DECOMPILATION OF CONTROL STRUCTURES BY MEANS OF GRAPH TRANSFORMATIONS *

Ulrike Lichtblau
Lehrstuhl Informatik II
Universität Dortmund
Postfach 500500
D-4600 Dortmund 50

ABSTRACT

Decompilation denotes the translation from lower level into higher level programming languages. Here we deal with the aspect of detecting higher level control structures, including loops with any number of exits, in line-oriented programs.
The detection is carried out on the control flow graph of the source program by means of so called wellstructuring transformations. We show that the iteration of these transformations always terminates in a time linearly depending on the number of vertices of the underlying control flow graph.

1. INTRODUCTION

Decompilation denotes the reverse of the compilation process, i.e. the translation from lower level into higher level programming languages. Simply embedding the source language into the target language is not considered to be sufficient. Rather, the higher target language should be exhausted. Therefore, the main problem of decompilation is the detection of high level structures in - at first sight - unstructured programs.
Decompilers are investigated in the literature since the middle of the sixties. Hopwood gave a survey of the subject [4].

Here we deal merely with one aspect of decompilation, namely the analysis of the control flow of source programs and its translation into semantically equivalent higher control structures of the target language.
The target languages which we consider are higher procedural ones like Ada or Modula-2. Their control structures include loops with any number of exits. The source languages

---

* This work was supported by Deutsche Forschungsgemeinschaft (grant Cl 53/3-2).

are line-oriented such as assembler languages or BASIC.
Programs written in one of these source languages can easily be represented by control flow graphs. We use this representation here.
The detection of higher control structures is described in terms of graph transformation rules. By application of a transformation rule a pattern of vertices and edges is replaced by a single vertex. Each pattern corresponds directly to a control structure of the target languages. Fig. 1 illustrates the idea of these transformation rules.

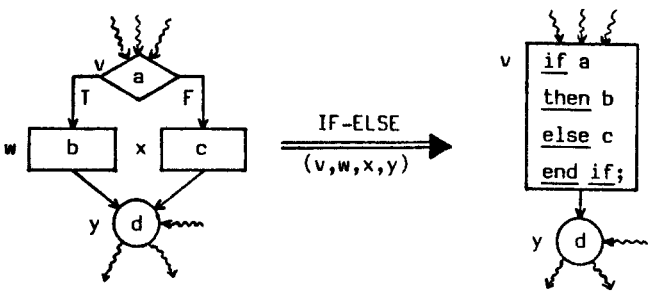

Fig. 1

Iteration of transformations based on such rules causes a control flow graph to decrease in size. The more higher structures it contains, the smaller it becomes.
We show that the process of reducing a control flow graph as far as possible always terminates in a time linearly depending on the number of vertices of the graph. Note that this holds although the size and the number of the patterns to be detected grow with the size of the input graph. The result can be obtained because it is possible to recognize the nesting of structures before starting the reduction process and because the system of wellstructuring transformation rules has the finite Church-Rosser property.

We do not explain the generation of target code here. However, this task can easily be solved by assigning labels to the vertices and edges of control flow graphs. In Fig. 1 such a labeling was assumed to exist. The edge labels should indicate 'true'- and 'false'-branches. The vertex labels should consist of simple conditions or simple statements at the beginning, and later on of target code which corresponds to the structure already detected. It is important to note that the transformation rules introduced here support the generation of code.
There are, of course, control flow graphs that by means of these transformation rules cannot be reduced to a single vertex. For code generation purposes they can be treated in the following way. Whenever there is no rule applicable, an edge is removed from the graph and a goto-statement is generated at the appropriate position. Afterwards, the process of reduction is restarted.

Rosendahl and Mankwald [8] use a similar idea of decompiling control structures by reducing the control flow graph and constructing the target program in the labels of its vertices. However, they only support the detection of loops with one exit. Further, they do not construct an efficient transformation algorithm.
The set of graph transformation rules introduced by Farrow, Kennedy and Zucconi [1] considers multiple exits from loops in an even more general way than is done in this paper. The derived reduction algorithm is of linear time complexity. These rules, however, do not support the translation of the recognized structure as they do not directly correspond to control structures of a higher programming language.

This paper states results from [5] and [6]. All proofs can be found there, as well as the details of target code generation.

2. PRELIMINARIES

A **graph** is a pair $G=(V,E)$, where V is a finite set of **vertices** and $E \subseteq V \times V$ is a set of directed **edges.**
Let $G=(V,E)$, $H=(V',E')$ be graphs.
For each vertex $v \in V$ we call $\mathbf{pred(v)} := \{w \in V \mid (w,v) \in E\}$ the set of all predecessors and $\mathbf{suc(v)} := \{w \in V \mid (v,w) \in E\}$ the set of all successors of v.
$\mathbf{d^+(v)} := \#\mathrm{pred}(v)$ is the **in-degree** and $\mathbf{d^-(v)} := \#\mathrm{suc}(v)$ is the **out-degree** of v.
A finite sequence of vertices $(v_0,\ldots,v_n)$, $n \geq 0$, is a **path** in G if $v_i \in V$ for $0 \leq i \leq n$ and $(v_i,v_{i+1}) \in E$ for $0 \leq i < n$.
A vertex $r \in V$ is called a **root** of G if there exists a path $(r,\ldots,v)$ in G for every $v \in V$.
$v \in V$ **dominates** $w \in V$ **with respect to** a root r of G if v occurs in every path $(r,\ldots,w)$ in G.
H is a **subgraph** of G if $V' \subseteq V$ and $E' = E \cap (V' \times V')$.
For each $V'' \subseteq V$ the graph $(V'', E \cap (V'' \times V''))$ is called the **subgraph** of G **induced by** V''.
An **isomorphism** i from G to H, denoted by $i: G \to H$, is a bijective mapping $i: V \to V'$ with the property $(i(v),i(w)) \in E'$ iff $(v,w) \in E$ for all $v,w \in V$.

Let X be a set and $R \subseteq X \times X$ a binary relation on X.
We denote by R^m, $m \in \mathbb{N}_0$, the m-fold iteration of R and by R^* the reflexive transitive closure of R.

3. CONTROL FLOW GRAPHS

In this section we introduce control flow graphs. We use them to carry out the detection of higher control structures since they are normal forms of line-oriented pro-

grams.

Definition :

A **control flow graph** is a 4-tupel $k=(V,E,r,s)$, where

(i) (V,E) is a graph

(ii) $V=\{r\}$ or $V=V_A+V_C+\{r,s\}$ and $V_A\neq\emptyset$

(iii) r is a root of (V,E)

(iv) $d^+(r)=0$ and $d^-(s)=0$

(v) $\forall v\in V_A\cup\{r\}: d^-(v)\leq 1$

(vi) $\forall v\in V_C : d^-(v)=2$.

K denotes the set of all control flow graphs and **SK** the subset of all graphs in K consisting of exactly one vertex.

There are four kinds of vertices in a control flow graph: a vertex r representing the start-statement of a program, a vertex s representing the stop-statement, vertices V_A corresponding to other statements and vertices V_C corresponding to conditions.

Example :

Fig. 2 shows a control flow graph $k\in K$.

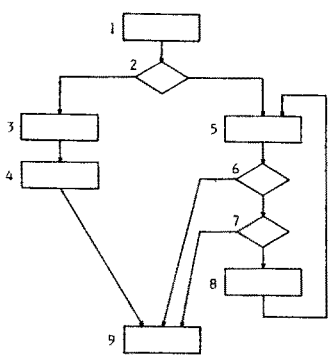

Fig. 2

4. A GRAPH TRANSFORMATION SYSTEM FOR STRUCTURE RECOGNITION PURPOSES

Here we introduce a system of transformation rules on the set of all control flow graphs. Each of these rules is applicable if and only if the given graph contains a pattern of vertices and edges that corresponds to a boolean operation on conditions or to one of the control structures 'sequence', 'if-then', 'if-then-else', 'loop' or 'while-loop', where a loop may have any number of exits. On application of a transformation rule the detected pattern is reduced to a single vertex.

Definition :

The **system of wellstructuring transformation rules T** is the collection of rules depicted in Fig. 3.

Notation :

Each transformation rule $t \in T$ consists of two graphs, the **left hand side** of t $L(t)=(V_{L(t)},E_{L(t)})$ and the **right hand side** $R(t)=(V_{R(t)},E_{L(t)})$ and two sets of vertices $I(t) \subset V_{L(t)}$ and $O(t) \subset V_{L(t)}$, the **in-vertices** and the **out-vertices** of t.
In Fig. 3 the arrow-heads mark the in-vertices and the circles symbolize the out-vertices.

The in-vertices and the out-vertices describe the allowed embedding of the left hand side of a rule into a control flow graph. In-vertices are the only vertices which may be connected to the outside by incoming edges, out-vertices are those which may be connected by outgoing edges.
Note that the out-vertices of a transformation rule do not belong to the pattern of the detected control structure. Rather, they are used to define the context.

Definition :

Let be $k=(V,E,r,s) \in K$.
A transformation rule $t \in T$ is said to be **applicable to** k **in** $v_1,\ldots,v_{m_t} \in V$ if there exists an isomorphism $i: L(t) \to G$, where G is the subgraph of (V,E) induced by $\{v_1,\ldots,v_{m_t}\}$ and

(i) $\forall v_j,\ 1 \leq j \leq m_t: \forall x \in V \setminus \{v_1,\ldots,v_{m_t}\}:$
$(x,v_j) \in E \Rightarrow i^{-1}(v_j) \in I(t)$
and $(v_j,x) \in E \Rightarrow i^{-1}(v_j) \in O(t)$

(ii) $\forall v_j,\ 1 \leq j \leq m_t:$
$v_j = r \Leftrightarrow i^{-1}(v_j) = r$
and $v_j = s \Leftrightarrow i^{-1}(v_j) = s$.

Applying a transformation rule means replacing the left hand side with the right hand side and embedding the latter. The replacing is done by removing certain vertices and adding certain edges. This is possible because $V_{R(t)} \subset V_{L(t)}$ for each $t \in T$. The embedding is straightforward since for every $t \in T$ $I(t) \cup O(t) \subset V_{R(t)}$.

Definition :

Let be $k=(V,E,r,s) \in K$ and $t \in T$ such that t is applicable to k in $v_1,\ldots,v_{m_t} \in V$ via the isomorphism $i: L(t) \to G$.
The **result of applying t to k in** $v_1,\ldots,v_{m_t}$ is defined by $\mathbf{t(v_1,\ldots,v_{m_t})(k)}:=$ (V',E',r',s'), where

- $V'=(V \setminus \{v_1,\ldots,v_{m_t}\}) \cup \{i(x) \mid x \in V_{R(t)}\}$
- $E'=(E \cap (V' \times V')) \cup \{(i(x),i(y)) \mid (x,y) \in E_{R(t)}\}$
- $r'=r$
- $s'=\begin{cases} r, & \text{if } t=BLK \\ s & \text{else} \end{cases}$.

The binary relation on K induced by T is denoted by T_R.

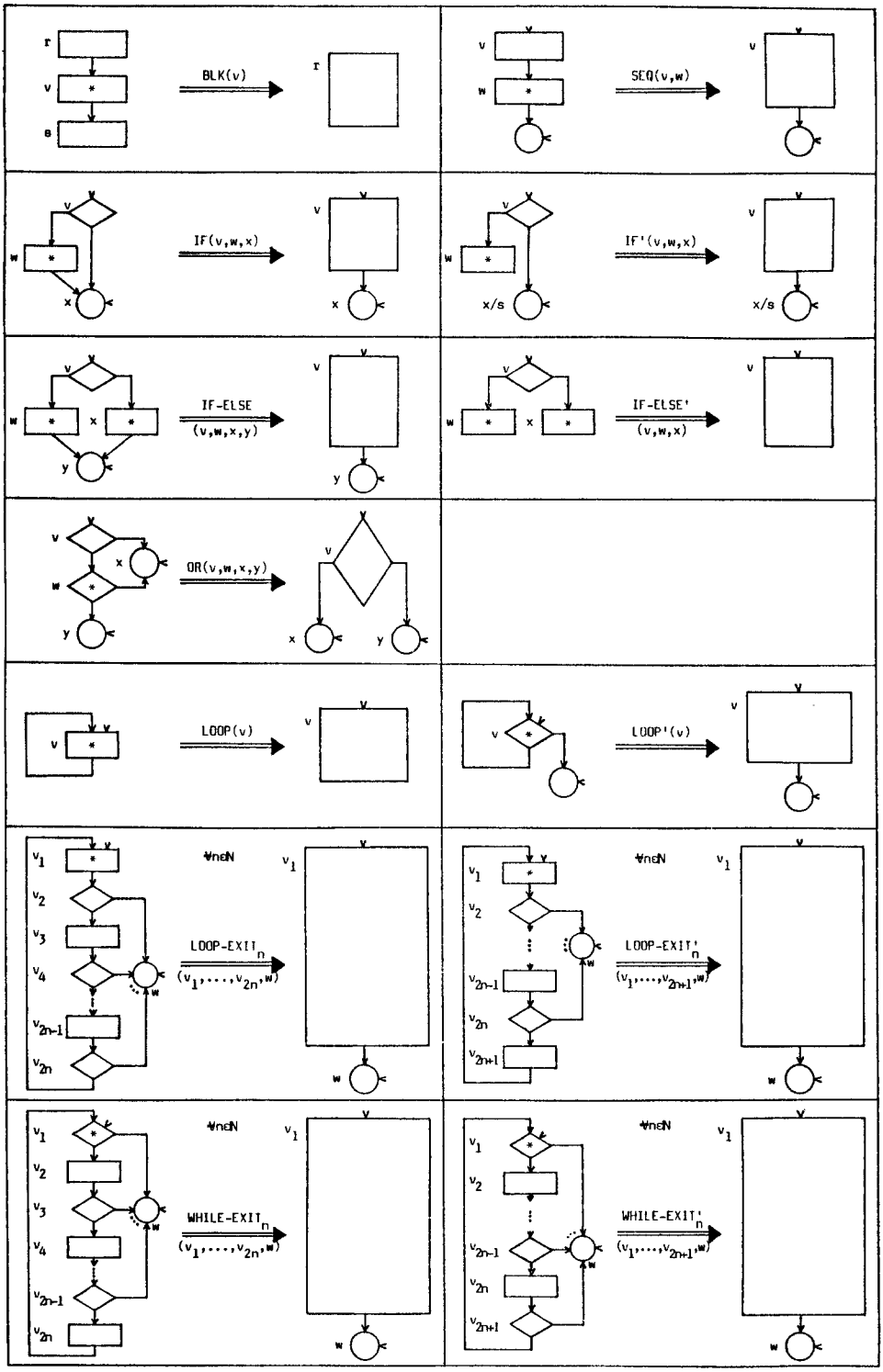
BLK(v)
SEQ(v,w)
IF(v,w,x)
IF'(v,w,x)
IF-ELSE
(v,w,x,y)
IF-ELSE'
(v,w,x)
OR(v,w,x,y)
LOOP(v)
LOOP'(v)
∀n∈N
LOOP-EXIT
LOOP-EXIT'
WHILE-EXIT
WHILE-EXIT'

Fig. 3

Notation :

If a transformation rule is applicable to a control flow graph, the vertices corresponding to those marked by '*' in Fig. 3 are called the **handles** of the rule; the set of all handles of $t \in T$ is denoted by **han(t)**.

Since a handle uniquely determines the set of vertices in which a transformation rule $t \in T$ is applicable we sometimes say that t is **applicable** to $k \in K$ **with handle** v and denote the result of the application by $t_{<v>}(k)$.

After a rule has been applied to a control flow graph the vertices corresponding to those appearing on the right hand side of the rule are referred to as its **resulting vertices**, **res(t)** standing for the set of all resulting vertices of $t \in T$.

Control flow graphs which by means of wellstructuring transformations can be reduced to a graph consisting of exactly one vertex are the interesting ones here. They represent line-oriented programs, the control flow of which can properly be expressed in terms of higher control structures.

Definition :

A control flow graph $k \in K$ is **wellstructurable** if there exists $sk \in SK$ such that $(k,sk) \in T_R^*$.

Example :

The control flow graph k shown in Fig. 2 is wellstructurable since on application of the sequence of transformation rules

$$SEQ(3,4) \bullet OR(6,7,9,8)^{(1)}_{\bullet} LOOP\text{-}EXIT_1(5,6,8,9)^{(2)}_{\bullet} IF\text{-}ELSE(2,3,5,9) \bullet BLK(2)$$

it is reduced to a certain $sk \in SK$.

The preliminary results at the marked positions are shown in Fig. 4.

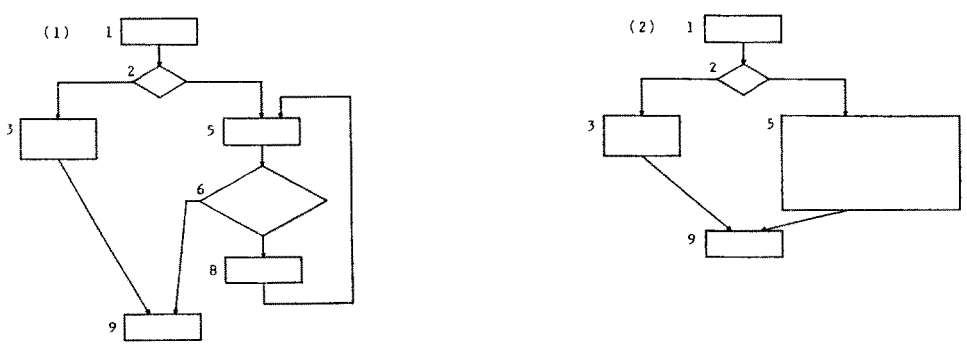

Fig. 4

A comparison of the transformation system T with known sets of rules for transforming control flow graphs yields the following results.

Each wellstructurable control flow graph is collapsible with respect to the definition of Hecht and Ullman [3]; the reverse does not hold.
The set of acyclic wellstructurable control flow graphs is properly covered by the set of acyclic SSFG-reducible ones defined by Farrow, Kennedy and Zucconi [1]. Considering arbitrary control flow graphs the inclusion does no longer hold.
The fully well structured control flow graphs of Rosendahl and Mankwald [8] are a proper subset of the wellstructurable ones.

5. PROPERTIES OF THE TRANSFORMATION SYSTEM

Here we collect some properties of the system of wellstructuring transformation rules T which prove to be useful in designing a linear time reduction algorithm based on T.

We start with the finite Church-Rosser property. It says that any sequence of transformation rules applicable to a control flow graph is of bounded length, and that the limit graph which can be derived from a given control flow graph is uniquely determined - and thus not dependent on the decision which of several applicable rules is chosen in a given situation.
The following formulation of the finite Church-Rosser property is due to Rosen [7].

Definition :
Let X be a set and $R \subset X \times X$ a binary relation on X.
(X,R) is **finite Church-Rosser (FCR)** if
(i) $\forall x \in X\ \exists m_x \in \mathbb{N}_0\ \forall y \in X, m \in \mathbb{N}_0$: $(x,y) \in R^m \Rightarrow m \leq m_x$
(ii) $\forall x,y,y' \in X$: $((x,y) \in R$ and $(x,y') \in R) \Rightarrow \exists z \in X$: $(y,z) \in R^*$ and $(y',z) \in R^*$.

Lemma 1 :
(K,T_R) is FCR.

Proof :
By considering all possible interferences of any two wellstructuring transformation rules (cf. [5]). ∎

It is an easy consequence of lemma 1 that one is free to assign priorities to the wellstructuring transformation rules. We do this in the following way.

Definition :
Let be **prio_1** := {BLK, SEQ, IF, IF', IF-ELSE, IF-ELSE', OR} $\subset$ T,
prio_2 := {LOOP, LOOP'}
$\cup$ {LOOP-EXIT$_n$ | $n \in \mathbb{N}$} $\cup$ {LOOP-EXIT$_n$' | $n \in \mathbb{N}$}
$\cup$ {WHILE-EXIT$_n$ | $n \in \mathbb{N}$} $\cup$ {WHILE-EXIT$_n$' | $n \in \mathbb{N}$} $\subset$ T .
Prio_i, i=1,2, is called the **class of wellstructuring transformation rules of priority i.**

When testing whether a transformation rule is applicable it is reasonable to try to find a handle first. Therefore, we next state easy to check conditions which are necessary for a vertex being a handle of a transformation rule of a certain priority.

Lemma 2 :
Let be $k=(V,E,r,s)\in K$ and $t\in T$ such that t is applicable to k with handle $v\in V$.
If $t\in prio\_1$, then $d^+(v)=1$.
If $t\in prio\_2$, then $d^+(v)>1$.

Proof :
By definition. ∎

We now turn to the question how the set of vertices which are potential handles of transformation rules changes during the reduction process.
Concerning transformation rules of priority 1 we get the following result: at most the resulting vertices of the rule applied last are new potential handles.

Lemma 3 :
Let be $k=(V,E,r,s)$, $k'=(V',E',r',s')\in K$, $v\in V\cap V'$, $t\in T$ and $t'\in prio\_1$ such that t is applicable to k, $t(k)=k'$ and t' is not applicable to k with handle v.
If t' is applicable to k' with handle v, then $han(t')\cap res(t)\neq\emptyset$.

Proof :
By verifying the following statement for each $t'\in prio\_1$ and each $t\in T$:
There are certain properties of a vertex v which are relevant to the decision whether t' is applicable with handle v. An application of t changes relevant properties of some vertices, but only of the resulting vertices of t. (cf. [6]) ∎

For transformation rules of priority 2 the answer is even easier: each vertex which may be used as a handle at any time during a reduction can be detected in the original control flow graph. This follows from lemma 2.

Furthermore, it is possible to sort the potential handles of transformation rules of priority 2 in such a way that for each nested loop the handle of the corresponding transformation rule appears before the handle of the rule detecting the outer loop. To show this we use a wellknown ordering on the vertices of a control flow graph. The definition is taken from Hecht [2].

Definition :
Let be $k=(V,E,r,s)\in K$.
A mapping $rPOSTORDER: V \to \{1,\ldots,\#V\}$ is called a **vertex numbering** of k if the ordering induced on V equals the reverse of the order in which each vertex was last visited during a depth-first search of k.

A depth-first search algorithm that computes vertex numberings is given in [2]. Note

that there exist different vertex numberings of a given control flow graph.

The reverse of any vertex numbering is an ordering of potential handles of transformation rules of priority 2 with the property described above.

Lemma 4 :

Let be $k=(V,E,r,s)$, $k_i=(V_i,E_i,r_i,s_i)\in K$, $i=1,2$, and $t_1,t_2\in prio\_2$ such that t_1 is applicable to k_1 with handle $v_1\in V_1$, t_2 is applicable to k_2 in $v_{21},\ldots,v_{2m},w\in V_2$, $m\geq 2$, with handle $v_2=v_{21}$ and (k,k_1), $(k_1,k_2)\in T_R^*$.
If there exists a subgraph $G'=(V',E')$ of (V_1,E_1) such that $v_1\in V'$ and G' is transformed into v_{2j}, $1\leq j\leq m$, during the transition from k_1 to k_2, then $\{v_1,v_2\}\subset V$ and $rPOSTORDER(v_1)\geq rPOSTORDER(v_2)$ for any vertex numbering $rPOSTORDER: V \to \{1,\ldots,\#V\}$ of k.

Proof :

By two claims:
v_2 dominates v_1 with respect to r. (By definition of the transformation rules.)
If x dominates y with respect to r, then $rPOSTORDER(x)\leq rPOSTORDER(y)$ for any vertex numbering rPOSTORDER. (By definition of a vertex numbering.) (cf. [6]) ∎

6. THE TIME COMPLEXITY OF THE REDUCTION PROCESS

In this last section we are concerned with the process of reducing a control flow graph by means of wellstructuring transformations as far as possible and, in particular, contracting each wellstructurable graph to a single vertex.
As our main result we show that this process always terminates in time linear in the number of vertices of the input control flow graph.
The algorithm which we present is based on the fact that, since (K,T_R) is FCR (lemma 1), transformation rules may be applied in any order. It uses two work-lists, each corresponding to a priority class and containing all vertices of the current control flow graph which can possibly be used as handles of transformation rules belonging to that class. These potential handles are determined by means of lemma 2. The potential handles of rules of priority 2 are ordered in such a way that handles of inner loops appear in the work-list before handles of outer loops (see lemma 4). During each iteration of the algorithm the first entry - say vertex v - is removed from the current work-list. The lists are treated in order of the priorities they correspond to. If there exists a transformation rule in the corresponding priority class that is applicable to the current control flow graph with handle v, then this rule is applied. After a successful application the work-lists are updated according to lemmas 3, 2 and 4.

Theorem :

Let be $k=(V,E,r,s)\in K$.

The problem of reducing k as far as possible by means of wellstructuring transformations can be solved in time $O(\#V)$.

Proof :

Consider the following algorithm REDUCE .

Input : $k=(V,E,r,s)\in K$

Output : $k'=(V',E',r',s')\in K$ such that

(i) $(k,k')\in T_R^*$

(ii) $\nexists t\in T$: t is not applicable to k'

(iii) $\nexists k''=(V'',E'',r'',s'')\in K$: if k" fulfills (i) and (ii), then $\#V''\geq\#V'$

Method :
```
Compute a vertex numbering rPOSTORDER: V → {1,...,#V};
prio_1_handle_1 := list of all x∈V with d+(x)=1,
                   in any order;
prio_2_handle_1 := list of all x∈V with d+(x)>1,
                   in order of decreasing values of rPOSTORDER;
kk := k;
while there exists 1≦i≦2 such that prio_i_handle_1 is nonempty
loop  if   prio_1_handle_1 is nonempty
      then perform actions on kk corresponding to priority 1;
      else perform actions on kk corresponding to priority 2;
      end if;
end loop;
k' := kk .
```

Refinements :

(1) perform actions on kk corresponding to priority i, $1\leq i\leq 2$:

```
v := first entry of prio_i_handle_1;
remove v from prio_i_handle_1;
if   there exists t∈prio_i such that t is applicable to kk with
     handle v
then kk := t<v>(kk);
     insert all x∈res(t) with d+(x)=1 into prio_1_handle_1
     at any position;
     if   i=2 and d+(v)>1
     then insert v into prio_2_handle_1
          at the new first position;
     end if;
end if;
```

```
(2) there exists tεprio_i such that t is applicable to kk with handle v,
    1≦i≦2 :
    case i is
    when 1 => verified := false;
              for all tεprio_1
              loop if   t is applicable to kk with handle v
                   then verified := true;
                        exit;
                   end if;
              end loop;
              there exists := verified;
    when 2 => verified := false;
              if   there is a vertex w that is a proper successor of
                   the potential loop at v
              then falsified := false;
                   x := v;
                   while not verified and not falsified
                   loop if   suc(x)\{w}={v}
                        then verified := true;
                        elsif yεsuc(x)\{w} is acceptable as a succes-
                              sor of x in a loop
                        then x := y;
                        else falsified := true;
                        end if;
                   end loop;
              end if;
              there exists := verified;
    end case;
(3) there is a vertex w that is a proper successor of the potential loop
    at v :
    falsified := false;
    if   d⁻(v)=2
    then x := v;
    elsif there exists yεsuc(v)
    then x := y;
    else falsified := true;
    end if;
    if   not falsified
    then if   there exists yεsuc(x) with d⁺(y)>1
         then w := y;
         elsif there exists yεsuc(x) with d⁻(y)≠1
         then w := y;
```

```
            elsif there exists y∈suc(x) with (y,v)∉E
            then  w := y;
            else  falsified := true;
            end if;
        end if;
        there is := not falsified;
```

Claim 1 :

Algorithm REDUCE is correct, i.e. it terminates and has the specified input/output-behaviour.

The main ideas of the proof of claim 1 are contained in the remarks at the beginning of this section.

Claim 2 :

Algorithm REDUCE can be implemented in such a way that its time complexity is O(#V).

Proof of claim 2 :

The computation of rPOSTORDER can be done in time O(#E) (cf. [2]). Since the out-degree of all vertices of a control flow graph is bounded by 2 we have O(#E)=O(#V) here.
Obviously, the initialization of the work-lists needs O(#V) steps.
Further, it is fairly clear that the number of iterations of the main while-loop appearing in REDUCE is in O(#V).
One yet has to look at the nested loops. There are two of them, both appearing in refinement (2).
The first is iterated a constant number of times.
For the second it is possible to show that the total number of its iterations (over all iterations of the main loop) is in O(#V). To prove this we observe that each vertex of the input control flow graph is only visited a constant number of times during all iterations of the inspected loop. The result depends on lemma 4.
So far we have seen that the total number of executions of any test and any simple statement appearing in REDUCE is in O(#V). The proof is completed by showing that each of these atomic components can be executed in constant time. This requires the development of an appropriate internal data structure, which for each vertex allows direct access to its predecessors and to its entries in the lists of predecessors of its successors as well as to its entry in a work-list. (cf. [6]) ∎

ACKNOWLEDGEMENT

The author would like to thank Prof. Dr. V. Claus for initiating this work and for helpful discussions.

REFERENCES

1. Farrow,R., Kennedy,K. and Zucconi,L. Graph grammars and global data flow analysis. Proceedings of the Seventeenth Annual IEEE Symposium on Foundations of Computer Science, Houston, Texas, Oct. 1976, 42-56.

2. Hecht,M.S. Flow Analysis of Computer Programs. North-Holland, New York, 1977, Chapt. 3.

3. Hecht,M.S. and Ullman,J.D. Flow graph reducibility. SIAM J. Comput. 1, 2 (June 1972), 188-202.

4. Hopwood,G.L. Decompilation. Ph.D. dissertation, University of California, Irvine, Feb. 1978.

5. Lichtblau,U. Graphtransformationen zur Erkennung Ada-ähnlicher Kontrollstrukturen in maschinennahen Programmen. Techn. Rep. 142, Abteilung Informatik, Universität Dortmund, 1982.

6. Lichtblau,U. Ein Algorithmus zur Erkennung höherer Kontrollstrukturen durch Graphtransformationen. Techn. Rep. 183, Abt. Informatik, Universität Dortmund, 1984

7. Rosen,B. Tree manipulating systems and Church-Rosser theorems. Journ. ACM 20, 1 (Jan. 1973), 160-187

8. Rosendahl,M. and Mankwald,K.P. Analysis of programs by reduction of their structure. In Graph-Grammars and Their Application to Computer Science and Biology, Claus,V., Ehrig,H. and Rozenberg,G. (Eds.). Springer-Verlag, Berlin, 1979, 409-417.