GRAPHS OF DATA FLOW DEPENDENCIES

P. Puhr-Westerheide

Gesellschaft für Reaktorsicherheit (GRS)mbH, Forschungsgelände, 8046 Garching, Federal Republic of Germany

Abstract. Data flow dependencies of a program are important for code-optimisation of compilers and for detecting errors in programs. The control flow of a program can be described by its control flow graph. The mutual dependencies of variables in a program can be represented by graphs, too. For each variable of a program a tree structure can be found that shows its dependency on other variables. Different kinds of tree structures depict the data flow dependencies more or less detailed, according to the more or less exhaustive use of the source code's information. The most accurate representation of data flow by tree structures can be achieved for individual paths of a program only. A less accurate representation results from the set of unordered program statements. In this case it is not possible to distinguish whether a statement precedes another or not. By use of the static control flow graph, trees can be established without regard to branching conditions. With these trees distinctions can be made between different actual values of the same variable. The methods of building tree structures that show the data flow of programs are suitable for data flow analysis by hand as well as for automatic analysis.

Keywords. Automatic testing, data flow dependencies, graph theory, programming control, computer debugging, trees, analyzing tools.

1. INTRODUCTION

It is the task of software programs to control computers in such way that they provide meaningful output data dependent on the offered input data. Since computers are involved in processes that are either relevant for the safety of human beings and objects or where malfunctions would imply high economical losses, the intention arose to prove in advance the correctness of software programs for any combination of input data. Although it seems to be impossible to show the correctness of any arbitrary software program, several partial approaches to that aim were developed. Among these, one particular part is considered here: the field of reconsideration of a program's data flow. The instrumentation of programs allows to have a look at the flow of data during its execution, but it is also possible to get information about data flow by the static analysis of a program. Errors and anomalies like a missing initialisation of a variable, a reference of a variable that is undefined, or a definition of a variable that has not been referenced after its previous definition may be discovered, and possible decouplings between data may be detected. The mutual influences of control flow and data flow cause some unpleasent problems preventing a strict separation between them. Therefore, some special intersections between them, like considerations on path conditions or mapping conditions depending on paths are not treated in detail in this paper.

The analysis of data flow has its roots not only in the field of safety considerations, but also in the design of code optimizing compilers. The FORTRAN analysing system, for example, /1/ contains a data flow analysis routine.

2. SEVERAL METHODS FOR THE ANALYSIS OF DATA FLOW

A very qualified method to represent the mutual dependencies of program data is the method of the Symbolic Execution. Each path of a program can be executed symbolically, i.e., the values of input variables of a program are represented by symbolical values instead of numerical or other actual values. Along an interesting selected path, variable values and path conditions are collected and, as far as possible, repeatedly substituted by existing symbolic values or expressions until the final or any other interesting point of the path is reached. The values of variables and the path condition, then, is given by formulae depending on symbolic input variables only. By evaluating these formulae, the correctness of a path execution can be shown in relation to its specification.

The symbolic execution is a very powerful method for finding errors in programs /2/. It shows exactly the mapping of data of program runs. Unfortunately, since the final expressions are frequently rather complex, they are often unwieldy and, therefore, difficult to resolve and to understand.

The analysis of data flow due to the method of /3/ and others is a less sophisticated method to investigate data mappings, but able to depict a lot of data flow errors or anomalies when existing. This method has its roots in the design of code optimizing compilers. It is based on the premises that the last action performed on a variable is represented by one of the three states, namely defined, undefined, and referenced, and that the sequential arrangement of these states due to a program path permits to detect anomalies as, for example, the state 'undefined' followed by the state 'referenced'. That means, the attempt is revealed of reading the value of a variable that was not even initialized.

A modest representation of data dependencies is given by most compilers as cross reference lists. Some of them present two kinds of list elements pointing out whether a variable is defined or referenced.

3. ELEMENTS OF DATA FLOW GRAPHS

As the representation of the control flow by its graph, the mutual dependencies of variables can be depicted as graphs, too. These graphs are composed of elementary partial graphs that can be extracted from the assignment statements. Supposed all assignment statements have the following configuration or can be matched to it:

$$v_d = f(v_{r1}, v_{r2}, \ldots, v_{rn})$$

v_d ... d-variable (defined variable)

v_r ... r-variable (referenced variable)

f ... mapping

Let V be the set of all names of variables standing in the statements of a program. Within this set, the values of the constants c_i are regarded as their names and, in the following, marked by the letter 'C'.

Whenever a variable with a name in V (say A) maps its value to a variable with a name in V (say B) during the execution of a statement (for example: A:=B), an element (A,B) RcVxV is generated. The set RcVxV is a relation on V. (1)
It can be established by a static analysis of the source code. Directed graphs are the elements of almost all data flow graphs mentioned in this paper+. The starting node of an elementary graph carries the name of a defined variable, its end node the name of a referenced variable. In the drawings the direction of edges is from top to bottom until otherwise noted. Statements containing more than one referenced variable lead to more than one elementary graphs of one statement tree of a variable by merging the starting nodes of them, as shown in fig.1.

A statement tree of a variable contains the same information as all entries concerning one statement in a cross reference list. The statement data trees of a set of statements can be linked together in different ways, as shown below.

4. WORST CASE DATA TREES

Without analysis of the control flow, the set of statements is not ordered with regard to the sequence of execution. Hence, it cannot be decided which assignment to a variable was relevant or not after the execution of the program. Therefore, when ignoring the control flow, all assignment statements must be considered in the same manner, even if the last assignment possibly overrides all preceding assignments. The concatenation of all statement trees of a variable in a program by merging their root nodes creates a tree structure that depicts all variables to be referenced when defining the root variable in any statement of the program. It does not show indirect references via the dependencies of the referenced variables on further variables. This disadvantage can be avoided by linking the corresponding trees of the referenced variables to it. A problem may arise when a first variable is referenced during the definition of a second and vice versa+; in this case, the linking never ends. The agreement to link the tree of a variable only once removes this problem. The resulting tree is called a 'worst case static tree of implicit variable' WSTIVV, it shows all possible dependencies of a variable on other variables in a program. The fact that a variable is not contained in a WSTIVV means that it is decoupled from the root variable and may not influence the value of it by any assignment. (Note that it may influence the value of it via the selection of paths by branchings, and that the presence of a variable in a WSTIVV does not mean that a reference to the root variable exists mandatory.) Every variable depicted as a leaf in the tree must either be an input variable, or a constant++, or it appears in another site as a node with successors; otherwise a reference of an uninitialized variable is detected. Fig.2 shows the construction of WSTIVVs for a program part.

A cross-reference-list whose list elements must be distinguishable between defined and referenced permits the construction of WSTIVVs.

+ except chap.5.4

++ A constant is regarded as a special case of a variable (see also chap.3).

+ Such a data graph is not a tree, as it contains cycles. However, by reasons of presentation, the cycles are cut off by splitting its start and end node into two nodes.

5. THE CONSTRUCTION OF DATA TREES WITH REGARD TO THE CONTROL FLOW GRAPH

5.1 Data Trees of Paths

The data mappings accomplished after a program run are related to the path that was run. A given path is a string of statements in the order of their execution. This information on the flow of control permits a decision on which of two statements is executed first, what was not possible when building WSTIVVs. In general, the selection of a path is dependent on the input data of a program. For that reason data trees built with regard to the execution sequence of a path are furthermore called dynamic trees of implicit variables of a variable (DTIVVs).

5.2 Data Trees Corresponding to a Whole Control Flow Graph

Another group of data trees with properties in between those of the DTIVVs and WSTIVVs does not need the consideration of actual values of input variables (i.e. the knowledge of a certain path) but of the static control flow graph. These trees are called STIVVs (static trees of implicit variables of a variable).

STIVVs of basic blocks. A basic block within a control flow graph "is a group of statements such that no transfer occurs into the group except to the first statement in that group, and once the first statement is executed, all statements in the group are executed sequentially" /4/. As described in chap.3, the trees of the defined variables are set up for every assignment statement within a basic block. After this step, the concatenation of the trees is following, beginning with the first statement of the basic block. (Now there is no need of merging the trees with the same roots). If any leaf of a new tree has the same name as the root of an already existing tree, it will be substituted by the already existing tree. After the concatenation process a tree with the same root that possibly exists already is erased. In the final step of processing a basic block, each leaf of all existing trees gets a dummy successor as an interface node (except leaves representing constants). Fig.3 shows the building of STIVVs of a basic block.

Concatenation of STIVVs of basic blocks. A sequence of basic blocks of the control flow graph without any branchings is a part of a path or an entire path. For such a given path, the STIVVs of the basic blocks are concatenated by replacing the dummy nodes by the corresponding STIVVs of the preceding basic block. The results of concatenation are the DTIVVs showing exactly all data mappings in the same sequence as performed on a path. The treatment of all paths in this way mostly has its limits in the big number of paths; however, with a loss of information, another method of concatenation using the whole control flow graph by-passes this difficulty. There is no problem with a branching in the control flow graph where the succeeding basic blocks simply inherit the STIVVs of their (common) predecessor. But when a basic block, at a conjunction of partial paths, has more than one predecessor, it is generally impossible to decide which predecessor transfers the control to it with the knowledge of the static control flow graph only. Hence, the preceding basic blocks must be considered equally. Each variable within a basic block that inherits its value of preceding basic blocks gets as many dummy node successors in the STIVV as there are edges entering the basic block (fig.4). In the course of concatenation of the STIVVs each dummy node must be replaced by the corresponding STIVV of a previous basic block. In the final STIVVs of that kind, the dependency of a node variable on all of its successors is not mandatory for any given program run.

Concatenation of STIVVs in strongly connected regions of a control flow graph. A special kind of conjunction of partial paths is given in strongly connected regions of the control flow graph. A strongly connected region is a subgraph of a graph in which each node is both its own successor and its own predecessor. In a reducible flow graph (see /5/) (roughly spoken: each loop formation has only one entry node), each latching node that feeds back the control to the entry node of a strongly connected region is unique.

Fig.5 shows the scheme of a simple, strongly connected region. The concatenation of the STIVV of basic block 3 with those of basic block 1 yields the STIVV of the exit basic block (exit STIVV), containing+ two nodes as dummies for the STIVV to be concatenated from the basic blocks that precede the entry basic block.

In the next steps the concatenation of STIVVs is continued until the STIVV of the latching basic block inherited the STIVV of the entry-basic block. Therewith the elementary STIVVs of the strongly connected region are found out. Since the number of runs through a strongly connected region is unknown in general, the presentation of data dependencies by STIVVs results in repetitions of its parts, as shown in fig.5. The repeated parts are the STIVVs of the latching basic block with the exception of their root node and that dummy node that represents the interface to the latching basic block. Fig.6 shows a strongly connected region with two latching basic blocks. Now three edges enter the entry basic block, two of them are latching edges. Since the control may be transferred either to basic block 3 or 4 each time when it is in basic block 2, it must be considered that the STIVV of the entry basic block inherit the STIVVs of basic block 3 and basic block 4 for each repetition. In this case, the repetitions in the STIVVs are twofold. For each strongly connected region a STIVV consists of a body and a repetition subgraph that is labelled and quoted only once in that STIVV.

+ In case the corresponding root variable was defined within basic block 1 or 3.

5.3 Relations between Data Trees of Program Paths and of a Whole Program

A characteristic of the STIVVs of the last chapter is that they contain as subgraphs the corresponding data trees of each path included in the control flow graph (DTIVVs). They are the whole of all DTIVVs of a variable in a program. The selection of a special DTIVV by hand can be done by drawing a line that intersects those edges of a STIVV that belong to the DTIVV, in the sequence of the execution of assignments as shown in fig.7a.

To each edge of a STIVV the number of the corresponding statement can be attached. It facilitates the search for the corresponding assignment. Fig.7b shows a STIVV composed of nodes, edges, and a third kind of elements that combine all edges corresponding to a partial path entering a basic block in the control flow graph.

5.4 Array Elements in Data Trees

Array elements are handled as normal variables; their index variables are regarded as variables that are referenced during the definition of an array element. Although it is a simplification, it is in accordance with the practice in several compilers (fig.8).

6. DATA TREES OF AFFECTED DATA

The replacement of relation (1) in chap.3 by the following:
"Whenever a variable with a name in V (say A) maps its value to a variable with a name in V (say B) during the execution of a statement (for example: A:=B), an element (B,A) RcVxV is generated. The set RCVxV is a relation on V."(2)

enables the generation of data graphs representing the influence of a variable over other variables, the affected variables.

Fig.9 shows the worst case static trees of affected variables of a variable (WSTAVV) for a program part.

7. CONCLUDING REMARKS

The presentation of data flow dependencies as graphs is an easy comprehensible way to show the mappings of data in a program. It permits the finding of certain data flow anomalies as well as possible decouplings of data. Data trees composed of cross-reference-lists give a simple overview on data mappings that is not very distinct because of the neglect of the control flow graph.

A clearer insight is given by data trees corresponding to the whole control flow graph of a program. The most exact presentation comes from data trees of single paths, but it has the disadvantage to show only the mappings of a special path out of a number of paths which frequently is huge, preventing the analysis of the whole of paths by quantitative reasons.

The presented considerations of this paper on data trees were made in connection with the development of an analysing system for program modules written in PEARL, a project that is supported by the BMFT+. Since there are so many relations between the methods of compiling and analyzing programs, the decision was made to choose an intermediate language as starting point for the analysis of modules. A particular PEARL-compiler translates PEARL-programs into strings of an intermediate language. The analyser is scheduled for carrying out some static analytic tasks, beginning with the structural analysis (see/6/) of the intermediate strings. It should become a tool that is particularly helpful for safety assessment. In connection with the analyzer a set of rules will be provided whose observation during program development will facilitate the analysis. Particular care is taken to provide a convenient presentation of the analysis results.

I would like to thank PDV (Project Prozeßlenkung mit DV-Anlagen) for its support and messrs. Grumbach and Märtz for reading the manuscript.

8. REFERENCES

/1/ Fosdick, L.D., and L.J. Osterweil.(1975) DAVE - A Fortran Program Analysis System. Dept. of Computer Science, Univ. of Colorado, Boulder, Colorado 80302.

/2/ King, J.C. (July 1976). Symbolic Execution and Program Testing. CACM vol.18 no.7.

/3/ Fosdick, L.D., and L.J. Osterweil (1976). Data Flow Analysis in Software Reliability. Computing Surveys, vol.8, no.3.

/4/ Aho, A.V., and J.D. Ullmann (1973). The Theory of Parsing. Translation and Compiling, vol.2, Poentice Hall, Englewood Cliffs, N.J., p.912.

/5/ Hecht, M.S., and J.D. Ullmann (Oct.1973). Analysis of a Simple Algorithm for Global Flow Problems. Conf. Record, ACM Symp.on Principles of Programming Languages, Boston, Mass., pp. 207-217.

/6/ Okroy, K. (June 1978). PDV-Bericht KfK-PDV 152, KfK Karlsruhe.

DISCUSSION

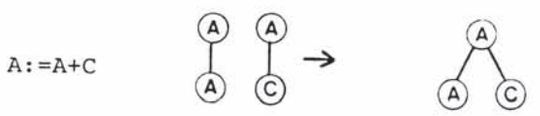
Grams: What types of programming errors can be detected by your method?

Puhr-Westerheide: For example variables which are not initialized or not referenced.

Maki: What is the advantage of this technique? A compiler can also find for example not initialized variables.

Puhr-Westerheide: Yes, you can also build the worst case data trees starting from the cross-reference-list of a compiler too.

+ BMFT ... Bundesministerium für Forschung und Technologie



assignement tree elements assignement variable tree

fig. 1. Merging the elementary graphs of an assignement statement

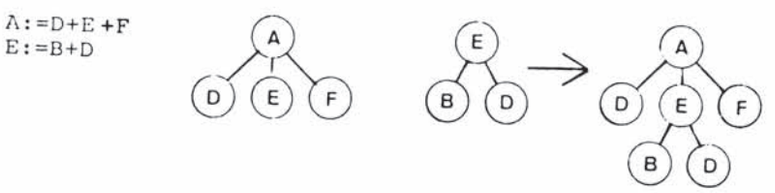


fig. 2. concatenation of STIVV's

Maki: So why do you use your technique?

Puhr-Westerheide: First: You can see which variables may affect a variable which is denoted in the root-variable. If you do this with the cross-reference-list of a compiler it is a very hard work.

Second: The information of the control flow graph is also used. You can also set up a very exact and clear data tree for only one path which shows exactly all mappings on one path. This cannot be done by a compiler.

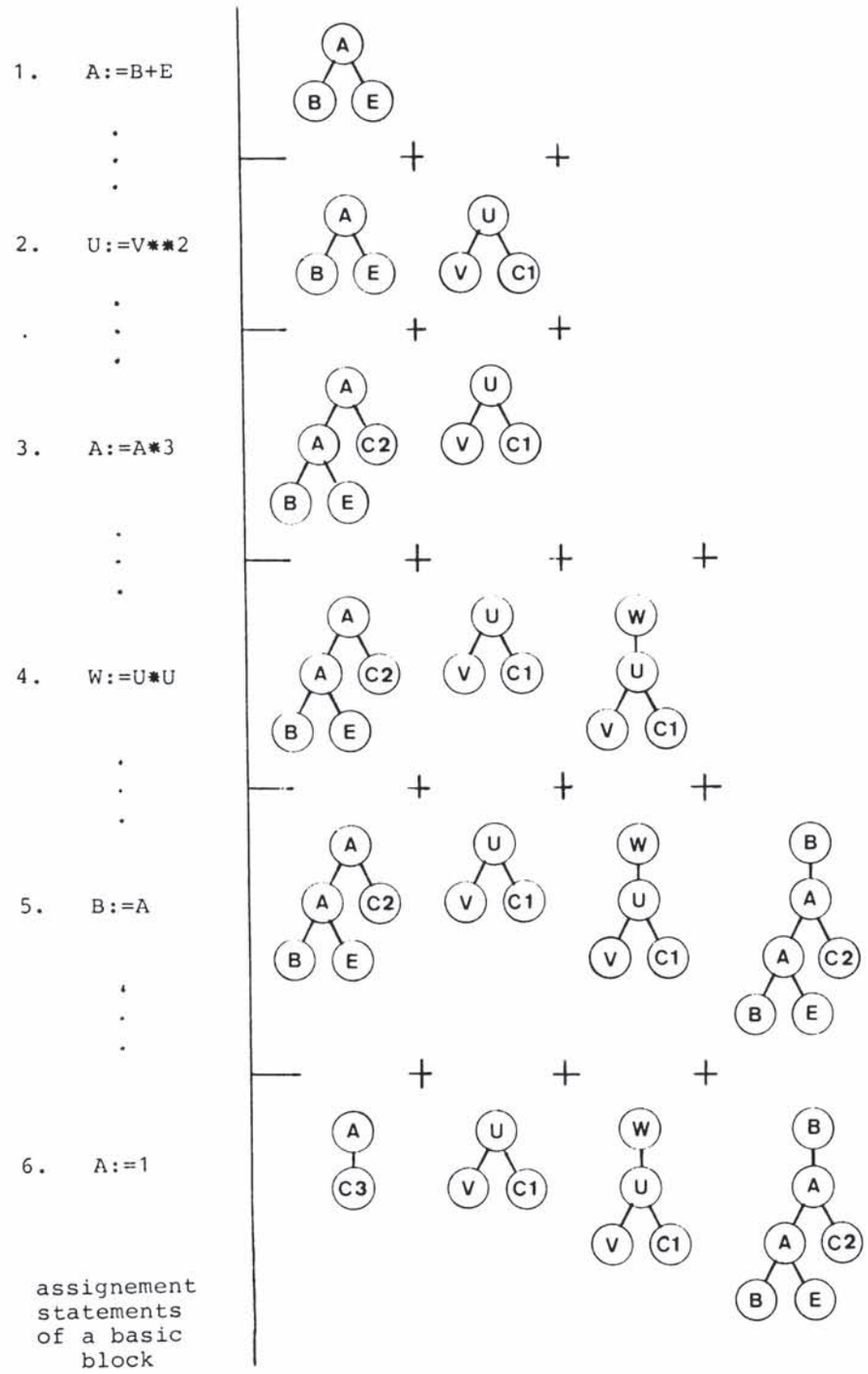


fig. 3. The generation of STIVV's of a basic block

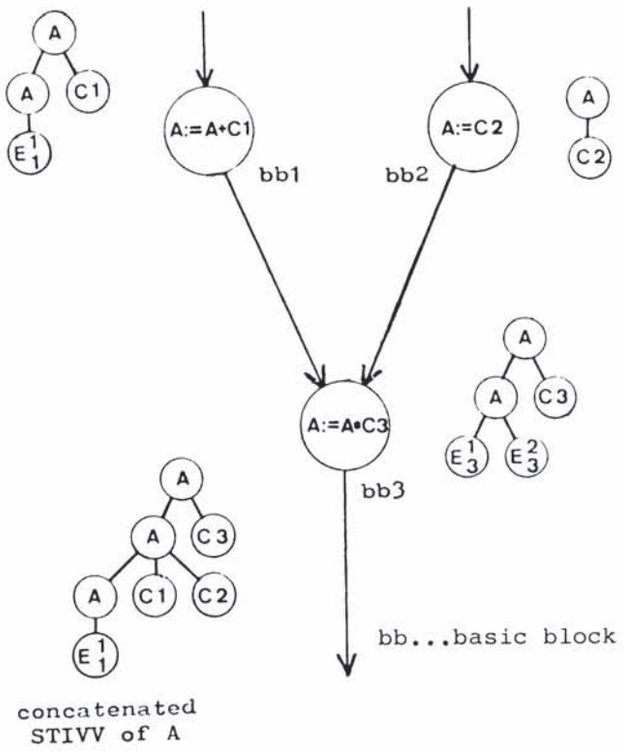


fig. 4. Concatenating the STIVV's of basic blocks

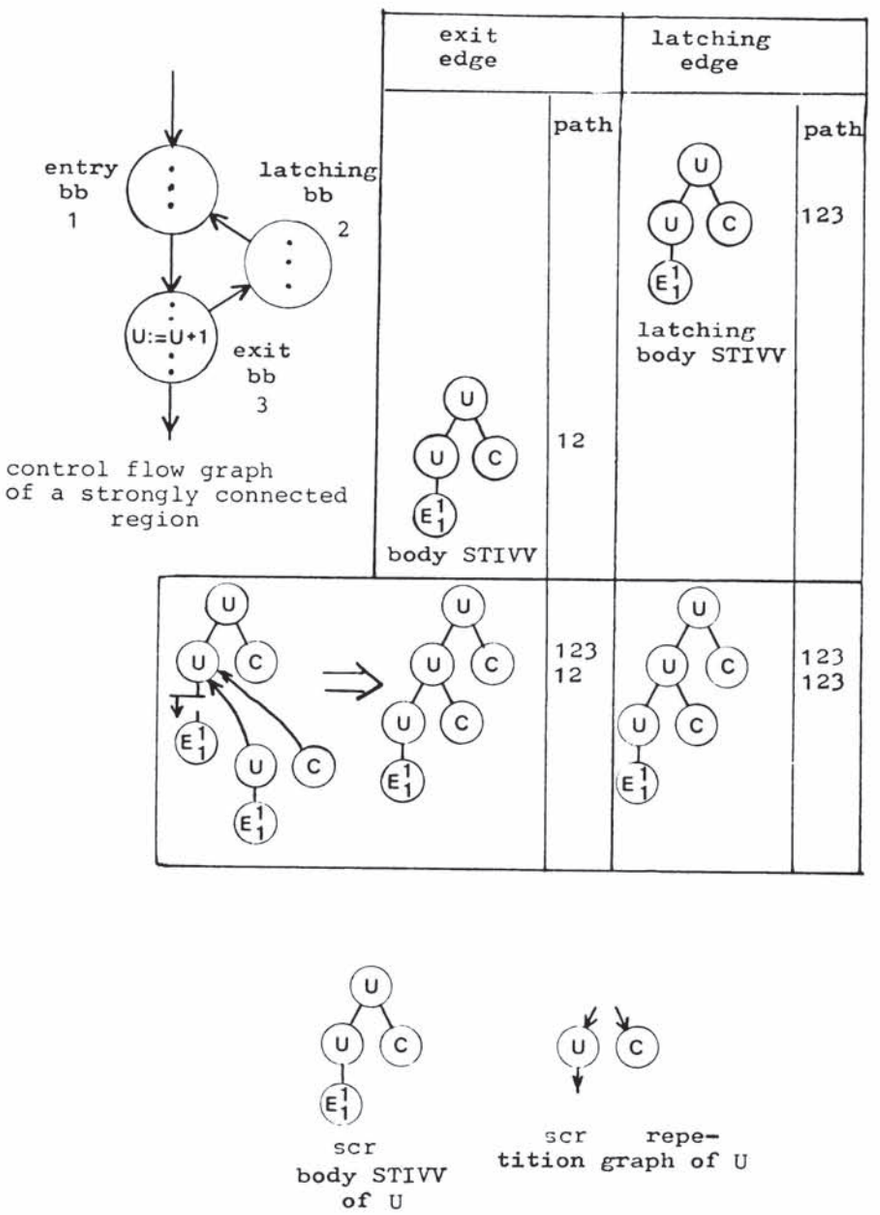


fig. 5. The generation of STIVV's of a strongly connected region (scr)

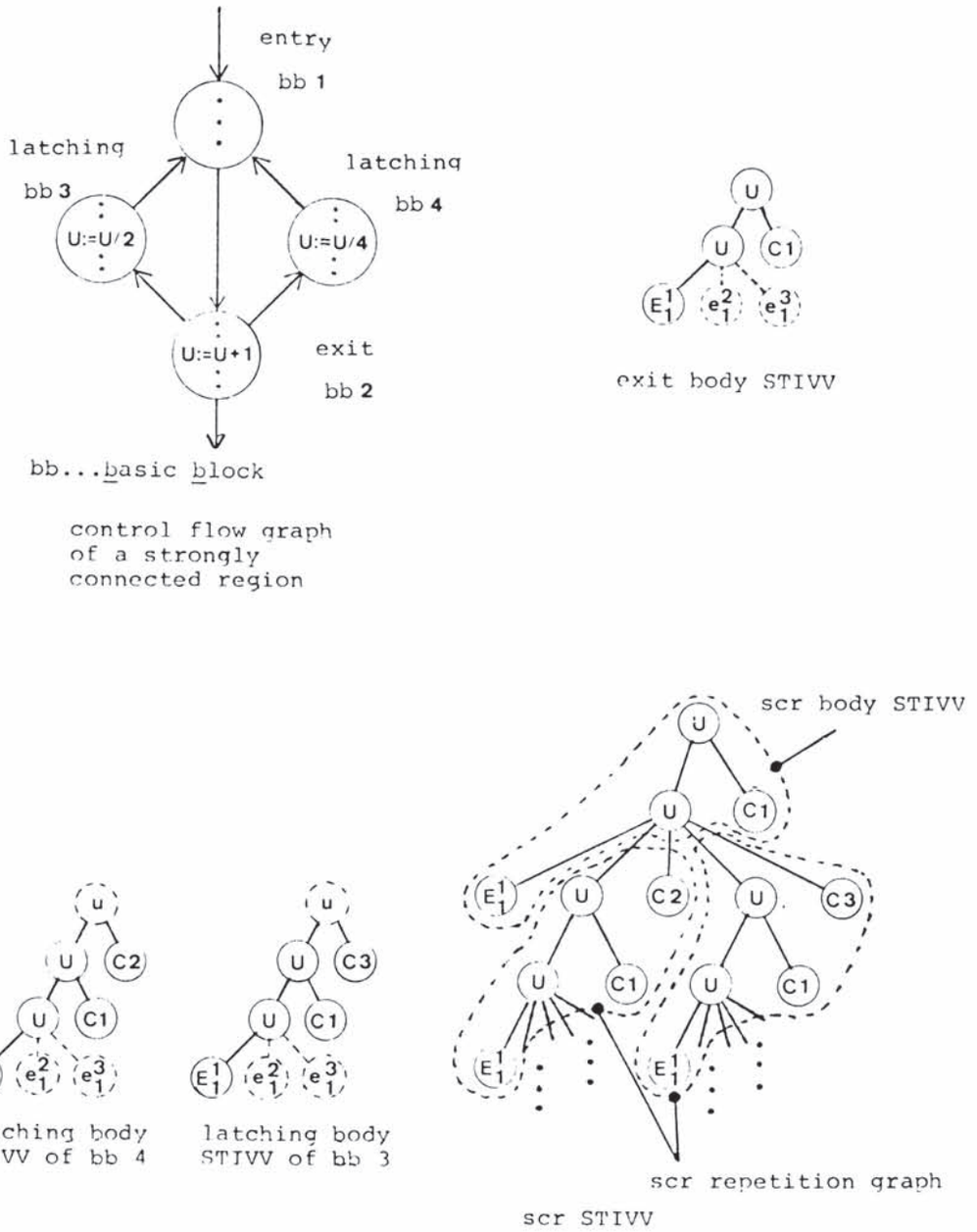


fig. 6. The generation of STIVV's of a scr containing two latching basic blocks

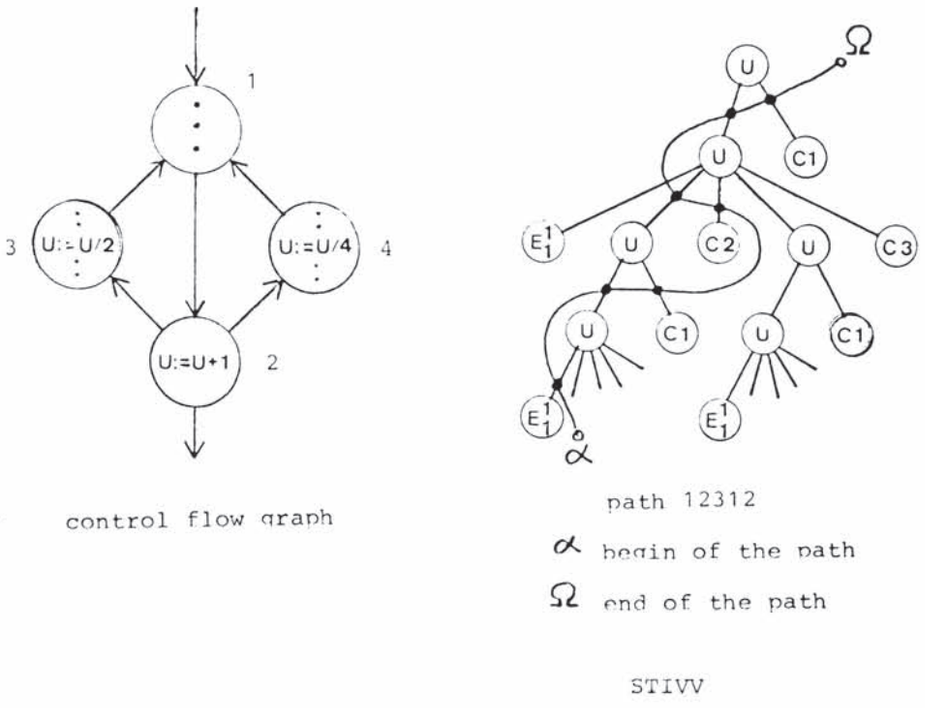


fig. 7a. A line labels a subgraph corresponding to a path

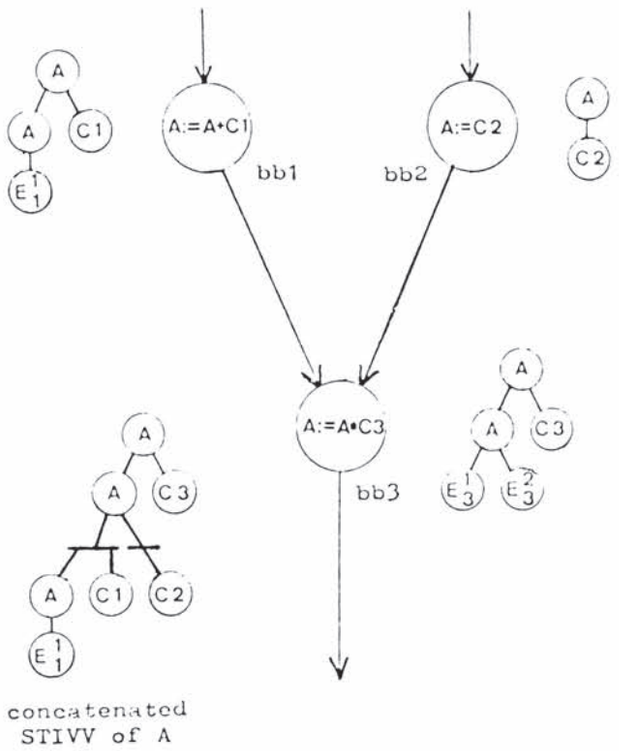


fig. 7b. STIVV of fig. 4, extended by a new element (see chap. 5.3)

A(I):=A(I-1)+A(I)

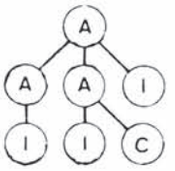


Fig. 8. STIVV of an array variable

U:=A+SIN(FI)
R:=A/8+T
T:=A+B

assignments

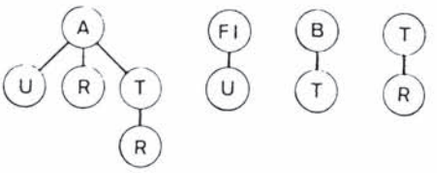


fig. 9. Trees of affected variables of a program part