KfK 3964
September 1985

Reuse of Software through Generation of Partial Systems

F. J. Polster
Institut für Datenverarbeitung in der Technik

Kernforschungszentrum Karlsruhe

KERNFORSCHUNGSZENTRUM KARLSRUHE

Institut für Datenverarbeitung in der Technik

KfK 3964

Reuse of Software through Generation of Partial Systems

Franz J. Polster

Kernforschungszentrum Karlsruhe GmbH, Karlsruhe

Kernforschungszentrum Karlsruhe GmbH
ISSN 0303-4003

Abstract:

One approach to improving software productivity is the development and reuse of general software for a given application area to avoid development of code. Frequently, for a particular application a partial system that supports only a subset of the capabilities of a general program system is sufficient.
The problem of constructing partial systems is addressed, where the program of a partial system is obtained by selecting only those code segments of the complete program that implement the capabilities needed. A heuristic for determining fragments of a program system, which can serve as the building blocks for the programs of partial systems, is presented.
The notion of "B-program" is introduced: a B-program contains in addition to the fragments themselves for each fragment substitute code and control information specifying the set of partial systems the fragment is relevant for. A representation of B-programs as a string is given, such that generating a partial system consists in scanning this string and selecting substrings.
A formal model for this type of program generation is developed: a B-program is viewed as an ordered tree with the substrings of the complete program as its leaves and the fragments as its non-leaf vertices; a "relevance" mapping indicates for each fragment vertex f whether or not f is relevant for a particular partial system; a mapping associates with each fragment its substitute. Generation of a partial system is defined in terms of pre-order traversal of a subtree of the B-program.
B-program reduction is dealt with: transformations for the elimination of superfluous vertices are presented, the issue of uniqueness and the problem of constructing a minimal reduced B-program are discussed.

Wiederverwendung von Software durch Erzeugung von Teilsystemen

Zusammenfassung:

Ein Ansatz zur Erhöhung der Software-Produktivität besteht in der Vermeidung von Neuprogrammierung durch Entwicklung allgemeiner Software für einen bestimmten Anwendungsbereich und deren wiederholte Verwendung. Für eine gegebene Anwendung genügt häufig ein Teilsystem, das nur eine Teilmenge der Fähigkeiten eines allgemeinen Programmsystems realisiert.
Das Problem der Konstruktion von Teilsystemen wird behandelt, wobei man das Programm eines Teilsystems durch Auswahl nur der Programmteile des vollständigen Programms erhält, die die benötigten Fähigkeiten implementieren. Ein heuristisches Verfahren zur Bestimmung von Fragmenten eines Programmsystems, die als Bausteine für die Programme der Teilsysteme dienen können, wird angegeben.
Der Begriff "B-Program" wird eingeführt: ein B-Programm enthält zu den Fragmenten selbst für jedes Fragment einen Ersatz und eine Spezifikation der Menge der Teilsysteme, für die das Fragment relevant ist. Eine Ausprägung von B-Programmen in Form von Zeichenketten wird angeben, so daß die Erzeugung eines Teilsystems im einmaligen Lesen dieses Strings und der Auswahl von Teilstrings besteht.
Ein formales Modell für diese Art von Programmerzeugung wird entwickelt: Ein B-Programm wird als ein geordneter Baum mit den Teilstrings des vollständigen Programms als Blätter und den Fragmenten als innere Knoten betrachtet; eine "Relevanz"-Abbildung gibt für jeden Fragment-Knoten f an, ob f für ein bestimmtes Teilsystem relevant ist oder nicht; eine Abbildung verknüpft mit jedem Fragment dessen Ersatz. Die Erzeugung eines Teilsystems wird als ein Aufsuchen der Knoten eines Teilbaumes des B-Programms in pre-order beschrieben.
Die Reduktion von B-Programmen wird behandelt: Transformationen zur Elimination von überflüssigen Knoten werden angegeben, Eindeutigkeitsfragen und das Problem der Konstruktion eines minimalen reduzierten B-Programms werden diskutiert.

CONTENTS

1. Introduction

Software productivity has become a critical problem: "the demand for new software is increasing faster than our ability to supply it, using traditional approaches" [5]; see also [13] on this issue.

As pointed out in [5], [30] one approach to improving software productivity is the reuse of software to avoid development of code. Reuse of software entails the design and implementation of general software systems, i.e. systems, which perform frequently used, common, and repetitive data processing tasks (also called "reusable functional collections" or "generic systems" [7]). Typical examples are operating systems, compilers, database management systems, mathematical subroutine packages.

By definition general software systems have to provide services for as wide a spectrum of applications of the respective application area as possible. Generality, however, cannot be accomplished without cost, such systems necessarily tend to become comprehensive and complex program systems, which often occupy a significant part of system resources and/or bring about a reduction in efficiency.

For a particular application in general an often small subset of the features provided by a program system P would suffice, so that the immediate use of P is at best wasteful and uneconomical, at worst impossible altogether, e.g. due to efficiency problems or limited resources. In order to avoid or at least reduce these problems with generalized software it is desirable to employ instead of a general program system P "versions" of P that provide exactly those features of P called for by the application at hand and consist only of the software components of P supporting them:

- This is one of the motivations for "SYSGEN" options of operating systems and research into families of operating systems [14], [15], [19], [21].
- Mary Shaw discusses in [20] the usefulness of and the benefits to be gained from having available for a programming language a "language contraction", i.e. a family of programmming languages produced by successively factoring out groups of features of the language: it is

shown that this is a technique for improving compilation efficiency, in particular, the sizes for the compilers corresponding to the sublanguages of a contraction are smaller than the size of the complete compiler implementing the full language.

- Similarly, "dedication" of database management systems, i.e. use of versions of a database management system that provide only subsets of the capabilities (in particular a subset of the operations of the user interface) supported by the complete database management system is presented in [17] as a way to benefit from general database packages also in environments that do not allow the use of the complete database management system due to memory restrictions, efficiency or economic considerations.

This work addresses the problem of tailoring a given program system P to the specific needs of an application through eliminating from P features not used by that application. In the following this type of program tailoring is referred to as program a d a p t a t i o n .
It is assumed that (i) program system P is given as a string over some alphabet and (ii) the versions of P can be characterized

- functionally, i.e. in form of a list of "algorithms" of P to be supported by a version
- quantitatively in terms of the values of "system parameters".

We will rely on the intuitive notion of

- an a l g o r i t h m as a set of one or more pieces of code required for the execution of some function provided by P
- a s y s t e m p a r a m e t e r of P as a substring of the program of P that represents the value(s) or size of a data object (e.g. values of variables, buffer sizes) of P and determines the degree, to which a function of P can be executed.
  For a database management system these may be figures like: the maximum number of predicates in queries, the maximal record-length for retrieval or update operations, an upper limit for the number of records to be sorted, maximum number of concurrent transactions, the size of the system buffer (cf. [17]).

Also, it is assumed that modifying system parameters, i.e. constants in definitional or assignment statements of the program text, requires only

replacing the old value with a new one (programs written in common programming languages have this property!).
This property, then, implies that substituting symbolic names, so-called p l a c e h o l d e r s , for system parameters in the source programs of versions of P, which support the same set of algorithms, results in identical strings. In other words such a string with placeholders represents instances of source programs with the same functional characteristic.
Therefore, for the adaptation of a general program P we at least conceptually start out with a program text including placeholders, called the c o m p l e t e p r o g r a m of P, and perform the following two steps (see fig. 1):

- Selection of the parts of the complete program implementing the algorithms required and their integration into the program of a p a r t i a l s y s t e m of P.
- Replacement of the placeholders in the program of a partial system with syntactically valid strings (e.g. constants). This yields the source program of a version of P.

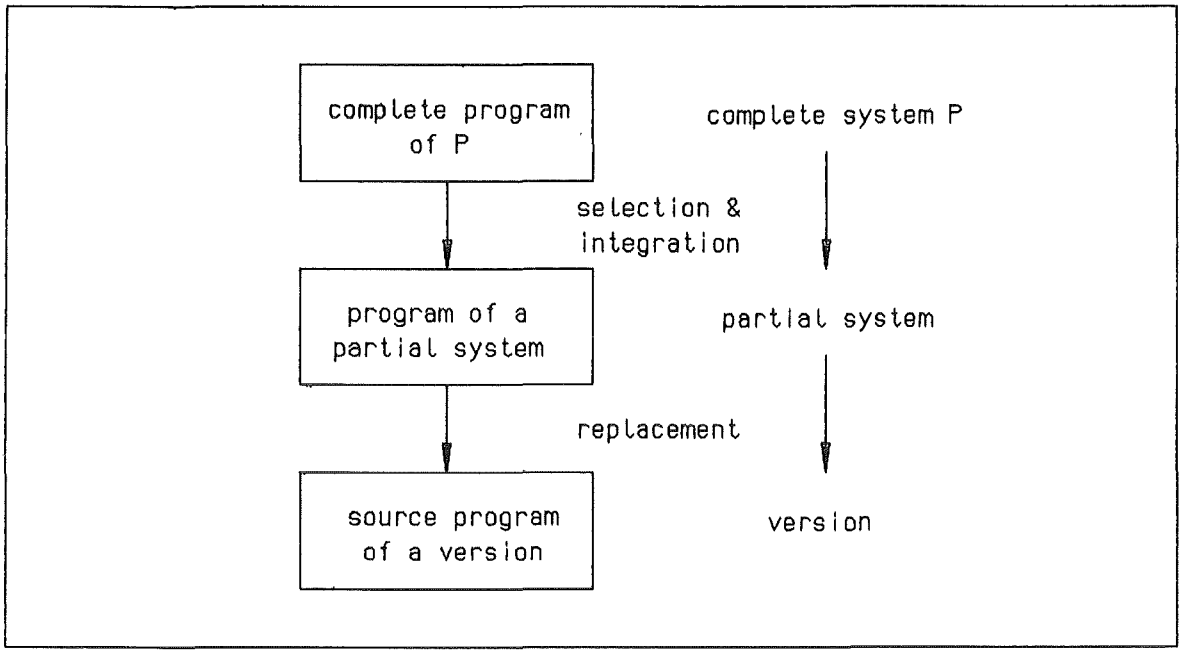


Fig. 1: Program adaptation

Notation:
The partial system providing all algorithms of P is called the c o m p l e t e s y s t e m (representing the set of instances of P).

The step of producing source programs is referred to as "dimensioning" (among other things, typically dimensions of arrays are fixed here!). Dimensioning is a simple task that can be implemented using e.g. a macro processor [6], [12], [18].
The focus of this paper is on the production of partial systems, i.e. program generation by means of code selection:
Section 2 shows that in general program adaptation entails generation of source programs out of a B-program ("base program").
Section 3 introduces and investigates the properties of the constituents of B-programs: fragments, substitutes, relevances.
Section 4 deals with the construction of B-programs: a heuristic for obtaining the fragments for a given program system and an implementation of the pertaining B-program as an expansion of the complete program is presented. Generation of a partial system consists in the selection of substrings of this extended program.
Section 5 formalizes these ideas and presents a theory of B-programs: a B-program is viewed as an augmented ordered tree with the substrings of the complete program as leaves and the fragments as non-leaf vertices; generation of a partial system is defined in terms of pre-order traversal of a subtree. This model provides the framework for proves about the generation algorithm and a rigorous treatment of B-program "reduction", i.e. the problem of constructing for a given B-program another B-program with a smaller number of vertices and edges, which still represents the same set of partial systems. The construction of a minimal reduced B-program and the issue of uniqueness are addressed.
The reader is referred to the appendix for an explanation of the basic concepts and notations of mathematics and computer science employed in this text.

For demonstration purposes the program system of fig. 2 will be used throughout this paper: it is called DBMS and sketches the implementation of a database management system, providing a one-tuple database interface with the six operations of table 1.

operation	semantics
OPEN	acquire a lock on a relation; in order to access the tuples of a relation the relation must be locked by the application program
CLOSE	release a lock; at the end of a transaction all locks acquired (with OPEN) must be released by the application program
FIND	select a set of tuples of a relation satisfying a qualification, make them available in a QSS
GET	retrieve a tuple of a QSS
INSERT	insert a tuple into a relation
DELETE	delete a tuple from a relation

Table 1: The operations supported by the example system DBMS

For the implementation of relations DBMS supports two storage structures (cf. variables FILE_TYPE of program units INSERT and DELETE of fig. 2), access paths are available in form of "sequential search", hashing or inverted file techniques.
There are two techniques for accessing data (variable ACCESS_TYPE of program unit GET): "sequential search" und "direct access" (employing lists of tuple identifiers TID).

Table 2 delineates the implementation of the operations of table 1 with the pertaining program fragments (statements, subroutine calls) in angular brackets; the right-most column gives the names of the algorithms of DBMS ('1' through '17').

```
PROCEDURE DBMS
   IF (OP<1 OR OP>6)
      THEN return 'operation unknown'
   CASE OP OF
     1: OPEN
     2: CLOSE
     3: FIND
     4: GET
     5: INSERT
     6: DELETE
   END
END
```

```
PACKAGE INDEXES
  INDEX_TABLE: ARRAY OF INTEGER
END
```

```
PROCEDURE OPEN
   OPEN_RF
   OPEN_IF
END
```

```
PROCEDURE CLOSE
   CLOSE_RF
   CLOSE_IF
END
```

```
PROCEDURE FIND
   USE INDEXES
   evaluate INDEX_TABLE
   STRTGY
   return qss
END
```

```
PROCEDURE STRTGY
   determine access-strategy and
    set ACCESS_TYPE
   CASE ACCESS_TYPE OF
     1: build seq.search qss
     2: BEGIN
        CASE FILE_TYPE OF
          1: calculate tid
          2: . . . . . .
             RETRIEVE_TID_LIST
             . . . . . .
        END
        build direct-access qss
        END
   END
END
```

```
PROCEDURE OPEN_RF
  . . . . . .

  GET
  . . . . . .
END
```

```
PROCEDURE CLOSE_RF
  . . . . . .

END
```

```
PROCEDURE OPEN_IF
  USE INDEXES
  . . . . . .
  GET
  . . . . . .
END
```

```
PROCEDURE CLOSE_IF
  USE INDEXES
  . . . . . .
END
```

```
PROCEDURE GET
   NEXT_TUPLE:
   CASE ACCESS_TYPE OF
     1: NEXT_SEQ
        . . . . . .
     2: NEXT_TID
        . . . . . .
   END
   IF (qualification is not satisfied)
      THEN GO TO NEXT_TUPLE
END
```

```
PROCEDURE NEXT_SEQ
   CASE FILE_TYPE OF
     1: next_1
     2: next_2
   END
END
```

```
PROCEDURE NEXT_TID
   return next tid of tid-list
END
```

```
PROCEDURE RETRIEVE_TID_LIST
   . . . . . .
END
```

```
PROCEDURE INSERT
  CASE FILE_TYPE OF
    1: INSERT_1
    2: INSERT_2
  END
  INSERT_TID
END
```

```
PROCEDURE DELETE
  CASE FILE_TYPE OF
    1: DELETE_1
    2: DELETE_2
  END
  DELETE_TID
END
```

```
PROCEDURE INSERT_1
  . . . . . .
END
```

```
PROCEDURE DELETE_1
  . . . . . .
END
```

```
PROCEDURE INSERT_2
  . . . . . .
END
```

```
PROCEDURE DELETE_2
  . . . . . .
END
```

```
PROCEDURE INSERT_TID
  USE INDEXES
  . . . . . .
END
```

```
PROCEDURE DELETE_TID
  USE INDEXES
  . . . . . .
END
```

Fig. 2: The example system DBMS

operation	implementation	algorithm
OPEN	- lock relation <OPEN_RF>	1
	- if inverted files exist for the relation, acquire locks and update INDEX_TABLE <OPEN_IF>	2
CLOSE	- release lock for relation <CLOSE_RF>	3
	- if inverted files exist for the relation, release locks and update INDEX_TABLE <CLOSE_IF>	4
FIND	- determine in INDEX_TABLE the available inverted files <evaluate INDEX_TABLE>	
	- determine access technique and create a subset (QSS) for sequential search <build seq.search qss>	5
	or direct access employing:	
	hashing <calculate tid>	6
	TID-list via inverted file <RETRIEVE_TID_LIST>	7
GET	- retrieve next tuple through: sequential search <NEXT_SEQ> according to	
	storage structure 1 <next_1> or	8
	storage structure 2 <next_2>	9
	direct access with a TID-list <NEXT_TID>	10
	- check, whether qualification is satisfied	11
INSERT	- insert a tuple according to storage	
	structure 1 <INSERT_1> or	12
	2 <INSERT_2>	13
	- determine in INDEX_TABLE the available inverted files and update them <INSERT_TID>	14
DELETE	- delete a tuple according to storage	
	structure 1 <DELETE_1> or	15
	2 <DELETE_2>	16
	- determine in INDEX_TABLE the available inverted files and update them <DELETE_TID>	17

Table 2: The algorithms of DBMS

A partial system of DBMS:

Let A be an application, e.g. a data entry program, requiring the DBMS operation INSERT only. We assume that storage structure 1 is suited for the rapid storage of tuples and therefore is used for the implementation of the relations to be updated by A; since (i) there are no retrieval operations to be supported and (ii) the maintenance of inverted files slows down update operations, no inverted files will be employed for A. Thus, algorithm 12 suffices for the implementation of operation INSERT for such an application. Due to the semantics of the DBMS interface (see table 1) A has to lock and unlock the relations to be accessed (operations OPEN, CLOSE), for these purposes only algorithms 1 and 3 respectively are necessary for this application (and not algorithm 2 or 4). Access to system catalogues (the call to GET in OPEN_RF!) requires algorithms 8 and 11.

The partial system of DBMS providing the operations OPEN, CLOSE and INSERT with these five algorithms is referred to as t_ins.

Remark: The program of partial system t_ins is shown in fig. 9, section 4.2.

2. Program generation through code selection

The eventual goal of generating a version of a program system P is the production of a load module implementing a subset of the capabilities provided by P. (Executable code may be either a "load module", that runs as a separate task, or a "linkable module", that is linked to other software. For the purpose of this paper this distinction is without any significance, and "load module" refers also to linkable modules!).
This section discusses techniques for code selection and justifies the implicit assumption of section 1 that program adaptation by means of code selection entails generation of source programs, unless we consider language- or machine-specific techniques, as e.g. manipulation of object modules.

We assume that executable code is constructed according to the general scheme of fig. 3 (cf. e.g. [22]):
A source program is given as a set of program units (procedures, functions, subroutines). A translator, e.g. a compiler, translates each program unit into an object module with the translator control program V_COMP specifying (among other things) the program units to be translated. A linker builds from the resulting object modules a load module as specified by a linker control program V_LINK.

With this scheme code selection can be done

- at link-time
- at translate-time
- before translate-time

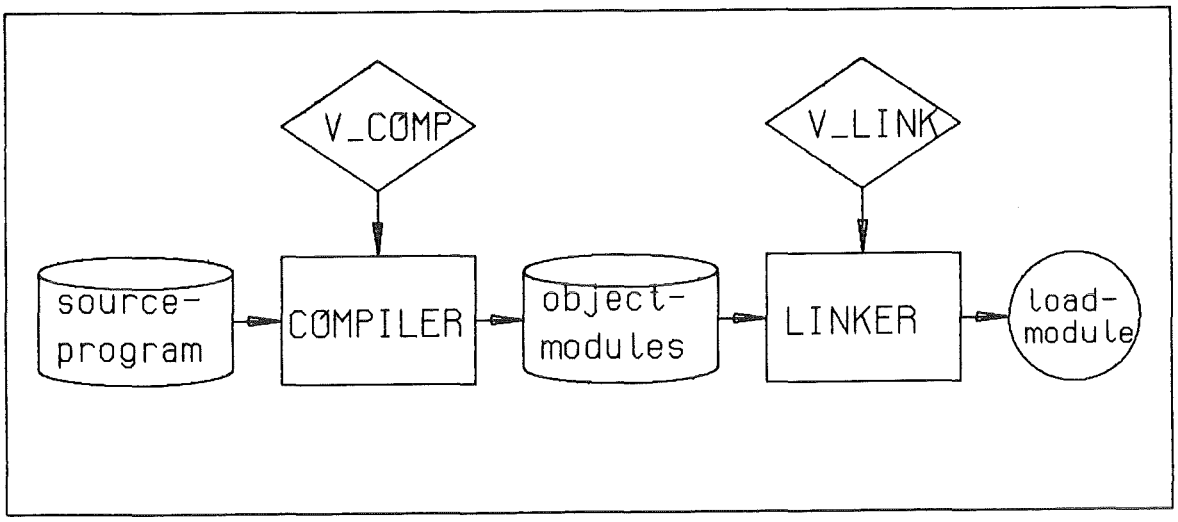


Fig. 3: Construction of executable code

2.1. Code selection at link-time

Code selection at link-time can be achieved through modifying the linker control program pertaining to the complete system: deleting an INCLUDE-statement from the control program has the effect of removing the specified object module (together with all modules referenced by this module only). I.e. code selection and integration at link-time consists of

- construction of a version-specific linker control program through selection of the relevant statements of the linker control program of the complete system
- initiation of a linker run

Code selection at link-time only, however, is in general not sufficient:

a) Mere removal of superfluous object modules will usually yield not-fault-tolerant load modules: an attempt to execute a function that is not implemented by a load module built in this way may lead to abnormal termination. In any case, the desired response, namely an indication, e.g. via a return code, of 'function is not implemented' cannot be achieved by simply deleting code from a load module. Rather, "substitute code" performing this task must be provided and included instead of removed object modules.

b) Code selection at link-time implies that the selectable unit is the object module. This is too coarse a granularity, however: e.g. the call of a module to a superfluous (and thus deleted) object module and the code usually associated with a call (e.g. convertion of values to be passed as actual parameters, checking and evaluating return codes) cannot be removed in this way, if some other part of the module is relevant.

c) Clearly, dimensioning cannot be achieved with this technique.

2.2. Code selection at translate-time

Code selection at translate-time can be achieved through modifying the translator control program of the complete system, such that only relevant object modules are produced.
Adaptation based on code selection at translate-time is more complex and time-consuming than the technique of section 2.1: in addition to the linker control program a version-specific translator control program has to be generated and the linker run is preceded by a translator run.
Yet, nothing is gained despite this higher degree of complexity: since each program unit corresponds to an object module the size of the selectable unit is the same as above, thus, code selection at translate-time does not solve any of the problems pointed out above.

2.3. Code selection before translate-time

According to section 2.1 the selectable unit being smaller than a program unit or object module is a necessary condition for the generation of partial systems without superfluous code. Code selection, thus, must take place before translate-time, i.e. version generation entails generation of source programs.
Since a source program can be considered a string of characters over some alphabet, the selectable unit may be any substring of the complete program, in particular a single statement of a program unit or even part of a statement. I.e. program adaptation can now be viewed as a general text manipulation task, namely the selection of substrings of a given

string, the complete program.
Another advantage of this technique is that it can easily be extended to implement the text replacement task of dimensioning.
These considerations suggest a scheme for version generation as displayed in fig. 4:

- a component B - p r o g r a m ("base program") comprises the substrings of the complete program necessary for the generation of partial systems together with their substitutes. It can be viewed as a representation of the set of versions of the complete system.
- a utility s e l e c t o r selects the strings of the B-program relevant for the version to be generated, integrates them into the program of a partial system and produces a source program by replacing the placeholders. The source program is processed as described above (fig. 3).
- A component V_DES contains a description of the version to be generated, it serves as the selector control program.

An implementation of program adaptation along these lines is described in [18].
The remainder of this paper investigates the problem of generating partial systems, in particular the nature and structure of B-programs.

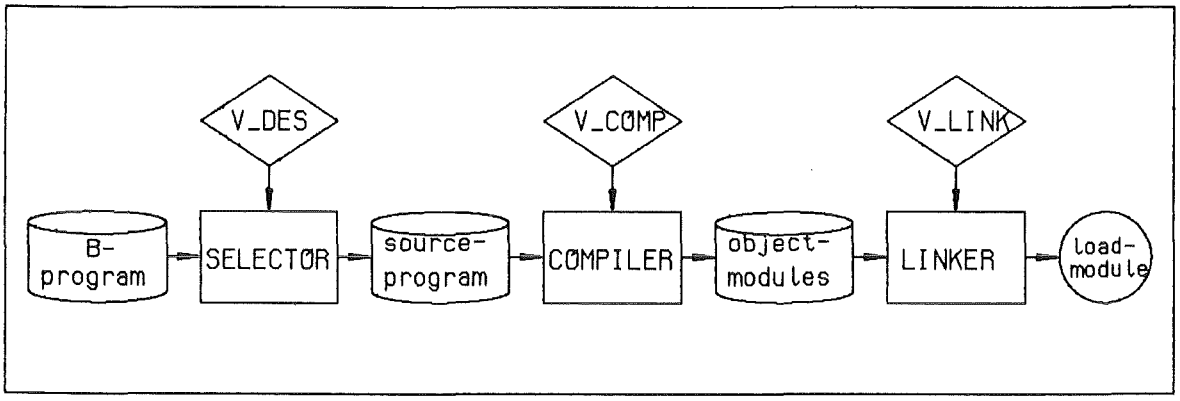


Fig. 4: Version generation

3. Concepts for the generation of partial systems

This section introduces the basic concepts for program adaptation, in particular the constituents of B-programs.

3.1. Definitions, terminology

Let T denote the set of partial systems of program system P. As postulated in section 2 a B-program contains the building blocks for the programs of the partial systems of T. In particular, it must make available the substrings of the complete program implementing the algorithms of P that are to be provided by these partial systems. To this end the notion of f r a g m e n t is introduced. A formal definition is developed in section 3.2, for the moment it is sufficient to think of a fragment as a substring of the complete program.

Let F denote the set of fragments of P. With each fragment f ε F information as to whether or not f is relevant for a given partial system t ε T must be associated. The "relevance" of a fragment can formally be thought of as a mapping of T into the set B:= {0, 1} of truth ("Boolean") values:

DEFINITION 1:
Let f ε F, g ε F:

- The mapping ρ_f: T --> B indicates whether or not a fragment f is r e l e v a n t for t ε T:

$$\rho_f(t) := \begin{cases} 0 : f \text{ is not relevant for } t \\ 1 : f \text{ is relevant for } t \end{cases}$$

  ρ_f is called the r e l e v a n c e of f, $\rho_f(t)$ the r e l e v a n c e v a l u e of f for the partial system t.
  f and g are said to have the s a m e r e l e v a n c e , iff $\rho_f \equiv \rho_g$ (cf. section E of appendix)
- A r e l e v a n c e e x p r e s s i o n is a relevance or a Boolean expression with relevances as operands. The Boolean operators are

defined for relevances in the obvious way, e.g.:

$$\rho_f \text{ OR } \rho_g\ (t) := \rho_f(t) \text{ OR } \rho_g(t)$$

Besides being a building block a fragment can also be viewed as a piece of code that is deleted from the complete program for the construction of some partial system. Therefore, as has been elaborated in section 2, substitute code must be associated with each fragment and these pieces of code must be components of a B-program, too.
Let Σ denote the set of strings over the alphabet of the programming language the complete program (and thus the programs of the partial systems!) are written in. We adopt the convention that the empty string, denoted: NIL, is element of Σ (cf. e.g. [4]).
Since a substitute must be a string of Σ the association of fragments with substitutes is expressed as a mapping F --> Σ:

DEFINITION 2:
The mapping σ: F --> Σ associates with each fragment f the s u b s t i - t u t e $\sigma(f)$ of f.
Note that due to NIL ε Σ a substitute $\sigma(f)$ may be the empty string.

Remark: Definition 2 says that each fragment is assigned exactly one substitute. As will be demonstrated in section 3.3, however, this does not necessarily exclude the possibility of having n>1 substitutes for a given substring of the complete program!

3.2. The fragment concept

This section explores the nature and properties of fragments and presents a formal definition of the fragment concept.

3.2.1. Fragment: program unit, sequence of statements

Fragments designate the parts of the complete program, which are required as building blocks for the programs of the partial systems in T. As discussed in section 2.1 it must be possible to have program units as well as single statements of a program unit as building blocks.
Examples:

- For partial systems of DBMS that do not support operation INSERT the program units INSERT, INSERT_1, INSERT_2, INSERT_TID are not relevant; therefore, in order to be able to construct such partial systems fragments are required, which comprise these program units and the relevances of which evaluate to 0 for such partial systems.
- Fig. 5 gives a more detailed presentation of program unit INSERT: Partial systems of DBMS for applications, which apply operation INSERT only to a subset of the storage structures supported by DBMS and/or do not require the maintenance of inverted files (cf. partial system t_ins), execute just a subset of the groups of statements marked with 1, 2 and 3 at the left in fig. 5. In order to be able to provide partial systems without dead code with respect to such applications, a fragment must be defined for each of these parts of program unit INSERT.

Convention:
Throughout this paper program lines pertaining to a fragment are marked with the name of that fragment in one or several columns at the left of the program text. E.g. the lines of code of fig. 5 with the name "1" (in the right column) form fragment 1.

```
name of
fragment

   | |          PROCEDURE INSERT
   | |
   | |D1|       S_RET: INTEGER
   | |D2|       Z_RET: INTEGER
   | |          FTYPE: INTEGER

    :             :   :   :   :   :   :   :   :

   | |          determine storage structure employed and
   | |          set variable FILE_TYPE
   | |          CASE FILE_TYPE OF
   | |            1: BEGIN
   | | 1|            /* insert:    storage structure 1 */
   | | 1|            INSERT_1( . . . . . ,S_RET)
   | | 1|            IF (S_RET # 0) THEN GO TO 930
   | |               END
   | |              2: BEGIN
   | | 2|            /* insert:    storage structure 2 */
   | | 2|            INSERT_2( . . . . . ,S_RET)
   | | 2|            IF (S_RET # 0) THEN GO TO 930
   | |               END
   | |          END
   | | 3|       /* update inverted files              */
   | | 3|       INSERT_TID( . . . . . ,Z_RET)
   | | 3|       IF (Z_RET # 0) THEN GO TO 990
   | |
   | |      900: /* REGULAR EXIT */
   | |          GO TO 999
   | |E1| 930: s_error-action
   | |E1|       GO TO 999
   | |E2| 990: z_error-action
   | |E2|       GO TO 999
   | |
   | |      999: RETURN
   | |          END
```

Fig. 5: Fragments as lists of statements of a program unit

The introduction of a fragment may necessitate the definition of additional, so-called "derived" fragments:

- the statements of fig. 5 marked E2 can be executed only, if those of fragment 3 are included in INSERT. Therefore in order to avoid unreachable code fragment E2 is required as a consequence of introducing fragment 3.

  Similarly the statements marked E1 can be executed only, if those of fragment 1 and/or 2 are included, which leads to the introduction of a fragment E1.
- Since variable Z_RET (S_RET) is referenced only by statements of fragment 3 (fragments 1 and 2), fragment D2 (D1) with definitional statements is introduced as indicated in fig. 5 in order to avoid partial systems with unreferenced program variables.

3.2.2. A formal definition

Fragments 1, 2 and 3 of fig. 5 are necessary for the generation of partial systems that support only a subset of the algorithms available in DBMS for the implementation of operation INSERT, for partial systems not realizing this operation at all a program unit INSERT is superfluous and should be omitted altogether.
To this end one might try to define additional fragments such that INSERT is completely "covered" with fragments and the relevances of these fragments evaluate to 0, when program unit INSERT is not relevant. This approach, however, is "unnatural", in that several fragments have to be processed in order to omit a single syntactic construct that forms a contiguous piece of text. Another flaw is that the substitute for a fragment becomes dependent on the partial system: e.g. for fragment 2 of fig. 5 the substitutes would be

- code producing a return code for partial systems that realize operation INSERT only with algorithm 12 (cf. fig. 9)
- the empty string NIL for partial systems without operation INSERT.

One way to avoid these difficulties is to allow fragments to comprise besides substrings of the complete program also fragments.
Then, in our example a fragment (represented by the empty left column of fig. 5) can be defined that contains fragments 1, 2, 3, D1, D2, E1 and E2 together with all the substrings of program unit INSERT not comprised by one of these fragments.
This is the rationale for the following definition:

DEFINITION 3:

- A f r a g m e n t f is a not empty list of strings $q \in \Sigma$ and fragments $f_i \neq f$; the fragments f_i are called the s u b f r a g m e n t s of f, f the e n c l o s i n g fragment for each f_i.
- A fragment g is called to be n e s t e d in f if and only if g is a subfragment of f or g is nested in a subfragment of f.

The "semantics" of a fragment can informally be described as follows:

- If fragment f is not relevant for a partial system, then for the generation of the program of that partial system f together with all

fragments nested in f and the strings comprised by them are replaced with $\sigma(f)$.

- Otherwise, the substrings comprised by f become part of the program of the partial system, the subfragments of f are processed in the same way as f.

According to the first statement fragments nested in a fragment that is not relevant for $t \varepsilon T$ do not contribute in any way to the program of t, i.e. they are implicitly assumed to be not relevant for t. This can be formally written as:

With $g \varepsilon F$ nested in f $\rho_f(t)=0$ implies $\rho_g(t)=0$ for each $t \varepsilon T$.

Often, however, stronger statements hold: the relevance of the fragment comprising program unit INSERT is equal to ρ_1 OR ρ_2, since for the implementation of the INSERT operation at least either fragment 1 or 2 must be provided!

Convention:

In this paper the name of a subfragment x of a fragment with the name f is the string 'f.x' .

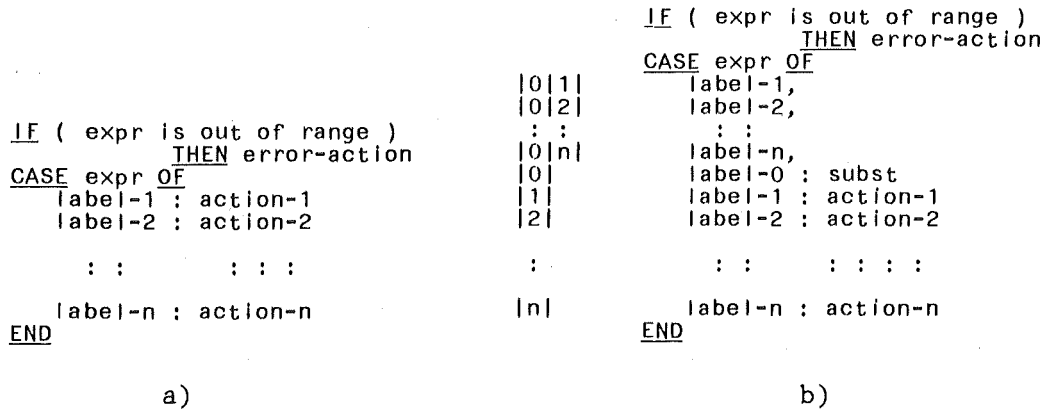
3.2.3. Fragment: part of statement

For the fragmentation of language constructs involving a list it is often useful to have a part of that list available for the construction of partial systems. Typical examples are CASE-constructs and definitional statements:

Execution of a CASE-construct consists in the execution of at most one out of several alternatives, therefore alternatives of CASE-constructs are "natural" candidates for fragments of a program system.
Fig. 6a depicts the general structure of a CASE-construct with n alternatives. The purpose of the IF-statement is to guarantee that the evaluation of the CASE-expression expr yields a legal value (cf. CASE-statement of PASCAL [10]).

A straightforward (and always viable) fragmentation of a CASE-construct consists of fragments for each of its alternatives, i.e. the n substrings action-i, $1 \leq i \leq n$, form a fragment each. In fig. 5 e.g. this leads to the fragments 1 and 2.

If, however, the substitutes for all alternatives are identical - a quite common situation in practice -, one can also proceed as follows, cf. fig. 6b:



a) b)

Fig. 6: Fragments of CASE-constructs

a) For $1 \leq i \leq n$ the string ' label-i : action-i ', the i-th alternative,

forms fragment i with the empty string as substitute: $\sigma(i)$=NIL.

b) In order to have the check for legal values of expr independent of the partial systems a fragment 0 with the subfragments 0.i, $1 \le i \le n$, is introduced. The common substitute subst is the only string of this fragment.

c) For $1 \le i \le n$ the relevance of a subfragment 0.i is the "negation" of the relevance of fragment i, i.e. $\neg\rho_i$:

$$\rho_{0.i}(t) := \neg\rho_i(t) = \begin{cases} 0 : \rho_i(t)=1 \\ 1 : \rho_i(t)=0 \end{cases}$$

Fragment 0 is relevant if and only if one of its subfragments is relevant:

$$\rho_0 \equiv OR_{i=1}^{n} \rho_{0.i}$$

d) The empty string NIL is the substitute for fragment 0 as well as its subfragments:

$$\sigma(0) = \sigma(0.i) := NIL \qquad 1 \le i \le n$$

For CASE-constructs with a large number of alternatives this kind of fragmentation is superior to the general approach:

- if alternative i is not relevant, the statements action-i w i t h label label-i are omitted without replacement (substitute is NIL!), i.e. the CASE-construct is "shortened" for such partial systems; also, duplication of the substitute subst is avoided.
- the CASE-construct becomes more "readable" in that it is immediately evident from the program text, which alternative is not implemented for a given partial system.

Notice that the strings comprised by fragment 0 and its subfragments are n o t substrings of the complete program! Also, 'label-0' is introduced as a "dummy" label in order to keep the fragmentation simple: otherwise, the commas of the label-list of fragment 0 would have to form separate fragments with rather complicated relevance expressions!

Remark: The fragments of program unit DBMS in fig. 7 are derived following this technique (see also fig. 9 for the program unit DBMS of partial system t_ins).

Fragments comprising a part of a program statement may also arise when several data objects with different relevances are declared in a single definitional statement. Let e.g. the declarations of fig. 5 be written as the single statement

S_RET,Z_RET,F_TYPE: INTEGER

Then, fragments D1 and D2 would be the substrings 'S_RET,' and 'Z_RET,' respectively.
Note that if all variables must be comprised by different fragments it is helpful to add (in analogy to the dummy label above) a "dummy" variable to the list of program variables in order to keep the fragmentation simple.

3.3. Substitutes

3.3.1. Definition of substitute code

Substitutes specify the actions of a partial system, when it is called to perform a function of the complete system that it is not intended to implement. In other words, the assignment of substitute code to fragments is an act of programming and is therefore the responsibility and the task of the system designer or programmer.
There may be some rules of the thumb, e.g. the substitutes for derived fragments such as fragments E1 and E2 of fig. 5 will usually be the empty string. In general, however, this task cannot be automated. Consider e.g. the INSERT operation of the example system and its implementation in fig. 5:

- It is sufficient and reasonable to assign fragments 1 and 2 substitute code with the only effect of producing a return code indicating "storage structure not accessible", say (cf. figures 8 and 9).
- In contrast, the substitute for fragment 3 (cf. $\sigma(8.3)$ of fig. 8) depends on the envisaged use of partial systems:
  NIL is the right choice, when it is guaranteed (as e.g. in "dedicated systems" [17]), that partial systems of DBMS without algorithm 14, i.e. lacking the capability of maintaining inverted lists, will never be called to insert tuples into relations that are implemented using

inverted files.

Otherwise $\sigma(3)$ must be code that "undoes" the insertion of the tuple into the storage structure (the effects of fragment 1 or 2, respectively) and generates an appropriate return code.

- In general there are several alternatives for the implementation of a substitute:

  The empty string as the substitute for program unit INSERT implies that for partial systems not supporting operation INSERT the call to INSERT in program unit DBMS must be replaced by a substitute indicating "operation not implemented", say (cf. figures 8 and 9).

  As an alternative the substitute for program unit INSERT could be a program unit with this task as its only purpose. A drawback of this approach is that the substitute is a program unit with the same name (and calling interface!) as program unit INSERT of the complete program: this complicates the manipulation and maintenance of the software system, e.g. the substitute program unit must be element of a separate "substitute library".

  A third option one might consider for the implementation of partial systems without operation INSERT is to replace both fragments 1 and 2 with their respective substitutes instead of completely replacing program unit INSERT. The problem here is that one cannot distinguish between the indication of "operation not implemented" and the indication of "operation only partially implemented"!.

3.3.2. Number of substitutes per substring

According to definition 2 a fragment is associated with exactly one substitute, in particular, the substitute of a fragment is independent of the partial systems. If for some reason the substitute of a substring q depends on the partial system and, thus, it should be necessary to provide for this string n>1 substitutes s^1, s^2, ..., s^n, this can be achieved by means of nested fragments as follows:

Define n fragments f^1, f^2, ..., f^n, such that f^{i+1} is the only element of f^i, i.e. (cf. section D of appendix)

$$f^1=<f^2>,\ f^2=<f^3>,\ \ldots,\ f^{n-1}=<f^n>$$

and set

$$f^n=<q>,\ \sigma(f^1):=s^1,\ \sigma(f^2):=s^2,\ \sigma(f^n):=s^n.$$

4. B-programs, generation of partial systems

With the concepts and techniques of the previous sections we now study the nature of B-programs and the process of generating a partial system (cf. figures 1 and 4). The problem of determining a set of fragments, a "fragmentation", is addressed and a heuristic procedure is presented. A representation of a B-program as an expansion of the complete program and the generation of partial systems is sketched. The informal discussion of this section provides the rationale for the abstract definition and formal treatment of "generation of partial systems" of section 5.

4.1. Determining fragmentations

Since a partial system should contain only relevant parts of the complete system, in particular no unreachable executable statements or declarations of unreferenced data objects (e.g. variables, arrays), the method of this section for identifying fragments is based on the ideas and techniques discussed in section 3 and relies heavily on flow analysis [9] of the complete program.
We use the following terminology:

- Often it will be the case that certain capabilities, i.e. algorithms, of the complete system are indispensable in that they must be provided by any partial system: an algorithm that is irrelevant for some partial system is called an "optional algorithm".
- We say that a fragment (with executable code) is e x e c u t e d , if at least one statement of the fragment is executed.

The method consists of four steps (as to the examples we assume that the program text of the example system is organized in lines as shown in fig. 7, also it is assumed that all algorithms are optional):

STEP 1:
For each program unit u of the program system define a fragment comprising u.
Explanations:

- Step 1 provides the means to select subsets of the program units of the

```
1|     PROCEDURE DBMS                                                  1
1|        . . . . . . . . .                                           1
1|        IF (OP<1 OR OP>6)                                           1
1|           THEN return 'operation unknown'                          1
1|        CASE OP OF                                                  1
1|0|1|      1,                                                        2
1|0|2|      2,                                                        3
1|0|3|      3,                                                        4
1|0|4|      4,                                                        5
1|0|5|      5,                                                        6
1|0|6|      6,                                                        7
1|0|        0: return 'operation not                                  8
1|0|                    implemented'                                 8
1|1|        1: OPEN                                                   9
1|2|        2: CLOSE                                                 10
1|3|        3: FIND                                                  11
1|4|        4: GET                                                   12
1|5|        5: INSERT                                                13
1|6|        6: DELETE                                                14
1|        END                                                        15
1|     END                                                           15
2|     PROCEDURE OPEN                                                16
2|        OPEN_RF                                                    16
2|1|     OPEN_IF                                                      17
2|     END                                                           18
3|     PROCEDURE FIND                                                19
3|        USE INDEXES                                                19
3|        evaluate INDEX_TABLE                                       19
3|        STRTGY                                                     19
3|        return qss                                                 19
3|     END                                                           19
4|     PROCEDURE STRTGY                                              20
4|        determine access-strategy and                              20
4|         set ACCESS_TYPE                                           20
4|        CASE ACCESS_TYPE OF                                        20
4|          1:                                                       20
4|1|           build seq.search qss                                  21
4|          2: BEGIN                                                22
4|2|           CASE FILE_TYPE OF                                     23
4|2|1|          1: calculate tid                                    24
4|2|2|          2: . . . . . . . .                                  25
4|2|2|             RETRIEVE_TID_LIST                               25
4|2|2|             . . . . . . . .                                  25
4|2|          END                                                   26
4|2|          build direct-access qss                               26
4|            END                                                   27
4|        END                                                       27
4|     END                                                           27
5|     PROCEDURE GET                                                28
5|        NEXT_TUPLE:                                               28
5|        CASE ACCESS_TYPE OF                                       28
5|          1: . . . . . . .                                        28
5|1|           NEXT_SEQ                                            29
5|1|           . . . . . . .                                       29
5|          2: . . . . . . .                                        30
5|2|           NEXT_TID                                            31
5|2|           . . . . . . .                                       31
5|        END                                                       32
5|3|     IF (qualifikation is not satisfied)                         33
5|3|        THEN GO TO NEXT_TUPLE                                   33
5|     END                                                          34
6|     PROCEDURE NEXT_SEQ                                          35
6|        CASE FILE_TYPE OF                                        35
6|          1:                                                     35
6|1|           next_1                                             36
6|          2:                                                     37
6|2|           next_2                                             38
6|        END                                                       39
6|     END                                                          39
7|     PROCEDURE NEXT_TID                                          40
7|       return next tid of tid-list                               40
7|     END                                                          40
```

Fig. 7: Fragmentation of the example system DBMS

```
 8|    PROCEDURE INSERT                                          41
 8|       CASE FILE_TYPE OF                                      41
 8|          1:                                                  41
 8|1|           INSERT_1                                        42
 8|          2:                                                  43
 8|2|           INSERT_2                                        44
 8|       END                                                    45
 8|3|    INSERT_TID                                             46
 8|    END                                                       47
 9|    PROCEDURE CLOSE                                          48
 9|       CLOSE_RF                                              48
 9|1|    CLOSE_IF                                               49
 9|    END                                                       50
10|    PROCEDURE DELETE                                         51
10|       CASE FILE_TYPE OF                                     51
10|          1:                                                 51
10|1|           DELETE_1                                       52
10|          2:                                                 53
10|2|           DELETE_2                                       54
10|       END                                                   55
10|3|    DELETE_TID                                            56
10|    END                                                      57
11|    PROCEDURE OPEN_RF                                       58
11|      . . . . . .                                           58
11|      GET                                                   58
11|      . . . . . .                                           58
11|    END                                                      58
12|    PROCEDURE CLOSE_RF                                      59
12|      . . . . . .                                           59
12|    END                                                      59
13|    PROCEDURE OPEN_IF                                       60
13|      USE INDEXES                                           60
13|      . . . . . .                                           60
13|      GET                                                   60
13|      . . . . . .                                           60
13|    END                                                      60
14|    PROCEDURE CLOSE_IF                                      61
14|      USE INDEXES                                           61
14|      . . . . . .                                           61
14|    END                                                      61
15|    PROCEDURE INSERT_1                                      62
15|      . . . . . .                                           62
15|    END                                                      62
16|    PROCEDURE INSERT_2                                      63
16|      . . . . . .                                           63
16|    END                                                      63
17|    PROCEDURE DELETE_1                                      64
17|      . . . . . .                                           64
17|    END                                                      64
18|    PROCEDURE DELETE_2                                      65
18|      . . . . . .                                           65
18|    END                                                      65
19|    PROCEDURE INSERT_TID                                    66
19|      USE INDEXES                                           66
19|      . . . . . .                                           66
19|    END                                                      66
20|    PROCEDURE DELETE_TID                                    67
20|      USE INDEXES                                           67
20|      . . . . . .                                           67
20|    END                                                      67
21|    PROCEDURE RETRIEVE_TID_LIST                             68
21|      . . . . . .                                           68
21|    END                                                      68
22|    PACKAGE INDEXES                                         69
22|1|    INDEX_TABLE: ARRAY OF INTEGER                         70
22|    END                                                      71
```

Fig. 7: Fragmentation of the example system DBMS (continued)

complete program (cf. section 3.2.1). Program units local to other program units give rise to nested fragments.
If we apply this rule to the example system we obtain the fragments 1 through 22 as shown in fig. 7.

- Identification of these fragments requires only a syntactical analysis of the complete program and, thus, is amenable to automation.

STEP 2:

For each fragment with statements that (i) implement an optional algorithm or (ii) the execution of which leads to the execution of an optional algorithm define subfragments comprising these statements.

Explanations:

- With step 1 the program units of the complete program are available as building blocks for partial systems. The purpose of this step is to make available as building blocks parts of program units that either implement or invoke directly or indirectly an optional algorithm (cf. section 2.1b).
- This is a recursive process in that it may be necessary to apply step 2 to fragments defined according to this step. Examples:
  a) Fragment 4 of step 1 (program unit STRTGY, fig. 7) contains code implementing algorithms 5 and 6 and a call to program unit RETRIEVE_TID_LIST, which implements algorithm 7. When STRTGY is invoked control is transferred to exactly one of the alternatives of the "outer" CASE-statement, thus, at first subfragments 1 and 2 of fragment 4 are introduced, where fragment 4.2 comprises the code of algorithm 6 and the call to program unit RETRIEVE_TID_LIST. Since with each execution of fragment 4.2 exactly one of these pieces of code is executed, step 2 must be applied also to fragment 4.2 yielding fragments 4.2.1 and 4.2.2.
  b) Execution of any alternative of the CASE-statement of fragment 1 leads to the execution of at least one algorithm supporting the respective operation, thus step 2 is applied to fragment 1 and yields the subfragments 1.1 through 1.6.
  Remark: Since the substitutes of these alternatives are identical the technique of section 3.2.3, fig. 6b, is employed!
- Obviously, in-depth knowledge of the internal design of the system and the "meaning" of program statements is indispensable for this step,

syntactical analysis of the complete program alone is insufficient: in general not every CASE-construct of the complete program gives rise to the definition of fragments and, vice versa, a fragment is not necessarily associated with a branch statement.
For instance, the fact that the call to OPEN_IF of fragment 2 (see fig. 7) is superfluous, if there are no inverted files to be locked - this is the reason for introducing fragment 2.1! - cannot be deduced from syntactical properties. Rather, knowledge of the tasks performed by program unit OPEN_IF (cf. section 1, table 2) is required.

- In general the set of subfragments of a fragment f introduced due to step 2 contains subsets X(f), such that with the execution of f exactly one fragment of X(f) is executed. Fragments with this property are called X - f r a g m e n t s of f, the other subfragments introduced in this step are called O - f r a g m e n t s .
  The sets of X-fragments of the example system:

  X(1) = { 1.1 , 1.2 , 1.3 , 1.4 , 1.5 , 1.6 } ,
  X(4) = { 4.1 , 4.2 } , X(4.2) = { 4.2.1 , 4.2.2 } ,
  X(5) = { 5.1 , 5.2 } , X(6) = { 6.1 , 6.2 } ,
  X(8) = { 8.1 , 8.2 } , X(10) = {10.1 ,10.2 } .

  The O-fragments: 2.1 , 5.3 , 8.3 , 9.1 , 10.3

STEP 3:
For each fragment f with statements that can be executed only when subfragments of f are executed define fragments comprising these statements.

STEP 4:
a) For each fragment f with declarations of data objects that are referenced only by statements of subfragments of f define fragments comprising these declarations.
b) For each global data object define a fragment comprising its declaration.

Explanations:
- As has been elaborated in section 3.2.1 in order to obtain partial systems without superfluous code in general after the definition of O- and X-fragments additional derived fragments must be introduced.

Step 3 completes the fragmentation of executable code, in step 4 fragmentation of definitional statements is done. Examples:
In fig. 5 application of steps 3 and 4 yielded fragments E1, E2 and D1, D2, respectively. In fig. 7 step 4 leads to fragment 22.1 with the declaration of the global data object INDEX_TABLE.

- After introducing a fragment due to step 3 the condition for applying this rule may be satisfied by other statements such that fragments comprising them must be defined, i.e. step 3 is in general an iterative process. This is true also for step 4, if definitional statements reference other definitional statements as e.g. type-declarations or separate specifications of initial values (e.g. DATA-statement of FORTRAN).
- Steps 3 and 4 require only flow analysis of the complete program. Program analysers (see e.g. [1]), thus, can at least aid in determining derived fragments.

Remarks:

a) Clearly, the crucial point is the definition of X- and O-fragments in step 2, with these fragments the optional algorithms of the complete system and, thus, to a large extent the set of partial systems are specified.

b) The fragments available with steps 1 and 2 can be viewed as an "initial solution" for a fragmentation, which is iteratively refined in steps 3 and 4. This refinement process is based on flow and syntactical analysis of the complete program only, semantic properties are not taken into account here. It, therefore, leads to a "finest" fragmentation for a given set of X- and O-fragments in that any additional decomposition of the fragments constructed does not increase the set of partial systems.

c) In general this method will lead to superfluous fragments: e.g. fragment E1 of fig. 5 is superfluous in that whenever the fragment comprising program unit INSERT (fragment 8 of fig. 7) is relevant, also the code of fragment E1 must be relevant and vice versa; the reader easily verifies that the same is true for fragment D1. In other words program unit INSERT and its subfragments E1, D1 have the same relevances (i.e. we have $\rho_8 \equiv \rho_{D1} \equiv \rho_{E1}$) such that fragments E1 and D1 can be deleted from F without altering (in particular without reducing)

the set of partial systems.

This observation leads to the investigation in section 5 of techniques for reducing B-programs, i.e. the elimination of superfluous fragments.

d) Since this method works "top-down" there can be no "overlapping" fragments, i.e. a substring of the complete program that is element of a certain fragment cannot be element of any other fragment. Also, each fragment has at most one enclosing fragment.

e) For this method in order to lead to partial systems without unreachable code it is necessary that each fragment satisfies the following m a x i m a l i t y property:

Let s be an executable statement and p‖s or s‖p a substring of the complete program. A fragment f comprising p must also include s, if the execution of s necessarily implies the execution of any statement of p and s is not already comprised by a fragment nested in a fragment enclosing f (cf. definition 3).

Therefore, determination of a fragment comprising a sequence of executable statements p of the complete program entails finding a maximal list of statements that are comprised by the same fragment (or no fragment at all) and include p such that flow of control occurs into the list only to the first statement and once the first statement is executed, all statements in the list are executed sequentially.

Program analysers as e.g. RXVP80 [2] or BRNANL [28] that provide information on the "basic blocks" [9] of program systems can aid in determining such maximal lists of statements.

4.2. Construction of B-programs, generation of partial systems

A B-program must meet at least the following requirements:

1) It must in some form contain the complete program of P, since it must be possible to generate the original complete system.
2) It must include a description of the fragmentation of P: as has been demonstrated above a fragmentation cannot be deduced from the complete program as such.
3) It must be possible to determine for each partial system t and fragment f the relevance value $\rho_f(t)$ and substitute $\sigma(f)$.

We assume, that the relevances and substitutes are given:

- as has been discussed in section 3.3 definition of substitutes is the responsibility of the system designers and/or programmers.
- the topic of constructing the relevances is beyond the scope of this paper and is dealt with elsewhere (chapter 4 of [16]; [29]; see also section 4.3).

For the construction of a B-program requirement 1 suggests to start out from the complete program of P and expand it by adding for each fragment f a description of the substring comprised by f, its relevance ρ_f and its substitute $\sigma(f)$. With the method of section 4.1 defining fragments and expanding the complete program can be combined and done as follows:

Definition of a fragment f comprising a substring q of the complete program with relevance ρ_f and substitute $\sigma(f)$ entails the replacement of q with the string

$$[\ \rho_f\ \sigma(f)\ q\ ]$$

called a b l o c k , where special symbols [and] indicate "begin-of-block" and "end-of-block" respectively. (Additional delimitors may be necessary to separate the three components of a block, this purely syntactical aspect can be neglected for the purpose of this discussion!) Since the method of section 4.1 produces only nested and no overlapping fragments definition of a subfragment f' of f comprising a string q' with

$q = q_1 \| q' \| q_2$ leads to the replacement of q' with the string

$$[\ \rho_{f'}\ \sigma(f')\ q'\ ]$$

such that after introducing f and f' the resulting expanded text contains instead of q the string

$$[\ \rho_f\ \sigma(f)\ q_1\ [\ \rho_{f'}\ \sigma(f')\ q'\ ]\ q_2\ ]$$

Note that "nested blocks" correspond to nested fragments!

From definition 3 and section 4.1 it follows that a B-program constructed in this way

- is a sequence of blocks, where the third component of a block, the "fragment component", itself may be a sequence of (i) substrings of the complete program and (ii) blocks
- contains the complete program in form of substrings
- any substring of the complete program appears in the B-program at most once, i.e. duplication of code does not occur.

Example: Fig. 8 displays the structure of the B-program for the example system constructed along these lines with the fragmentation of fig. 7: the complete program is partitioned into 71 substrings q_i, $1 \leq i \leq 71$, where q_i represents the concatenation of the lines of fig. 7 with the integer i at the right margin.

The process of generating the program of a partial system t, can, then, in principle be thought of as selecting and concatenating substrings of a B-program:

Starting with the first block the blocks of the B-program are evaluated as follows: the relevance value of the fragment is determined. In case the fragment is not relevant for t the substitute component of this block is appended to the program text produced so far (the empty string is assumed as the initial value of the program to be generated); otherwise the fragment component is "processed":

- if it is a substring of the complete program, this string is appended to the program text generated so far
- if it is a list of substrings and blocks the substrings are appended to the program text generated so far, blocks are evaluated, i.e. for each of these blocks as just described the relevance value is determined, ..., etc.

$[\ \rho_1\ \text{NIL}\ q_1\ [\ \rho_{1.0}\ \text{NIL}\ [\ \rho_{1.0.1}\ \text{NIL}\ q_2\ ]\ [\ \rho_{1.0.2}\ \text{NIL}\ q_3\ ]\ [\ \rho_{1.0.3}\ \text{NIL}\ q_4\ ]$
$[\ \rho_{1.0.4}\ \text{NIL}\ q_5\ ]\ [\ \rho_{1.0.5}\ \text{NIL}\ q_6\ ]\ [\ \rho_{1.0.6}\ \text{NIL}\ q_7\ ]\ q_8\ ]$
$[\ \rho_{1.1}\ \text{NIL}\ q_9\ ]\ [\ \rho_{1.2}\ \text{NIL}\ q_{10}\ ]\ [\ \rho_{1.3}\ \text{NIL}\ q_{11}\ ]$
$[\ \rho_{1.4}\ \text{NIL}\ q_{12}\ ]\ [\ \rho_{1.5}\ \text{NIL}\ q_{13}\ ]\ [\ \rho_{1.6}\ \text{NIL}\ q_{14}\ ]\ q_{15}\ ]$
$[\ \rho_2\ \text{NIL}\ q_{16}\ [\ \rho_{2.1}\ \text{NIL}\ q_{17}\ ]\ q_{18}\ ]\ [\ \rho_3\ \text{NIL}\ q_{19}\ ]$
$[\ \rho_4\ \text{NIL}\ q_{20}\ [\ \rho_{4.1}\ \text{NIL}\ q_{21}\ ]\ q_{22}$
$[\ \rho_{4.2}\ \text{NIL}\ q_{23}\ [\ \rho_{4.2.1}\ \text{NIL}\ q_{24}\ ]\ [\ \rho_{4.2.2}\ \text{NIL}\ q_{25}\ ]\ q_{26}\ ]\ q_{27}\ ]$
$[\ \rho_5\ \text{NIL}\ q_{28}\ [\ \rho_{5.1}\ \sigma(5.1)\ q_{29}\ ]\ q_{30}\ [\ \rho_{5.2}\ \sigma(5.2)\ q_{31}\ ]\ q_{32}$
$[\ \rho_{5.3}\ \text{NIL}\ q_{33}\ ]\ q_{34}\ ]$
$[\ \rho_6\ \text{NIL}\ q_{35}\ [\ \rho_{6.1}\ \sigma(6.1)\ q_{36}\ ]\ q_{37}\ [\ \rho_{6.2}\ \sigma(6.2)\ q_{38}\ ]\ q_{39}\ ]\ [\ \rho_7\ \text{NIL}\ q_{40}\ ]$
$[\ \rho_8\ \text{NIL}\ q_{41}\ [\ \rho_{8.1}\ \sigma(8.1)\ q_{42}\ ]\ q_{43}\ [\ \rho_{8.2}\ \sigma(8.2)\ q_{44}\ ]\ q_{45}$
$[\ \rho_{8.3}\ \text{NIL}\ q_{46}\ ]\ q_{47}\ ]\ [\ \rho_9\ \text{NIL}\ q_{48}\ [\ \rho_{9.1}\ \text{NIL}\ q_{49}\ ]\ q_{50}\ ]$
$[\ \rho_{10}\ \text{NIL}\ q_{51}\ [\ \rho_{10.1}\ \text{NIL}\ q_{52}\ ]\ q_{53}\ [\ \rho_{10.2}\ \text{NIL}\ q_{54}\ ]\ q_{55}$
$[\ \rho_{10.3}\ \text{NIL}\ q_{56}\ ]\ q_{57}\ ]\ [\ \rho_{11}\ \text{NIL}\ q_{58}\ ]\ [\ \rho_{12}\ \text{NIL}\ q_{59}\ ]\ [\ \rho_{13}\ \text{NIL}\ q_{60}\ ]$
$[\ \rho_{14}\ \text{NIL}\ q_{61}\ ]\ [\ \rho_{15}\ \text{NIL}\ q_{62}\ ]\ [\ \rho_{16}\ \text{NIL}\ q_{63}\ ]\ [\ \rho_{17}\ \text{NIL}\ q_{64}\ ]$
$[\ \rho_{18}\ \text{NIL}\ q_{65}\ ]\ [\ \rho_{19}\ \text{NIL}\ q_{66}\ ]\ [\ \rho_{20}\ \text{NIL}\ q_{67}\ ]\ [\ \rho_{21}\ \text{NIL}\ q_{68}\ ]$
$[\ \rho_{22}\ \text{NIL}\ q_{69}\ [\ \rho_{22.1}\ \text{NIL}\ q_{70}\ ]\ q_{71}\ ]$

$\sigma(5.1)=\sigma(5.2)$:= return 'illegal access-type'
$\sigma(6.1)=\sigma(6.2)=\sigma(8.1)=\sigma(8.2)$:= return 'storage structure not accessible'

Fig. 8: The B-program of the example system

Note that due to the one-to-one correspondence of fragments and blocks the order of

- the fragments without enclosing fragment
- the components of each fragment

determines the order, in which the substrings of the complete program are concatenated to form the programs of the partial systems. I.e. being a list (and not just a simple set!) is an essential property of a fragment (see definition 3)!

Example:

Fig. 9 shows the program of partial system t_ins. It is the result of applying this procedure to the B-program of fig. 8, when the following relevance values, and only these, are equal to 1 for t=t_ins:

$\rho_1(t)$, $\rho_{1.0}(t)$, $\rho_{1.0.3}(t)$, $\rho_{1.0.4}(t)$, $\rho_{1.0.6}(t)$, $\rho_{1.1}(t)$, $\rho_{1.2}(t)$, $\rho_{1.5}(t)$, $\rho_2(t)$, $\rho_5(t)$, $\rho_{5.1}(t)$, $\rho_{5.3}(t)$, $\rho_6(t)$, $\rho_{6.1}(t)$, $\rho_8(t)$, $\rho_{8.1}(t)$, $\rho_9(t)$, $\rho_{11}(t)$, $\rho_{12}(t)$, $\rho_{15}(t)$.

A detailed presentation of the implementation of B-programs as an expansion of the complete program and the pertaining algorithm for the generation of partial systems is given in [18].

```
1|     PROCEDURE DBMS
1|
1|        IF (OP<1 OR OP>6)
1|           THEN return 'operation unknown'
1|        CASE OP OF
**
1|0|3|      3,
1|0|4|      4,
**
1|0|6|      6,
1|0|        0: return 'operation not
1|0|                implemented'
1|1|        1: OPEN
1|2|        2: CLOSE
**
1|5|        5: INSERT
**
1|        END
1|     END
2|     PROCEDURE OPEN
2|        OPEN_RF
**
2|     END
5|     PROCEDURE GET
5|        NEXT_TUPLE:
5|        CASE ACCESS_TYPE OF
5|          1: . . . . . .
5|1|           NEXT_SEQ
5|1|           . . . . . .
5|          2: . . . . . .
**            return 'illegal access-type'
5|        END
5|3|     IF (qualifikation is not satisfied)
5|3|        THEN GO TO NEXT_TUPLE
5|     END
6|     PROCEDURE NEXT_SEQ
6|        CASE FILE_TYPE OF
6|          1:
6|1|           next_1
6|          2:
**            return 'storage structure not accessible'
6|        END
6|     END
8|     PROCEDURE INSERT
8|        CASE FILE_TYPE OF
8|           1:
8|1|           INSERT_1
8|           2:
**            return 'storage structure not accessible'
8|        END
**
8|     END
9|     PROCEDURE CLOSE
9|        CLOSE_RF
**
9|     END
11|    PROCEDURE OPEN_RF
11|      . . . . . .
11|      GET
11|      . . . . . .
11|    END
12|    PROCEDURE CLOSE_RF
12|      . . . . . .
12|    END
15|    PROCEDURE INSERT_1
15|      . . . . . .
15|    END
```

Fig. 9: The program of the partial system t_ins
(**: marks substitute code, within program units only!)

4.3. The set of partial systems

From the preceding sections it can be seen that in general not any arbitrary subset of the set F of fragments of a program system can be used to build a "correct" partial system. Rather, interdependencies among fragments reflecting e.g. the flow of control must be observed. Consider the following examples from section 3, fig. 5:

a) The fact that (i) execution of fragment 3 may lead to the execution of fragment E2 and (ii) the statements of E2 make sense only together with those of fragment 3 implies the equality of the respective relevances: $\rho_3 \equiv \rho_{E2}$

b) Similarly, flow analysis decrees, that whenever one of the alternatives of the CASE-statement of fig. 5 is part of a partial system also fragment E1 must be incorporated:

$$\rho_{E1} \equiv \rho_1 \text{ OR } \rho_2 .$$

c) Analogously, the relevances of fragments D1 and D2 satisfy

$$\rho_{D2} \equiv \rho_3 \qquad \rho_{D1} \equiv \rho_1 \text{ OR } \rho_2 .$$

d) From b) and c) follows $\rho_{D1} \equiv \rho_{E1}$!

In [29] the notion of "fragment system", basically a directed acyclic graph, is introduced as a formal model of such interdependencies among relevances:

- it is shown that there exists a minimal subset $C \subseteq F$ of fragments such that for each $f \in F$ there is a subset $C(f) \subseteq C$ with

$$\rho_f \equiv \text{OR}_{g \in C(f)} \rho_g$$

- the elements of C are a subset of the X- and O-fragments, they are called "characteristic" fragments
- an algorithm for the construction of C(f) for $f \in F\text{-}C$, i.e. a representation of relevences in terms of relevances of characteristic fragments, is given.

Thus, with $n=|C|$ each partial system $t \in T$ can be represented by an element $\tau(t) \in B^n$, where the components of $\tau(t)$ are the values of the characteristic fragments.

In general, however, the reverse does not hold: correct partial systems have to satisfy a set of constraints of the form

$$OR_{g\varepsilon C1}\rho_g(t)=1 \Longrightarrow OR_{g\varepsilon C2}\rho_g(t)=1$$
$$OR_{g\varepsilon C1}\rho_g \equiv OR_{g\varepsilon C2}\rho_g$$

with $C1,C2 \subseteq C$, such that the set T of partial systems of a program system can be thought of as a subset of B^n.

In [29] it is shown that such constraints can be mechanically derived from the fragment system. However, constraints may also reflect semantic properties of the program system (cf. section 4.1, remark c), as the following example shows.

Example:

From the description of the DBMS user-interface in section 1 it follows that each version of DBMS supporting operation OPEN must also provide operation CLOSE. This implies that any version implementing algorithm 1 must also include algorithm 3, which can be formally expressed as the constraint $\rho_{11} \equiv \rho_{12}$.

5. A formal model for the generation of partial systems

We now present a formalization of the ideas developed more or less intuitively up to this point:
A B-program from above is viewed as a particular instance of an augmented tree, an "abstract B-program", and generation of partial systems is defined in terms of preorder tree-traversal. This provides the groundwork for the formal treatment of one aspect of B-program construction, namely B-program reduction, i.e. simplification of B-programs through elimination of superfluous fragments.

5.1. Abstract B-programs

As above F denotes the set of fragments, $Q \subseteq \Sigma$ be the set of substrings of the program system. According to definition 3 a fragment $f \in F$ is a list of substrings and fragments, i.e. (cf. section D of appendix)

$$f = < f[1], f[2], \ldots, f[n] > \qquad n \geq 1$$

with either $f[i] \in F$ or $f[i] \in Q$.

The nesting of fragments and the association of substrings with fragments can be expressed as a relation S on the set F+Q:

$$S = \{ (f,g) \mid f \in F, g \in F+Q, g \text{ is element of } f \}.$$

The method of section 4 produces fragmentations such that (cf. section 4.1, remark d)

- each fragment has at most one enclosing fragment
- each substring of the complete program, i.e. each element of Q, is element of at most one fragment.

Therefore, if there are k fragments $f_1, \ldots, f_k$ without an enclosing fragment, the graph (F+Q,S) is a set of k ordered trees with these k fragments as roots. Adding to F the "pseudo-fragment" $r:=< f_1, \ldots, f_k >$ (this list is a fragment according to definition 3!) yields a single ordered tree (cf. section H of appendix) with

- the substrings of the program system as the leaves
- the pseudo-fragment r as its root
- the fragments as the other non-leaf vertices.

A B-program of section 4, thus, can be viewed as an instance of an "abstract B-program":

DEFINITION 4:
T be the set of partial systems of a program system, Σ be the set of strings over the alphabet of the programming language the program is written in. Let F, Q and Σ be not empty sets with $Q \subseteq \Sigma$ and $F \cap Q = \emptyset$; $S \subseteq P \times P$ be a relation on $P := F + Q$, σ and ρ be mappings σ: F --> Σ and ρ: $T \times F$ --> B.
(P,S,σ,ρ) is called an a b s t r a c t B - p r o g r a m , if B1, B2 and B3 are satisfied:
B1: (P,S) is an ordered tree with the elements of Q as its leaves
B2: $(f,g) \in S$, $g \in F$, $\rho(t,f)=0$ ==> $\rho(t,g) = 0$
B3: root r of (P,S) satisfies: $\rho(t,r) = 1$ for each $t \in T$ and
$\sigma(r) = NIL$

Remark: Due to $F \subseteq P$ property B1 implies that S is a relation on $F \times P$!

As explained above the B-programs of section 4 can be interpreted as ordered trees, i.e. they satisfy property B1.
The mapping ρ represents the set of relevances of a B-program:

$$\rho(t,f) := \rho_f(t)$$

B2 is the property of nested fragments of section 3.2.2!
B3 gives the definitions of the relevance and substitute of the pseudo-fragment: these values are needed for the formal definition of program generation below. Notice that $\sigma(r)$, since r is always relevant, can be any arbitrary value.
B3 does not imply that a partial system necessarily contains code of the complete system. It is possible that all successors of the root of (P,S) are fragments and that these are not relevant for a particular partial system. Then, this partial system is composed only of the substitutes of the successors of the root.
In particular, definition 4 accommodates the extreme case of the empty string as program of a partial system!

The B-programs of section 4 are special cases of abstract B-programs:
- Definition 4 allows for fragments f and g of a B-program with $\rho(t,f) = \neg\rho(t,g)$ for all $t \in T$, i.e. the relevance of f may be the

negation of that of g: $\rho_f \equiv \neg\rho_g$.
Such an equation can express the fact that the complete system provides capabilities, which cannot coexist in a running system (cf. "restrictive characteristics" in [8]). I.e. an abstract B-program can be a representation of the set of partial systems for rather "inhomogeneous" software systems that cannot be modeled as fragment systems, relevances, thus, can be more complicated than the Boolean expressions of section 4.3

- In section 4 due to step 1 the successors of the root are always fragments, whereas the root of an abstract B-program can have as successors elements of F as well as Q!
- The set Q of definition 4 is just a not empty set of strings, without further restrictions. In particular, it may contain elements representing identical strings or strings with common substrings: in the terminology of section 4 this means that a substring of the complete program may appear as element of more than one fragment, i.e. for the implementation of a B-program code may be duplicated.

Example:
Fig. 10 shows the ordered tree (F+Q,S) of the abstract B-program based on the fragmentation of fig. 7:

- rectangles represent the elements of F, i.e. the fragments; circles stand for the elements of Q, i.e. the substrings of the complete program.
- the name of a leaf representing the substring q_i is the index i, the name of a non-leaf vertex is the name of the corresponding fragment. The name of the pseudo fragment is "DBMS".
  (Due to the different graphical representation of leaf and non-leaf vertices no ambiguity can arise from the fact that in the examples of this paper the name of a fragment and the index of a substring can be identical!)
- the left-to-right ordering of the successors of a vertex f in fig. 10 depicts the order $\leq$ defined on the successor set SUCC(f) (cf. section H of appendix).

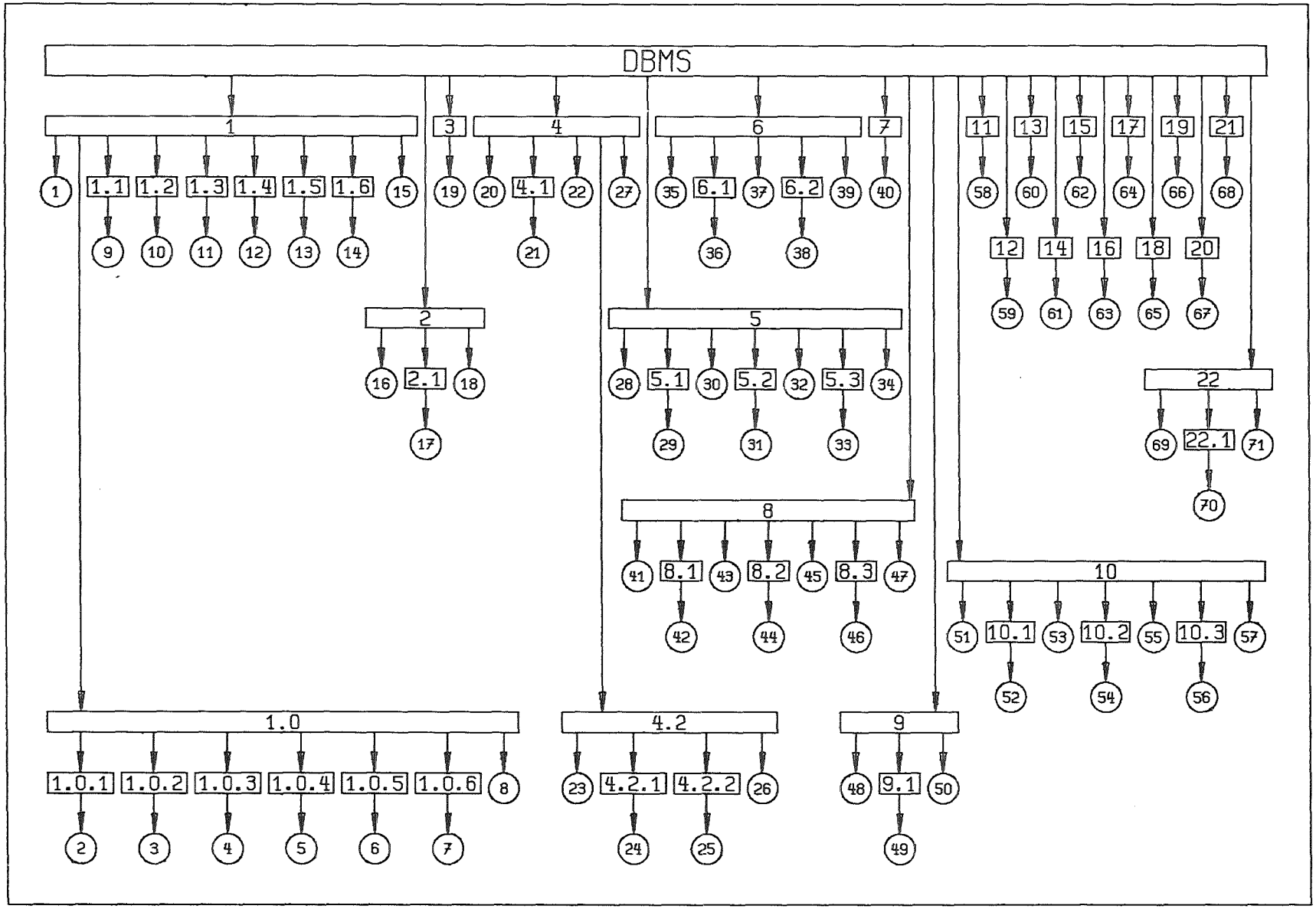


Fig. 10: The ordered tree of the B-program of the example system

5.2. Program generation

It is easy to verify that what in section 4.2 has informally been described as generation of partial systems can be perceived as a traversal of the ordered tree of an abstract B-program:

Algorithm GPS (generation of partial systems):
Input : B-program BP=(P,S,σ,ρ), t ε T
Output: The program text PROG=GPS(t,BP) of the partial system t
Method:

```
            r be the root of (P,S)
            PROG = NIL
            EVAL(t,r)
with
            PROCEDURE EVAL(t,x)
               IF (x ε F)
                  THEN /* evaluate non-leaf vertex */
                       IF (ρ(t,x)=0)
                          THEN PROG = PROG ‖ σ(x)
                          ELSE FOR I=1 TO |SUCC(x)|
                               DO
                               EVAL(t,x[I])
                               END
                  ELSE /* evaluate leaf            */
                       PROG = PROG ‖ x
            END
```

Remark:
It is the concatenation operations of this algorithm that requires Σ and Q of a B-program to be sets of string-valued objects. Note, however, that for algorithm GPS Q ⊆ Σ is no precondition!

Generation of a partial system consists of the "evaluation" of a subset of the vertices of the B-program, where e v a l u a t i o n of a vertex x for t ε T, denoted P(t,x), is defined as follows:

$$P(t,x) := \begin{cases} \text{determine } \rho(t,x) & : x \in F \\ \text{PROG} = \text{PROG} \parallel x & : x \in Q \end{cases}$$

DEFINITION 5:

- Let $BP=(P,S,\sigma,\rho)$ be a B-program, $P=F+Q$, r the root of (P,S) and $t \varepsilon T$. The subtree (P_t,S_t) of (P,S) with

  $P_t := \{ p \mid p \varepsilon P,\ p=r$ or there is $x \varepsilon PRED(p)$ with $\rho(t,x)=1 \}$

  $S_t := S^*(P_t \times P_t)$

  is called the subtree of BP r e l e v a n t for t.
- $SUCC_t(x)$ denotes the set of successors of vertex x with respect to (P_t,S_t):

  $$SUCC_t(x) := SUCC(x)^*P_t$$

Example: Figure 11 depicts the subtree relevant for t_ins.

The following statements are an immediate consequence of property B2:

- $x \varepsilon P_t^*F$ and $\rho(t,x)=0$ ==> x is a leaf of (P_t,S_t) (i.e. $SUCC_t(x)=\emptyset$)
- x is a leaf of (P_t,S_t) ==> ($x \varepsilon P_t^*F$ and $\rho(t,x)=0$) or $x \varepsilon Q$

THEOREM 1:

For the generation of a partial system t algorithm GPS evaluates exactly the vertices of the relevant subtree (P_t,S_t). To this end (P_t,S_t) is traversed in preorder.

Proof:

The first part of this theorem follows immediately from the remarks to definition 5 and algorithm GPS.

As to the order of visiting the vertices of the tree observe that algorithm GPS evaluates

- root r as the first vertex
- immediately after evaluation of any non-leaf vertex x of (P_t,S_t), i.e. of x with $|SUCC_t(x)|>0$, the successors of x are evaluated in the order $\leq$ defined on SUCC(x) (cf. section H of appendix!).

This sequence of evaluations can formally be described as:

PREORDER(t,r)

with

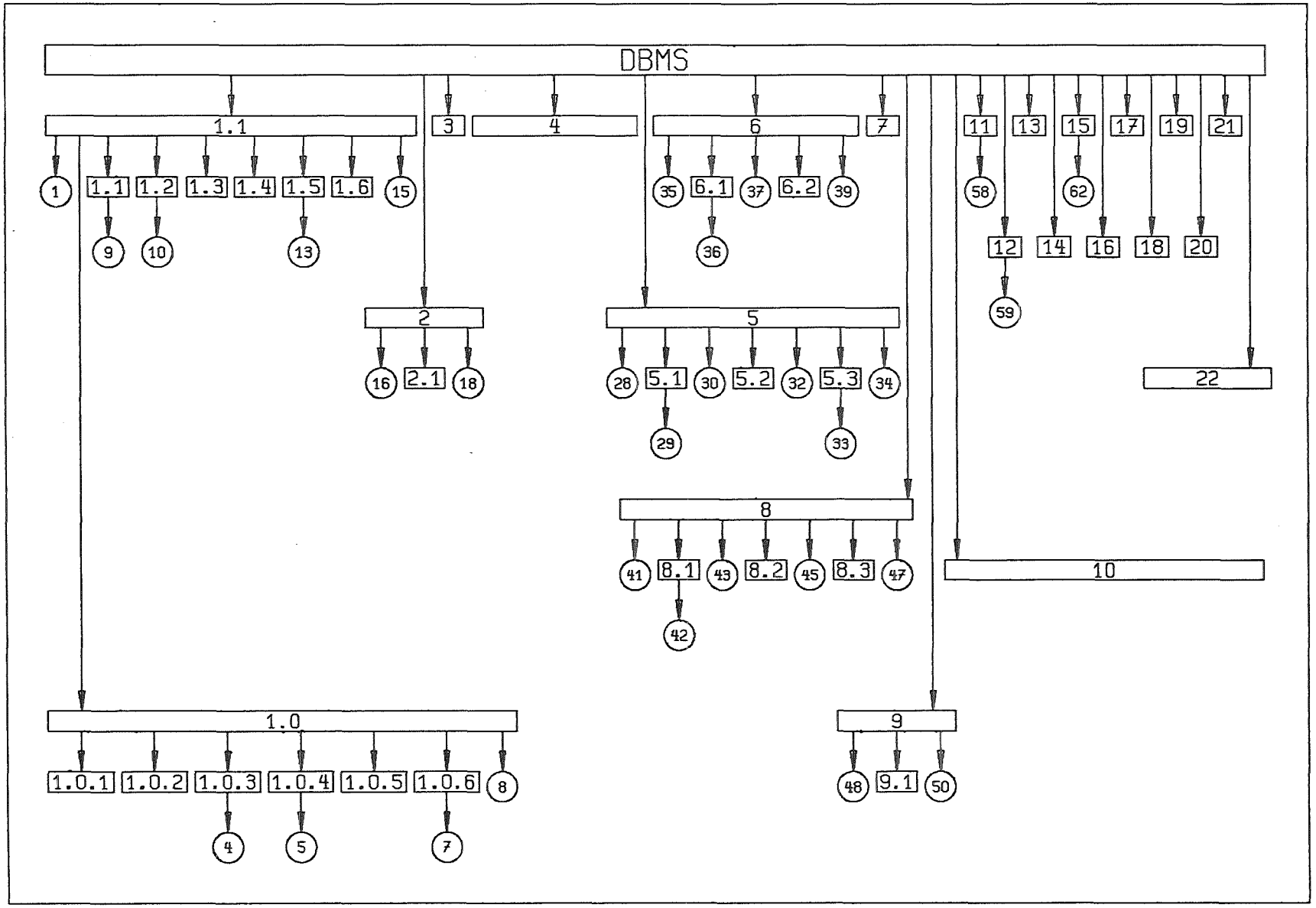


Fig. 11: The subtree relevant for partial system t_ins

```
PROCEDURE PREORDER(t,x)
    P(t,x)
    FOR I=1 TO |SUCC_t(x)|
    DO
        PREORDER(t,x[i])
    END
END
```

(We assume that for n<m

FOR I = m TO n DO statements END

is equivalent to the empty statement, i.e. 'statements' is not executed!) This is the definition of traversal of ordered trees in preorder (cf. e.g. [3], [11], [23]).

□

Notation:

$<_{BP}$ denotes the order induced on the vertices of a B-program BP=(P,S,σ,ρ) by preorder traversal of (P,S), i.e.:

$x <_{BP} y$ <==> traversing (P,S) in preorder vertex x is visited before vertex y.

Remark: The order ≤ defined on the successor sets (cf. section H of appendix) is embodied in $<_{BP}$ in the following sense:

$x \varepsilon SUCC(k), y \varepsilon SUCC(k), x \leq y \quad ==> \quad x <_{BP} y$

It follows immediately from theorem 1 that for each t ε T the string GPS(t,BP) is the concatenation of

- the leaves q ε Q of (P_t,S_t)
- the substitutes of the vertices f ε F that are leaves of (P_t,S_t).

Furthermore, with $x_1,x_2,x_3 \varepsilon \Sigma$ (remember NIL ε Σ!):

- $q_1,q_2 \varepsilon Q^*P_t$, $q_1 <_{BP} q_2 \quad ==> \quad GPS(t,BP) = x_1 \| q_1 \| x_2 \| q_2 \| x_3$
- $q \varepsilon Q^*P_t$, $f \varepsilon F$ is leaf of (P_t,S_t), $q <_{BP} f \quad ==>$

$$==> \quad GPS(t,BP) = x_1 \| q \| x_2 \| \sigma(f) \| x_3$$

I.e., the order in which the substrings of the complete program and substitutes are concatenated is implicitly given with the abstract B-program, thus the program text of a partial system is unambiguously determined with the set of relevant fragments:

Corollary:
A partial system t ε T is uniquely determined with the set

$$F_t := \{ f \mid f \varepsilon F, \rho(t,f)=1 \}$$

5.3. Reduced B-programs

It has been pointed out that constructing fragmentations according to the scheme of section 4 can lead to fragments that are required whenever their enclosing fragments are required and that such fragments can be omitted without altering the set T of partial systems (section 4.1c). In terms of abstract B-programs: an abstract B-program with a smaller number of vertices can be employed for the generation of the same set of partial systems.

DEFINITION 6:
Let BP=(P,S,σ,ρ) and BP'=(P',S',σ',ρ') be two B-programs. BP' is a r e d u c e d B-program with respect to BP if conditions R1 and R2 are satisfied:
R1: $|P'| < |P|$
R2: GPS(t,BP) = GPS(t,BP') holds for each t ε T.

We present three operations on B-programs for transforming a given B-program BP into a reduced B-program BP'.
These "transformations" explicitly refer to the order ≤ defined on the successor sets of BP and BP' in terms of the index mappings ν and ν' of BP and BP', respectively (cf. section H of appendix).
In the following SUCC(f) and PRED(f) denote the successors and predecessors of a vertex f with respect to the ordered tree of B-program BP (and not that of BP'!).

TRANSFORMATION 1:

Let BP=(P,S,σ,ρ) be a B-program, P=F+Q, and f,g ε F with

T1.1: $(f,g) \varepsilon S$

T1.2: $\rho_f \equiv \rho_g$

BP':=(P',S',σ',ρ') with P'=F'+Q' is derived from BP as follows (cf. fig. 12):

$F' := F - \{f,g\} + \{f'\}$ with $f' \neg\varepsilon F$, $\quad Q' := Q$

$S' := S - \{(x,y) \mid (x,y)\varepsilon S,\ x=f \text{ or } x=g\} - \{(x,f) \mid x\varepsilon PRED(f)\}$
$\quad + \{(f',x) \mid x\varepsilon P,\ x\neq g,\ (f,x)\varepsilon S \text{ or } (g,x)\varepsilon S\} + \{(x,f') \mid x\varepsilon PRED(f)\}$

For each t ε T:

$$\rho'(t,x) := \begin{cases} \rho(t,x) & : x \varepsilon F'-\{f'\} \\ \rho(t,f) & : x = f' \end{cases} \qquad \sigma'(x) := \begin{cases} \sigma(x) & : x \varepsilon F'-\{f'\} \\ \sigma(f) & : x = f' \end{cases}$$

With m=|SUCC(g)| and n=|SUCC(f)|:

$$\nu'(x) := \begin{cases} \nu(x) & : x \varepsilon P'-(SUCC(f)+SUCC(g)+\{f'\}) \\ \nu(f) & : x = f' \\ \nu(x) & : x \varepsilon SUCC(f),\ \nu(x)<\nu(g) \\ \nu(x)+\nu(g)-1 & : x \varepsilon SUCC(g) \\ \nu(x)+m & : x \varepsilon SUCC(f),\ \nu(x)>\nu(g) \end{cases}$$

Explanation: Two vertices f and g are merged into a new vertex f' such that the successors of f and g, i.e. the vertices SUCC(f)+SUCC(g)-{g}, become the sucessors of f', see fig. 12. Notice that f' is the only predecessor in (P',S') for each of these vertices and that the other vertices and edges remain unchanged. The graph (P',S'), thus, forms a tree again.

Due to the definition of ν' the order of the vertices relative to each other remains unchanged, in particular:

$$f'[i] = \begin{cases} f[i] & : 1 \leq i \leq \nu(g)-1 \\ g[i+1-\nu(g)] & : \nu(g) \leq i \leq \nu(g)+m-1 \\ f[i+1-m] & : \nu(g)+m \leq i \leq m+n-1 \end{cases}$$

Remark:

It was pointed out in section 5.1 that for the B-programs of section 4 the successors of the root are always fragments. Application of this transformation with f as the root of (P,S) may yield a B-program BP' with an element of Q' as successor of the root of (P',S')!

TRANSFORMATION 2:

Let BP=(P,S,σ,ρ) be a B-program, P=F+Q, and f,g,h ε F with

T2.1: $f,g \in SUCC(h)$

T2.2: $\nu(f)+1=\nu(g)$

T2.3: $\rho_f \equiv \rho_g$

BP':=(P',S',σ',ρ') with P'=F'+Q' is derived from BP as follows (cf. fig. 13):

$F' := F - \{f,g,h\} + \{f',h'\}$ with $f',h' \notin F$, $Q' := Q$

$S' := S - \{(x,y) \mid (x,y) \in S,\ x=f \text{ or } x=g \text{ or } x=h\} - \{(x,h) \mid x \in PRED(h)\}$
$+ \{(f',x) \mid x \in P,\ (f,x) \in S \text{ or } (g,x) \in S\}$
$+ \{(h',x) \mid x \in P,\ (h,x) \in S,\ x \neq f,\ x \neq g\} + \{(h',f')\}$
$+ \{(x,h') \mid x \in PRED(h)\}$

For each t ε T:

$$\rho'(t,x) := \begin{cases} \rho(t,x) & : x \in F'-\{f',h'\} \\ \rho(t,h) & : x=h' \\ \rho(t,f) & : x=f' \end{cases} \qquad \sigma'(x) := \begin{cases} \sigma(x) & : x \in F'-\{f',h'\} \\ \sigma(h) & : x=h' \\ \sigma(f) \| \sigma(g) & : x=f' \end{cases}$$

With n=|SUCC(f)|:

$$\nu'(x) := \begin{cases} \nu(x) & : x \in P'-(SUCC(h)+SUCC(g)+\{f',h'\}) \\ \nu(f) & : x = f' \\ \nu(h) & : x = h' \\ \nu(x) & : x \in SUCC(h),\ \nu(x)<\nu(f) \\ \nu(x)+n & : x \in SUCC(g) \\ \nu(x)-1 & : x \in SUCC(h),\ \nu(x)>\nu(g) \end{cases}$$

Explanation: Two neighboring vertices f and g with predecessor h are merged into a new vertex f' such that the successors of f and g, i.e. the vertices SUCC(f)+SUCC(g) become the successors of f', see fig. 13. Notice that f' is the only predecessor in (P',S') for each of these vertices and that the other vertices and edges remain unchanged (up to renaming). The graph (P',S'), thus, is a tree again.

Due to the definition of ν' the order of the vertices relative to each other remains unchanged. In particular, the successors of f' and h' in (P',S') are ordered as follows (m=|SUCC(g)|):

$$f'[i] = \begin{cases} f[i] & : 1 \le i \le n \\ g[i-n] & : n+1 \le i \le m+n \end{cases} \qquad h'[i] = \begin{cases} h[i] & : 1 \le i \le \nu(f)-1 \\ f' & : i=\nu(f) \\ h[i+1] & : \nu(f)+1 \le i \le |SUCC(h)|-1 \end{cases}$$

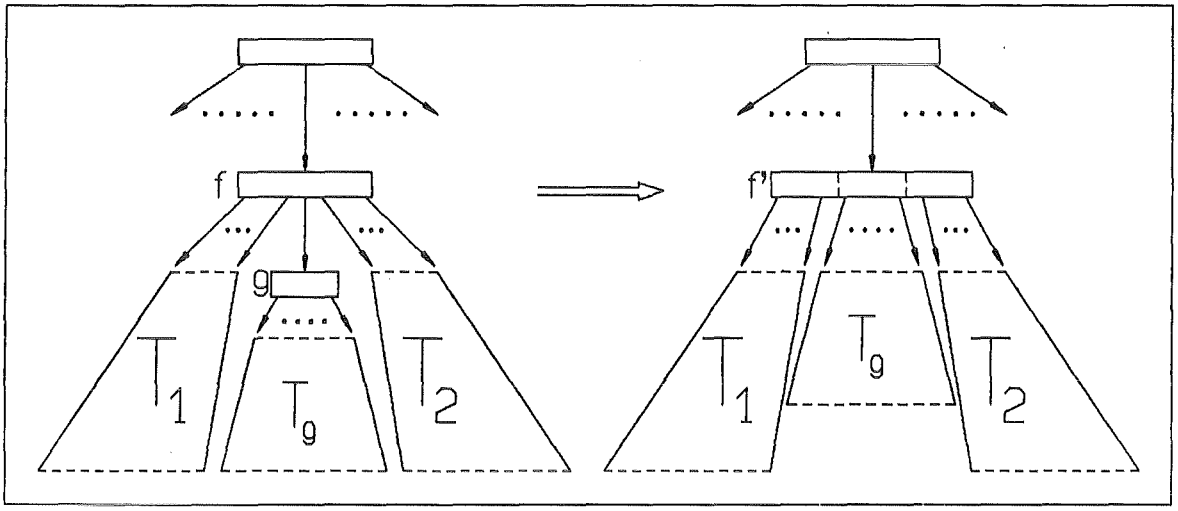


Fig. 12: Reduction of a B-program via transformation 1

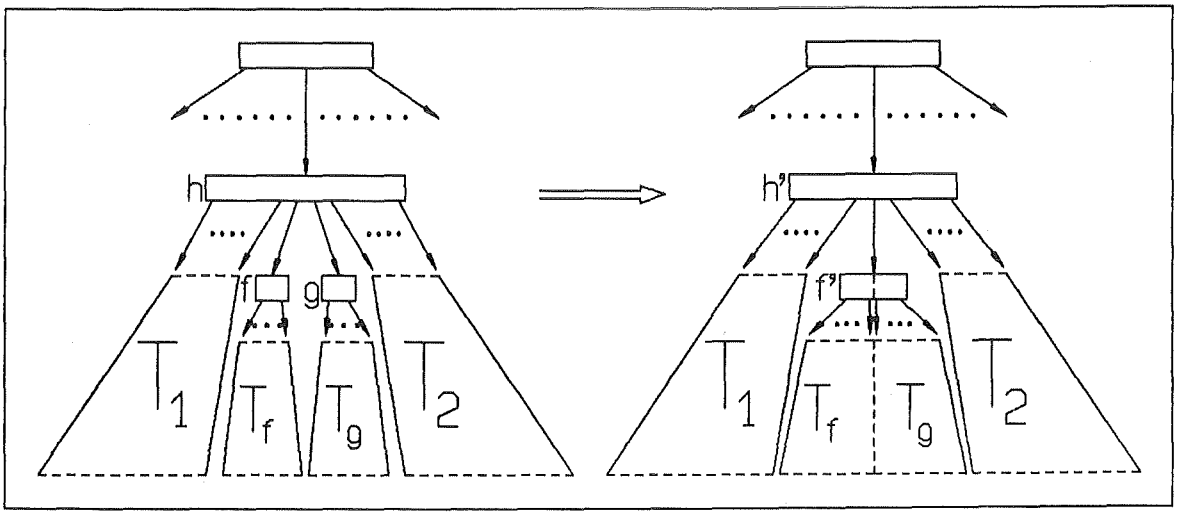


Fig. 13: Reduction of a B-program via transformation 2

TRANSFORMATION 3:

Let BP=(P,S,σ,ρ) be a B-program, P=F+Q, and f ε F and u,v ε Q with

T3.1: $u,v \in SUCC(f)$

T3.2: $\nu(u)+1=\nu(v)$

BP':=(P',S',σ',ρ') with P'=F'+Q' is derived from BP as follows:

$F' := F - \{f\} + \{f'\}$ with $f' \neg\varepsilon F$

$Q' := Q - \{u,v\} + \{q\}$ with $q = u \| v$

$S' := S - \{(x,f) \mid x \in PRED(f)\} - \{(f,x) \mid x \in SUCC(f)\}$
$+ \{(x,f') \mid x \in PRED(f)\}$
$+ \{(f',q)\} + \{(f',x) \mid x \in P, x \in SUCC(f), x \neq u, x \neq v\}$

For each t ε T:

$$\rho'(t,x) := \begin{cases} \rho(t,x) & : x \in F'-\{f'\} \\ \rho(t,f) & : x = f' \end{cases} \qquad \sigma'(x) := \begin{cases} \sigma(x) & : x \in F'-\{f'\} \\ \sigma(f) & : x = f' \end{cases}$$

$$\nu'(x) := \begin{cases} \nu(x) & : x \in P'-(SUCC(f)+\{q,f'\}) \\ \nu(u) & : x = q \\ \nu(f) & : x = f' \\ \nu(x) & : x \in SUCC(f),\ \nu(x)<\nu(u) \\ \nu(x)-1 & : x \in SUCC(f),\ \nu(x)>\nu(v) \end{cases}$$

Explanation: In analogy to transformation 2 here two neighboring l e a v e s with a common predecessor are merged into a single new leaf vertex the only predecessor of which is the predecessor of the leaves being merged. Since the other vertices and edges remain unchanged (up to renaming), the graph (P',S') is a tree again.

The successors of f' in (P',S') are ordered as follows:

$$f'[i] = \begin{cases} f[i] & : 1 \le i \le \nu(u)-1 \\ q & : i = \nu(u) \\ f[i+1] & : \nu(u)+1 \le i \le |SUCC(f)|-1 \end{cases}$$

Remark: From the definitions of ν' it follows immediately, that these transformations are order-preserving, i.e. we have:

$$p_1,p_2 \in P^*P',\ p_1 <_{BP} p_2 \ ==>\ p_1 <_{BP'} p_2$$

THEOREM 2:
Let $BP'=(P',S',\sigma',\rho')$ be the result of applying one of the three transformations to B-program $BP=(P,S,\sigma,\rho)$. Then:
a) BP' is a B-program and
b) BP' is a reduced B-program with respect to BP.

Proof:
a) As has been pointed out with each transformation (P',S') is an ordered tree, i.e. B1 holds for BP'.
Due to the definitions of ρ' and σ' BP' satisfies also properties B2 and B3. Thus, BP' is a B-program.

b) Each transformation replaces two vertices with a single new vertex, i.e. $|P'|=|P|-1$. The remainder of this proof shows for each transformation that $GPS(t,BP)=GPS(t,BP')$ holds for each $t \in T$. (The notation is the one used with the specification of the respective transformation!)

b1) transformation 1:
Because of theorem 1 it is sufficient to show that the result of EVAL(t,f) (GPS applied to BP) is identical to EVAL(t,f') (GPS applied to BP'), i.e. the strings appended to PROG are identical.
Due to T1.2 there are two cases to be considered:
Case 1: $\rho'(t,f')=\rho(t,f)=\rho(t,g)=0$
Because of $\sigma(f)=\sigma'(f')$ nothing is to be shown here.
Case 2: $\rho'(t,f')=\rho(t,f)=\rho(t,g)=1$
Utilizing the definition of f' and T1.1 EVAL(t,f') and EVAL(t,f) are equivalent to the sequences of statements of the left and right columns, respectively (with $n:=|SUCC(f)|$, $m:=|SUCC(g)|$):

```
EVAL(t,f'):                          |  EVAL(t,f):
-------------------------------------+-----------------------------
FOR I=1 TO ν(g)-1                    |  FOR I=1 TO ν(g)-1
DO                                   |  DO
   EVAL(t,f'[I])                     |     EVAL(t,f[I])
END                                  |  END
                                     |
FOR I=ν(g) TO ν(g)+m-1               |
DO                                   |
   EVAL(t,f'[I])                     |  EVAL(t,g)
END                                  |
                                     |
FOR I=ν(g)+m TO m+n-1                |  FOR I=ν(g)+1 TO n
DO                                   |  DO
   EVAL(t,f'[I])                     |     EVAL(t,f[I])
END                                  |  END
```

With the explanation to transformation 1 it is straightforward to see that

- the first and last iteration statements of both columns respectively are equivalent
- the second iteration statement of the left column (EVAL(t,f')) is equivalent to

```
FOR I=1 TO m
DO
   EVAL(t,g[I])
END
```

which in turn is equivalent to EVAL(t,g) of the right column.
This concludes the proof of the theorem for transformation 1.

b2) transformation 2:
In analogy to b1) it must be shown that EVAL(t,h) (for BP) and EVAL(t,h') (for BP') are identical. Because of T2.2 and with the explanations concerning the successors of h it is sufficient to proof that execution of the sequence

```
EVAL(t,f)
EVAL(t,g)
```

is equivalent to the execution of EVAL(t,f'):
Case 1: $\rho'(t,f')=\rho(t,f)=\rho(t,g)=0$
Here, because of $\sigma'(f')=\sigma(f)\|\sigma(g)$ nothing is to be shown.
Case 2: $\rho'(t,f')=\rho(t,f)=\rho(t,g)=1$
With the explanations concerning the successors of f' in (P',S') the

equivalence follows in analogy to b1) directly from the definition of EVAL.

b3) transformation 3:
Because of $\sigma'(f')=\sigma(f)$ and $q = u \| v$ EVAL(t,f) (GPS applied to BP) and EVAL(t,f') (GPS applied to BP') are equivalent and the theorem holds also for transformation 3.
□

5.4. Reducing B-programs

For various reasons it is desirable to reduce a B-program as much as possible:

- Section 4 sketches the implementation of abstract B-programs as expansions of complete programs. Obviously, the smaller the number of fragments of the complete program and, thus, the number of blocks of the corresponding B-program, the less additions (delimitors, relevances, substitute code) are made to the complete program and the easier it is to read and understand the B-program. This is an important aspect, when maintenance of the program system must be done by programmers in terms of editing (the respective substrings of) the B-program.
- Implementations of abstract B-programs based on data structures commonly used to represent directed graphs (see e.g. [3]) may store the strings of the leaves (i.e. the elements of Q) and the substitutes of fragment vertices in files. For obvious reasons, the number of files should be as small as possible.
- Reducing a B-program is a means to speed up the process of generating partial systems:
  Generation of a partial system $t \in T$ out of a B-program $BP=(P,S,\sigma,\rho)$ involves evaluation of the vertices of the relevant subtree (P_t,S_t). Evaluation of a vertex (cf. section 5.2) implies (i) determining the relevance value and locating its successors and/or (ii) looking up and appending a piece of code (substitute code or a substring of the complete program). Clearly, the length of the text to be generated is given with t and the cost (in time) of appending as such is not a function of the number of vertices. However, the task of "looking up a piece of code" is performed for each leaf of the relevant subtree of t. Since this can be a time-consuming operation, it may e.g. involve locating and accessing a file on disc, one should try to minimize the number of vertices to be evaluated.
  Note that for each partial system the relevant subtree of a B-program BP', which is constructed from BP according to one of the transformations 1 through 3, is either identical to the one of BP (up to

renaming) or contains at least one vertex less. In particular, with these transformations the number of leaves can never increase.

Reducing B-programs by means of transformations 1-3 is a multistep procedure as the following examples demonstrate.

Examples:

a) Figure 14a shows the subtree of an abstract B-program based on the fragmentation of program unit INSERT in figure 5. According to section 4.1, remark c, fragments D1 and E1 have the same relevance as their enclosing fragment 8 (program unit INSERT): $\rho_8 \equiv \rho_{D1} \equiv \rho_{E1}$. Therefore, transformation 1 can be applied twice resulting in two vertices being removed (figure 14b). Now it is possible to apply transformation 3 twice to eliminate another two vertices via merging leaves of vertex 8', which leads to the subtree of figure 14c.

b) The fact that STRTGY is invoked if and only if FIND is executed implies $\rho_3 \equiv \rho_4$. Therefore, the B-program of figure 7 can be reduced by means of transformation 2 (figure 15b) and transformation 3 (figure 15c).

Given a B-program it is natural to try to find a minimal reduced B-program, i.e. a reduced B-program with a minimal number of vertices. We can show that under the condition that only transformations 1-3 are employed there is a unique minimal reduced B-program and that the order, in which these transformations are applied for its construction, is irrelevant.

Notation: We write: $B \rightarrow B'$, if B' is obtained from B-program B according to transformation 1 or 2 or 3.

DEFINITION 7:

A list $<B_1, B_2, \ldots, B_n>$ of B-programs is called a r e d u c t i o n s e q u e n c e for B_1, if RS1 and RS2 hold:

RS1: $B_{i-1} \rightarrow B_i$ for $1<i\leq n$

RS2: Neither transformation 1 nor 2 nor 3 can be applied to B-program B_n

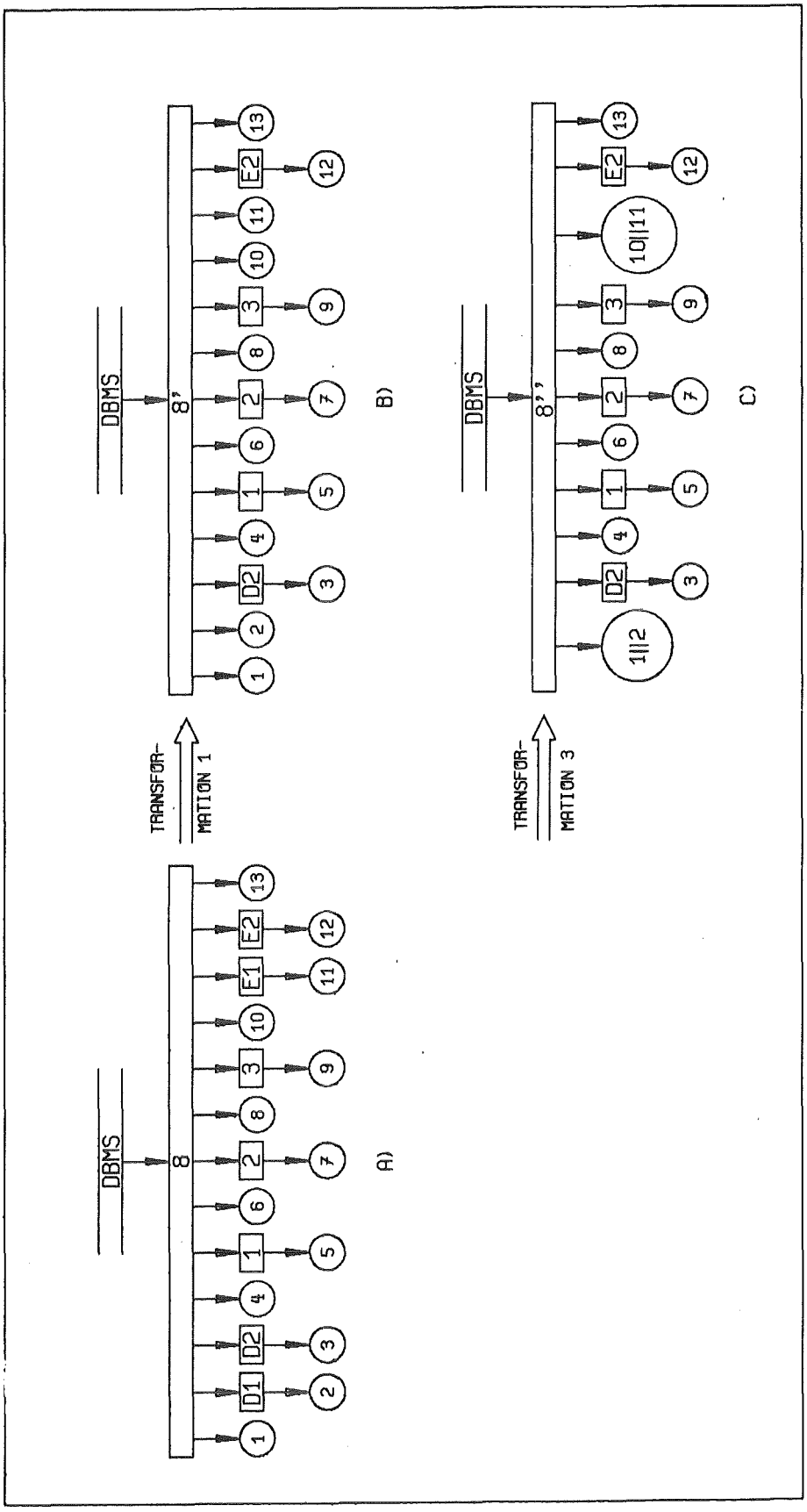


Fig. 14: Application of transformations 1 and 3

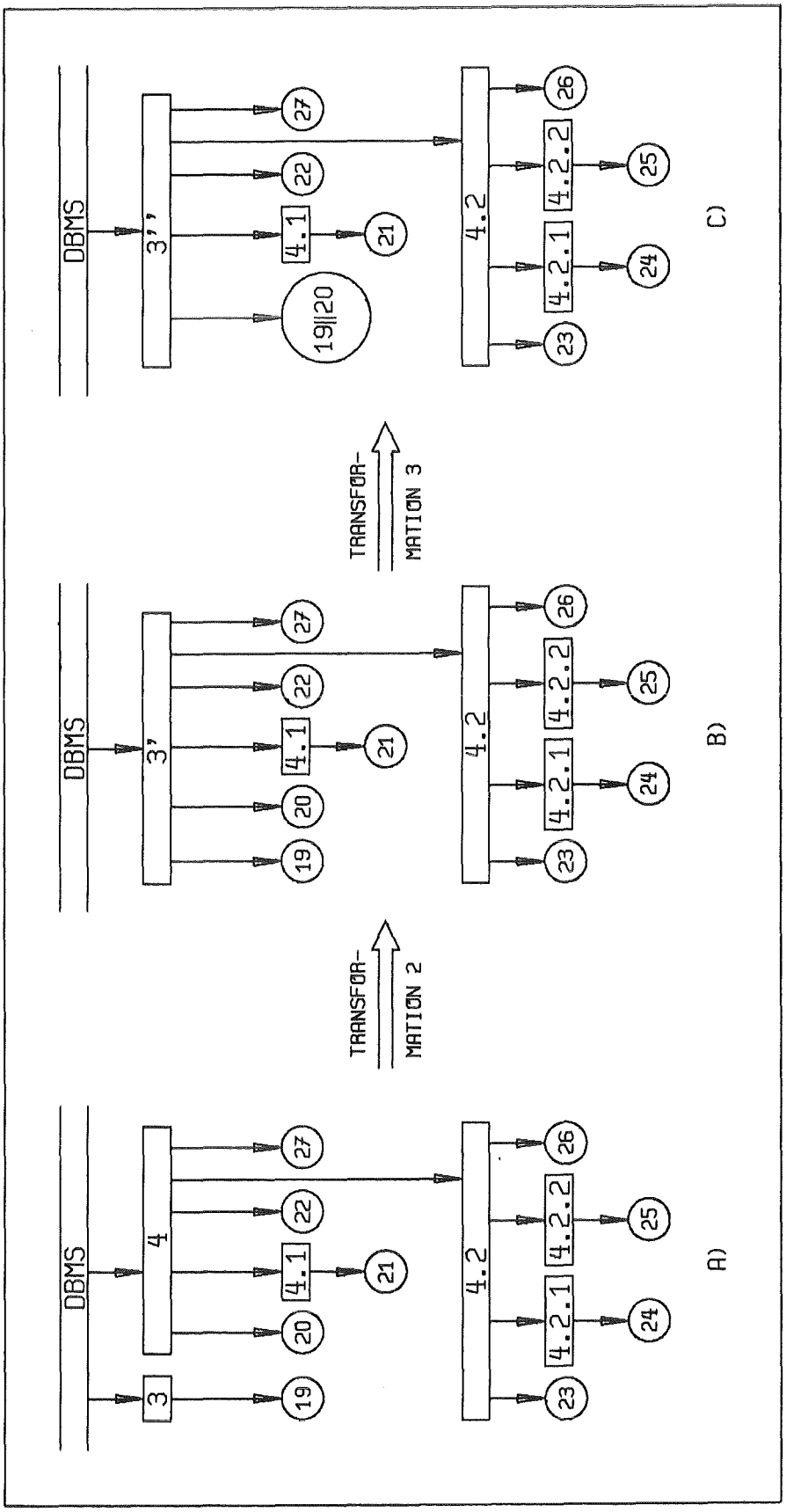


Fig. 15: Application of transformations 2 and 3

THEOREM 3:

Let B be a B-program and

$$<B=B_{1,1}, \ldots, B_{1,k}>$$
$$<B=B_{2,1}, \ldots, B_{2,j}>$$

two reduction sequences for B.

Then $B_{1,k}$ and $B_{2,j}$ are identical up to renaming, denoted: $B_{1,k} \simeq B_{2,j}$, and k=j.

Proof:

The proof is by induction on the number of vertices of B.

Inductive hypothesis: the statement of the theorem holds for B-programs with n vertices, $n \geq 2$.

Basis: n=2, this case is vacuous.

Inductive step:

Assume the inductive hypothesis is true for B-programs with up to n vertices.

B be a B-program with n+1 vertices, let there be $m \geq 1$ pairs of vertices that can be merged with one of the transformations. I.e. the reduction sequences for B are

$$<B=B_{1,1}, B_{1,2}, \ldots >$$
$$\vdots$$
$$<B=B_{m,1}, B_{m,2}, \ldots >$$

where the B-programs $B_{i,2}$, $1 \leq i \leq m$, have n vertices each such that the inductive hypothesis holds for these B-programs.

If m=1 the statement of the theorem holds also for B.

For m>1, without loss of generality, it is sufficient to show for the reduction sequences

$$<B_{1,1}, B_{1,2}, \ldots, B_{1,k}>$$
$$<B_{2,1}, B_{2,2}, \ldots, B_{2,j}>$$

that $B_{1,k} \simeq B_{2,j}$ holds. Note that from this follows immediately k=j since each transformation eliminates exactly one vertex!

Let (u_1,v_1) be the pair of vertices merged by $B \rightarrow B_{1,2}$ into vertex u^1 of $B_{1,2}$ and (u_2,v_2) the vertices merged by $B \rightarrow B_{2,2}$ into vertex u^2 of $B_{2,2}$.

Case 1: $\{u_1,v_1\}*\{u_2,v_2\} = \emptyset$

There are reduction sequences

$$<B_{1,2}, B_{1,3}, \ldots >$$
$$<B_{2,2}, B_{2,3}, \ldots >$$

such that $B_{1,2} \rightarrow B_{1,3}$ merges the pair (u_2,v_2) and $B_{2,2} \rightarrow B_{2,3}$ merges the pair (u_1,v_1), and thus $B_{1,3} \simeq B_{2,3}$ (the transformations are order-preserving, see remark of section 5.3).

Case 2: $\{u_1,v_1\}*\{u_2,v_2\} \neq \emptyset$

The situations to be considered are illustrated in fig. 16 (we assume without loss of generality $u_2=v_1$):

(1), (2) and (3) show the situations that can arise when v_1 is a successor of u_1 (without loss of generality); (4) refers to the situation, where these vertices are neighbors: they must be either all leaves or all fragments (the latter is not shown).

For each of these situations there are reduction sequences

$$<B_{1,2}, B_{1,3}, \ldots >$$
$$<B_{2,2}, B_{2,3}, \ldots >$$

such that $B_{1,2} \rightarrow B_{1,3}$ merges the pair (u^1,v_2) and $B_{2,2} \rightarrow B_{2,3}$ merges the pair (u_1,u^2). Here, too, we have $B_{1,3} \simeq B_{2,3}$.

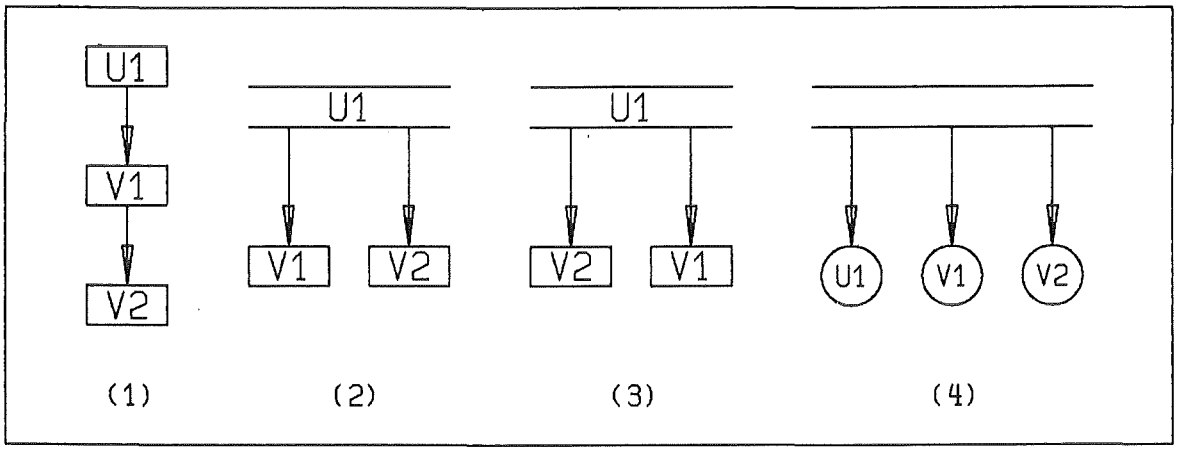


Fig. 16: Merging of vertices in proof of theorem 3

Since $<B_{1,2}, B_{1,3}, \ldots, B_{1,k}>$ is a reduction sequence, due to the induction hypothesis for any reduction sequence $<B_{1,2}, \ldots, B_{1,r_1}>$ holds $B_{1,k} \simeq B_{1,r_1}$. Analogously, for any reduction sequence $<B_{2,2}, \ldots, B_{2,r_2}>$ holds $B_{2,j} \simeq B_{2,r_2}$.

Because of $B_{1,3} \cong B_{2,3}$ this implies $B_{1,r_1} \cong B_{2,r_2}$, therefore: $B_{1,k} \cong B_{2,j}$ and k=j. This proves the statement of the theorem also for m>1.
□

As an immediate consequence we have shown
Corollary:
Provided that only transformations 1, 2 and 3 are employed for reducing a B-program B there exists exactly one (up to renaming) minimal reduced B-program B_{min} with respect to B. B_{min} is obtained by applying these transformations in any order.

Caveat:
Transformations 1-3 provide means for removing superfluous fragments, which may be introduced with the method of section 4.1. We are not aware of other operations for B-program reduction that can formally be specified in terms of B-programs only. This, however, is n o t meant to say that there are no other techniques, which could lead to reduced B-programs smaller than the minimal B-program B_{min} of the corollary.
In fact, as the example below illustrates, since transformations 2 and 3 merge only vertices that are immediate neighbors it is possible to arrive at smaller reduced B-programs, if changing the order of the vertices is allowed. The order of the vertices of a B-program, however, determines the order of the substrings of the complete program, see section 5.2. I.e. not every arbitrary reordering of vertices is allowed. Rather, it seems to be necessary to introduce the notion of "permissible reordering": intuitively, reordering vertices is permissible if and only if the corresponding changes in the order of the substrings leave the complete program semantically unchanged, i.e. program execution must not be affected by rearranging the respective textual components of the complete program:

- In general the order of separately compilable program units of a program system is irrelevant and can be changed arbitrarily.
- Common programming languages allow to change the order of definitional statements within a program unit without affecting program execution.

A precise definition of what constitutes a "permissible reordering" seems to be dependent on the programming language the complete program is written in.

The following example demonstrates how reordering of vertices can be employed for B-program reduction.

Example:

We assume that the ordering of the program units of the example system is irrelevant. Then it is permissible to modify the ordered tree of figure 10 such that fragment 11 becomes the right neighbor of fragment 2. Note that this corresponds to textually rearranging the program of fig. 7 such that program unit OPEN_RF immediately succeeds program unit OPEN.

Then, because of $\rho_2 \equiv \rho_{11}$ (this identity reflects the fact that whenever OPEN is invoked also OPEN_RF is executed and that OPEN is the only program unit calling OPEN_RF) this B-program can further be reduced by means of transformations 2 and 3.

Remarks:

- Reordering of vertices of B-program can be viewed as a fourth transformation, which in essence consists only in defining a new mapping ν' without merging vertices, i.e. the ordered trees (P,S) and (P',S') are isomorphic [3]!
- Such a "reordering-transformation" would entail a generalization of definition 6:
  Condition R2 could no longer be interpreted as postulating the equality of the program texts GPS(t,BP) and GPS(t,BP'). Rather, it had to denote the equivalence of the "behavior" of these two programs at run-time: given the same input both programs must produce identical results.
- If reordering is allowed for reducing B-programs, the extent to which reduction is possible depends on the order, in which these four transformations are applied. This is illustrated in fig. 17, where we assume $\rho_2 \equiv \rho_3$:
  - Immediate application of transformation 2 yields a reduced B-program with 11 vertices that cannot be further reduced, fig. 17b.
  - Reordering of vertices 2 and 3 first (fig. 17c), however, leads to the B-program of fig. 17d, which can by means of transformation 3 be reduced to a B-program with 10 vertices (fig. 17e).
- This example also demonstrates that, when reordering is allowed as a fourth transformation, there may be reduction sequences, which do not lead to a minimal reduced B-program. I.e. the order, in which these four transformations are applied, is essential for the construction of

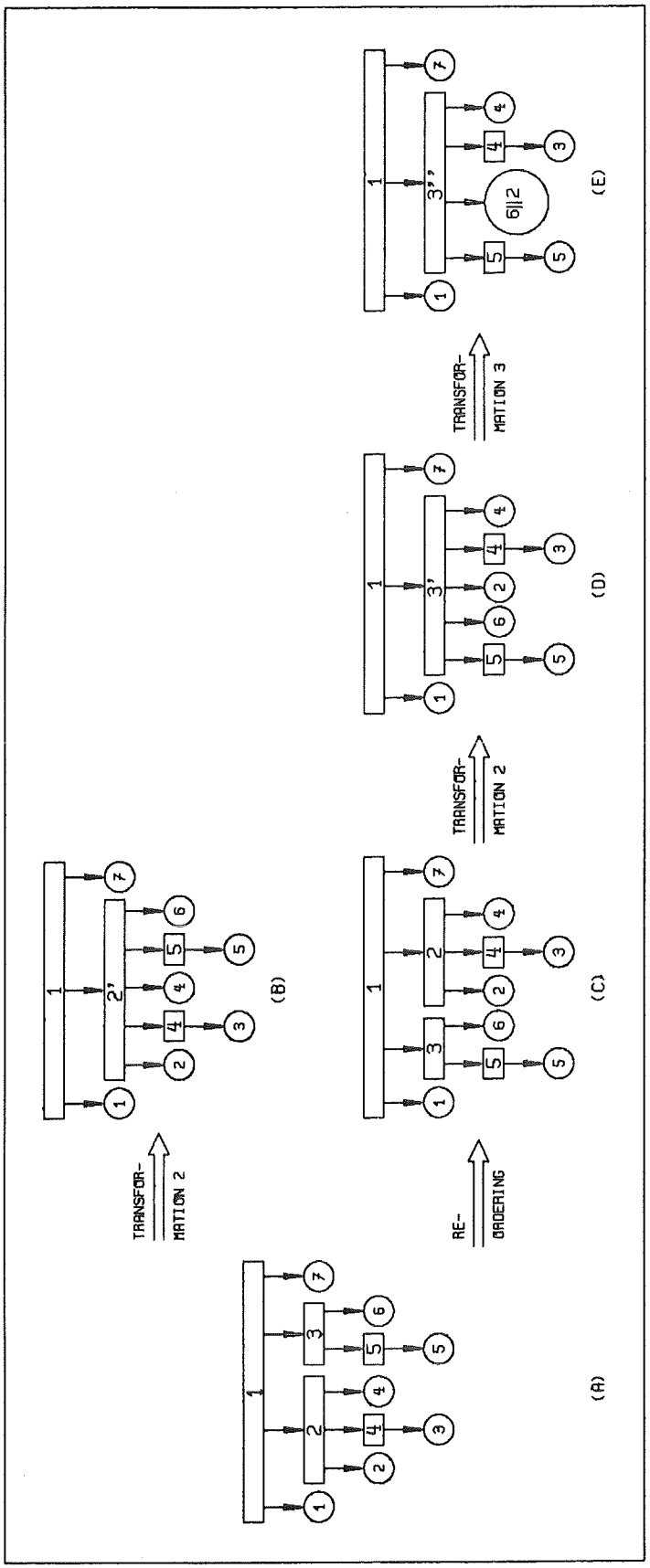


Fig. 17: Reducing B-programs with reordering of vertices

a minimal reduced B-program!

6. Conclusions

A special instance of reuse of software - construction of program systems out of parts of an existing software system - has been dealt with. The motivation for this work is the fact that for a particular application frequently only a subset of the capabilities provided by program systems as e.g. operating systems, compilers, database management systems, business application packages, is relevant, i.e. that partial systems of such general software systems are sufficient in many situations.
A formal model for the generation of partial systems of a software system has been presented:
The notion of B-program, an augmented ordered tree, has been introduced as a representation of the set of partial systems that can be generated. It offers two types of building blocks for the program of a partial system: substrings of the program of the software system and substitutes. They are considered uniformly as strings over some alphabet.
Generation of partial systems has been defined in terms of traversing this tree and concatenating strings associated with the vertices visited.

This model is a generalization of conditional compilation, preprocessing techniques as e.g. the "compile-time operations" of [27] and various mechanisms employed in "customizing systems" (see e.g. [24], [8]). Also, a B-program can be viewed as a "metaprogram" in the sense of [25], [26], i.e. a program, which "at once possesses several distinct implementations and the processing of which results in the choice of just one possible implementation and the production of a program module" [26].

The ideas and concepts of this formal model have been tried and put to work in a system for the generation of partial systems of a database management system [17], [18]. Here, a general purpose macro processor has been employed for the implementation of algorithm GPS and serves as the selector utility (cf. section 2.3, fig. 4). Therefore, the blocks of

B-programs take the form of macro calls. A generation run has four steps:

1) it is checked, whether the partial system to be generated is a correct one, i.e. whether the description passed to the generator (V_DES of fig. 4) satisfies the constraints characterizing the set of partial systems [29].
2) generation of job control programs (cf. section 2.3)
3) the actual generation of a source program
4) production of a load module: compilation and linking according to the respective job control program generated in step 2.

This program generator exploits the fact that our model makes no specific assumptions as to the nature of "code": in [18] it is shown that not only step 3, but also the first two steps can be viewed as instances of the problem of generating a partial system, consequently these subtasks, too, are implemented using the selector utility. The program generator, therefore, contains several B-programs: (1) a B-program representing the constraints characterizing the set of partial systems; (2) a B-program representing the set of control programs for the linker and translator utilities each; (3) the B-program representing the set of partial systems of the database management system itself.

We have presented a heuristic method for the systematic construction of a B-program for a given software system. It has been demonstrated that this task in principle cannot be automated, rather human interaction is required. Subtasks amenable to computerization have been pointed out.
Future research should be in the area of computer-aided construction of B-programs, tools supporting or even automating the following subtasks should be developed:

- given a set of X- and O-fragments derivation of a fragmentation for the software system at hand based on an analysis of the syntactical structure as well as data and control flow
- definition and administration of substitutes
- determination of fragments with identical relevances (cf. also [29])
- B-program reduction, i.e. construction of a minimal B-program

It would be interesting to see to which extent such tools become language-dependent (remember e.g. that the task of B-program reduction is independent of programming languages, if only transformations 1-3 are employed!).

Also, research is needed to develop ways of ensuring that the partial systems that can be generated from a given B-program are syntactically and semantically correct: We suspect that with the method of section 4.1 for the determination of a fragmentation the syntactic correctness of partial systems is a consequence of the syntactic correctness of the complete program. As to semantic correctness, since partial systems are composed of parts of an existing software system a "natural" approach would be to try to reuse the efforts and means for the validation and verification of the complete system (e.g. correctness proves, test drivers, test data and results [1]).
Another problem is that of verifying that the partial systems that can be generated are free of superfluous code.
Clearly, investigations along these lines entail a precise and formal specification of the procedure of determining fragments.

APPENDIX

This section gives the basic definitions and notations used in this paper.
Let M and N be sets:

A: The cardinality of M, denoted: $|M|$, is the number of elements in M.
$\emptyset$ denotes the e m p t y s e t , i.e. $|\emptyset|=0$.

B: $M*N$ denotes the intersection, $M+N$ the union of M and N.
The Cartesian product of M and N is the set

$$M \times N := \{ (m,n) \mid m \in M, n \in N \}$$

C: A set $R \subseteq M \times N$ is called a (binary) relation between M and N.
A p a r t i a l o r d e r on M is a relation $R \subseteq M \times M$ such that:

- $(x,x) \in R$ for each $x \in M$ (R is reflexive)
- $(x,y) \in R, (y,x) \in R \Longrightarrow x=y$ (R is antisymmetric)
- $(x,y) \in R, (y,z) \in R \Longrightarrow (x,z) \in R$ (R is transitive)

D: With R an order on M a set $L \subseteq M$ is called a l i s t , if for each pair $(x,y) \in L \times L$ either $(x,y) \in R$ or $(y,x) \in R$. L[i] denotes the i-th element of list L, L is written using angular brackets:

$$L = < L[1], \ldots, L[i], \ldots >$$

E: A m a p p i n g $f: M \dashrightarrow N$ is a relation $f \subseteq M \times N$ such that

$$(x,y) \in f, (x,z) \in f \Longrightarrow y=z$$

Two mappings $f: M \dashrightarrow N$, $g: M \dashrightarrow N$ are said to be e q u a l , denoted: $f \equiv g$, if $f(x)=g(x)$ holds for each $x \in M$.

F: A d i r e c t e d g r a p h is a pair G=(M,R), where M is a set and R a binary relation R $\subseteq$ M×M. The elements of M are called the v e r t i c e s , the elements of R the e d g e s of G.
Let k,k_1,k_2 be vertices of a directed graph G=(M,R):

- The p r e d e c e s s o r s of k in G are the vertices of the set PRED(k):

$$PRED(k) := \{ x \mid x \in M, (x,k) \in R \}$$

- The s u c c e s s o r s of k in G are the vertices of the set SUCC(k):

$$SUCC(k) := \{ x \mid x \in M, (k,x) \in R \}$$

- A p a t h P from x to y is a list of $n \geq 2$ vertices k_i, $1 \leq i \leq n$, with $(k_i,k_{i+1}) \in R$ for $1 \leq i \leq n-1$ and $k_1=x$, $k_n=y$. P is a c y c l e if x=y.

G: A t r e e is a directed graph T=(M,R) such that:

1. T has no cycles
2. there is exactly one vertex $r \in M$ with $PRED(r)=\emptyset$; r is the r o o t of T
3. $k \in M, k \neq r \Longrightarrow |PRED(k)|=1$
4. for each vertex $k \in M, k \neq r$ there exists a path from r to k.

A vertex $k \in M$ without successors, i.e. $|SUCC(k)|=0$, is called a l e a f of T.

H: A tree T=(M,R) is an o r d e r e d t r e e , if for each $k \in M$ the set SUCC(k) is a list.

- We use the symbol $\leq$ to denote the order defined on the vertices of each of the successor sets, i.e.:

  for each $k \in M$ and $x \in SUCC(k)$, $y \in SUCC(k)$ we have $x \leq y$ or $y \leq x$

  Remark: In general $\leq$ is a partial order on M, because for $x \in SUCC(k_1)$, $y \in SUCC(k_2)$ with $k_1 \neq k_2$ neither $x \leq y$ nor $y \leq x$ must hold!
- The i-th successor of k, i.e. the i-th element of list SUCC(k), is denoted k[i], i.e. for $x \in SUCC(k)$, $y \in SUCC(k)$ with x=k[i], y=k[j] we have: $x \leq y \Longleftrightarrow i \leq j$
- for $k \in M$ with predecessor k' $\nu(k)$ denotes the i n d e x of k in the list SUCC(k'): $k = k'[\nu(k)]$
  Note that ν is a mapping $M \longrightarrow \{ i \mid 1 \leq i \leq \max_{k \in M} |SUCC(k)| \}$! It is called the i n d e x m a p p i n g of the ordered tree.

- with $k_1, k_2 \in SUCC(k)$ vertex k_1 is the left neighbor of k_2 and k_2 the right neighbor of k_1, if $\nu(k_1)+1=\nu(k_2)$.
  Note that only vertices with a common predecessor can be neighbors!

REFERENCES

[1] Adrion, W.R.; et al.: Validation, Verification, and Testing of Computer Software. ACM Computing Surveys 14,2 (June 1982), p.159-192

[2] Andrews, C.L.; DeHaan, W.R.: RXVP80: The Verification and Validation System for FORTRAN (RXVP80/FORTRAN) and COBOL (RXVP80/COBOL). Proceedings: SOFTFAIR - a Conference on Software Development Tools, Techniques and Alternatives. Arlington, VA, USA. July 25-28, 1983. IEEE Comput. Soc. Press, p. 38-47

[3] Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Company, 1974

[4] Aho, A.V.; Ullman, J.D.: The Theory of Parsing, Translation and Compiling (vol. 1). Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1972

[5] Boehm, B.W.: Improving software productivity. Proc. IEEE COMPCON 81. Washington D.C., Sept. 15-17, 1981, p. 2

[6] Cole, A.J.: Macro Processors. Cambridge University Press, Cambridge, Great Britain, 1981

[7] Freeman, P.: Reusable Software Engineering: Concepts and Research Directions. In: Freeman, P.; Wasserman, A.I. (ed.): Tutorial on Software Design Techniques (4th edition), p. 63-76. IEEE Computer Society Press, 1983

[8] Gordon, R.D.: The Modular Application Customizing System. IBM Systems Journal 19,4(1980), p. 521-541

[9] Hecht, M.S.: Flow Analysis of Computer Programs. North Holland, New York, 1977

[10] Jensen, K.; Wirth, N.: PASCAL User Manual And Report. 2nd ed., Springer Verlag. Berlin, 1978.

[11] Knuth, D.E.: The Art of Computer Programming, vol. 1. Addison-Wesley Publishing Company, 1969

[12] Kernighan, B.W.; Plauger, P.J.: Software Tools. Addison-Wesley Publishing Company, 1976.

[13] Musa, D.J. (ed.): Stimulating Software Engineering Progress - a Report of the Software Engineering Planning Group. ACM Software Engineering Notes 8,2 (Apr. 1983), p. 29-54

[14] Nehmer, J.: Betriebssysteme für Kleinrechner. Angewandte Informatik, 1/77, p. 1-14

[15] Parnas, D.L.; Handzel, G.; Würges, H.: Design and Specification of the Minimal Subset of an Operating System Family. IEEE Transactions on Software Engineering 2,4 (Dec. 1976) p. 301-307

[16] Polster, F.J.: Generierbare Datenbanksysteme. Doctoral Dissertation, University of Karlsruhe, Fakultät für Informatik, Fed. Rep. Germany, Dec. 1982. (In German, also available as report KfK 3479, Kernforschungszentrum Karlsruhe, Fed. Rep. Germany, Febr. 1983)

[17] Polster, F.J.: Dedication of general database software to specific applications. Proceedings IEEE Computer Societies 7th International Computer Software and Applications Conference (COMPSAC '83), Chicago, Nov 7-11, 1983, p. 201-210

[18] Polster, F.J.: Adaptation of Program Systems Through Code Selection. Kernforschungszentrum Karlsruhe, Fed. Rep. Germany, Sept. 1983, submitted for publication.

[19] Rüb, W.; Schrott, G.: Automatische Generierung problemangepaßter Prozeßrechner-Betriebssysteme. Angewandte Informatik 1/80, p.7-17

[20] Shaw, M.: Reduction of Compilation Costs Through Language Contraction. CACM 17,5 (May 1974), p. 245-250

[21] Schrott, G.: Generation of dedicated realtime operating systems by dialogue. Proc. IFAC/IFIP Workshop on Real Time Programming, p. 145-151. Eindhoven, Netherlands. June 1977

[22] Tannenbaum, A.S.: Structured Computer Organization. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1976

[23] Wirth, N.: Algorithmen und Datenstrukturen. B.G.Teubner, Stuttgart, 1979

[24] Winston, L.E.: A Novel Approach to Computer Application System Design and Implementation. Hewlett-Packard Journal, April 1981, p. 13-18

[25] Flon, L.; Cooprider, L.W.: Metaprogramming - Prospects for the Practical Reuse of Software. Technical report TR-112, Computer Science Department, University of Southern California, Los Angeles, 1982

[26] Flon, L.; Raeder, G.: Metaprogramming - Language and Examples. Technical report TR-113, Computer Science Department, University of Southern California, Los Angeles, 1982

[27] PL/I Checkout and Optimizing Compilers: Language Reference Manual. GC33-0009-3, IBM United Kingdom Laboratories, 1974

[28] Fosdick, L.D.: BRNANL, a Fortran Program to Identify Basic Blocks in Fortran Programs. Report CU-CS-040-74, Department of Computer Science, University of Colorado, Boulder. March 1974.

[29] Polster, F.J.: A Theory of Partial Systems. A translation of chapter 4 of [16], submitted for publication.

[30] IEEE Transactions on Software Engineering, Special Issue on Software Reusability, vol. 10, No. 5.