KYBERNETIKA — VOLUME 33 (1997), NUMBER 3, PAGES 333 - 356

SOFTWARE COMPLEXITY AS A SOFTWARE QUALITY INDICATOR (THE THREE DOUBTS AND THE LIGHT AT THE END OF TUNNEL)

JIŘÍ VANÍČEK

The known methods of program complexity measurement, based on flowgraphs is critized. The new method of the complexity measurement for imperative programs, based on lucid decomposition, is proposed.

Motto: It is not very reasonable to be an absolute optimist. The light at the end of tunnel could be the spotlights of an arriving train.

1. INTRODUCTION

1.1. Software quality standardization

The aim of the paper is to announce some new views and new results concerning to the problem of measurement of software complexity, which is one of the most significant indicators of software quality. The proofs of the theorems introduced on the paper, detail discussion of the problem stated, and the examples the author try and hope to publish on the separate monography.

In the International Standard *ISO/IEC 9126 Software Product Evaluation – Software Quality Characteristics and Subcharacteristics*, the software quality is defined as the totality of characteristics of a software entity that bear on its ability to satisfy stated and implied needs (see also general definition of quality in ISO 8402). The following six quality characteristics are defined from the external point of view:

functionality as the extent to which the software in use provides function which meet stated and implied needs under specified conditions;

reliability as the extent to which software in use maintains its level of performance when used under specified conditions;

usability as the effort needed to use system containing software, and the user's satisfaction with the software, under specified conditions;

efficiency as the resources used by a system containing software in order to achieve the required performance under specified conditions;
maintainability as the effort needed to make specified modification to software and
portability as the effort needed to transfer the software to another environment.

The modification of this definitions can be given from the *internal point of view* in the terms of *set of attributes of the software that influence the respective properties* of the software. The number of *subcharacteristics* is also defined.

In the two series of prepared standards:

1. *Software Quality Chracteristics and metrics*
   - ISO/IEC 9126-1 Part 1: Quality characteristics and subcharacteristics
   - ISO/IEC 9126-2 Part 2: External metrics and
   - ISO/IEC 9126-3 Part 3: Internal metrics
2. *Software Product Evaluation*
   - ISO/IEC 14598-1 General Overview
   - ISO/IEC 14598-2 Planning and Management
   - ISO/IEC 14598-3 Process for Developers
   - ISO/IEC 14598-4 Process for Acquires
   - ISO/IEC 14598-5 Process for Evaluators and
   - ISO/IEC 14598-6 Evaluation Modules

the attempt to prepare the framework for the evaluation of quality of software product is given.

The level of various quality characteristics and subcharacteristics can be predicted in the particular stages of the software life cycle, including the development phase by various *measurable entities* of the software. More details about ISO/IEC JTC 1 standardization activities may be found in the Czech language in a less technical form in Vaníček [22].

1.2. Software complexity and its importance

Today more then 200 various measures for the quality indicators and predictors are proposed. More then 100 of them are *metrics for software complexity*. It is clear for everybody that the *software complexity measures have the key role and a direct influence to all quality characteristic mainly to software reliability, mainatainability and portability and also to such important external variables as is for example the effort needed for for the developing and implementation of the software*. To have a satisfactory measure for software complexity is therefore *the main problem in the software quality theory*.

In the monography of Zuse [26] the detail list and in Zuse [28] the summary of the complexity metrics is given and the analysis of the properties of the complexity metrics from the measurement theoretic view (see Krantz et al [10], Luce et al [12] and Roberts [20]) is reported.

The results of Zuse analysis and also the analysis of many other authors turned out that these metrics, including the sophisticated ones, often do not coincide with the intuitive notion of program complexity and/or lead to serious problems when the complexity of the combined program is to be determined from the complexity of its modules by the natural concatenation rules. Also the practical experiences with these measures points to the fact, that trivial measures, such as a number of lines of source code of the program or Halstead [5] measures, often leads to the better correlation of such external variables as the number of failures, the effort needed for program development or maintenance, than these sophisticated complexity measures. The author of this paper believes that there are three main sources of problems. Before describing these problems and explaining how to avoid them, some basic abstract concepts should be recalled.

2. SOME THEORETICAL BACKGROUNDS AND CONVENTIONS

2.1. Graphs

By the word *graph* in this paper we shall mean a simple directed graph, defined as a ordered triple (N, E, φ), where N is a nonempty finite set of nodes, E a nonempty finite set of edges and φ a total injective mapping from E into $N \times N$. Sometimes we shall identify the ordered pair of nodes $(u, v) \in N \times N$ directly with the edge φ^{-1} if some $n \in N$ such $e \in E$ that $\varphi(e) = (u, v) \in N \times N$ exists. Using this convention we shall also denote the graph as a ordered pair (N, E) only. The number of edges $||\{v \in N : (u, v) \in E\}|| = outdeg(u)$, with the origin on the node $u \in N$ is called a *out-degree* of the node u, the number of edges with its end on the node $u \in N$ the *in-degree* of $u \in N$ and denoted by $indeg(u)$.

The sequence of nodes $(u = n_1, n_2, \ldots, n_k = v)$ of the graph (N, E) we shall call the *path from u to v* if for each $j = 1, \ldots, k-1$ there exists a edge e_j from n_j to n_{j+1} such that $\varphi(e_j) = (n_j, n_{j+1})$. The graph (N, E) is said to be *connected* if for each two nodes $u, v \in N$ there exists a sequence of nodes $(m_1 = u, \ldots, m_r = v)$ such that for each $i = 1, \ldots r-1$ there is an edge $e_i \in E$ from m_i to m_{i+1}: $(\varphi(e_i) = (e_i, e_{i+1}))$ or from m_{i+1} to m_i: $(\varphi(e_i) = (e_{i+1}, e_i))$. The graph is said to be *strongly connected* if for each two nodes $u, v \in N$, there exists a path from u to v. It is clear, that the connected graph is not necessarily strongly connected. The graph (N, E) we shall call to be *acyclic* if for each $u \in N$ there exists no nontrivial path from u to u. All nodes $v \in N$ such that there exists a edge $e \in E$ from $u \in N$ to v we shall call *children* of the node $u \in N$. For details see Behzad et al [2] or Nešetřil [14].

The ordered triple (N, E, r) we shall call a *hierarchy* if (N, E) is a connected and acyclic graph, which has only one node $r \in N$, called a *root of the hierarchy* such that $indeg(r) = 0$. The nodes of the hierarchy, which have no children $(outdeg(u) = 0)$ we shall call *leafs of the hierarchy*. If in the connected and acyclic graph (N, E) each node $v \in N$: $indeg(v) \leq 1$ (v has no more than one node $u \in N$ – parent of v –, such that v is the child of u), then we shall call (N, E) to be a *tree*. Each tree has a uniquely defined root node r, such that (N, E, r) is a hierarchy. But not all hierarchies are trees as in a hierarchy can exist nodes, which are children of more then one parent nodes.

2.2. Flowgraph of the program

A majority of the complexity metrices are based on the properties of so called *flowgraph* of the program, which is defined as a (directed) graph (N, E) with a distinguished start node s and terminal node t, with the following three properties:
(1) s is the unique node in N whose in-degrec is zero, (2) t is the unique node in N whose out-degree is zero and (3) for any node $v \in N$ there exists a path from s to t, going through v. For each program its flow graph can be derived uniquely by the simple algorithm, but different programs and also different program frames can have the same flowgraph (the correspondence is not a bijection).

The intuitive meaning of the nodes on the flowgraph is that s is the starting point and t is the terminal point of the program. Unconditional program statements are those u with $indeg(u) = outdeg(u) = 1$, the predicate statements are those v with $indeg(v) = 1$ and $outdeg(v) > 1$ and the fuse points on the control path are those w with $indeg(w) > 1$ and $outdeg(w) = 1$. Sometimes it is conventional to represent unconditional statements only by the edges of a flowgraph and so called *reduced flowgraph* (N', E') in which all nodes $u \in N$ with $indeg(u) = outdeg(u) = 1$ are left out results. In this reduced flowgraph (N', E') there is

$$N' = \{v \in N : indeg(v) \neq 1 \vee outdeg(v) \neq 1\},$$

and a set of its edges E' contain all pairs of nodes (u, v) for which there exists a path from u to v in the graph (N, E), containing only the nodes being left out. Also the mapping from programs into the set of its reduced flowgraphs is not a bijection. For details see Zuse [26].

2.3. Measurement theory

The measure theoretic view is based on the theory of empirical and numerical relational systems and scale. The more detail and complete presentation of this theory can be found on Krantz et al [10] or Roberts [20].

Let $\mathcal{A} = (A, R_1, \ldots, R_n, \oplus_1, \ldots, \oplus_m)$ be an empirical system, where A is not empty set of objects, $R_i, i = 1 \ldots n$, are some k_i-nary relations on A, and $\oplus_j, j = 1, \ldots, m$, are binary operations on A. Let $\mathcal{B} = (B, S_1, \ldots, S_n, \bullet_1, \ldots, \bullet_m)$ be a formal relational system, where B is, for example a set $\Re$ of real numbers (or real vectors) and S_i and $\bullet_j$ the respective relations and operations on B. The measure μ is a total mapping of A into B which is a homeomorphism, which means that it preserves all relations and operations $(R_i(a_1, \ldots, a_{k_i}) \Leftrightarrow S_i(\mu(a_1), \ldots, \mu a_{k_i}))$ and $\mu(a \oplus b) = \mu(a) \bullet \mu(b)$ for all $i = 1, \ldots, m$ and all $a, b, a_1, \ldots, a_{k_i} \in A$). Then the triple $(\mathcal{A}, \mathcal{B}, \mu)$ is said to be a *scale*. If two relational systems $\mathcal{A}$ and $\mathcal{B}$ are given, the *representation problem* is the question of the existence of the measure μ such that $(\mathcal{A}, \mathcal{B}, \mu)$ is a scale of a given type.

As a example we can imagine various priority relations $R_1, \ldots, R_n$ and various concatenation operations $\oplus_1, \ldots, \oplus_m$ on the set of empirical objects and the respective inequalities and arithmetical operations between numbers which are the metrics quantifying respective properties of empirical objects.

Let $(\mathcal{A}, \mathcal{B}, \mu)$ be a scale. A total mapping g of $\mu(A)$ into B is called an *admissible transformation* if and only if $(\mathcal{A}, \mathcal{B}, g \circ \mu)$, where $g \circ \mu$ is defined by $(g \circ \mu)(x) = g(\mu(x))$, is also a scale. Scales can be classified according to the admissible transformations. For the software complexity theory, the following two classes of scales are of main interest: (1) the *ordinal scale*, for which all strictly increasing function is an admissible transformation, and (2) the *ratio scale*, for which the admissible transformation are the functions of the form $g(x) = a \cdot x$, where $a \in \Re^+$ is a positive real number. For the software complexity measures the following empirical relational systems and scales are frequently used:

1. The *relational systems* $\mathcal{P}^\star = (\boldsymbol{P}^\star, \star\succeq)$, and $\mathcal{P} = (\boldsymbol{P}, \succeq)$, where $\boldsymbol{P}^\star$ and $\boldsymbol{P}$ are sets of all programs and all flowgraphs, respectively and $\star\succeq, \succeq$ the relations *"to be equally or more complex"* for programs and flowgraphs, respectively. It is clear that the relations $\star\succeq, \succeq$ are weak orders on $\boldsymbol{P}^\star$ and $\boldsymbol{P}$, respectively, which means that the relations are reflexive and transitive, but it is possible for two programs or flowgraphs that there exist $P_1^\star$ and $P_2^\star$ such that $P_1^\star \star\succeq P_2^\star, P_2^\star \star\succeq P_1^\star$ and $P_1^\star \not\equiv P_2^\star$ for programs, or there exist $P_1, P_2 \in \boldsymbol{P}$ such that $P_1 \succeq P_2, P_2 \succeq P_1$ and $P_1 \not\equiv P_2$ for flowgraphs. For such objects we shall further use the notation $P_1^\star \approx P_2^\star$ or $P_1 \approx P_2$. There is a relation $\approx = (\succeq \cap \preceq)$ as a subset of $\boldsymbol{P}^\star \times \boldsymbol{P}^\star$ or $\boldsymbol{P} \times \boldsymbol{P}$ is an equivalence on the set $\boldsymbol{P}'$ (where $\boldsymbol{P}' = \boldsymbol{P}^\star$ or $\boldsymbol{P}' = \boldsymbol{P}$) of all programs or flowgraphs in question and generates a decomposition of $\boldsymbol{P}'$ into a factor set $\boldsymbol{P}'|_\approx$ of equally complex programs or flowgraphs. The respective *ordinal scales* are $(\boldsymbol{P}^\star, (\Re^+, \geq, +), \mu)$ and $(\boldsymbol{P}, (\Re^+, \geq, +), \mu)$, where $\Re^+$ is the set of positive real numbers, $\geq$ a natural order on $\Re^+$ and μ the complexity measure.
2. The *additive relational systems* $(\boldsymbol{P}^\star, \star\succeq, \oplus^\star)$ and $(\boldsymbol{P}, \succeq, \oplus)$, where the two relations have been described beforehend, and $\otimes$ and $\oplus$ are a concatenation operation of two programs (for examples a sequence or a `if then else` combination of two program modules and two flowgraphs, respectively). The respective *additive ratio scales* are $((\boldsymbol{P}^\star, \star\succeq, \oplus^\star), (\Re^+, \geq, +), \mu)$ and $((\boldsymbol{P}, \succeq, \oplus), (\Re^+, \geq, +), \mu)$, where $+$ is a addition of real numbers in $\Re^+$.

3. FIRST DOUBT: PROGRAMS OR FLOWGRAPHS?

3.1. Theoretical conditions for transparency of the compexity measures

The first weak point in software complexity measures is the following: Almost all complexity metrics are based on the properties of flowgraphs. The respective relational system and scale are built in majority cases *not on the set of programs, but on the set of their flowgraphs.* The reason why this is done is that flowgraphs being matematically clear defined objects can be handled more easily and the concatenation of flowgaphs is easy to define and is closed (for any two flowgraphs their concatenation exists). For programs, this, of course, is not fulfilled without additional consideration. If we investigate software measures based on flowgraphs, the question, however, is whether it is sufficient to discuss them on the flowgraphs. Let $P^\star$ be a set of all programs written in a given imperative programming language

(such as PASCAL, C, ADA, ...) and $\star\succeq$ the complexity relation on $P^\star$ and P, the set of its flowgraphs. *Let ϕ be the mapping that maps every program to its flowgraph.* The following conditions denoted PF1, PF2, PF3, PF4 plays a fundamental role (see also Zuse and Bollmann–Sdorra [27]):

- PF1: For all $P^\star, P_1^\star, P'^\star, P_1'^\star \in \boldsymbol{P}^\star$:($\phi(P^\star) = \phi(P_1^\star) \wedge \phi(P'^\star) = \phi(P_1'^\star) \wedge P^\star\star\succeq P'^\star) \Rightarrow P_1^\star\star\succeq P_2^\star$. (Programs with the same flowgraph may be substituted for each other with respect to its complexity).
- PF2: For all $P^\star, P'^\star \in \boldsymbol{P}^\star : \phi(P^\star) = \phi(P'^\star) \Rightarrow P^\star \approx P^\star$ (there is $P^\star \succeq P'^\star \wedge P'^\star \succeq P^\star$). (Programs with the same flowgraph are equally complex).
- PF3: For all $P^\star, P_1^\star, P'^\star, P_1'^\star \in \boldsymbol{P}^\star$: If $\phi(P^\star) = \phi(P'^\star)$, $\phi(P_1^\star) = \phi(P_1'^\star)$ and $P^\star \otimes P_1^\star$ and $P'^\star \otimes P_1'^\star$ both exist, then $\phi(P^\star \otimes P_1^\star) = \phi(P'^\star \otimes P_1'^\star)$. (The flowgraph of combined programs depend only on the flowgraph of each program).
- PF4: For all $Q, Q' \in \boldsymbol{P}$ there exist $P^\star, P'^\star \in \boldsymbol{P}^\star$ with $Q = \phi(P^\star), Q' = \phi(P'^\star)$ such that $P^\star \otimes P'^\star$ exists. (The programming language is sufficiently rich, that programs with any two flowgraphs can be combinated).

If $\star\succeq$ is transitive and F2 holds, then F1 holds. If $\star\succeq$ is reflexive and F1 holds, then F2 holds.

The following theorems have been proved for clarification of the situation:

Theorem 1. Let $((\boldsymbol{P}^\star, \star\succeq,), (\Re^+, \geq), \psi)$ be a scale. The relation $\succeq$ on $\boldsymbol{P}$ such that $P^\star\star\succeq P'^\star \Leftrightarrow \phi(P^\star) \succeq \phi(P'^\star)$ for all $P^\star, P'^\star \in \boldsymbol{P}^\star$ and a mapping μ from $\boldsymbol{P}$ to $\Re^+$ with $\psi = \mu \circ \phi$ such that $((P, \succeq), (\Re, \geq), \mu)$ is a scale, and such that both scales have the same set of admissible transformations, exists if and only if PF1 and PF2 hold.

Theorem 2. If PF1 holds then there exists a relation $\succeq$ on $\boldsymbol{P}$ satisfying the condition $P^\star\star\succeq P'^\star \Leftrightarrow \phi(P^\star) \succeq \phi(P'^\star)$ for all $P^\star, P'^\star \in \boldsymbol{P}^\star$ and such that for every total mapping of $\boldsymbol{P}$ into $\Re^+$ the following holds: $((\boldsymbol{P}^\star, \star\succeq), (\Re^+, \geq), \mu \circ \phi)$ is an ordinal scale if and only if $((\boldsymbol{P}, \succeq), (\Re^+, \geq), \mu)$ is an ordinal scale.

Theorem 3. Let $\bullet$ be some binary operation on $\Re^+$ and $((\boldsymbol{P}^\star, \star\succeq, \otimes), (\Re^+, \geq, \bullet), \psi)$ be a scale. If PF1, PF2, PF3 and PF4 hold then there exists a relation $\succeq$ and an operation $\oplus$ on $\boldsymbol{P}$ such that for all $P^\star, P'^\star \in \boldsymbol{P}^\star$: $P^\star\star \succeq P'^\star \Leftrightarrow \phi(P^\star) \succeq \phi(P'^\star)$ and $\phi(P^\star \otimes P'^\star) = \phi(P^\star) \oplus \phi(P'^\star)$, whenever $P^\star \otimes P'^\star$ exists and there exists a total mapping μ of $\boldsymbol{P}$ into $\Re^+$ such that $\psi = \mu \circ \phi, ((\boldsymbol{P}, \succeq, \oplus), (\Re^+, \geq, \bullet), \mu)$ is a scale and both scales have the same admissible transformations.

Theorem 4. If PF1, PF3 and PF4 hold then there exists a relation $\succeq$ and operation $\oplus$ on $\boldsymbol{P}$ such that for all $P^\star, P'^\star \in \boldsymbol{P}^\star$: $P^\star\star\succeq P'^\star \Leftrightarrow \phi(P^\star) \succeq \phi(P'^\star)$ and $\phi(P^\star \otimes P'^\star) = \phi(P^\star) \oplus \phi(P'^\star)$, whenever $P^\star \oplus P'^\star$ exists, such that for every total mapping $\boldsymbol{P}$ into $\Re^+$ the following holds: $((\boldsymbol{P}^\star, \star\succeq, \otimes), (\Re^+, \geq, +), \mu \circ \phi)$ is a ratio scale if and only if $((\boldsymbol{P}, \succeq, \oplus), (\Re^+, \geq, +), \mu)$ is a ratio scale.

Theorem 5. If PF1, PF2, PF3 and PF4 holds, then there exists $\succeq$ and $\oplus$ on $\boldsymbol{P}$ with $P^\star \star \succeq P'^\star \Leftrightarrow \phi(P^\star) \succeq \phi(P'^\star)$ for all $P^\star, P'^\star \in \boldsymbol{P}$ and with $\phi(P^\star \otimes P'^\star) = \phi(P^\star) \oplus \phi(P'^\star)$ for all $P^\star, P'^\star \in \boldsymbol{P}^\star$, whenever $P^\star \oplus P'^\star$ exists, such that the following statements are equivalent:

- For all $P^\star, P'^\star \in \boldsymbol{P}^\star : (\exists Q, Q1, Q2 \in \boldsymbol{P} : (\phi(P^\star) = Q \circ (Q_1 \circ Q_2)) \wedge (\phi(P'^\star) = (Q \circ Q_1) \circ Q_2)) \Rightarrow P^\star \approx P'^\star$.
- For all $Q, Q_1, Q_2 \in \boldsymbol{P} : Q \circ (Q_1 \circ Q_2) \approx (Q \circ Q_1) \circ Q_2$.

The conditions PF1, PF2, PF3 and PF4 therefore give the complete answer to the question if it is possible to study only the complexity of a flowgraph. The question whether these conditions are fulfilled or not is not a theoretical question, but rather a question of the empirical observation and intuition of experts. There are a strong grounds for the idea that these conditions are not fulfilled also for very simple programs. In the case where we have reason to believe that some of the assumptions PF1, PF2, PF3 or PF4 do not hold, flowgraph complexity does not make sense anyway.

3.2. Short case demonstration

For the clarification of our pessimism let us consider the following pseudo-pascal program frames:

```
program P1;
        begin if c then P
                   else Q;
              R
          end P1.
```

```
program P2;
  label L;
  begin if c then begin P;
                        goto L
                  end
             else begin Q;
                    L: R
                  end
    end P2.
```

```
program P3;
 label L1;
begin
   if p1 then begin S1;
      while p3 do begin S3;
                        S4;
                  L1: S5;
                        S6
                  end
               end
   else begin S2;
              if p2 then goto L1
              else S8
```

```
program P4;
 label L1, L2;
begin
   if p then begin S1;
          while p3 do begin S3;
                            S4;
                      L1: S5;
                            S6
                      end
                 L2: S7;
            end
      else begin S2;
             if p2 then goto L1
```

```
        end                                          else begin S8;
  S7                                                       goto L2
end P3.                                                end
                                          end
                                  end P4.
```

It is clear that P1 has the same flowgraph as P2, and P3 the same flowgraph as P4, but probably everybody will consider the program P2 as more complex then P1 and P4 more complex as P3, even though programs P2, P3 and P4 are ugly.

4. SECOND DOUBT: SHOULD COMPLEXITY MEASURES REALLY BE ADDITIVE?

4.1. Full additivity and extensive structures

The key problem in program complexity measure theory is the question of the full additivity of the complexity measure with respect to such a concatenation as is for example the sequential concatenation $P_1 \oplus_{seq} P_2$ defined by $P_1 \oplus_{seq} P_2 =$ **begin** P_1; P_2 **end** and the selection combination $P_1 \oplus_{sel_C} P_2 =$ **if** C **then** P_1 **else** P_2. In Zuse [26] and Zuse [29] and Zuse [30] it is proven, that for the existence of the positive additional ratio scale, for programs and/or flowgraph, described in the Section 2, the necessary and sufficient condition is that $(\boldsymbol{P}, \succeq, \oplus)$ is a so-called *closed extensive structure*, (see Krantz et al [10]) which means that the following set of axioms are satisfied:

- $\succeq$ is a weak order on $\boldsymbol{P}$
- $\oplus$ is weakly associative operation on $\boldsymbol{P}$ $(P_1 \oplus (P_2 \oplus P_3) \approx (P_1 \oplus P_2) \oplus P_3)$
- (weak monotonicity): $P_1 \succeq P_2 \Leftrightarrow P_1 \oplus P_3 \succeq P_2 \oplus P_3 \Leftrightarrow P_3 \oplus P_1 \succeq P_3 \oplus P_2$ for all $P_1, P_2, P_3 \in \boldsymbol{P}$
- (Archimedean axiom): If $P_1 \succeq P_2$, than for any $P_3, P_4 \in \boldsymbol{P}$ there exists a natural number n, such that $n \circ P_1 \oplus P_3 \succeq n \circ P_2 \oplus P_4$, where $n \circ P$ is defined by the induction: $1 \circ P = P$; $(n+1) \circ P = P \oplus (n \circ P)$.

The last *Archimedean axiom* is the most problematic point in the closed extensive structure demand. There are two equivalent formulation of this requirement:

- The pair of elements $(A, B) \in \boldsymbol{P} \times \boldsymbol{P}$ is called *anomalous pair* if $A \not\approx B$ and either for all natural number n there is

$$((n+1) \circ B \succ n \circ A) \wedge ((n+1) \circ A \succ n \circ B)$$

  or for all natural number n there is

$$(n \circ A \succ (n+1) \circ B) \wedge (n \circ B \succ (n+1) \circ A).$$

  The requirment is that there do not exist any anomalous pair.
- If we shall call the sequence A, $A \oplus A = 2 \circ A$, $3 \circ A, \ldots$ the *standard sequence*, the Archimedean axiom says that every strictly bounded $(\exists B \in \boldsymbol{P} \, \forall n : B \succ n \circ A)$ sequence is finite.

The closed extensive structure is called to be *positive* if in addition the property $P_1 \oplus P_2 \succeq P_1$ for all $P_1, P_2 \in \boldsymbol{P}$ holds. The main theorem of Krantz et al [10], Section 3.2, says that $((\boldsymbol{P}, \succeq, \oplus), (\Re^+, \geq, +), \mu)$ is a scale, which means that for all P_1, $P_2 \in \boldsymbol{P}$ the following holds:

- (I) : $P_1 \succeq P_2$ if and only if $\mu(P_1) \geq \mu(P_2)$
- (II): $\mu(P_1 \oplus P_2) = \mu(P_1) + \mu(P_2)$.

if and only if $(\boldsymbol{P}, \succeq, \oplus)$ is a positive closed extensive structure. Another function η satisfies the conditions (I) and (II) if and only if there exists some $c \in \Re^+$, such that $\eta = c \cdot \mu$.

The problem is with the empirical validation of the Archimedean axiom, whether the requirement, that very much simple problems in its the concatenation become more complex, than one very unlucid and complicated problem, really describe the situation in real life. For example if a very large, but structured program is more complex to understand than the short one with some multiple entry, parallel or overlapping loops.

4.2. Additivity or wholeness?

The second problem is if the additivity of the complexity measure really conforms to the opinion of experts. Many authors, for example Weyuker [23], Prather [18] and Pressman [19], p. 324, suggest the property of *wholeness*, that says that the complexity of the sum must be at least as big as the sum of complexity of the parts, that is, $\mu(P_1 \oplus P_2) \geq \mu(P_1) + \mu(P_2)$. This opinion is also confirmed by the well known so called COCOMO model (see Boehm [3]) in which the relation between the effort E to write a program and $L(P)$ which is the lenght of its code is estimated by the equation $E = a \cdot L(P)^b$, where $b > 0$ is for various programs estimated by numbers from the interval $(1.05, 1.20)$, depending on the type of the program.

In all of referenced works of Zuse it is substantiated that wholeness is a requirement without any empirical meaning, and thus, it is impossible to validate in the terms of programs or its flowgraphs. The additive measure for program complexity is validated if the axioms of the closed extended structure is fulfilled, and for such external variables, as is the effort or the number of faults in the program, the transformation of the program complexity by means of strictly monotonic function f from $\Re^+$ into $\Re^+$ is suggested. This approach leads to now nonadditive scales, where on $\Re^+$ a new operation $\bigcirc$ is defined by $a \bigcirc b = f^{-1}(f(a) + f(b))$. If, for example, $f(t) = \log(t)$, than $\bigcirc = \cdot$, where $\cdot$ is a multiplication on $\Re^+$. If $f(t) = \tanh^{-1}(t) = \log((1+t)/(1-t))/2$, then $a \bigcirc b = (a+b)/(1+a \cdot b)$ and we obtain the addition of relativistic velocity. In Roberts [20] it is shown, that if such a new scale $((\boldsymbol{P}, \preceq, \oplus), (\Re, \geq, \bigcirc), \mu)$ is also a ratio scale, then the function f must have a form $f(x) = a \cdot x^b$ for some $a, b \in \Re^+$. Therefore, this result gives an excellent theoretical background for COCOMO formula.

However the experiences of experts has shown that the growth of the effort can not be estimated only from the effort for the components which are parts of the whole software complex. There are two main objections against additivity of complexity measures and derived general formulas like the COCOMO one:

- If the number of components in the concatenation process is very large, the growth of the effort is steeper, and the reliability of the concatenated component decrease more strictly.
- The growth of the effort and fall of the reliability is not only a function of the complexity of the software components, but also depend very much on the level of coupling between the components, which can be measured, for example, by the amount of shared global data, number of parameters passed between the modules, shared subordinate modules, and generally by so-called fan-in and fan-out data flow complexity. See for example Ovideo [15], or Troy and Zweben [21]. Therefore the complexity $\mu(P_1 \oplus P_2)$ of the concatenation cannot be only a function of two variables $\mu(P_1)$ and $\mu(P_2)$ only, but as a function of three variables $\mu(P_1)$, $\mu(P_2)$ and some data flow and coupling complexity $\delta(P_1, P_2)$ between P_1 and P_2. This point of view casts doubt upon the additivity of complex measures and of course also the associativity axiom for the concatenation operation.

4.3. Limited extensive structures, using local ordered semigroups

The first objection can be partially solved using the concept of concatenation operation, which is not closed, based on the theory of *ordered local semigroups* and *extensive structures with no essential maximum* (see Krantz et al [10] Sections 2.2 and 3.4). The principle of this idea is that only entities which are not arbitrarily large can be created. In the analogy if our problem is not to concatenatiate programs, but rods, we can form a new rod $a \oplus b$ from rods a and b only if we have enough room to do so. If our room is not sufficiently large, we can go into a corridor, and when that fails we can go on a the campus walk, but sooner or later we are forced to stop concatenating, and a classical theory of unlimited addition cannot actually be performed due to the practical limitation of space.

In this alternative theory, we shall add the new relation $\mathcal{B}$ on $\boldsymbol{P}$, such that $(P, Q) \in \mathcal{B}$ denotes that P can be concatenated, and thus $P \oplus Q \in \boldsymbol{P}$ exists. The most sensitive problem is now the formulation of the Archimedean property with these limited conditions. For this purpose the more convenient is the following alternative formulation of the Archimedean axiom: *Every strictly bounded standard sequence is finite.*

In this theory the following definition is proposed and the related theorem can be proven for replacing of the unlimited addition condition for the existence of ordinal additive scale:

Definition 1. Let $\boldsymbol{A}$ be a nonempty set, $\succeq$ and $\mathcal{B}$ a binary relations on $\boldsymbol{A}$ (subsets of $\boldsymbol{A} \times \boldsymbol{A}$) and $\oplus$ a binary operation from $\mathcal{B}$ into $\boldsymbol{A}$. Let us denote for $P, Q \in \boldsymbol{A}$: $P \succ Q \Leftrightarrow (P \succeq Q \wedge Q \not\succeq P)$. The quadruple $(\boldsymbol{A}, \succeq, \mathcal{B}, \oplus)$ is a *local extensive structure with no essential maximum* if and only if the following six axioms are satisfied for all $P_1, P_2, P_3 \in \boldsymbol{A}$:

1. The relation $\succeq$ is a weak order on $\boldsymbol{A}$.
2. $((P_1, P_2) \in \mathcal{B} \wedge (P_1 \oplus P_2, P_3) \in \mathcal{B}) \Rightarrow (P_2, P_3) \in \mathcal{B}$
   $\wedge (P_1, P_2 \oplus P_3) \in \mathcal{B} \wedge (P_1 \oplus P_2) \oplus P_3 = P_1 \oplus (P_2 \oplus P_3)$.

3. $((P_1, P_3) \in \mathcal{B}) \wedge (((P_1, P_2) \in \mathcal{B}) \Rightarrow ((P_3, P_2) \in \mathcal{B}) \wedge (P_1 \oplus P_3 \succeq P_3 \oplus P_2))$
4. $P_1 \succ P_2 \Rightarrow (\exists\, Q \in \boldsymbol{A} : (P_2, Q) \in \mathcal{B} \wedge P_1 \succeq P_2 \oplus Q)$.
5. $(P_1, P_2) \in \mathcal{B} \Rightarrow P_1 \oplus P_2 \succ P_1$.
6. Every strictly bounded ($\equiv_{\text{Def}}\ \exists Q \in \boldsymbol{A}\, \forall n : Q \succ P_n$) standard ($\equiv_{\text{Def}}\ P_{n+1} = n \circ P_n$ for $n = 1, \ldots$) sequence is finite.

Theorem 6. Let $(\boldsymbol{A}, \succeq, \mathcal{B}, \oplus)$ be a local extensive structure with no essential maximum. Then there exists a function μ from $\boldsymbol{A}$ into $\Re^+$ such that for all $P_1, P_2 \in \boldsymbol{A}$, the following holds:

- (I) : $P_1 \preceq P_2$ if and only if $\mu(P_1) \geq \mu(P_2)$;
- (II): $(P_1, P_2) \in \mathcal{B} \Rightarrow \mu(P_1 \oplus P_2) = \mu(P_1) + \mu(P_2)$.

If another function η satisfies (I) and (II) then there exists a $c \in \Re^+$such that for all nonmaximal $Q \in \boldsymbol{A}$ there is $\eta(Q) = c \cdot \mu(Q)$ (ratio scale property).

The present representation of ratio scale, in this classical sense, assumes that there is no maximal element with respect to $\succeq$. It is also possible to formulate a similar theory if there exists some maximal element. Such an element $Z \in \boldsymbol{P}$ is called an *essential maximum* relative to $\succeq$ and $\oplus$ if and only if it is maximal respective to $\succeq$ and there exists some $P \in \boldsymbol{P}$ such that $(Z, P) \in \mathcal{B}$. In the definition of extensive structure with essential maximum the last two axioms have to be replaced by:

- 5-Max. For all $P_1, P_2 \in \boldsymbol{P}, (P_1, P_2) \in \mathcal{B}$ if P_1 is not an essental maximum then $P_1 \oplus P_2 \succ P_1$.
- 6-Max. If the standard sequence is strictly bounded by an element that is not an essential maximum, then the sequence is finite.

The set of all essential maxima can be characterized by equations $Z \approx Z \oplus P$ for $(Z, P) \in \mathcal{B}$, and if Z and Z' are two essential maxima, then $Z \approx Z'$. The restriction of the local extensive structure $(\boldsymbol{P}, \succeq, \mathcal{B}, \oplus)$ with some essential maxima to $(\boldsymbol{A}', \succeq', \mathcal{B}', \oplus')$, where $\boldsymbol{A}'$ is the set of all nonmaximal elements of $\boldsymbol{A}$ and $\succeq', \mathcal{B}', \oplus'$ the restrictions of $\succeq, \mathcal{B}, \oplus$ to $\boldsymbol{A}' \times \boldsymbol{A}'$, respectively is an extensive structure with no essential maximum. Two different strategies for the measures on the local extensive structures with an essential maximum can be used:

- To ignore the maximal elements and leave the measure for these elements undefined;
- If μ is an additive function on $\boldsymbol{A}'$ and g a strictly increasing function from $(0, 1)$ to $\Re^+$, to define $\Phi(P) = g^{-1}(\mu(P))$ for nonmaximal P, and $\Phi(Z) = 1$ if Z is any essential maximum. The Φ satisfies the formula $\Phi(P_1 \oplus P_2) = g^{-1}(g(\Phi(P_1) + \Phi(P_2)))$ if $P_1 \oplus P_2 \succ Z$ and if we introduce the convention $g(1) = \infty$, $g^{-1}(\infty) = 1$ and $t + \infty = \infty$ for $t \in \Re$, then also for $P_1 \approx Z$, where Z is an essential maximum. To obtain the usual relativistic addition formula for velocities, the choice is $g = \tanh^{-1}$.

4.4. Independency conditions, decomposition facilities and program coupling

The second objection is *much more serious.* It is directly connected with the principal question whether the concatenation $P \oplus Q$ on the set $\boldsymbol{P}'$ of programs or flowgraphs is transparent with respect to the relation $\approx = (\succeq \cap \preceq)$, (to be *"equally complex"*) or not. That means if a program consists of several modules whether it is possible to compute the overall complexity from the complexity of the single modules. The following theorems contain a complete answer to this principle question (see also Zuse and Bollmann–Sdorra [27]):

Theorem 7. Let $((\boldsymbol{P}', \succeq, \oplus), (\Re, \geq), \mu)$ be a ordinal scale on the empirical relational structure $(\boldsymbol{P}', \succeq, \oplus)$ and $\approx = (\succeq \cap \preceq)$. Then:

- The binary operation $\bullet$ on $\mu(\boldsymbol{P}') \subseteq \Re$, such that $\mu(P, Q) = \mu(P) \bullet \mu(Q)$ for $P, Q \in \boldsymbol{P}'$ exists if and only if for all $P_1, P_2, P_3 \in \boldsymbol{P}'$: $P_1 \approx P_2 \Rightarrow (P_1 \oplus P_3 \approx P_2 \oplus P_3) \wedge (P_3 \oplus P_1 \approx P_3 \oplus P_2)$.
- The operation $\bullet$, which is one to one in each variable exists if and only if for all $P_1, P_2, P_3 \in \boldsymbol{P}'$: $P_1 \approx P_2 \Leftrightarrow P_1 \oplus P_3 \approx P_2 \oplus P_3 \Leftrightarrow P_3 \oplus P_1 \approx P_3 \oplus P_2$.
- The operation $\bullet$, which is not decreasing in each variable exists if and only if for all $P_1, P_2, P_3 \in \boldsymbol{P}'$: $P_1 \preceq P_2 \Rightarrow (P_1 \oplus P_3 \preceq P_2 \oplus P_3) \wedge (P_3 \oplus P_1 \preceq P_3 \oplus P_2)$.
- The operation $\bullet$, which is strictly monotonic in each variable exists if and only if for all $P_1, P_2, P_3 \in \boldsymbol{P}'$: $P_1 \preceq P_2 \Leftrightarrow P_1 \oplus P_3 \preceq P_2 \oplus P_3 \Leftrightarrow P_3 \oplus P_1 \preceq P_3 \oplus P_2$.

A new, weaker concept, of *independence* in software complexity measurement theory then of Weyuaker [23] can be proposed:

Definition 2. The relational system $(\boldsymbol{P}, \preceq, \oplus)$ is said to be *decomposable* if and only if there exists functions $f : \boldsymbol{P} \to \Re$, $g : \boldsymbol{P} \to \Re$ and $F : f(\boldsymbol{P}) \times g(\boldsymbol{P}) \to \Re$ such that for all $P, Q, R, S \in \boldsymbol{P}$: $P \oplus B \preceq R \oplus S \Leftrightarrow F(f(P), g(Q)) \geq F(f(R), g(S))$. If the function F is strictly increasing in each variable, then this relational system is said to be *monotonically decomposable.*

Definition 3. The relational system $(\boldsymbol{P}, \preceq, \oplus)$ is *independent* if and only if for all $P, Q, R, S \in \boldsymbol{P}$: $((P \oplus R \preceq Q \oplus R) \Leftrightarrow (P \oplus S \preceq Q \oplus S)) \wedge ((R \oplus P \preceq R \oplus Q) \Leftrightarrow (S \oplus P \preceq S \oplus Q))$.

Definition 4. Let $\preceq$ be a weak order on the set $\boldsymbol{P}$. The subset $\boldsymbol{A} \subset \boldsymbol{P}$ is called to be *order-dense* with respect to $\preceq$ if and only if for all $P, Q \in \boldsymbol{P}$: $P \prec Q$ there exists some $A \in \boldsymbol{A}$ such that $Q \preceq A \preceq Q$.

The following theorem can be proven:

Theorem 8. Let $\boldsymbol{P}$ be finite, countable or contain some finite or countable order-dense subset with respect to a weak order $\preceq$. Then:

- There exists a total mapping μ of $\boldsymbol{P}$ into $(\Re, \geq)$ such that $P \preceq Q \Leftrightarrow \mu(P) \geq \mu(Q)$ for all $P, Q \in \boldsymbol{P}$.
- $(\boldsymbol{P}, \preceq, \oplus)$ is monotonically decomposable if and only if it is independent.

The discussion at the beginning of this section leads to very serious doubts that the conditions on Theorems 7 and 8 can be satisfied if the program modules, which is to be combined are not fully independent. The effect of data sharing and control links between the program modules leads to the conclusion, that with the exception of the unrealistic situation of complete independency between concatenating software components the assumptions of Theorem 7 and the independency condition in Definition 3 are not satisfied in the real world. Therefore the endeavour to create a satisfactory complexity measure system for software on the base of binary concatination operation of its modules is looked upon by the author with great pessimism. The author has the conviction, that for the combination of software modules, the trinary operation, such as the glue operation of directed graph by the additional hierarchicaly superior node on which the coupling between the concatenating components is described, must be used. This principle will be described more clearly in the last Section 6 of this paper.

5. THIRD DOUBT: IS THE SENSITIVITY TO UNSTRUCTUREDNESS AT AN APPROPRIATE LEVEL?

5.1. Cyclomatic number and derived measures

For the comp'exity of individual software module many various measures, mostly based on the analysis of flowgraphs is used (see for example Zuse [26], Zuse [28]). The measures of Halstead [5] and similar measures for flowgraphs, such as number of its nodes, are invariant with respect to the elementary operation *to move the node from one to the another place of the flowgraph.* It is clear, that this type of measures does not support the structural complexity of the software, which is in strong contradiction with the experiences in software engineering.

To describe the complexity on the appropriate level, various measures are suggested. The most primitive is the so-called *Cyclomatic number* by McCabe is defined by $\mu_c = ||E|| - ||N|| + 2$, where $||E||$ and $||N||$ are numbers of edges and nodes, respectivelly. This measure, which is equal to the number of linearly independent paths in the strongly connected graph (to make a flowgraph strongly connected a virtual edge $T \rightarrow S$ had to be inserted), is sensitive to the number of predicates in the program, but not to their relation, and therefore not against unstructuredness and nesting depth.

The *Essential cyclomatic number* substracts from the cyclomatic number the number of proper subgraphs with single-entry and single-exit nodes. This leads to a reduction process of the graph G replacing such single entry and single exit structural subgraph by a single node. The subgraphs concidered for this reduction are only

the *D-structured subgraphs* (sequence; selection: `if-then-else` or `case`; iteration: `while-do` or `repeat-until`) and not other general prime subgraphs with single-entry and single-exit nodes. For the flowgraph which is D-structured, which means it is built only using the listed constuctions, the essential cyclomatic number is equal to 1 and therefore the measure is too coarse. The second objection against this measure is that it estimates the complexity of a relatively reasonable construction such as the loop with multiple exit points with the same weight as absolutely unadmissible control constructions, such as a multiple-entry loop or overlapping loops.

5.2. Interval analysis and reducibility

Hecht [7] defines a measure based upon the interval analysis which measures the nesting depth of loops. It is also used to estimate the effort of data flow analysis algorithms in compilers (see Aho et al [1]). The interval $I(h)$ of the flowgraph G is a subset of nodes constructed for a node h by the algorithm: $I(h) := \{h\}$;
`while` there exists a node $v \neq s, v \notin I(h)$, all predecessors of v are in $I(h)$ `do`
$I(h) = I(h) \cup h\{v\}$
`endwhile`.

The node h is a header of the interval. For the partitioning G into a unique set of disjoint itervals, the header nodes are selected by the algorithm:
`construct` $I(s)$;
`while` there exists a node h, which is not yet a member of any interval, and all its predecessors are already members of some interval `do`
`construct` $I(h)$
`endwhile`.

The partition of a flowgraph G into intervals leads to the construction of a new flowgraph $I(G)$ by the following rules:

- The nodes of $I(G)$ correspond to the intervals of G.
- The start node of $I(G)$ is the interval of G that contain the start node of G.
- In $I(G)$ there is an edge from the interval U to the different interval V if and only if in G there is an edge from some node in U to the header of V.

The flowgraph in the sequence $G, I(G), I(I(G)), \ldots$ is the *limit flowgraph* if it is equal to its interval partition. The flowgraph is called *reducible* by intervals if its limit graph is trivial (single node without self-loop). Every flowgraph which is not reducible contains some multiple-entry loop and therefore is not D-structured. On the other hand, multiple-exit loops are reducible, and so are some other unstructured flowgraphs; for example abnormal selections of path (jumps between the branches on the selection) are also reducible. Also some more complicated unstructural constructions which are not use loops, for example the construction of Yourdon [25] and other constructions listed by Williams [24], are reducible. Reducibility is therefore weaker condition than D-structuredness and also BJ-structuredness, sometimes called semistructuredness (structuredness with respect to Dijkstra constructions and the Bohm–Jacopiny construction of the loop with multiple exits – `exit` or `quit` in the backtracking dictionary). For reducible flowgraph G the measure $\mu_{idsl}(G)$, defined as a interval derived sequence length can be used. This measure is sensitive

to nesting depth of natural loops and admits multiple-exit loops, is not defined for multiple-entry loops, but it is inconveniently tolerant with respect to some other unstructural constructions than multiple entry loops (for example the construction of Yourdon [25]). For irreducible flowgraphs, this measure does not have any tools for classifying the degree of unstructuredness on more levels then two.

Many other complexity measures are used to try to estimate the level of nesting complexity and unstructuredness in an appropriate way. For example for each node v of the flowgraph G the set of all paths from all successors of v to the terminal node have a uniquely defined node $\inf(v)$ called the greatest lower bound of v. The set of nodes on all paths from v to $\inf(v)$ is called the *context* of v. For all nodes on the context of node v some *additional complexity* can be defined for example by Harrison and Magel [6]. Piwowarski [16] has combined this kind of additional complexity measure with the modified McCabe complexity and has worked out some measure, which is relatively fine and sensitive to unstructuredness. Howewer, these measures are not able to distinguish sufficiently the various kinds of unstructuredness, and it seems to the author that the penalty for unacceptable unstructuredness is not sufficiently severe in this class of measures.

5.3. Vector complexity measures

Also, some non-scalar measures are used, such as the *predicate nesting vector* which contains three parts: (1) The *predicate execution number*, which shows for each node v nesting depth of predicates that determine the execution of v; (2) The *number of loop predicates*, which is the number of loops which contain v; (3) The *degree of unstructuredness*, which quantifies the unstructuredness of the context, described in the last paragraph. If we add the predicate node p itself to its own range which is the difference for non-loop predicates, we reach the extended context $extcont(v)$ of the node v. Basically, two predicates p and q are structured with respect to each other if every path from p to $extcont(q)$ enters this extended context at exactly one node $entry(q)$. Such a three part vector measure is of course not convenient from the point of view of measurement theory and additivity.

We can close this section with the conclusion that besides the general problem whether or not it is sufficient to investigate flowgraphs in place of programs or not, *today there is not a flowgraph complexity measure available which is able to support on a satisfactory level the commonly-accepted view of program complexity.*

6. THE ATTEMPT TO AVOID PROBLEMS. THE LUCID HIERARCHY AS A PROGRAM MODEL

6.1. Program hierarchy

After the critique in the last three sections we shall try to offer a new way for the study of program complexity, based on the theory of Langefots [11] of lucid structures and top-down design, and the principles of hierarchical design of software from Jackson [9]. The general idea of this attempt is the decomposition of the complicated and unlucid problem, which is difficult to understand in its entirety into

several lucid problems using given methods of decomposition, with the limitation for the number of components into which the decomposition is realized in that one level. If the number of components is too large, it is necessary to arrange the decomposition procedure into several levels in such a way, that each decomposition is lucid. The measure of program complexity is the man's effort which is needed for this decomposition and for the understanding of each part.

Jackson [9] proposes for a D-structured programs' design a graphical representation as directed graph, which is a tree with nodes or edges colored depending on the decomposition rule used (sequence, selection or iteration). The root of this tree is the whole program, and leafs are individual statements. Unfortunately this simple model cannot be directly used for such programs, which contain subroutines and/or functions, which are called in several points, loops with multiple exits, recursive subroutines. It is also not applicable for unstructured (irreducible) programs. Therefore the generalization of the tree structure to the more general notion of *program hierarchy* is necessary. Some ideas of the proposed attempt are based also on Ph.D. thesis of Dvořák [4] and Pospíšil [17].

Let us suppose, that the relation of subroutine calls is acyclic, and there does not exist any subroutine, without any call, and accessable only by `goto` statement into this subroutine. Let us further assume that the recursive procedure definition can be transformed using the transcription rule: DefRec $\rightarrow$ `HEAD if P then OUT else In`, where `HEAD, OUT, IN` are nonterminals, generating the heading, the sequence of statements which does not contain a recursive call and the statements which may contain the internal call, respectively. Program, subroutines, recursive procedures (further abbr. recursions), blocks, loops, selections, subroutine and function calls, exits, and all other jumps which can occur in program, will be called *actions*. The entire program is the action of the hierarchical level zero. For all actions of the level i, the following holds: Each successors of this action in its decomposition, its children, with except of subroutines and recursions have the level $i+1$. The level of the subroutine and recursion is defined as $\min(j_1, j_2, \ldots, j_n)$, where $j_1, \ldots, j_n$ are levels of the action's call. Let us assume that the natural order $\prec$ of the set $\mathcal{A}$ of all actions, which is generated by the order in which they are listed in the program, is given. If we use an (ordered) graph terminology to describe the decomposition of the program action as an edge on the graph, the program decomposition can be investigated as a hierarchy with the nodes ordered by this natural order in which statements are listed in the source program text (so called *ordered hierarchy*). The following definition is proposed for the formalization of our situation:

Definition 4. The quadruple $(\mathcal{A}, \mathcal{H}, r, \Xi)$ is called to be a *program hierarchy* if and only if $(\mathcal{A}, \mathcal{H}, r)$ is a ordered hierarchy with the natural order $\prec$, $\mathcal{A}$ is the set of action of the program r, $\mathcal{H}$ is the decomposition relation on the set of program actions and Ξ is a subset of $\mathcal{A} \times \mathcal{A}$ and if and only if the following holds *(intuitive meaning is described in italics)*:

1. The decomposition $\mathcal{A} = \mathcal{PG} \cup \mathcal{SR} \cup \mathcal{BL} \cup \mathcal{ST} \cup \mathcal{SQ} \cup \mathcal{SL} \cup \mathcal{WH} \cup \mathcal{RP} \cup \mathcal{SC} \cup \mathcal{RD} \cup \mathcal{RO} \cup \mathcal{RI} \cup \mathcal{RC} \cup \mathcal{EX} \cup \mathcal{GT}$ is given.
   ($\mathcal{PG} = \{r\}$ denotes a whole program, $\mathcal{SR}$ the set of all subroutine nodes, $\mathcal{BL}$

blocks of code with local data structure declarations, $\mathcal{ST}$ simple statements, $\mathcal{SQ}$ sequence decompositions, $\mathcal{SL}$ selections (`if then else` *or* `case` *,) $\mathcal{WH}$* `while` *loops; $\mathcal{RP}$* `repeat` *loops, $\mathcal{SC}$ subroutine calls, $\mathcal{RD}$ recursion definitions, $\mathcal{RO}$ the subtrees of the recursion without internal call, $\mathcal{RI}$ subtrees of the recursion contained its internal call, $\mathcal{RC}$ internal recursion calls, $\mathcal{EX}$* `exit (quit)` *statements from loops and $\mathcal{GT}$ other (unstructural)* `goto` *statements.)*

2. If we denote $(\mathcal{A}, \mathcal{H}')$ the restriction of the graph $(\mathcal{A}, \mathcal{H})$ to the set of edges $\mathcal{H}' = \{(u, v) \in \mathcal{H} : u \notin \mathcal{SC}\}$. Let $\Xi \subset \mathcal{A} \times \mathcal{A}$ be a relation on $\mathcal{A}$ (the set of edges, which is to be added to the structurogram or r if r is not D-structured) such that the following for all $(a_1, a_2 \in \mathcal{A} \wedge (a_1, a_2) \in \Xi)$ holds:
(*Natural limitation of unstructural decompositions, described by the set Ξ: All jumps are allowed in the main program and inside subroutines only, and are leafs on the hierarchy, and from such a leaf only one edge is allowed, and is allowed only to some predecessor of this edge on the hierarchy.*)
   - There is $a_1 \in \mathcal{EX} \vee a_1 \in \mathcal{GT}$.
   - Then exists a connected component of $(\mathcal{C}, \mathcal{H}_C)$ of $(\mathcal{A}, \mathcal{H}')$ such that $a_1, a_2 \in \mathcal{C}$.
   - If a_2 is a root of connected component $(\mathcal{C}, \mathcal{H}_C)$ and $c_1 \in \mathcal{C}$, then $c_1 \in \mathcal{EX}$.
   - If exists $a_3 \in \mathcal{A}$ such that $(a_1, a_3) \in \mathcal{H}$, then $(a_1, a_3) \in \Xi$.
   - If $a_1 \in \mathcal{EX}$, then a_2 is preccessor of a_1.
   - There exists some $a_3 \in \mathcal{A}$ such that $(a_3, a_2) \in \mathcal{H}$.

3. For all $a_1, a_2, a_3, \in \mathcal{A}$ holds:
(*Natural conditions for structured decomposition: It is only one root program; if the program is the set of modules, the virtual root action for its collection shall be added; each cycle have only one child, its body; recursions have two children, one without and the second with the recursion call; the subroutine call has only one child, which is a subroutine or recursion; simple statement, internal recursion call, exit and goto statement are leafs.*)
   - $\mathcal{PG} = \{r\}$.
   - $a_1 \in \mathcal{WH} \cup \mathcal{RP} \wedge (a_1, a_2) \in \mathcal{H}, (a_1, a_3) \in \mathcal{H} \Rightarrow a_2 = a_3$.
   - $(a_1 \in \mathcal{RD} \wedge (a_1, a_2) \in \mathcal{H} \wedge (a_1, a_3) \in \mathcal{H} \wedge a_1 \prec a_3$ in the natural order $\prec) \Rightarrow (\{b \in \mathcal{A} : (a_1, b) \in \mathcal{H}\} = \emptyset \wedge a_2 \in \mathcal{RO}, a_3 \in \mathcal{RI})$.
   - $(a_2 \in \mathcal{RO} \cup \mathcal{RI} \wedge (a_1, a_2) \in \mathcal{H}) \Rightarrow a_1 \in \mathcal{RD}$.
   - $(a_1 \in \mathcal{SC} \wedge (a_1, a_2) \in \mathcal{H} \wedge (a_1, a_3) \in \mathcal{H}) \Rightarrow (a_2 = a_3 \wedge a_2 \in \mathcal{SR} \cup \mathcal{RD})$.
   - $(a_1, a_2) \in \mathcal{H} \wedge a_2 \in \mathcal{SR} \cup \mathcal{RD}) \Rightarrow a_1 \in \mathcal{SC}$.
   - $a_1 \in \mathcal{ST} \cup \mathcal{RC} \cup \mathcal{EX} \cup \mathcal{GT} \Rightarrow \{(b : (a_1, b) \in H\} = \emptyset$.

The set $\mathcal{F} = \mathcal{ST} \cup \mathcal{SC} \cup \mathcal{RC}$ is called the set of *function nodes*, the set $\mathcal{D} = \mathcal{WH} \cup \mathcal{RP} \cup \mathcal{SL} \cup \mathcal{RD}$ the set of *decision nodes*. The set of all nodes which are neither function nor decision nodes is called the set of *virtual nodes* and is denoted by $\mathcal{V}$. The set of all children of conditional nodes we shall call the set of *conditional nodes* and denote by $\mathcal{C}$. It is clear that $\mathcal{F} \cup \mathcal{D} \cup \mathcal{V} = \mathcal{A}$, but each of its sets can have a non-empty intersection with $\mathcal{C}$. The program hierarachy is called to be *complete*

if each leaf of the hierarchy $(\mathcal{A}, \mathcal{H}, r)$ is an element of the set of functional nodes $\mathcal{F} = \mathcal{ST} \cup \mathcal{SC} \cup \mathcal{RC}$.

Not all types of constructions in the program have to be used. If the progamming language does not have enough tools for some structured constructions, but some *standard constructions* using `goto`, which corresponds to such costructions as `while, repeat, case, exit,...` can be identified, such usage of `goto` statement can be considerred as a respective structural construction.

The *concept of program is not formalized* in this paper. However for each imperative programming language some formal grammar for program frames (programs without type and data descriptions) can be given, and transcription rules which transform the program text in an algorithmic way into the program hierarchy can be proposed. Such an algorithm takes a linear course through the program text, and using the stack for the initialized but not yet finished program actions. The concrete implementation of such a algorithms for various programming languages have been developed by fellow workers and students of the author. The output of such an algorithm is the program hierarchy a dynamical data structure.

6.2. Program schema and transformation of program hierarchies

For all complete hierarchies, the concept of *program schema*, which is analogy of this concept from Zohar Manna [13], can be defined, in which to each function node corresponds some function action, and to each conditional node some predicate action. Our schema is howewer more rich, as it contains also abstract nodes, corresponding to some "virtual" actions which are realized, according to the Dijkstra's philosophy, by means of its subordinate components.

For complete program hierarchies and corresponding program schemas, extended program hierarchy and extended program schema respectively, can be defined in natural way. For the given input data sequence $\mathcal{INP}$ the interpretation of the program schema (in the terminology of Zohar Manna, more precisly the interpretation of program schemas signature) can be described. The interpretation is intuitively clear, but its formal description is relatively time-consuming, due to the variability of action items and various rules for selecting the successor of the given node depending of its type. However, such a formalization using the concept of the so-called successor depending of the given set of currently inappropriate nodes was given. The sequence of nodes $(r, a_1, a_2, \ldots)$ called a *control path* can be derived, and for the given input is uniquely defined. The corresponding sequence $(p_1, p_2, \ldots)$, where p_j, $j = 1, \ldots$ is the corresponding function action for functional nodes and is the empty set $\emptyset$ for other nodes is called the *computing history* for all the given input $\mathcal{INP}$. Two program schemas $\mathcal{PS}_1, \mathcal{PS}_2$ with the same signature and its interpretation are called to be *functionally equivalent*, if they have the identical computing history for each admissible input. The complete program hierarchy $\mathcal{H}_1$ is said to be *transformable* to the complete program hierarchy $\mathcal{H}_2$ (or $\mathcal{H}_2$ to be a transformation of $\mathcal{H}_1$) if and only if $\mathcal{H}_2$ is a extension of $\mathcal{H}_1$ and for each program schema of $\mathcal{H}_1$ there exists its extended program schema which is functionaly equivalent to the original one.

The following special type of program hierarchy transformation will be used in the last subsection. Let $v \in \mathcal{A}$ be a node such that two edges $u, w \in \mathcal{A}$ exists such that $(u, v) \in \mathcal{H}$ and $(w, v) \in \Xi$. Let us make a copy of the node w and of the whole subtree with the root w and add this copy into the program hierarchy as a successors of the node w. If we now use the **goto** statement (if the set of nodes which have been copied coincide with the set of all successor of the youngest common predecessor $\inf(w, v)$ of the nodes v and w, **exit** can be used) to the first successor of copied nodes which have not been copied, the computation history will be preserved. This method of transformation is called *the transformation by means of node copies.*

6.3. Complexity measure for well-designed programs

Definition 5. The program hierarchy and each its program schema is called well-designed if and only if using the notation from the Definition 4 the following holds:

$$(u, v) \in \Xi \Rightarrow u \in \mathcal{EX}.$$

For such program hierarchies there is $\mathcal{GT} = \emptyset$.

Definition 6. Let $(\mathcal{A}, \mathcal{H}, r, \Xi)$ is the program hierarchy, $x \in \mathcal{A}$ a node and $T(x) = (T, K, x)$ the subtree of the hierarchy $(\mathcal{A}, \mathcal{H} \cup \Xi, r)$ which contain only the node x and its children on the relation $\mathcal{H} \cup \Xi$. This subtree is called a *realization of the action* x. Let k be a natural number, called the *level of lucidity*. The realization of the action x is called *k-lucid* if and only if $||K|| \leq k$, where $||K||$ is the number of edges on the realization $T(x) = (T, K, x)$ of the action x.

The numb r k (level of lucidity) should be considered as a psychological limitation for the understanding of the entire decomposition of the action x. If this limitation is exceeded, the additional transformation on the next hierarchical level is necessary. The choice of the constant k is an open question; however our experience leads to the strong opinion, that k should not be very large and surely should have only one decimal digit. Perhaps, the choice k in the interval from 5 to 7 (7 for experienced professionals) is recommendable.

Now we are able to define the decomposition complexity for the realization of each action.

Definition 7. Let $T(x)$ be the realization of a action x and $|K|$ the number of edges on the realization $T(x) = (T, K, x)$. The *decomposition complexity* $DC(x)$ of x is defined by:

- If $||K|| = 0$ then $DC(x) = 0$.
- If $||K|| = 1$ then $DC(x) = 1$.
- If $||K|| > 1$ then

$$DC(x) = (||K|| + k - 3)\mathrm{div}(k - 1) = trunc((||K|| + k - 3)/(k - 1)),$$

where k is the level of lucidity and div the operation of integer division with the truncation of the remainder.

This formula counts the minimal necessary number of lucid realizations, which are functionally equivalent to the given realization of $T(x)$, which can be generally unlucid.

Sometimes it can be useful to count a decomposition complexity in such structures, which have been unnecessarily decomposed with respect to lucidity, as less complex than follows immediately from this definition. For this porpose the following definition can be used:

Definition 8. Let $T(x)$ be the realization of a action x and $DC(x)$ the decomposition complexity of x defined in Definition 7. Let $\sqsubset$ be some order on the set of action with the following properities:

- if u is a successor of x, then $u \sqsubset x$.
- if u, v are both children of the same action x and $DC(u) < DC(v)$, then $u \sqsubset v$.

It is clear that such an order exists, but in general is not uniquelly defined. Let us define a new *reduced decomposition complexity* (more precisely reduced decomposition complexity with respect to $\sqsubset$) $RDC(x)$ for all actions step by step by the order $\sqsubset$ as follows:

- $(DC(x) > 1 \vee DC(x) = 0) \Rightarrow RDC(x) = DC(x)$
- If $DC(x) = 1$, let $(u_1, u_2, \ldots)$ be children of x in the selected order $\sqsubset$ and $n\max$ the maximal index such that

$$E(x) + \sum_{i=1}^{n\max} \sharp T(u_i) \sharp < k,$$

  where $\sharp T(z) \sharp$ denotes the number of edges on the realization $T(z)$ of the action z. In this case we shall define $RDC(x) = 1$ and $RDC(u_i) = 0$ for all $i = 1, \ldots n\max$. If such a sequence of children does not exist, we shall put $RDC(x) = DC(x)$.

Theorem 9. There exists an algorithm for the generation of the complete program hierarchy for each program or program frame and to compute for this program hierarchy the decomposition complexity (or reduced decomposition complexity) for each lucidity level k, given by natural number.

Now we are able to *propose* the *global complexity measure for each well-designed program hierarchy* by the formula:

$$\Upsilon((\mathcal{A}, \mathcal{H}, r, \Xi)) = \alpha . \sum_{x \in \mathcal{A}} DC(x) + \beta . \sum_{x \in \mathcal{A}} TC(x) + \gamma . \sum_{x \in \mathcal{A}} IC(x) + \delta . \sum_{x \in \mathcal{A}} EC(x),$$

where

- $DC(x)$ is the *decomposition complexity* of action corresponding to the node x, defined by the Definition 7. If it is appropriate this term can be replaced by the reduced decomposition complexity $RDC(x)$, which is defined on the Definition 8.
  This contribution is zero if x is the leaf in the program hierarchy.
- $TC(x)$ is the natural number defined depending on the set of various types of nodes in the program hierarchy. One of various possible suggestions for example is:
  * $TC(x) = 0$ for $x \in \mathcal{PR} \cup \mathcal{SR} \cup \mathcal{BL} \cup \mathcal{ST}$
  * $TC(x) = 1$ for $x \in \mathcal{SL} \cup \mathcal{RD} \cup \mathcal{RO}$
  * $TC(x) = 2$ for $x \in \mathcal{WH}$ (or $TC(x) = 2 + c$, where c is the number of nodes in $\mathcal{WH} \cup \mathcal{RP} \cup \mathcal{RC} \cup \mathcal{RI}$ on the shortest path from the root r and node x)
  * $TC(x) = 3$ for $x \in \mathcal{RP} \cup \mathcal{RC} \cup \mathcal{RI}$ (or $TC(x) = 3 + c$, where c is determined as before)
  * $TC(x) = 4$ for $x \in \mathcal{EX}$ (or $TC(x) = 4 + c$, where c is the number of nodes in $\mathcal{WH} \cup \mathcal{RP} \cup \mathcal{RC} \cup \mathcal{RI}$ on the shortest path from the youngest common predecessor $\inf(x, y)$ of x and y, where $(x, y) \in \Xi$ is the corresponding exit relation — *c is the depth of the exit.*)

  But this evaluation is dependent on subjective taste and can be the subject of discussions.
- $IC(x)$ is the *internal complexity* of the functional and decision node x, which can be measured by some formula of the Halstead [3] type counting the number of elementary items on the statement and on the predicate, which is necessary to compute in the respective node.
  This contribution is of course equal zero on virtual nodes.
- $EC(x)$ is the *external complexity* of the node x which can be measured by the ammount of fan-in and fan-out data, including parameters and global data using some appropriate measures from Henry and Kafura [8] or others listed on Zuse [26]).
  This contribution can be very significant in the root node $\mathcal{PG} = \{r\}$, where it describes the complexity of data input and output of the program and in virtual nodes in $\mathcal{V} \subset \mathcal{A}$ such as subroutines or blocks, where cohesion and coupling can be taken into consideration.
- $\alpha, \beta, \gamma, \delta \in \Re$ are nonnegative real numbers which can be considered as *weights* for the individual views for the global complexity of the program, *depending on preferences.*

The concept of global complexity has the sense only for entire real programs with data structures and statements, not for program frames. If our choice is $\gamma = \delta = 0$ a global complexity can be called the *structural complexity* of the program hierarchy and denoted as

$$\Omega((\mathcal{A}, \mathcal{H}, r, \Xi)) = \alpha . \sum_{x \in \mathcal{A}} DC(x) + \beta . \sum_{x \in \mathcal{A}} TC(x).$$

This measure can be used not only for entire program but also for program frames. If our choice is $\alpha = \beta = \delta = 0$, $\gamma = 1$ or $\alpha = \beta = \gamma = 0$, $\delta = 1$ the words *internal* or *external complexity* of the program can be used, respectively.

6.4. Complexity for not well-designed programs

As it has been demonstrated in the last section the unstructuredness can have a drastically different influence to the program complexity and lack of understability, depending on the context in which the unstructural `goto` statement is used. Therefore it is not a way to solve the problem on a adequate level for programs which are not well-designed using some general penalty constant for $TC(x)$ when $x \in \mathcal{GT}$.

To solve the problem in the case when $\mathcal{GT} \neq \emptyset$ and there exists some $(u, v) \in \Xi$ such that $u \in \mathcal{GT}$, we propose transformation by means of node copies, mentioned at the end of Subsection 6.2, be used. The following theorem can be proved:

Theorem 10. For each not well-designed program with the program hierarchy $\mathcal{H}$ and its program schema there exists a transformation into the well-designed program hierarchy $\mathcal{H}'$ such that this hierarchy is well-designed, and $\mathcal{H}$ and $\mathcal{H}'$ and respective program schemas are functionally equivalent. Such a program hierarchy transformation can be realized by an algorithm.

The formal description of the algorithm is relatively long and complicated. The two different cases, the jump back and the jump ahead with the respect to the relation of the left way through the hierarchy, should be separated. Roughly speaking, the idea is to copy whole subtree of the target of unstructural `goto` statement by means of a node copy transformation in a different context.

Now we are able to define a *decomposition complexity* and also the *structural and global complexity for not well-structured program hierarchies* by the equation

$$\Omega(\mathcal{H}) = \Omega(\mathcal{H}'), \quad \Upsilon(\mathcal{H}) = \Upsilon(\mathcal{H}'),$$

where $\mathcal{H}'$ is the well-designed functional equivalent transformation of $\mathcal{H}$ from the Theorem 10. And $\Omega(\mathcal{H}')$ and if needed, also $\Upsilon(\mathcal{H}')$ is to be computed using the formulas in Subsection 6.3.

This method gives the adequate penalty to such unstructural constructions as is for example multiple entry or overlapping loops, which coincides to the general intuitive feeling of experts.

6.5. A brief estimation of the proposed method

The proposed method has been implemented for several program languages and sample programs with the relatively good results. The obtained complexity measure values are able to distinguish in a very subtle way the programs with respect to understability of its function from the program code, and the results are, in our opinion, not comparable with any complexity measure proposed before.

The open question of course is the choice of the weight coefficients (α, β, γ and δ) between structural, internal and external complexity and between the contribution

of various types of nodes on the program hierarchy. Our present recomendation is to use this flexibility of the proposed method for the tuning of the proposed metrics to our subjective intuitive feeling for the complexity, which may be different for various groups of experts. On the other hand, our experience leads to the opinion, that the proposed complexity measure is not very sensitive with respect to the choice of the lucidity level constant k. Its choice, of course, is also in question.

The main limitation of the proposed measure is the absence of the data complexity review in the method. The measure is also oriented mainly to classical imperative programming languages and is probably not adequate for the measure of complexity of programs written in object-oriented languages. However, the object-oriented software complexity measures are, as is shown for example in Zuse [29], in the present time only in the beginning stages and cannot be considered satisfactory.

The autor submits the method as a modest contribution for the effort to design an adequate complexity metrics for software on the prepared standard or technical report ISO/IEC 9126-3: Information Technology. Software Product Evaluation. Part 3: Internal metrics. This contribution in slightly modifyied form was recorded as ISO/IEC, JTC1, 7/6-Dub-12 by the working group ISO/IEC, JTC1, SC7, WG6 "Information Technology. Software Engineering. Evaluation and Metrics", on its Dublin meeting on November, 1995.

(Received December 21, 1995.)

REFERENCES

[1] A. V. Aho, R. Sethl and J. D. Ullman: Compilers: Principles, Techniques, and Tools. Addison–Wesley, Reading, MA 1986.

[2] M. Behzad and G. Chartrand: Introduction to the Theory of Graphs. Allyn and Bacon, Boston 1971.

[3] B. W. Boehm: Software Engineering Economics. Prentice Hall, Englewood Cliffs 1981.

[4] L. Dvořák: Program Complexity Measurement (in Czech). Ph.D. Thesis. ČVUT FEL & ČSAV, Praha 1987.

[5] N. H. Halstead: Elements of Software Science. First edition. Elsevier, North Holland, New York 1977.

[6] W. A. Harrison and K. I. Magel: A complexity measures based on nesting level. SIGPLAN Notices *16* (1981), 3, 63–84.

[7] M. S. Hecht: Flow Analysis of Computer Programs. Elsevier, North Holand, New York 1977.

[8] S. Henry and D. Kafura: Software Metrics based on information flow. IEEE Trans. Software Engrg. 7 (1981), 5, 510–518.

[9] M. A. Jackson: Principles of Program Design. Academic Press, London – New York – San Francisco 1975.

[10] D. H. Krantz, D. R. Luce, P. Suppes and A. Tversky: Foundation of Measurement. Vol. 1. Additive and Polynomial Representations. Academic Press, San Diego 1971.

[11] B. Langefors: Theoretical Analysis of Information Systems. Studentlitteratur, Lund 1973.

[12] D. R. Luce, D. H. Krantz, P. Suppes and A. Tversky: Foundation of Measurement. Vol. 3. Representation, Axiomatization and Invariance. Academic Press, San Diego 1990.

[13] Z. Manna: Mathematical Theory of Computation. McGraw Hill, New York 1974.

[14] J. Nešetřil: Graph Theory (in Czech). Mathematical Workshop SNTL 13, Praha 1979.
[15] E. I. Oviedo: Control flow, data flow and programmers complexity. In: Proc. of COMPSAC 80, Chicago 1980, pp. 146–152.
[16] P. Piwowarski: A nesting level complexity measure. SIGPLAN Notices *17* (1982), 9,44–50.
[17] J. Pospíšil: Program Structure Complexity Measurement (in Czech). Ph.D. Thesis. VUT Brno 1988.
[18] R. E. Prather: An axiomatic theory of software complexity measure. Comput. J. *27* (1984), 4, 340–347.
[19] R. S. Pressman: Software Engineering: A Practitioner's Approach. McGraw Hill, New York 1992.
[20] F. S. Roberts: Measurement Theory with Applications to Decisionmaking, Utility, and the Social Sciences. Encyclopedia of Mathematics and its Applications. Addison Wesley, Reading, MA 1979.
[21] D. Troy and S. Zweben: Measurement the quality of structured design. J. System Software *2* (1981), 113–120.
[22] J. Vaníček: It the software standardization possible (in Czech)? Magazín ČSN *10* (1995), 205–210 and *11* (1995), 225–228.
[23] E. J. Weyuker: Evaluation software complexity measures. IEEE Trans. Software Engrg. *14* (1989), 9, 1357–1365.
[24] M. H. Williams: Flowchart schemate and the problem of nomenclature. J. Comput. *26* (1983), 3.
[25] E. Yourdon: Techniques of Program Structure Design. Prentice Hall, Englewood Cliffs 1975.
[26] H. Zuse: Software Complexity – Measures and Methods. De Gruyter Berlin – New York 1991.
[27] H. Zuse and P. Bollmann–Sdorra: Measurement theory and software measures. In: Formal Aspects of Measurement (T. Denvir, R. Herman and R. Whitty, eds.), Workshops in Computing, Springer–Verlag, Berlin 1992, pp. 219–259.
[28] H. Zuse: Complexity metrices/analysis. Software complexity metrics/analysis. In: Encyclopedia of Software Engineering (J. J. Marciniak, ed.), Vol. I. Wiley, New York 1994, pp. 131–166.
[29] H. Zuse: Foundation of the validation of object–oriented software measures. In: Deutsche Universitätsverlag DUV, Gaber–Vieweg–Westdeutscher Verlag, Berlin 1994.
[30] H. Zuse: Foundation of validation, prediction of software measures. In: Proc. of the Annual Oregn Workshop on Software Metrics, Silver Fall State Park 1994, pp. 1–16.

Doc. RNDr. Jiří Vaníček, CSc., Katedra inženýrské informatiky, Stavební fakulta ČVUT (Department of Informatics, Faculty of Civil Engineering, Czech Technical University), Thákurova 7, 116 29 Praha 6, and Úřad pro státní informační systém (The Office for the State Information System), Havelkova 22, 130 00 Praha 3. Czech Republic.