UNIVERSITY OF JOENSUU
DEPARTMENT OF COMPUTER SCIENCE
REPORT SERIES A

Algorithms for Counting Unadjusted Function Points from Dataflow Diagrams

Raimo Rask

Report A-1991-1

UDK 681.3.06:519.68 ACM D.2.8
ISSN 0789-7316
ISBN 951-708-016-6

TABLE OF CONTENTS

Algorithms for Counting Unadjusted Function Points from Dataflow Diagrams

Raimo Rask
Department of Computer Science
University of Joensuu
PO Box 111
80101 Joensuu, Finland
rask@cs.joensuu.fi

September 30, 1991

ABSTRACT

To calculate a cost estimate of a software system we should be able to use the descriptions produced by the current CASE-tools to measure automatically the size of the system. Current software development environments do not, however, provide enough support for the measurement of a system description during the analysis phase of software development. This paper gives *algorithms* for calculating Albrecht's *unadjusted function point value* based on dataflow and entity-relationship diagrams that are used during Structured Analysis.

1. INTRODUCTION

The development of software metrics started from code level in the environment of third generation languages (e.g. McCabe 1976, Halstead 1977). The automation of code level metrics has been easy by scanning the written code and producing metrics values. Although it has been possible to use graphical upper level description techniques (e.g. DeMarco 1978, Lundeberg & al. 1981, Jackson 1983) and upper level measurement techniques (e.g. Albrecht 1979, DeMarco 1982) one has not been able to automate the measurement process of these top level specifications in the absence of required automation tools.

The introduction of CASE-tools in software development has radically changed the dynamics of software creation (Banker & al. 1990) transforming the process of software development into a more automated, rigorous and standardized engineering discipline. Grady (1990) considers CASE-tools as one of the greatest promises for automating the measurement required by project management.

Currently the best CASE-tools cover all life-cycle phases of software engineering. The most visible part of these CASE environments are, however, the diagramming tools for the analysis phase. If we now are able to define reliably the size of a software system by a product metric based on these graphical descriptions and if we are able to automate the counting process, we can also easily estimate the cost of the software system using process metrics.

We believe that, in future, gathering work product data and work effort data demands such a systems development work-bench by which we can automatically gather the data (Rask 1990). The collected data should be gathered into a project database which should come as a part of the repository of the current CASE-tools. From the project database we can get the data for estimation purposes.

In chapter 2, we first give a short description of the classification of the user functions used in Albrecht's (1979) Function Point Analysis. Chapter 3 clarifies the transaction concept by which we can bound the effects of user transactions in the dataflow diagrams of Structured Analysis. In chapter 4 we then introduce the exact algorithms to count the value of unadjusted function points from Structured Analysis descriptions. These algorithms can be used as a basis for automating the size estimation of software systems during the analysis phase, i.e. at the requirements specification level of software development. Chapter 5 is a summary of our work and states the future directions of our project.

2. FUNCTION CLASSIFICATION IN FUNCTION POINT ANALYSIS

Albrecht (1979) has developed a widely applied measurement method which is called *Function Point Analysis* (FPA). FPA measures the size of an application from an end-user perspective. A *function point* is defined as one end-user business function. The user functions are organized into the following five groups (Albrecht & Gaffney 1983, Development Centre Productivity Measurement Guide 1984, Dreger 1989): external input types, external output types, external inquiry types, logical internal file types and external interface file types. The counting process contains the following phases:

- identification and classification of user functions,
- defining the complexity of each function,
- calculating an unadjusted function point value,

- specifying the general application characteristics, and
- counting the final function point value.

To analyze these functions, we first identify all unique functions and classify them into the groups above. Then we weight each function by its level of complexity: simple, average, or complex. The sum of these weights is called a *total of unadjusted function points* (TUFP). Table 1 shows the different *weights* for each function type (Albrecht & Gaffney 1983, Dreger 1989).

Table 1. FPA function types and possible weights

Function Type	Simple	Complexity Average	Complex
External Input	3	4	6
External Output	4	5	7
Logical Internal File	7	10	15
External Interface File	5	7	10
External Inquiry (Input)	3	4	6
External Inquiry (Output)	4	5	7

The classification of inputs, outputs, and inquiries depends on the number of files referenced and on the number of data items referenced. Correspondingly, the classification of logical internal files and external interfaces depends on the data items referenced and on the logical record formats of the files or logical relationships between the files. The *classification factors* for user functions are given in Table 2. The complete classification principles are found in the Appendix of this report.

Table 2. Classification factors for user functions

Function Type	Files referenced	Data items referenced	Logical record formats/relationships
External Input	x	x	
External Output	x	x	
Logical Internal File		x	x
External Interface File		x	x
External Inquiry (Input)	x	x	
External Inquiry (Output)	x	x	

The implementation of a logical system may vary depending on goals and hardware/software environment. For this reason, Albrecht (1979) has defined 14 environmental *adjusting factors* the sum of which, the total degree of influence (TDI), is used when counting the *final function point value* (FFPV). This is counted by the following formula:

$$FFPV = TUFP * (0.65 + TDI/100).$$

In this report we will concentrate only on counting the size of a software system: the value of unadjusted function points from the descriptions of Yourdon's (1989) Structured Analysis.

3. TRANSACTIONS IN DATAFLOW DIAGRAMS

A software system implements automated user procedures which are instances of real world user transactions. A transaction in the system is initiated by some event in the environment. The model of the software system thus consists of the descriptions of a transaction set. To measure the cost of the system during the analysis phase it is essential to find the user transactions and to define the effort required by the realization of these transactions, i.e. the cost of realization. To achieve this cost estimate we have to use a graphical description from which we may reliably trace and extract each transaction and the measurable components of these transactions to define the size of the system, the basis for cost estimation.

If we are using *Structured Analysis* (SA), developed by DeMarco (1978), we describe the dynamic behaviour of a system by a *dataflow diagram* (DFD) which consists of dataflows, processes, data stores and terminators. The static description of the system is described currently by an *entity-relationship diagram* (ERD) and by a data dictionary (Yourdon 1989).

In the DFD, a *transaction* consists of elementary functions or primitive processes which are directly chained to each other via dataflows, and of data stores and terminators which are connected to these functions (Shoval 1988, Shoval & Manor 1990). A user transaction is identified by an input-flow or by an output-flow. A dataflow from an external terminator to a process signifies an event by which a user triggers an input transaction. A dataflow to a terminator signifies an event by which a user triggers an output transaction. One of the primitive processes that constitute a transaction, is the trigger, i.e. a function connected to the terminator which is the cause of an event.

Figure 1 represents two user transactions. In the transaction triggered by R, the trigger f1 is the starting function and the chain effect goes forward from f1 to f2. In the other transaction triggered by S, the trigger function f4 is the last function on its chain, and cannot be executed before f3 is executed.

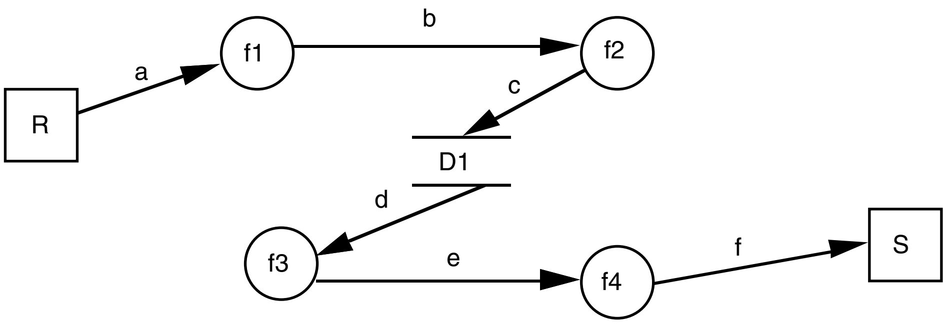


Figure 1. Two user transactions (Shoval 1988)

The measurement of a system should be objective and algorithmic (Conte & al. 1986), i.e. the metric should be computed precisely according to an algorithm; its value should not change due to changes in time, place, or observer. According to this requirement, we need exact algorithms and working practices for the counting process.

4. ALGORITHMS FOR COUNTING UNADJUSTED FUNCTION POINTS

In the following we will consider each of the function types of FPA one after the other. We will give exact algorithms to identify functions in the DFD and to find their properties needed in the complexity classification. We will also give one reduced example of each function type to illustrate the counting process. The rules without algorithms are also given by Rask (1991). Hence, an unadjusted function point value for a system can be evaluated early at the specification phase. Moreover, our method can be fully automated, and therefore it can be implemented with available CASE-tools.

4.1. External Input and Output Types

The *inputs* in a DFD come from terminators and the *outputs* go to terminators. We can find the required *input data items* in the input dataflow structure which is recorded in a data dictionary. The *files referenced* by an input transaction are the logical internal files corresponding to data stores of the dataflow diagram *inside the transaction*. If the input transaction updates the data store using two or more different keys, the data store is counted as two or more logical internal files depending on the number of these different keys (see section 4.3).

The possible alternatives in an input flow have effect on the number of unadjusted function points: it is not the same thing if we have one input flow with three alternatives (for example d = [a|b|c]) or if we have three separate input flows. Our *working practice* must pay attention to this situation so that the cost estimates are useful.

Algorithm 1 can be used to automate finding and weighting of the inputs. An obvious upper bound for the complexity of procedure weights_of_inputs is O(m) if the number of input dataflows is *m*. Correspondingly, an obvious upper bound for the complexity of procedure find_logical_files_of_inputs is O(n) if the number of dataflows in the processed dataflow diagram is *n*. Thus, the complexity of Algorithm 1 is O(mn).

Algorithm 1. Finding and weighting of inputs.

procedure weights_of_inputs (input_list, input_weight_list):
-- input_list imports the input dataflows
-- input_weight_list exports the weights of the input dataflows
-- local variable logical_file_list contains all logical files referenced
-- local variable n counts the number of internal logical files
-- local variable s counts the number of data items (size) of the input dataflow f
-- local variable w counts the weight of the input dataflow f
for all m input flows f=(t,p) with terminator t and process p in input_list **do**
    logical_file_list : = nil;
    find_logical_files_of_inputs (p, logical_file_list);
    n : = length(logical_file_list); s : = |f|; w : = input_weight(n, s);
    input_weight_list : = input_weight_list ∪ w.

```
procedure find_logical_files_of_inputs (v, logical_file_list):
-- v imports the address of primitive process or data store of the transaction
-- logical-file-list contains the internal logical files for the input dataflow
-- <d,key(f)> is an internal logical file of data store d with the key of dataflow f
-- <d,key(d)> means an internal logical file of data store d with its primary key
-- (v,v') means a directed edge from v to v'
for all dataflows f connected to v do
    if f is connected to a data store d then
        if f is named then
            if <d,key(f)> is not in logical_file_list then
                add <d,key(f)> to logical_file_list;
        elsif <d,key(d)> is not in logical_file_list then
            add <d,key(d)> to logical_file_list;
    elsif f=(v, v') and v' is a process and (v,v') is unmarked then
        mark (v,v');
        find_logical_files_of_inputs (v', logical_file_list).
```

The *output data items* and the required *file references* for complexity classification are found in the same way as we represented in connection with external inputs. In our view every output must *correspond* to some input flow in a DFD; we can not assume that an output includes an input as in Figure 1 because this input is a part of the application as a user function.

Algorithm 2. Finding and weighting of outputs.

```
procedure weights_of_outputs (ouput_list, output_weight_list):
-- output_list imports the output dataflows
-- output_weight_list exports the weights of the output dataflows
-- local variable logical_file_list contains all logical files referenced
-- local variable n counts the number of internal logical files
-- local variable s counts the number of data items of the output dataflow f
-- local variable w counts the weight of the output dataflow f
for all m output flows f=(p,t) with process p and terminator t in output_list do
    logical_file_list : = nil;
    find_logical_files_of_outputs (p, logical_file_list);
    n : = length(logical_file_list); s : = |f|; w : = output_weight(n, s);
    output_weight_list : = output_weight_list ∪ w.
```

```
procedure find_logical_files_of_outputs (v, logical_file_list):
-- v imports the address of primitive process or data store of the transaction
-- logical-file-list contains the internal logical files for the output dataflow
-- <d,key(f)> is an internal logical file of data store d with the key of dataflow f
-- <d,key(d)> means an internal logical file of data store d with its primary key
-- (v',v) means a directed edge from v' to v
for all dataflows f connected to v do
    if f is connected to a data store d then
        if f is named then
            if <d,key(f)> is not in logical_file_list then
                add <d,key(f)> to logical_file_list;
        elsif <d,key(d)> is not in logical_file_list then
            add <d,key(d)> to logical_file_list;
    elsif f=(v', v) and v' is a process and (v',v) is unmarked then
        mark (v',v);
        find_logical_files_of_outputs (v', logical_file_list).
```

Algorithm 2 can be used to automate finding and weighting of outputs. The algorithm is similar to Algorithm 1 except for the direction of dataflows. In Algorithm 2 we must go backwards the dataflows i.e. to count incoming dataflows to processes. Thus the complexity of Algorithm 2 is also O(mn).

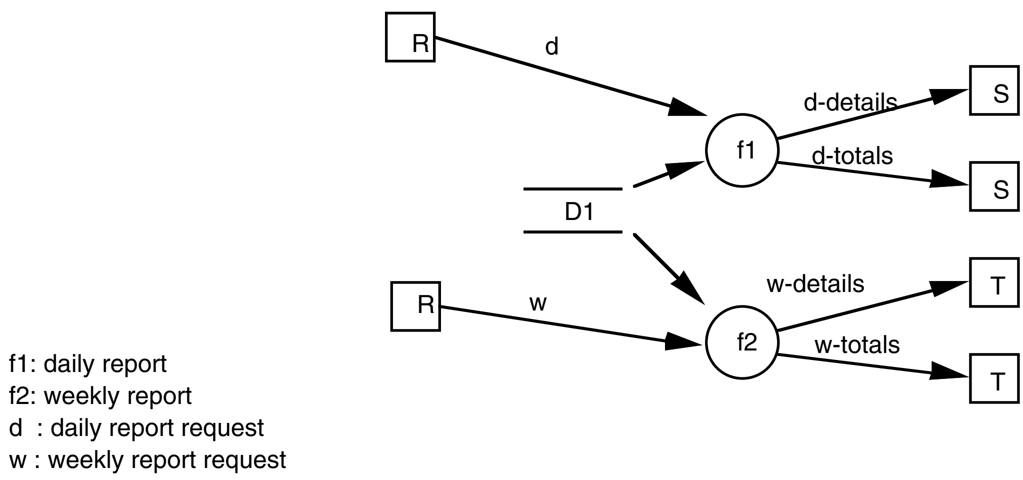


Figure 2. An example of inputs and outputs

Figure 2 contains two different transactions. The first transaction consists of one input (d from R), two outputs (d-details and d-totals to S), one process (f1), and one data store (D1). The data store is also contained in the second transaction. Here we can find at least six unadjusted function points for external inputs: there are two inputs (d, w) and the complexity of an input is at least three. Similarly, there are at least 16 unadjusted function points for external outputs: there are four outputs and the complexity of an output is at least four. The exact value of unadjusted function points depends on the number of data items referenced and this information can be found in the data dictionary as described above. The complexity values used in the example are the minimum values; all possible numeric values of the complexities are represented in Table 1 and in the Appendix.

4.2. External Inquiry Types

An *inquiry* includes both input and output dataflow. An *inquiry input* enters the application boundary and an *inquiry output* leaves the application boundary. Both parts of an inquiry (input and output) are associated with the same terminator at the topmost level of the dataflow diagram. To *identify* an inquiry we should present an inquiry dataflow by a *double directed edge*. Although Yourdon (1989) uses this double directed edge, the meaning may be different from our principle. If an inquiry is given by two directed edges, it is difficult to make difference between inquiries and input and output types.

We can decompose an inquiry into subinquiries and after decomposition the input and output dataflows of the decomposed inquiry may be associated with separate processes. A separate input or a separate output flow of an inquiry can not further be divided into separate input or output flows. The *weighting* of an inquiry input and output is done in the same way as in the case of external inputs and outputs. As a result we have to choose the worst (input or output) part for function point counting.

Algorithm 3 can be used to automate finding and weighting of inquiries. Because an inquiry consists of an input part and an output part it is possible that procedure weights_of_inquiries uses procedures find_logical_files_of_inputs and find_logical_files_of_outputs from Algorithms 1 and 2 to count weights for input and output parts of inquiry flows. The complexity of Algorithm 3 thus is O(2mn) = O(mn) if the number of dataflows in the processed dataflow diagram is n and the number of inquiry dataflows is m.

Algorithm 3. Finding and weighting of inquiries.

procedure weights_of_inquiries (inquiry_list, inquiry_weight_list):
-- inquiry_list imports the inquiry dataflows
-- inquiry_weight_list exports the weights of the inquiry dataflows
-- local variable logical_file_list contains all logical files referenced
-- local variable w counts the weight of the inquiry flow f = $<f_i,f_o>$
-- local variable n_i counts the number of internal logical files for input part f_i
-- local variable s_i counts the number of data items (size) of the input part f_i
-- local variable w_i counts the weight of the input part f_i
-- local variable n_o counts the number of logical files for output part f_o
-- local variable s_o counts the number of data items (size) of the output part f_o
-- local variable w_o counts the weight of the output part f_o
for all m inquiry flows f = $<f_i,f_o>$ in inquiry_list **do**
    select input part f_i = (t,p) from inquiry flow f;
    logical_file_list : = nil;
    find_logical_files_of_inputs (p, logical_file_list);
    n_i : = length(logical_file_list); s_i : = $|f_i|$; w_i : = input_weight(n_i, s_i);
    select output part f_o = (p,t) from inquiry flow f;
    logical_file_list : = nil;
    find_logical_files_of_outputs (p, logical_file_list);
    n_o : = length(logical_file_list); s_o : = $|f_o|$; w_o : = output_weight(n_o, s_o);
    w : = max(w_i, w_o)
    inquiry_weight_list : = inquiry_weight_list $\cup$ w.

Often an inquiry is part of an uppdate as described in Figure 3.

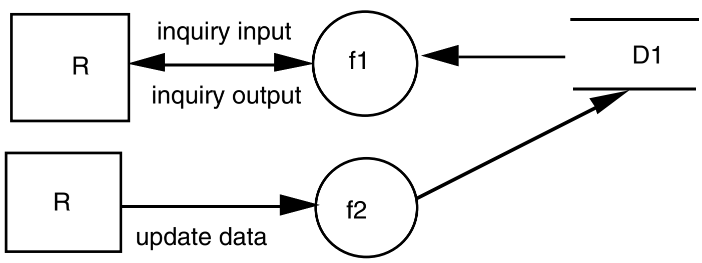


Figure 3. A typical inquiry as part of update

Figure 3 describes two transactions: one for an inquiry and one for an update. The complexity value for the inquiry is at least four since the complexity of the inquiry input is at least three and the complexity of the inquiry output is at least four. Thus the worst of these complexities is at least four. For the input of the update the complexity is at least three.

4.3. Logical Internal File Types

In a DFD each data store corresponds to one or more *logical internal files*. Based on Rask (1991) we should take into account the different keys by which data is accessed from data stores to count unadjusted function points from DFDs. Access keys must be contained in the data store and must be present in the corresponding dataflows; this kind of rule is proposed by Cowan (1990), also. Where the records are to be read or written sequentially 'next' or 'last' could be specified. The *number of different keys* used corresponds to the number of logical internal files of each data store. It is reasonable to assume that each dataflow contains only one key (or a combined key) i.e. one logical record and one logical record format. Sometimes, of course, it is reasonable to describe a varying part for a record, for example r = @k + a + b + [c|d|e]. If a data store contains different record formats the referenced dataflow should contain the name of one of these records. If the record format of a data store is singular we may leave the referenced dataflow unnamed. For each logical file the number of referenced data items can be found by considering dataflows around the data store. If a dataflow is unnamed we understand that all the data items of the data store are referenced and the key of this dataflow is the primary key of the data store.

The *relationships* which a data store (a logical internal file) participates in may increase the value of unadjusted function points, also. To understand the relationships between data stores we must know the *entity model* (entity-relationship diagram) of the application area. This model should direct the drawing of the DFD and we can find the relationships directly from the entity model. This idea is supported also by the requirement (Yourdon 1989) that every store on the DFD must correspond to an object type, or a relationship, or an associative object type. The logical internal files defined by the different keys of a data store have the *same relationships* as the original data store. For example, if the data store BOOKS is accessed by the ISBN of a book or by a name of a book the relationships of these logical files correspond to the relationships of the entity BOOK in the entity model.

Algorithm 4. Finding and weighting of logical internal files.

```
procedure weights_of_logical_files (data_store_list, file_weight_list):
-- data_store_list imports the data stores
-- file_weight_list exports the weights of the internal logical files
-- local variable logical_record_list contains all logical records of a data store
-- local variable r counts the number of relationships for a data store d
-- local variable l counts the number of logical record formats for a logical file
-- local variable n counts the maximum of r and l
-- local variable s counts the number of data items (size) of the logical file f
-- local variable w counts the weight of the logical file f
for all p data stores d in data_store_list do
    r := number of the relationships of d counted from ER-diagram;
    logical_record_list := nil;
    find_logical_records_of_data_stores (d, logical_record_list);
    for all logical files f in logical_record_list do
        l := number of logical record formats of f;
        n := max(r,l);
        s := |f|;
        w := file_weight(n, s);
        file_weight_list := file_weight_list ∪ w.

procedure find_logical_records_of_data_stores (d, logical_record_list):
-- d imports the address of the data store in the DFD
-- logical-record-list contains the logical records for the data store d
-- <d,f,key(f)> means a logical record of data store d with name and key of
-- dataflow f
-- <d,d,key(d)> means a logical record of data store d with name d and primary
-- key
for all dataflows f connected to d do
    if f is named then
        if <d,f,key(f)> is not in logical_record_list then
            add <d,f,key(f)> to logical_record_list;
    elsif <d,d,key(d)> is not in logical_record_list then
        add <d,d,key(d)> to logical_record_list.
```

Algorithm 4 can be used to automate finding and weighting logical internal files based on DFD and ERD. The complexity of algorithm 4 in the worst case is

$O(p(2q+p)) = O(pq+p^2) = O(\max(pq,p^2))$ if the number of data stores in the processed DFD is *p* and the number of processes is *q*; in the worst case each data store is connected to each process and there is a relationship between each data store. As it is reasonable to assume that $q > p$ we can say that the complexity in the worst case is O(pq).

The total number of unadjusted function points is the better the more *normalized* the data stores are; Gane and Sarson (1979) propose that the data base schema should be formed by dividing each data store in a DFD by normalizing rules.

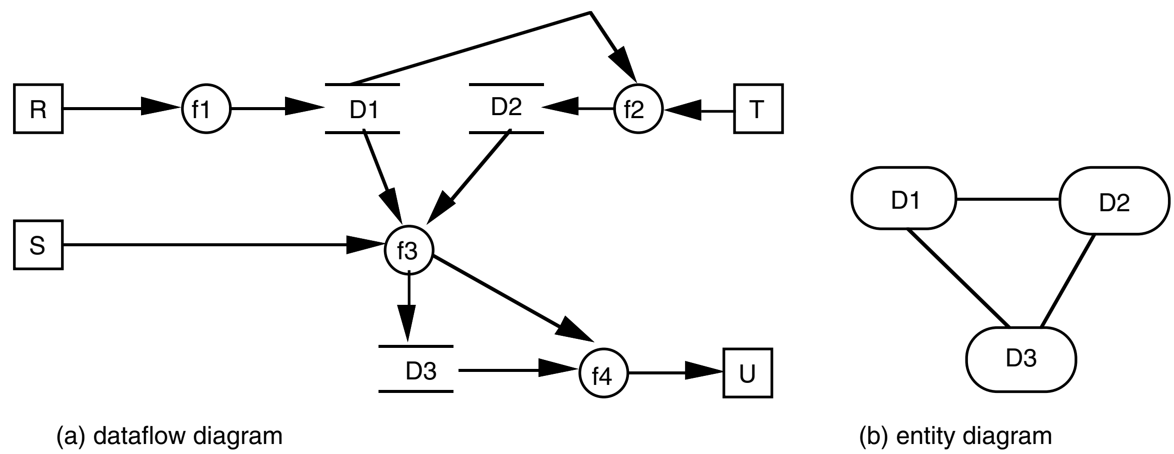


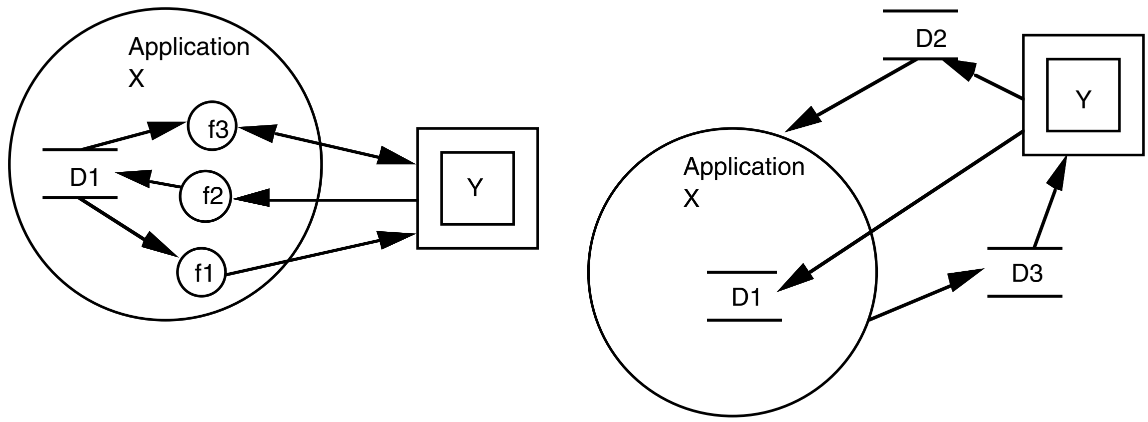
Figure 4. Logical internal files and their relationships

In Figure 4(a) we have three transactions. If we know the corresponding entity model of Figure 4(b) we can conclude that there is a relationship between data stores D1 and D2, D1 and D3, and D2 and D3. If we have only one key in each data store and the number of data items referenced is e.g. at most 19, the weight for each of these logical internal files is seven; each file has only two logical relationships.

4.4. External Interface File Types

The basic symbols of a DFD are not able to distinguish an *external application*, with which the information is changed, from other terminators. Of course we could mark an external application on a repository only but why not give a special graphical symbol for it? If we want to describe the external application explicitly we can use, for instance, *double rectangles* for these application terminators. So, we could represent the interface files in a DFD as in Figure 5.

Figure 5(a) describes *transaction files* that update logical internal files of application X (input) or transfer update information to application Y (output and inquiry). Transaction files are described by *dataflows*, not by data stores. It is not possible to describe explicitly the *conversion* of transaction files in a DFD. It is, however, possible to compare input and output dataflows.



(a) Interfaces with transactions (b) Interfaces with a master file

Figure 5. Principles of interface files

If the input flow of the process which writes the transaction file differs from the output flow (the dataflows have different names or the names of the decomposed dataflows do not match the names of the components of the composed dataflow) then we may assume that a conversion was made and we may count one interface type in addition to one output type. Correspondingly, if the output flow of the process which reads the transaction file differs from the input flow we may assume that a conversion was made and we may count one interface type in addition to one input type. For the input part of an inquiry we do not check conversion if the inquiry flow is not composed into separate dataflows at the lowest level of the DFD.

To record all the transaction types we must obtain the number of types in a dataflow. We describe different transaction types by a selection of alternatives, for example t=[a|b|c] where a, b, and c are transaction types. When we count the number of unadjusted function points we do not count the number of relationships to other files but the *number of logical record formats* of the transaction file. Here we have two possibilities: we may interpret each transaction type as a record format or we may interpret each transaction type as a separate transaction. Of course the interpretation has influence to the number of unadjusted function points. If we interpret each transaction type as a separate transaction (this is recommended by Dreger (1989)) we have to divide the transaction dataflow into separate dataflows by *decomposing* the original transaction process into separate processes corresponding to the transaction types. This decomposition may also improve the quality and reliability of the system because we have to specify each subtransaction separately; when we automate the counting process we can test the completeness.

In Figure 5(b) we have described the different interpretations for *master files*. D1 is an internal file, D2 and D3 are *external files* (Yourdon (1989) uses a term external store). Data store D2 is *produced* by application Y and *used* by application X. Data store D3 is produced by application X and used by application Y. Here D2 would receive interface credit only and D3 would receive both file and interface credit although D3 is an external file. D3 is, however, created by application X and, based on Dreger (1989), it should receive also file credit. If the same external file appears more than once on the context diagram level, we recommend to count it only once as a file and as an interface (reusability). Data store D1 in Figure 5(b) is *shared* between applications X and Y; both applications are performing maintenance as though D1 were its own internal file. Thus D1 is counted both as a file and as an interface. To indicate a shared file interface we need a *direct dataflow connection* between applications, i.e. between the context diagram of our object system and the terminator that describes the external application. Each external file is understood as one logical file and to contain only one logical record format. The number of logical record formats of a shared file interface means the number of record formats of the logical files that are based on the key of the dataflow by which the shared file is referenced.

If all the transactions in Figure 5(a) require a conversion there are one output, one input, one inquiry, and three interfaces; therefore, the number of unadjusted function points is at least 26. If there is no conversions the number of unadjusted function points is at least 11. In Figure 5(b) we have an example of three master file interfaces, one of which is shared; the number of unadjusted function points for these interfaces is at least 15. In addition, both D1 and D3 produces at least seven points as a file.

Algorithm 5 can be used to automate finding and weighting external interface files based on DFD and ERD. We have not presented here the main procedure weights_of_interfaces that calls sequentially the procedures presented below.

Algorithm 5. Finding and weighting of external interface files.

procedure weights_of_interface_inputs(interface_input_list, input_weight_list,interface_file_weight_list):
-- interface_input_list imports the interface input dataflows
-- input_weight_list exports the weights of the interface input dataflows
-- interface_file_weight_list exports the weights of the transaction files
-- local variable logical_file_list contains all logical files referenced
-- local variable n counts the number of internal logical files
-- local variable s counts the number of data items (size) of the input dataflow f
-- local variable w counts the weight of the input dataflow f
for all m_1 interface input flows f=(e,p) with external application e and process p in interface_input_list **do**
    logical_file_list : = nil;
    find_logical_files_of_inputs(p,logical_file_list);
    n : = lenght(logical_file_list); s : = |f|; w : = input_weight(n,s);
    input_weight_list : = input_weight_list ∪ w;
    if f is converted **then**
        n : = number of logical record formats of f;
        w : = interface_file_weight(n,s);
        interface_file_weight_list : = interface_file_weight_list ∪ w.

procedure weights_of_interface_outputs(interface_output_list, output_weight_list,interface_file_weight_list):
-- interface_output_list imports the interface output dataflows
-- output_weight_list exports the weights of the interface output dataflows
-- interface_file_weight_list exports the weights of the transaction files
-- local variable logical_file_list contains all logical files referenced
-- local variable n counts the number of internal logical files
-- local variable s counts the number of data items of the output dataflow f
-- local variable w counts the weight of the output dataflow f
for all m_2 interface output flows f=(e,p) with external application e and process p in interface_output_list **do**

```
        logical_file_list : = nil;
        find_logical_files_of_outputs(p,logical_file_list);
        n : = lenght(logical_file_list); s : = |f|; w : = output_weight(n,s);
        output_weight_list : = output_weight_list ∪ w;
        if f is converted then
                n : = number of logical record formats of f;
                w : = interface_file_weight(n,s);
                interface_file_weight_list : = interface_file_weight_list ∪ w.

procedure weights_of_interface_inquiries(interface_inquiry_list,
inquiry_weight_list,interface_file_weight_list):
-- interface_inquiry_list imports the inquiry dataflows
-- inquiry_weight_list exports the weights of the inquiry dataflows
-- interface_file_weight_list exports the weights of the transaction files
-- local variable logical_file_list contains all logical files referenced
-- local variable w counts the weight of the inquiry flow f = <fi,fo>
-- local variable ni counts the number of internal logical files for input part fi
-- local variable si counts the number of data items (size) of the input part fi
-- local variable wi counts the weight of the input part fi
-- local variable no counts the number of internal logical files for output part fo
-- local variable so counts the number of data items (size) of the output part fo
-- local variable wo counts the weight of the output part fo
for all m3 interface inquiry flows f = <fi,fo> in interface_inquiry_list do
        select input part fi = (t,p) from inquiry flow f;
        logical_file_list : = nil;
        find_logical_files_of_inputs (p, logical_file_list);
        ni : = length(logical_file_list); si : = |fi|; wi : = input_weight(ni, si);
        select output part fo = (p,t) from inquiry flow f;
        logical_file_list : = nil;
        find_logical_files_of_outputs (p, logical_file_list);
        no : = length(logical_file_list); so : = |fo|; wo : = output_weight(no, so);
        w : = max(wi, wo);
        inquiry_weight_list : = inquiry_weight_list ∪ w.
        if fo is converted then
                n : = number of logical record formats of fo;
                w : = interface_file_weight(n,so);
                interface_file_weight_list : = interface_file_weight_list ∪ w.
```

```
procedure weights_of_interface_external_files(external_file_list,
interface_file_weight_list,file_weight_list):
-- external_file_list imports the interface external data stores
-- interface_file_weight_list exports the weights of the interface files
-- file_weight_list exports the weights of the internal logical files
-- local variable r counts the number of relationships for the external file
-- local variable s counts the number of data items (size) of the external file
-- local variable w counts the weight of the external file
for all p1 external data stores d in external_file_list do
        r := number of the relationships of d counted from ER-diagram;
        s := |d|;
        w := interface_file_weight(r, s);
        interface_file_weight_list := interface_file_weight_list ∪ w.
        if d is sent to external application e then
                w := file_weight(n,s);
                file_weight_list := file_weight_list ∪ w.

procedure weights_of_interface_shared_files(shared_file_list,
interface_file_weight_list):
-- shared_file_list imports the interface shared data stores
-- interface_file_weight_list exports the weights of the interface files
-- local variable r counts the number of relationships for a shared file d
-- local variable l counts the number of logical record formats for the logical
-- file determined by key(f)
-- local variable n counts the maximum of r and l
-- local variable s counts the number of data items (size) of the shared file d
-- local variable w counts the weight of the logical file f
-- <d,key(f)> means those logical files of data store d with key f
for all p2 shared data stores d in shared_file_list connected with dataflow f to
external application e do
        r := number of the relationships of d counted from ER-diagram;
        l := number of logical record formats of <d,key(f)>;
        n := max(r,l);
        s := |d|;
        w := interface_file_weight(n, s);
        interface_file_weight_list := interface_file_weight_list ∪ w.
```

The complexity of Algorithm 5 in the worst case is $O(m_1q+m_2q+2m_3q+p_1p+p_2p) = O(q(m_1+m_2+m_3)+p(p_1+p_2))$ when the number of data stores in the DFD is p, the number of processes is q, the number of interface inputs is m_1, the number of interface outputs is m_2, the number of interface inquiries is m_3, the number of external data stores is p_1, and the number of shared data stores is p_2.

5. SUMMARY AND FUTURE PLANS

We believe that Function Point Analysis together with a description produced by Structured Analysis methodology gives a *working practice* that possibly helps to understand better the modelling process. The algorithms presented here enable us to *automate*, during analysis phase, estimation of the size of the final software. Currently we are building a prototype system with DFD and ERD interface to analyze the specification level descriptions using measurement methods like the FPA (Albrect 1979) reported in this paper, Bang metrics (DeMarco 1982), and MARK II metric (Symons 1988).

REFERENCES

Albrecht A. J. 1979. Measuring Application Development Productivity. *Proceedings, Joint SHARE/GUIDE/IBM Application development symposium*, October, 83-92.

Albrecht A. J. & Gaffney J. E. 1983. Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering*, SE-9(6), November, 639-648.

Banker & al. 1990. *Managing Development Productivity of the Computer Aided Software Engineering (CASE) Process with Dynamic Life Cycle Trajectory Metrics.* Center for Research on Information Systems, New York University, Information Systems Department, Leonard N. Stern School of Business.

Cowan J. B. 1990. Quality Assurance Potential of Analyst/Designer workbenches. *Information and Software Technology*, 32(1), January/February, 46-52.

DeMarco T. 1978. *Structured Analysis and Systems Specification.* Yourdon Press.

DeMarco T. 1982. *Controlling Software Projects*. Yourdon Press.

Development Centre Productivity Measurement Guide 1984. IBM.

Dreger J. B. 1989. *Function point analysis*. Prentice-Hall.

Gane C. P. and Sarson T. 1979. *Structured Systems Analysis: Tools and Techniques*. Prentice-Hall.

Grady R.B. 1990. Work-product Analysis: The Philosopher's Stone of Software. *IEEE Software*, March, 26-34.

Halstead M. H. 1977. *Elements of Software Science*. Elsevier.

Jackson M. 1983. *Systems Development*. Prentice-Hall.

Lundeberg M. & al. 1981. *Information Systems Development - A Systematic Approach*. Prentice-Hall.

McCabe T. J. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4), 308-320.

Rask R. 1990. The Management and Evaluation of Systems Development. *Abstracts of the Seminar on Design Automation of Computer Hardware and Software* (Ed. Iiro Hartimo), Helsinki University of Technology, Faculty of Electrical Engineering, Laboratory of Signal Processing and Computer Technology.

Rask R. 1991. Counting Function Points from SA Descriptions. *The Papers of The Third Annual Oregon Workshop on Software Metrics* (Ed. W. Harrison), Silverton, Oregon, March 17-19.

Shoval P. 1988. ADISSA: Architectual Design of Information Systems Based on Structured Analysis. *Information Systems*, 13(2), 193-210.

Shoval P. and Manor O. 1990. Software Engineering Tools Supporting ADISSA Methodology for Systems Analysis and Design. *Information and Software Technology*, 32(5), 357-369.

Software Tools 1990. The Practical Use of Software Metrics. Wembley Conference Centre, London, 12-14 June, Blenheim Online.

Symons R. C. 1988. Function Point Analysis: Difficulties and Improvements. *IEEE Transactions on Software Engineering*, 14(1), January, 2-11.

Yourdon E. 1989. *Modern Structured Analysis*. Prentice-Hall.

APPENDIX - Detail and summary sheets of FPA

Detail Sheet: Outputs

	1-5 data items referenced	6-19 data items referenced	20 or more data items referenced
0 or 1 file(s) referenced	Simple (4)	Simple (4)	Average (5)
2 or 3 files referenced	Simple (4)	Average (5)	Complex (7)
4 or more files referenced	Average (5)	Complex (7)	Complex (7)

Detail Sheet: Inputs

	1-4 data items referenced	5-15 data items referenced	16 or more data items referenced
0 or 1 file(s) referenced	Simple (3)	Simple (3)	Average (4)
2 files referenced	Simple (3)	Average (4)	Complex (6)
3 or more files referenced	Average (4)	Complex (6)	Complex (6)

Detail Sheet: Inquiries

Output Part:	1-5 data items referenced	6-19 data items referenced	20 or more data items referenced
0 or 1 file(s) referenced	Simple (4)	Simple (4)	Average (5)
2 or 3 files referenced	Simple (4)	Average (5)	Complex (7)
4 or more files referenced	Average (5)	Complex (7)	Complex (7)

Input Part:	1-4 data items referenced	5-15 data items referenced	16 or more data items referenced
0 or 1 file(s) referenced	Simple (3)	Simple (3)	Average (4)
2 files referenced	Simple (3)	Average (4)	Complex (6)
3 or more files referenced	Average (4)	Complex (6)	Complex (6)

Detail Sheet: Files

	1-19 data items referenced	20-50 data items referenced	51 or more data items referenced
1 logical record format/relationship	Simple (7)	Simple (7)	Average (10)
2-5 logical record formats/relationships	Simple (7)	Average (10)	Complex (15)
6 or more logical record formats/relationships	Average (10)	Complex (15)	Complex (15)

Detail Sheet: Interfaces

	1-19 data items referenced	20-50 data items referenced	51 or more data items referenced
1 logical record format/relationship	Simple (5)	Simple (5)	Average (7)
2-5 logical record formats/relationships	Simple (5)	Average (7)	Complex (10)
6 or more logical record formats/relationships	Average (7)	Complex (10)	Complex (10)

Summary Sheet

Business Function	Number	Complexity		Factor		Totals: Line Totals	Totals: Group Totals
OUTPUTS	O_4	Simple	*	4	=	O_4 * 4 = A	
	O_5	Average	*	5	=	O_5 * 5 = B	
	O_7	Complex	*	7	=	O_7 * 7 = C	
TOTAL							A + B + C
INQUIRIES	Q_4	Simple	*	4	=	Q_4 * 4 = D	
	Q_5	Average	*	5	=	Q_5 * 5 = E	
	Q_6	Complex	*	6	=	Q_6 * 6 = F	
	Q_7	Complex	*	7	=	Q_7 * 7 = G	
TOTAL							D + E + F + G
INPUTS	I_3	Simple	*	3	=	I_3 * 3 = H	
	I_4	Average	*	4	=	I_4 * 4 = I	
	I_6	Complex	*	6	=	I_6 * 6 = J	
TOTAL							H + I + J
FILES	F_7	Simple	*	7	=	F_7 * 7 = K	
	F_{10}	Average	*	10	=	F_{10} * 10 = L	
	F_{15}	Complex	*	15	=	F_{15} * 15 = M	
TOTAL							K + L + M
INTERFACES	U_5	Simple	*	5	=	U_5 * 5 = N	
	U_7	Average	*	7	=	U_7 * 7 = O	
	U_{10}	Complex	*	10	=	U_{10} * 10= P	
TOTAL							N + O + P

TOTAL UNADJUSTED FUNCTION POINTS = A+B+C+D+E+F+G+H+I+J+K+L+M+N+O+P