BEGIN
    division.left_descendant := division(father);
    division.right_descendant := brother;
    END
  END


selection:
  BEGIN
  division.structure_type := iteration;
  division.left_descendant := division(father);
  division.right_descendant := brother;
  IF branch is the left descendant of its ancestor THEN
    division.condition := father_condition
  ELSE division.condition := NOT(father_condition);
  END


iteration:
  BEGIN
  IF father = NIL THEN
    division := brother
  ELSE
    BEGIN
    division.left_descendant := brother
    division.right_descendant := division(father);
    division.condition := NOT (father.condition)
    IF branch is the left descendant of its ancestor THEN
      division.structure_type := selection;
    ELSE division.structure_type := iteration;
    END
  END
END
```

3. Decomposition of regular expressions

3.1. Regular expressions

Given a set of symbols (an alphabet) A, regular sets are:
- subsets of A,
- the set of concatenations of all pairs of words p and q from two regular sets,
- union of regular sets,
- the set of all sequences of words from a set including the null word.

Regular sets are denoted by *regular expressions*, which are formed from
- the null word Λ symbol,
- the empty set $\emptyset$ symbol,
- constants from A,
- variables for regular sets,
- the concatenation pq,
- the union $p+q$,
- the iteration p^* denotes the set of all possible concatenations of words from p including Λ.

(see [1, 2 and 9])

Example: $A=\{I, J, K, L, M, N\}$, a possible regular expression is $L(I+K)^*NI$, possible members of this regular set are: *LNI, LIIIINI, LIKKIIKKKIINI*

The following obvious properties will be used throughout without explicit reference:

$$p+p=p=p+\emptyset, \qquad p+q=q+p,$$

$$p\Lambda=\Lambda p=p, \qquad p^*p^*=p^*=(p^*)^*.$$

The following identities are proved in [9]:

Properties 3.1.

(1) $p\emptyset=\emptyset p=\emptyset$,
(2) $\emptyset^*=\Lambda$,
(3) $\Lambda^*=\Lambda$,
(4) $(p+q)r=pr+qr$, $p(q+r)=pq+pr$,
(5) $pp^*=p^*p$,
(6) $p^*=pp^*+\Lambda$,
(7) $(p+q)^*=(p^*q^*)^*$,
(8) $(pq)^*p=p(qp)^*$,
(9) $(p^*q)^*p^*=(p+q)^*$, $p^*(qp^*)^*=(p+q)^*$,
(10) $(p^*q)^*=(p+q)^*q+\Lambda$, $(pq^*)^*=p(p+q)^*+\Lambda$.

In what follows, we extend the formalism of regular expressions by an iteration operator acting on two expressions:

Definition 3.2 (binary iteration operator). $p * q = (pq)^* p = p(qp)^*$ (the second equation holds by (8)).

We assume the following precedence between operators:

1st: unary $*$,
2nd: concatenation,
3rd: binary $*$,
4th: $+$.

E.g.: $a + b * cd = a + (b * (cd))$

The introduction of a binary iteration operator is essential for the results developed in the following section. In fact, the central Theorem 3.5 is a direct consequence of some of the following identities for binary iteration:

Properties 3.3.

(11) $p * q = pq(p * q) + p$,
(12) $p * q = p * (q * p) = p * (q * (p * \cdots$,
(13) $\Lambda * p = p^*$,
(14) $p * \Lambda = pp^*$,
(15) $p * \emptyset = p$,
(16) $\emptyset * p = \emptyset$,
(17) $(p * q)q = p(q * p)$,
(18) $q * p = q(p * q)q + q$,
(19) $pq * r = (p * qr)q = p(q * rp)$,
(20) $p * qr = (pq * r)rp + p = pq(rp * q) + p$,
(21) $(p * q) * r = p * (q + r) = (p * r) * q$,
(22) $p * (q * r) = pq((r + p) * q)qp + pqp + p$,
(23) $(p + q) * r = (\Lambda * qr)(p * (r * q))(\Lambda * rq) + q * r$.

Proof. (references to Properties 3.1.)

(11) $p * q = (pq)^* p = (pq(pq)^* + \Lambda)p = pq(pq)^* p + p = pq(p * q) + p$.
(12) $p * (q * p) = p * (q(pq)^*) = (pq(pq)^*)^* p = (pq)^* p = p * q$, by (10), (6)
The second equation follows from a recursive application of the first one.
(13) and (14) by definition of $p * q$.
(15) $p * \emptyset = p(\emptyset p)^* = p\emptyset^* = p\Lambda$.
(16) $\emptyset * p = \emptyset(p\emptyset)^* = \emptyset\Lambda = \emptyset$.
(17) $(p * q)q = (pq)^* pq = pq(pq)^* = p(q * p)$.
(18) $(q * p) = qp(q * p) + q = q(p * q)q + q$, by (11), (17).
(19) $pq * r = (pqr)^* pq = (p * qr)q$, $pq * r = pq(rpq)^* = p(q * rp)$.
(20) $p * qr = p(qr * p)p + p = pq(r * pq)p + p = (pq * r)rp + p$, by (18), (19), (17);
$p * qr = pqr(p * qr) + p = pq(rp * q) + p$, by (11), (19).

(21) $(p*q)*r=((pq)^*pr)^*(pq)^*p=(pq+pr)^*p$, by (9);
$=(p(q+r))^*p=p*(q+r)$
$=(p*r)*q$ by $p+q=q+p$.

(22) $p*(q*r)=p((q*r)*p)p+p=p(q*(r+p))p+p$ by (18), (21)
$=p(q((r+p)*q)q+q)p+p$ by (18)
$=pq((r+p)*q)qp+pqp+p$

(23) $(p+q)*r=(p+q)(r*(p+q))(p+q)+p+q$ by (18)
$=(p+q)((r*q)*p)(p+q)+p+q$ by (21)
$=p((r*q)*p)p+p$ (:I)
$+p((r*q)*p)q$ (:II)
$+q((r*q)*p)p$ (:III)
$+q((r*q)*p)q+q$ (:IX)

I: $p((r*q)*p)p+p=p*(r*q)$ by (18)

II: $p((r*q)*p)q=(p*(r*q))(r*q)q$ by (17)

III: $q((r*q)*p)p=q(r*q)(p*(r*q))$ by (17)

IV: $q((r*q)*p)q+q=q((r*q)(p*(r*q))(r*q)+r*q)q+q$ by (18)
$=q(r*q)(p*(r*q))(r*q)q+q(r*q)q+q$
$=q(r*q)(p*(r*q))(r*q)q+q*r$ by (18)

$(p+q)*r=(q(r*q)+\Lambda)(p*(r*q))((r*q)q+\Lambda)+q*r$
$=(\Lambda*qr)(p*(r*q))(\Lambda*rq)+q*r$ □

Finally the following rule is a direct consequence of the Theorem of Arden [3] (by substitution): If Λ is not in pq then $X=p*q$ is the unique solution of $X=pX+q$ (not used in what follows).

3.2. The division transformation

In what follows, we use expression functions e, f etc. in the following sense: In $f(x)$, x is a variable (or perhaps a constant) for a regular expression and $f(x)$ is a regular expression where x appears exactly once. If $x=y$ then $f(x)$ and $f(y)$ are identical regular expressions. In particular we use $\boldsymbol{a}$, $\boldsymbol{b}$ as symbols for atomic expressions. I.e., $a(x, y)$ means one of the following: $x+y$, $y+x$, $x*y$, $y*x$, xy, yx.

Example. $e(x)=l+K*xN$.

Definition 3.4. The *division* of a regular expression $e(x)$ by x is an expression s, such that the regular set denoted by $e(x)$ can be written as

$$e(x)=k(x*s)l+m$$

where x does not appear in k, s, l or m.

The division of $e(x)$ by x will be written $s=N_x e(x)$.

The regular set represented by $N_x e(x)$ is the set of all words that may appear between two occurrences of x in a word of the set $e(x)$. The division may be derived by a purely formal transformation of a regular expression. The rules of the following theorem give the result of a division transformation with respect to a variable, expressed in terms of the division with respect to the next higher nesting level. The division can be computed by applying these rules recursively from x upwards. The different possible cases of operators of the regular algebra must be distinguished.

Theorem 3.5.

(1) $N_x e(xq) = q(N_y e(y))$,
(2) $N_x e(qx) = (N_y e(y))q$,
(3) $N_x e(q+x) = (N_y e(y)) * q$,
(4) $N_x e(x * q) = (N_y e(y)) + q$,
(5) $N_x e(q * x) = q * (N_y e(y))$,
(6) $N_x x = \emptyset$.

Proof. Let r stand for $N_y e(y)$, i.e., the division of the next higher nesting level. To prove the rules, we show that if the division of the next higher nesting level is known, i.e. if $e(y)$ may be written as $k(a(x, q) * r)l + m$, then it may also be written as $k'(x * b(q, r))l' + m'$, where the atomic expression b is the division by x. The following identities are direct consequences of the properties 3.3 of the binary iteration operator.

(1) $k(xq * r)l = k(x * qr)ql$, by (19).
(2) $k(qx * r)l = kq(x * rq)l$, by (19).
(3) $k((q+x) * r)l = k(\Lambda * qr)(x * (r * q))(\Lambda * rq)l + k(q * r)l$, by (23).
(4) $k((x * q) * r)l = k(x * (q+r))l$, by (21).
(5) $k((q * x) * r)l = k((q * r) * x)l$, by (21).
$k((q * x) * r)l = k(q * r)(x * (q * r))(q * r)l + k(q * r)l$, by (18).
(6) $x = x * \emptyset$, by (15). □

Example. In this and the following examples we consider regular expressions on an alphabet $\{I, J, K, L, M, N\}$: The expression $L + ((I + N) * K)M$ has possible occurrences such as L, $IKIKNKIKNM$, IM, $IKIKIKM$, etc.

The division by N (which appears only once in the expression) is:

$$
\begin{aligned}
N_x L + ((I+x) * K)M &= (N_x(L + (x * K)M)) * I && (3)\\
&= (N_x(L + xM) + K) * I && (4)\\
&= (MN_x(L+x) + K) * I && (1)\\
&= (M(N_x x * L) + K) * I && (3)\\
&= (M(\emptyset * L) + K) * I && (6)\\
&= (M\emptyset + K) * I\\
&= K * I
\end{aligned}
$$

One may observe from this example that symbols standing outside the outermost iteration $(l+x)*K$ got lost during the division transformation. This is stated by the following:

Corollary 3.6 (outermost iteration). *Let $P=f(q*r)$, where f is an expression function without iteration operator, then*

$$N_qP=r \quad \textit{and} \quad N_rP=q.$$

Proof. (1) $N_xf(x)=\emptyset$. In fact $N_yy=\emptyset$ (by Theorem 3.5) and the result of the division operation of an expression without iteration is a nesting of expressions e of the type $xe(y)$, $e(y)x$, $e(y)*x$. But all of these are $\emptyset$ for $e(y)=\emptyset$. Thus, the result of their nesting is $\emptyset$.

(2) $N_qf(q*r)=N_qf(x)+r=\emptyset+r=r$, $N_rf(q*r)=q*N_qf(x)=q*\emptyset=q$ (Theorem 3.5.) □

Corollary 3.7. $N_x(p*e(x))=f(p)$ *for some f.*

Proof. Following Theorem 3.5, the division by a sub-expression is an expression of the division by the next higher nesting level. On the other hand, the division by $e(x)$ is p. Thus, the division is an expression of p. □

This corollary means that the result of a division transformation may always be written as an expression function of one of the sub-expressions of the highest nesting level. This property is essential as it assures that the "reverse" division $N_p(N_x(p*e(x)))$ is always defined. We will frequently make use of it without explicit reference.

3.3. The decomposition transformation

The result Corollary 3.6 shows that only the part of an expression inside its outermost iteration operation appears in the result of a division transformation. This motivates us to focus on expressions with an iteration at the outermost nesting level, e.g. $p*p'$. We call such expressions *closed* regular expressions. We will use capitals for closed expressions and frequently write them as $P=p*p'$, $Q=q*q'$ etc.

Definition 3.8. For a closed regular expression $P=e(x)$, we call (left) *decomposition* by x the closed expression $x*N_xe(x)$. We use the symbol $M_xe(x)$ for the decomposition of $e(x)$ by x.

By analogy, we call *right* decomposition the closed expression $N_xe(x)*x$ and use the symbol $M'_xe(x)$ for it.

Example. Let $P=(L*I)*K(I+J)$ be an expression on the alphabet $\{I, J, K, L, M, N\}$, let $x=I+J$, then $P=(L*I)*Kx$, and $M_x(L*I)*Kx=(I+J)*N_x((L*I)*Kx)=(I+J)*(L*I)K$

The decomposition $M_x e(x)$ denotes the regular set of all words, starting and ending with x, that appear within one of the words of the set $e(x)$.

Lemma 3.9. $N_x(p*e(x))=N_x(e(x)*p)$.

Proof. For some f, $N_x(p*e(x))=f(N_y(p*y))=f(p)$, and $N_x(e(x)*p)=f(N_y(y*p))=f(p)$ (by Corollary 3.6). □

As a consequence of this lemma, we can frequently ignore the difference between left and right decompositions. Thus, the following results will be formulated for left decomposition (we call them simply decompositions). The generalization to right decompositions is straightforward.

In Theorems 3.13 and 3.14 we will show that in a precise sense the division of a closed expression loses no information. i.e., the original expression may be retrieved by another decomposition and the result of a decomposition is independent of possible intermediate decomposition transformations. To prove these central results, the following lemmas will be needed. They are of interest mainly for this particular problem. ($a(q, r)$ stands for an atomic expression of q and r).

Lemma 3.10. *Let* $P=p*a(q, r)$ *and* $Q=M_qP$. *Then* $P=M_pQ$.

Lemma 3.11. *Let* $P=p*a(q, r)$, $Q=M_qP$, $R=M_rP$. *Then* $R=M_rQ$.

Proof. Both lemmas are verified by comparison of all possible cases for $a(q, r)$:

P	$Q=M_qP$	M_pQ	$R=M_rP$	M_rQ
$p*qr$	$q*rp$	$p*qr$	$r*pq$	$r*pq$
$p*rq$	$q*pr$	$p*rq$	$r*qp$	$r*qp$
$p*(q+r)$	$q*(p*r)$	$p*(q+r)$	$r*(p*q)$	$r*(p*q)$
$p*(q*r)$	$q*(r+p)$	$p*(q*r)$	$r*(q*p)$	$r*(q*p)$
$p*(r*q)$	$q*(r*p)$	$p*(r*q)$	$r*(p+q)$	$r*(p+q)$

□

Lemma 3.12. *Let* $P=p*e(f(q))$. *Then* $N_qP=N_q(M_{f(q)}P)$.

Proof. The nesting level of x in an expression $e(x)$ shall be defined as follows: the nesting level of x in $a(x)$ shall be 1 and if the nesting level of y in $e(y)$ is n then

that of x in $e(a(x, r))$ is $n+1$. The lemma can be proved by induction on the nesting level of q

$n=1$: $P=p*e(a(q, r))$, then $N_qP=b(r, N_{a(q,r)}P)$,
$N_q(M_{a(q,r)}P)=N_q(a(q, r))*N_{a(q,r)}P)=b(r, N_{a(q,r)}P)$.

$n\to n+1$: $P=p*e(f(a(q, \cdot)))$, let $s=f(a(q, r))$,

$N_qP=b(r, N_{a(q,r)}P)$,

$N_q(M_sP)=b(r, N_{a(q,r)}M_sP)=b(r, N_{a(q,r)}P)$

(by induction assumption) □

Example. $P=(L*I)*K(I+x)=(L*I)*Kf(x)$, $Q=M_{f(x)}P=M_y((L*I)*K)y=(I+x)*(L*I)K$, $N_xQ=N_x((I+x)*(L*I)K)=(L*I)K*I$ and $N_xP=N_x((L*I)*K(I+x))=(L*I)K*I$.

Theorem 3.13 (symmetry). *Let $P=p*e(q)$, $Q=M_qP$. Then $P=M_pQ$.*

Proof. By induction on nesting level of q in $e(q)$:

(i) $e(q)=q$ then $Q=q*p$ and $N_pQ=q$,
(ii) $e(q)=a(q, r)$ then proved by lemma,
(iii) assume true for nesting level n:

$$P=p*p'=p*e(a(q, r)) \text{ and } Q=q*q'=q*b(r, N_{a(q,r)}P).$$

Let $s=a(q, r)$, $S=M_sP=s*s'$ and $S'=s'*s$, then

$Q=M_qS$ (by Lemma 3.12)

$S'=M_{s'}Q$ (by Lemma 3.10)

$M_pQ=M_p(M_{s'}Q)$ (by Lemma 3.12)
$=M_pS'=M_pS=P$ (by induction assumption). □

Example. $P=(L*I)*K(I+J)$, $Q=M_{i+j}P=(I+J)*(L*I)K$, $M_{L*I}Q=(L*I)*N_x((I+J)*xK)=(L*I)*K(I+J)$.

Theorem 3.14 (transitivity). *Let $P=e(q, r)$ a closed expression. $Q=M_qP$, $R=M_rP=r*r'$. Then $R=M_rQ$ or $R=M'_{r'}Q$.*

Proof. Several possible cases will be distinguished.

(i) $P=p*e(q)$ and $q=f(r)$, then proved by Lemma 3.12.
(ii) $P=p*e(r)$ and $r=f(q)$ then $Q=M_qR$ (by Lemma 3.12) and $R=M_{r'}Q$ (by Theorem 3.13).
(iii) $P=e(r)*f(q)$ then $P=M_pQ$ (by Theorem 3.13).

$R = M_r Q$ (by Lemma 3.12)

(iv) $P = p * e(q, r)$, we write $e(q, r)$ as $f(a(s, t))$, where $s = g(q)$ and $t = h(r)$. Let $S = M_s P = s * s'$ and $T = M_t P$, then

(a) $S = M_{s'} Q$ (by Lemma 3.12 and Theorem (3.13),

(b) $T = M_t S$ (by Lemma 3.11),

(c) $R = M_r Q$ (by Lemma 3.12). □

Example. $P = (L * x) * K(I + J)$, $Q = M_{I+J} P = (I + J) * (L * x)K$, $R = M_x P = M_x((L * x) * K(I + J)) = x * (L * K(I + J))$, $M_x Q = M_x((I + J) * (L * x))K = x * (L * K(I + J)) = R$. □

Finally, as a result of Corollary 3.6: For $P = p * p'$ we get $M_p P = P$ (reflexivity).

These results may be interpreted as follows: Let $P \sim Q$ stand for the fact that P is a decomposition of Q, i.e. there is a p such that $Q = e(p)$ and $P = M_p Q$ or $P = M'_p Q$. Then

(1) $P \sim P$,

(2) if $P \sim Q$ then $Q \sim P$,

(3) if $P \sim Q$ and $R \sim P$ then $R \sim Q$.

I.e., the relation $\sim$ has the characteristics of an equivalence relation. As with any equivalence relation, the decomposition relation $\sim$ induces a partition of closed regular expressions into equivalence classes. All expressions of the same class may be obtained from each other by decomposition transformations. One might wonder what formal representations should be given to such an equivalence class. This formal representation will be defined in the following section by the concept of *regular structures.*

3.4. Regular structures

Definition 3.15. A *regular structure* is a connected circuitless graph (a tree). For each edge $\{u, v\}$ we also consider two directed (or *incident*) edges (u, v) and (v, u). Each vertex is said to be of one of the following types:

terminal, with one incident edge.

sequential, with 3 incident edges. With respect to a sequential vertex v, a bijective *SUCCESSOR* mapping is defined on its 3 incident edges. I.e., each incident edge has exactly one successor and one predecessor incident edge.

branching, with 3 incident edges. One of the incident edges is said to be the *selection edge,* the two others are said to be *iteration edges.*

The graphical representation is shown in Fig. 11.

Definition 3.18. The *partial view* $V(u, v)$ of a directed edge (u, v) is a regular expression:

– If v is terminal, then $V(u, v)$ is the word associated with v.

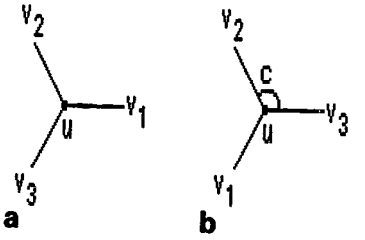


Fig. 11. Graphical notation of regular structures (a) Sequential vertex with incident edges: $(u, v_3) =$ SUCCESSOR(u, v_2), $(u, v_2) =$ SUCCESSOR(u, v_1), $(u, v_1) =$ SUCCESSOR(u, v_3). (b) Branching vertex with selection edge (u, v_1) and iteration edges (u, v_2) with condition c, (u, v_3) with condition NOT c.

- If v is sequential, then $V(u, v) = V(v, w)V(v, z)$, where (w, v) is the successor of (u, v) and (z, v) is the successor of (w, v).
- If v is a branching vertex:
  - if (u, v) is the selection edge, then $V(u, v) = V(v, w) + V(v, z)$,
  - if (u, v) is an iteration edge, then $V(u, v) = V(v, w) * V(v, z)$, where (w, v) is the selection edge.

Definition 3.17. The *local view* $W(u, v)$ of a directed edge (u, v) is a regular expression: $W(u, v) = V(u, v) * V(v, u)$.

In Fig. 12 the regular structure corresponding to the example of definition 3.8 is shown. Example local views are:

$$W(v, u) = (L * I) * K(I + J) \quad (P \text{ in example of Definition 3.8}),$$

$$W(v, w) = (I + J) * (L * I)K \quad (Q \text{ in example of Theorem 3.13}),$$

$$W(u, I) = I * (L * K(I + J)) \quad (R \text{ in example of Theorem 3.14}).$$

For any of the local views, all other local views are decompositions. More precisely, if (u, v) and (w, z) are edges of a regular structure and u is closer to (w, z) than v (i.e., (v, u) points to the direction of (w, z)), then $W(u, v) = M_{V(u,v)}W(w, z)$. I.e., any partial view $V(v, u)$ is the division by the 'reverse' partial view $V(u, v)$ of any local view whose edge is on the side of the graph pointed to by (v, u).

Example. In the regular structure of Fig. 12, $V(u, v)$ is the division by $V(v, u)$ of any of the local views: $W(v, w)$, $W(w, v)$, $W(v, K)$, $W(K, v)$, $W(w, I)$, $W(I, w)$, $W(w, J)$, $W(J, w)$ and (trivally) $W(u, v)$, $W(v, u)$.

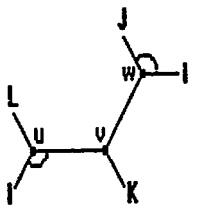


Fig. 12. Regular structure of the example to Definition 3.8.

Theorems 3.13 and 3.14 and their proof can be understood most easily when reference is made to the concept of regular structure.

3.5. N-ary regular expressions

Concatenation and union operators are associative. I.e., $(ab)c = a(bc)$ may be written abc and $(a+b)+c = a+(b+c)$ may be written $a+b+c$. However, in the division transformation we make explicit use of the fact that operators are binary. In fact, $N_x e((x+p)+q) = (N_y e(y) * q) * p$ whereas $N_x e(x+(p+q)) = (N_y e(y) * (p+q))$. Although the result is the same regular set according to Properties 3.3, the expressions are formulated differently. This kind of ambiguity may be removed by introducing a division transformation on N-ary regular expressions. As usual, we write

pqr for $(pq)r = p(qr)$ and $p+q+r$ for $(p+q)+r = p+(q+r)$.

To generalize binary to N-ary iteration in the same style, we note that:

$(\cdots(p_1 * p_2) * p_3) * \cdots) * p_N = p_1 * (p_2 + p_3 + \cdots + p_N)$ (by a recursive application of Property 3.3(21)).

We may agree to write this as

$$p_1 * p_2 * p_3 * \cdots * p_N = (\cdots(p_1 * p_2) * p_3) * \cdots) * p_N.$$

In this N-ary iteration p_2 to p_N may be permutated, but not p_1.

Then we generalize Theorem 3.5 as follows:

$$N_x e(p_1 \cdots p_J x p_{J+1} \cdots p_N) = p_{J+1} \cdots p_N N_y e(y) p_1 \cdots p_J,$$

$$N_x e(p_1 * \cdots p_J * x * p_{J+1} \cdots * p_N) = p_1 * \cdots * p_J * p_{J+1} * \cdots * p_N * N_y e(y),$$

$$N_x e(x * p_1 * \cdots * p_N) = p_1 + \cdots + p_N + N_y e(y),$$

$$N_x e(p_1 + \cdots p_J + x + p_{J+1} \cdots + p_N) = N_y e(y) * p_1 * \cdots * p_J * p_{J+1} * \cdots * p_N.$$

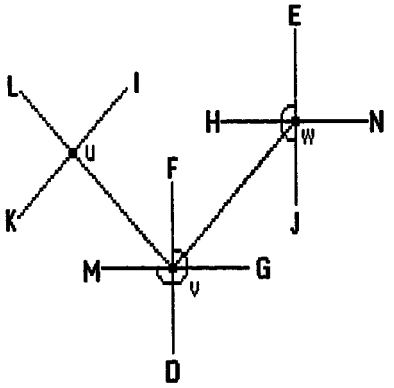


Fig. 13. N-ary regular structure of the example in section 3.5.

The *regular structure concept* may be generalized to allow vertices with more than three incident edges. A sequential vertex may have n incident vertices with a SUCCESSOR function. For a branching vertex, one of its n incident edges is of selection type, all others are iteration edges.

Example. An N-ary regular structure is shown in Fig. 12. An example of local views is:

$$W(u, v) = (M + D + G + (N * E * H * J) + F) * ILK.$$

3.6. Header, trailer and remainder

The division $s = N_x e(x)$ was defined by $e(x) = k(x * s)1 + m$. The expressions k, l, m may be formally derived, as is seen from the proof of Theorem 3.5. If the regular set $e(x)$ may be written as $e(x) = k(x * s)l + m$ where k, s, l and m do not contain x (i.e., s is the division $N_x e(x)$) then we call $k = H_x e(x)$ the *header*, $l = T_x e(x)$ the *trailer* and $r = R_x e(x)$ the *remainder* of $e(x)$ with respect to x.

Headers and trailers may be derived by concatenation of the expressions obtained for different atomic expressions:

(1) $H_x e(xq) = H_y e(y)$, $T_x e(xq) = qT_y e(y)$.
(2) $H_x e(qx) = H_y e(y)q$, $T_x e(qx) = T_y e(y)$.
(3) $H_x e(q + x) = H_y e(y)(\Lambda * qN_y e(y))$, $T_x e(q + x) = (\Lambda * N_y e(y)q)T_y e(y)$.
(4) $H_x e(x * q) = H_y e(y)$, $T_x e(x * q) = T_y e(y)$.
(5) $H_x e(q * x) = H_y e(y)(q * N_y e(y))$, $T_x e(q * x) = (q * N_y e(y))T_y e(y)$.
(6) $H_x x = \Lambda$, $T_x x = \Lambda$.
(7) $R_x e(xq) = R_x e(qx) = R_x e(x * q) = R_y e(y)$.
(8) $R_x e(q + x) = R_x e(q * x) = R_y e(y) + H_y e(y)(q * N_y e(y))T_y e(y)$.
(9) $R_x x = \emptyset$.

These formulas are verified from the proof of Theorem 3.5.

Example Let $P = L * (K + J * N)$. Then,

$$N_x(L * (K + J * x)) = J * (L * K).$$

$$\begin{aligned} H_x(L * (K + J * x)) &= L(\Lambda * KL)(J * (L * K)) = (L * K)(J * (L * K)) \\ &= ((L * K) * J)J \\ &= (L * (K + J))J. \end{aligned}$$

$$\begin{aligned} T_x(L * (K + J * x)) &= (J * (L * K))(\Lambda * LK)L \\ &= (J * (L * K))(L * K) = J((L * K) * J) \\ &= J(L * (K + J)). \end{aligned}$$

$$\begin{aligned} \boldsymbol{R}_x(L*(K+J*x)) &= L+L(K*L)L+L(\Lambda*KL)(J*(L*K))(\Lambda*LK)L \\ &= L*K+(L*K)(J*(L*K))(L*K) \\ &= (L*K)*J=L*(K+J) \qquad \square \end{aligned}$$

Clearly, the rules listed above lack the elegance of Theorem 3.5. Their application to the example seems to be a complicated method to obtain a simple result. This may suggest that header and trailer transformation are not yet fully understood. The results Lemma 3.12 and Theorem 3.13 seem to suggest that headers and trailers are of minor interest. In fact, the original expression may be gained back without using them. However, we state these rules for the sake of completeness.

Conclusion

This paper presents one possible approach to Jackson's "structure clash problem". We do not claim to give the definite solution to the problem and the results presented here should be viewed in the context of preceding research. For example, a structure clash may be resolved on an operating system level by a mechanism like UNIX "pipes". Durieux [7, 8] handles the problem by formalizing directly Jackson's program inversion on the level of finite automata. The scheduling of co-routines corresponds to a cartesian product of automata.

It is not our purpose to enter into a debate on what is methodologically the best way to handle the problem. We think that the "division" concept gives a good feeling of what happens in restricted cases of inversion: e.g. the transformation of a program reading a sequential file into a subroutine which is called for each record. The concept of "general (regular) structures" gives a vision of program and data structures which is independent of a particular starting point. We argue that it may serve as an example of one of the main paradigms of current computer science: abstraction.

The concepts developed in this paper are not restricted to the context of JSP. Their applications to program design in general should be investigated in more detail. Some possible further applications are:

– In process control real time programming, suspend operations are frequent and the "natural" start point of program is not always as clear as in business systems. It might then be interesting to design the program from any one of the suspend points and to "decompose" afterwards for the implementation.

– Even in conventional programmming, the point from which the program may be designed most easily may be a location other than the starting point: e.g. call of subprogram, read etc. One might imagine the possibility of writing a "local view" of an algorithm with a compiler capable of decomposition transformations.

– Suppose a program is being written for the processing of an intermediate file. One might wish to have a new version of the same program that works as a subroutine.

- The local view from a critical point of a program might be helpful in debugging a piece of code.
- The problem of error recovery in syntax analysis might be tackled by using the concept of local views. In fact, it is possible to have a piece of syntax specified by a regular expression $e(p)$ where p is a unique symbol. Then, if p is the current symbol the decomposition, $\dot{M}_p e(p)$ is the syntax as it appears at this moment.

Although we give a precise formal transformation for data and program structure, we do not claim to have solved all aspects of the structure clash problem. In addition, this paper has suggested a number of open questions which could be the subject of further research. For example:
- How could the transformations presented be extended to transform a program with several suspend points into a structured program where *all* of these are mapped to the beginning of the program? In terms of regular algebra this means to define a transformation analog to the division for an expression $P(x)$ where the variable x appears several times.
- How should the formalism be extended to procedure calls? There seems to be a symmetry between calling and being called.
- The concept of "header" and "trailer" should be explored further, in particular to generate open and close routines.
- Regular structure are defined as circuitless graphs. How could the same graphs with circuits be interpreted? This would introduce a rather peculiar type of recursiveness.
- An algebraic foundation for transformation of program structures should be developed, in analogy to regular algebra for data structures.

Acknowledgement

Yves Pigneur (University of Lausanne) contributed to this work by critical reading of the first draft versions. Joe Oates (ISDOS Incl.) corrected the first version. One of the referees suggested the use of regular algebra; Giovanni Coray (EPFL) gave me some advice on this subject. Susan Jongeneel (Veillon) corrected the English of the second version and made several suggestions to improve the formulation of the paper. I would like thank them all for their contribution.

References

[1] A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation and Compiling, Vol. 1 Parsing* (Prentice-Hall, Englewood Cliffs, NJ, 1972).
[2] M.A. Arbib, *Theories of Abstract Automata* (Prentice-Hall, Englewood Cliffs, NJ, 1969).
[3] D.N. Arden, Delayed logic and finite state machines, Proc. 2nd Ann. Symp. on Switching Circuit Theory and Logical Design, 1961.
[4] C. Böhm and G. Jacopini, Flow diagrams, Turing machines and languages with only two formation rules, *Comm. ACM* **9** (1966) 366–371.

[5] A. Cohen, *Structure, Logic and Program Design* (Wiley, Chichester, 1983).
[6] N. Deo, *Graph Theory with Applications to Engineering and Computer Science* (Prentice-Hall, Englewood Cliffs, NJ, 1974).
[7] J.-L. Durieux, Ebauche de formalisation de la construction de Jackson-Hughes, *Actes Colloques Génie Logiciel* **3** (1986) 295–304.
[8] J.-L. Durieux and D. Dzierzgowski, About methods, Report Tool Use. Cert. T2, CERT, Toulouse, 1987.
[9] A. Ginzburg, *Algebraic Theory of Automata* (Academic Press, New York, London, 1968).
[10] J.W. Hughes, A formalization and explication of the Michael Jackson method of program design, *Software Practice and Experience* (1979) 191–202.
[11] M. Jackson, *Principles of Program Design* (Academic Press, London, 1975).
[12] K. Orr, *Structured Systems Development* (Yourdon Press, New York, 1977).
[13] J.-D. Warnier, *Logical Construction of Programs* (Van Nostrand Reinhold, New York, 1976).
[14] J.-D. Warnier, *Construction et Transformation de Programmes* (Les éditions d'organisation, Paris, 1983).