RAIMO RASK

AUTOMATING ESTIMATION OF SOFTWARE SIZE DURING THE REQUIREMENTS SPECIFICATION PHASE - Application of Albrecht's Function Point Analysis within structured methods

Joensuu 1992

Automating estimation of software size during the requirements specification phase: application of Albrecht's Function Point Analysis within structured methods. - Joensuu: University of Joensuu, 1992. 128 p. (Joensuun yliopiston luonnontieteellisiä julkaisuja 28 = University of Joensuu. Publications in Sciences, 28. ISSN 0781-0342).

ISBN 951-708-099-9

KEYWORDS: CASE, Function point analysis, JSD, product metrics, SA, software engineering, software metrics, software size estimation, structured methods, transformation.

The purpose of this thesis is to demonstrate how to measure software size reliably as early as possible. The size estimates provide a basis for estimation of software costs. Algorithms are developed for calculating an unadjusted value of Albrecht's Function Point Analysis (FPA) from descriptions of Structured Analysis (SA) and Jackson's System Development (JSD). Another goal of this study is to develop a method-independent environment for software development by specifying transformation rules between different description techniques. In this study the examination is restricted to SA dataflow diagrams and JSD specification diagrams. Rules are given for transformation between these description techniques. The algorithms used are validated by carrying out a set of experiments. First, the FPA algorithms are verified in relation to SA specifications. Next, the FPA algorithms are verified in relation to JSD specifications. Last, it is shown that the algorithmic transformation from JSD to SA preserves the unadjusted function point value of FPA. The experiments use case material from the literature, random SA specifications, and test material produced by computer science students. Based on the validation, it is argued that FPA algorithms can be developed to SA and JSD specifications. It is also possible to transform one system description into another and then apply FPA to the resulting transformation.

Raimo Rask, Department of Computer Science, University of Joensuu, P.O. Box 111, SF-80101 Joensuu, Finland. E-mail: rask@cs.joensuu.fi.

PREFACE

This thesis is the result of research carried out during 1988-1992 as a teacher and researcher at the Department of Computer Science, University of Joensuu. The research was partly funded by the University of Joensuu and the Finnish Cultural Foundation, Regional Foundation of North Karelia.

My interest in software metrics goes back to an issue of IEEE Transaction on Software Engineering (SE-5(2), March 1979), which was a commemorative issue in honor of Dr. Maurice H. Halstead. The copy of this issue was part of the material I received from Professor Aarni Perko (University of Jyväskylä) when I started my licentiate studies. The papers in that issue started me collecting and reading software metrics systematically. At the end of 1985 I finished my licentiate thesis on the topic "Measurement of internal and external complexity of a software module".

In 1986 I started as a senior lecturer (in systems development) at the University of Joensuu. This obliged me to shift my interest into systems level descriptions and metrics and to think more deeply about the process of software development. Discussions with Professor Kalle Lyytinen in late 1987 strengthened my opinion of the necessity to focus on the early phases of software development in metrics development. Dr. Rade Adamov's (University of Zürich) visit also had a positive impact on my starting doctoral studies at the end of 1988. The final inspiration, however, came from Associate professor Jorma Sajaniemi.

I have been aided in my work by many teachers, researchers, and students, to whom I owe my thanks. First, I am greatly indebted to Kalle Lyytinen and Jorma Sajaniemi for their advice and encouragement during this research. I also thank Martti Penttonen, and the referees Jon Turner (New York University) and Reijo Sulonen (Helsinki University of Technology) for evaluating my research.

The work of several students who helped to implement my ideas has been important. In particular, I thank Petteri Laamanen for his excellent programming skills, which were needed to implement a CASE tool that we have named JoUCASE, Joensuu University CASE. This system has been the basis for various measurements and data analysis reported in this thesis.

Finally, to my wife Liisa, our children Hermanni and Tiuku, and our dog Tiltu I would like to express my respect and appreciation for their patience and friendship during these years.

TABLE OF CONTENTS:

List of Abbreviations 8
List of Algorithms 8
List of Equations 8
List of Figures 9
List of Tables 10

1. INTRODUCTION 11

1.1. Software Measurement and Metrics 11
1.2. Albrecht's Function Point Analysis 12
1.2.1. Finding and Weighting of FPA Functions 13
1.2.2. Minimum Structures for Calculating TUFPV 16
1.3. Approach Taken 18
1.3.1. General Problem Formulation 18
1.3.2. Thesis Outline 19

2. APPLICATION OF ALBRECHT'S FPA TO SA SPECIFICATIONS 21

2.1. Components of Dataflow and Entity-Relationship Diagrams 21
2.2. Identification and Weighting of User Functions 25
2.2.1. External Input and Output Types 26
2.2.2. External Inquiry Types 31
2.2.3. Logical Internal File Types 34
2.2.4. External Interface File Types 37
2.3. Summary 44

3. APPLICATION OF ALBRECHT'S FPA TO JSD SPECIFICATIONS 46

3.1. Components of the JSD Specification Diagram 46
3.2. Identification and Weighting of User Functions 49
3.2.1. External Input and Output Types 49
3.2.2. External Inquiry Types 54
3.2.3. Logical Internal File Types 57
3.2.4. External Interface File Types 60
3.3. Summary 67

4. METHODICAL EQUIVALENCE 69

4.1. Transformation between JSD and SA 69
4.1.1. Transformation from JSD to SA 70
4.1.2. Transformation from SA to JSD 75
4.2. Summary 76

5. VALIDATION OF THE FPA ALGORITHMS AND THE TRANSFORMATION RULES 77

5.1. Validation of FPA Algorithms for SA Descriptions 78
    5.1.1. A Preliminary Test Using a Well-known Test Material 78
    5.1.2. An Extended Test Using Random SA Test Material 83
        5.1.2.1. Comparison of Algorithms 2.1-2.5 with Function Bang 76
        5.1.2.2. Comparison of Algorithms 2.1-2.5 with Mark II 83
5.2. Validation of FPA Algorithms for JSD Descriptions 95
    5.2.1. Application of Algorithms 3.1-3.4 to JSD after Transformation from SA 95
    5.2.2. Application of Algorithms 3.1-3.4 to JSD before Transformation to SA 96
5.3. Properties of Algorithmic Transformation 101
5.4. Summary of the Results 102

6. SUMMARY AND CONCLUSIONS 104

REFERENCES 106

APPENDICES

A - Model for Measuring and Controlling Software Productivity 110
B - FPA Detail and Summary Sheets 113
C - Software System 115
D - Example of a Dataflow Diagram 116
E - Example of FPA Data Gathering from SA 117
F - Example of a JSD Specification Diagram 119
G - Example of FPA Data Gathering from JSD 120
H - Notation for the Algorithms 122
I - DeMarco's Function Bang Metric 124
J - Mark II Function Points 125
K - Summary of Potential Size Metrics 127
L - Example of an ER-diagram 128

LIST OF ABBREVIATIONS:

CASE = Computer-Assisted Software (System) Engineering
CFP = Corrected count of Functional Primitives
DFD = Dataflow Diagram
ERD = Entity-Relationship Diagram
FFPV = Final Function Point Value
FP = Function Point
FPI = Function Point Index
FPA = Function Point Analysis
JSD = Jackson System Development
LOC = Lines Of Code
SA = Structured Analysis
TCA = Technical Complexity Adjustment
TUFPV = Total Unadjusted Function Point Value
UFP = Unadjusted Function Points

LIST OF ALGORITHMS:

Algorithm 2.1. Finding and weighting of inputs in SA 28
Algorithm 2.2. Finding and weighting of outputs in SA 30
Algorithm 2.3. Finding and weighting of inquiries in SA 33
Algorithm 2.4. Finding and weighting of logical internal files in SA 35
Algorithm 3.5. Finding and weighting of external interface files in SA 40
Algorithm 3.1. Finding and weighting of inputs in JSD 51
Algorithm 3.2. Finding and weighting of outputs in JSD 53
Algorithm 3.3. Finding and weighting of inquiries in JSD 56
Algorithm 3.4. Finding and weighting of logical internal files in JSD 59
Algorithm 3.5. Finding and weighting of external interface files in JSD 63
Algorithm 4.1. Transformation from JSD to SA 74

LIST OF EQUATIONS:

(1.1) $y = f(x_1,x_2,...,x_n)$ 12
(1.2) $TUFPV = \sum F_i * w_i$ 15
(1.3) $FFPV = TUFPV * (0.65 + TDI/100)$ 16
(5.1) $Z = \sqrt{-2 \ln U_1} \cos(2\pi U_2)$ 86
(A.1) $\text{Productivity} = \dfrac{\text{Outputs produced by the process}}{\text{Inputs consumed by the process}}$ 110
(A.2) $C = K + M * P$ 112
(A.3) $C = K + M * P^E$ 112
(A.4) $m(t) = 2 * C * a * t * \exp(-a * t^2)$ 112
(A.5) $a = 1/2t_d^2$ 112
(I.1) $CFP = \sum w_i * CFPI_i$ 124
(I.2) $CFPI_i = (TC_i * \log_2 (TC_i))/2$ 124
(J.1) $UFP = N_I W_I + N_E W_E + N_O W_O$ 125
(J.2) $UFP1 = 0.44N_I + 1.67N_E + 0.38N_O$ 125
(J.3) $UFP2 = 0.58N_I + 1.66N_E + 0.26N_O$ 125
(J.4) $TCA = 0.65 + C * TDI$ 126
(J.5) $FPI = UFP * TCA$ 126

LIST OF FIGURES:

Figure 1.1. Connections between applications 15
Figure 1.2. Minimal structures for weighting inputs, outputs, inquiries, and files 17
Figure 1.3. Minimal structures for weighting external interface files 17
Figure 1.4. Structure of the thesis 19
Figure 2.1. Basic symbols of the dataflow diagram 22
Figure 2.2. Composite input dataflow 22
Figure 2.3. Composite output dataflow 23
Figure 2.4. A context diagram 23
Figure 2.5. Components of an ERD 24
Figure 2.6. Data dictionary notation 24
Figure 2.7. Two user transactions 25
Figure 2.8. Counting diverging input dataflows 27
Figure 2.9. Input and output transactions 31
Figure 2.10. A typical inquiry as part of an update 32
Figure 2.11. Logical internal files and their relationships 37
Figure 2.12. Principles of an interface file type 37
Figure 2.13. Components for counting unadjusted function points from SA 44
Figure 3.1. Basic symbols of a JSD specification diagram 46
Figure 3.2. Example of the JSD specification diagram 47
Figure 3.3. Example of a merged dataflow 47
Figure 3.4. Example of a JSD structure diagram 48
Figure 3.5. Example of JSD inputs and outputs 50
Figure 3.6. Inquiry and interactive functions 55
Figure 3.7. Proposal for an inquiry symbol 55
Figure 3.8. Logical internal files in a specification diagram 60
Figure 3.9. Principles of interfaces 61
Figure 3.10. Components for counting unadjusted function points in JSD 68
Figure 4.1. Example of a dataflow diagram after transformation 69
Figure 4.2. Transformation of a JSD specification to an SA dataflow diagram 70
Figure 4.3. Basic structures of external input/output transformations for a model process 71
Figure 4.4. Transformation of model processes connected with one dataflow 71
Figure 4.5. Feedback between model processes 72
Figure 4.6. Connection of a function process to a model process 72
Figure 4.7. Merging the dataflows 73
Figure 4.8. Generalization of Figure 4.1 73
Figure 4.9. Example of a dataflow diagram 75
Figure 4.10. Two interpretations of the dataflow diagram in Figure 4.9 75
Figure 5.1. Data tokens around a process 80
Figure 5.2. Transactions in counting Mark II 81
Figure 5.3. Data generation and measurement process 84
Figure 5.4. Two examples of test dataflow diagrams 85
Figure A.1. Estimation of the software size, cost and productivity 111
Figure A.2. Principles of a control and estimation system 112
Figure C.1. Model for a software system 115

LIST OF TABLES:

Table 1.1.	Summary of Function Point Distribution	13
Table 2.1.	Procedures, complexity, and calling structure of Algorithm 2.1	27
Table 2.2.	Procedures, complexity, and calling structure of Algorithm 2.2	29
Table 2.3.	Procedures, complexity, and calling structure of Algorithm 2.3	33
Table 2.4.	Procedures, complexity, and calling structure of Algorithm 2.4	35
Table 2.5.	Procedures, complexity, and calling structure of Algorithm 2.5	40
Table 3.1.	Procedures, complexity, and calling structure of Algorithm 3.1	51
Table 3.2.	Procedures, complexity, and calling structure of Algorithm 3.2	53
Table 3.3.	Procedures, complexity, and calling structure of Algorithm 3.3	56
Table 3.4.	Procedures, complexity, and calling structure of Algorithm 3.4	58
Table 3.5.	Procedures, complexity, and calling structure of Algorithm 3.5	63
Table 5.1.	Properties of the object subsystems	79
Table 5.2.	Summary of SA measurements for a case taken from the literature	80
Table 5.3.	Correlation between the metrics	82
Table 5.4.	Internal and external complexity indicated by Mark II	83
Table 5.5.	Properties of the dataflow diagrams	86
Table 5.6.	Mean correlations per diagram type: TUFPV(SA) vs. Function Bang	87
Table 5.7.	Total correlations per distribution: TUFPV(SA) vs. Function Bang	88
Table 5.8.	Results of Regression Analysis: TUFPV(SA) vs. Function Bang	89
Table 5.9.	Mean correlations per diagram type: TUFPV(SA) vs. unweighted Mark II (UFP)	91
Table 5.10.	Total correlations per distribution: TUFPV(SA) vs. unweighted Mark II (UFP)	91
Table 5.11.	Results of Regression Analysis: TUFPV(SA) vs. unweighted Mark II (UFP)	92
Table 5.12.	Internal complexity indicated by Mark II	93
Table 5.13.	Mean correlations per diagram type: TUFPV(SA) vs. Mark II with weights	93
Table 5.14.	Total correlations per distribution: TUFPV(SA) vs. Mark II with weights	93
Table 5.15.	Results of regression analysis: TUFPV(SA) vs. UFP1	94
Table 5.15.	Results of regression analysis: TUFPV(SA) vs. UFP2	94
Table 5.17.	Summary of JSD measurements	96
Table 5.18.	FPA values from JSD laboratory exercises: TUFPV(JSD-1)	98
Table 5.19.	FPA values from JSD laboratory exercises: TUFPV(JSD-2) & TUFPV(JSD-3)	99
Table 5.20.	Values from SA exercises: TUFPV(SA)	99
Table 5.21.	Correlation between TUFPV(SA) and TUFPV(JSD): independent descriptions	100
Table 5.22.	Values from JSD to SA transformations: TUFPV(SA)	102
Table 5.23.	Correlation between TUFPV(SA) and TUFPV(JSD): direct transformation	102
Table K.1.	Summary of potential code-level size metrics	127
Table K.2.	Summary of potential design-level size metrics	127
Table K.3.	Summary of potential specification-level size metrics	127

1. INTRODUCTION

As a part of computer-based information systems, software has become important to organizations and will become more so in the future. Software is obtained either by developing it from the very beginning or by purchasing a package. In either case, in order to identify requirements, and to configure, test and document the application software, a development project needs to be formed.

The development of application software is made easier, or more productive, by using special methods and tools, and by taking into account organizational and human factors.[1] From the project management point of view, it must be possible to control the productivity. To do this we need software metrics for the process of development as well as the product of development. Of these metrics, one of the most important is the complexity or size of particular software because this affects the resources that must be allocated to the project, the establishment of a realistic schedule, and replanning of the project based on the progress achieved.[2]

1.1. Software Measurement and Metrics

Metrics can be defined as scales or units in which a quantitative attribute can be measured (Hunter 1990). A *software metric* is a measure of the extent or degree to which a software product or process possesses and exhibits a given attribute. Thus, metrics can be divided into two functional categories (Conte et al. 1986, IEEE Standard 982.2 1988, Hunter 1990, Kitchenham 1990a): process metrics and product metrics.

Process metrics quantify attributes of the development process and of the development environment. They are often concerned with resources. *Product metrics* are measures of the software product. They can be based on requirements, design, documentation, or source code. It may, however, be difficult to classify certain metrics as either process- or product-related.

A model for the software development process from the view of measurement may be represented by the general form (Conte et al. 1986):

---

[1] A good framework for the study of software productivity is given by Nunamaker & Chen (1989).

[2] This thesis concentrates on size measurement, but in Appendix A is given a wider model for measuring and controlling productivity.

(1.1) $y = f(x_1, x_2, \dots, x_n)$,

where the *dependent* variable y is the process metric of interest, such as the total effort or development cost. This is a function of the *independent* variables $x_1, x_2, \dots, x_n$, which can be either product- or process-related.

The metrics used must be validated and automated. The application of software metrics requires a carefully planned model or framework for controlling the software development. We must know why and what we measure and how we measure it. We must also know when the measure is applied[1].

This study concentrates on Albrecht's (1979) estimation method - Function Point Analysis - which sizes an application by counting and weighting all end-user business functions for cost estimation purposes.[2] Software costs, correspondingly, are understood as the effort required to implement the software product.

1.2 Albrecht's Function Point Analysis

Albrecht (1979) has developed a measurement method called *Function Point Analysis* (FPA). FPA sizes an application from the perspective of an end-user. A *function point* is defined as one end-user business function. 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 calculate an *unadjusted value* for the function points, the user functions are first identified, classified and weighted. The 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.

---

[1] Appendix A contains a short description of the measurement process applicable to all life-cycle phases.

[2] Appendix K summarizes potential "size" metrics for all life-cycle phases.

Table 1.1 contains the Function Point distribution of 13 applications. This distribution may be used as a general guideline when counting function point values.

Table 1.1. Summary of Function Point Distribution (Dreger 1989).

		Per Cent Contribution				
Application	Function Points	Output	Inquiries	Input	Files	Interfaces
Meat Processing	654	30	7	28	35	0
Share Purchase	166	18	8	37	28	9
Corporate Accounting	2047	18	4	34	45	0
Job Costing	485	18	2	26	52	2
Utility Rates	1777	28	6	37	30	0
Work in Process	528	34	0	26	32	9
Purchased Material	108	27	0	26	29	19
Retail Parts	662	17	22	21	40	0
Wholesale Distribution	714	23	20	18	39	0
Message Switching	129	13	22	49	16	0
Teller System	604	28	12	39	21	0
Money Transfer	105	55	0	18	7	20
Library Issue	186	48	0	31	21	0
WEIGHTED AVERAGE	628	24	8	31	36	1

1.2.1. Finding and weighting of FPA functions

An *external input* is a unique user data or control input that enters the application boundary and also updates (adds to, changes, or deletes from) a logical internal file (Dreger 1989). An external input may also trigger an output which is not a part of an external inquiry even though the input does not update any logical internal file. An input is considered unique if it has a different format or has the same format as another input but requires different processing logic. For example, transactions such as add, delete, or change may have exactly the same screen format, but they should be counted as unique inputs if they require different processing logic (Albrecht 1979). The complexity classification of input functions depends on two things: the number of files referenced and the number of data items referenced.

An *external output* is a unique user data or a control output which is procedurally generated and leaves the application boundary. An output is considered unique if it has a different format or has the same format as another output but requires a different processing logic. According to Dreger (1989), each report format, total level, detail level, and different processing period needs to be counted separately. The complexity classification of output functions depends on the same things as the classification of input types: the number of files referenced and the number of data items referenced.

External inquiries include each unique input/output combination where the on-line input causes and generates an immediate on-line output from a data base or a master file (Dreger 1989). The inquiry performs no updates[1]; data is not entered, except for control purposes, and therefore only transaction logs are altered (Albrecht 1979). An inquiry is considered unique if it has a format different from other inquiries either in its input or in its output or if it has the same format, both in input and in output, as another inquiry but requires a different processing logic in either. Both parts are classified separately, and the complexity of an inquiry function is the greatest of the complexities of its input and output parts (Dreger 1989).[2]

A *logical internal file* refers to a major logical group of user data or control information maintained within the application boundary (Albrecht 1979, Dreger 1989). According to Dreger (1989), each key or data base view is counted as one logical internal file. For example, a customer file requiring a separate index file because of the access method would be considered one logical file, not two. An alphabetical index file to aid in establishing customer identity, however, would be counted as a separate file (Albrecht 1979). The complexity classification of each file depends on two things: the number of data items actually required by the application and either the number of record formats within the file or the number of logical relationships this file participates in to meet application requirements.

External interface files consist of major logical files or similar logical groups of user-approved data or control information within the application boundary that is sent to, shared with, or received from another application (Albrecht & Gaffney 1983, Dreger 1989). Each interface file must also be a logical internal file of the application itself, to some of its external applications, or both - or it must be a transaction file. Complexity classification principles for the external interfaces are the same as for logical internal files, although a transaction file is not a file per se.

Figure 1.1 summarizes the possible connections between two applications X and Y (Dreger 1989). *Interfaces* can represent any of the following three situations:[3]

---

[1] Albrecht and Gaffney (1983) say that an external inquiry type is entered only to direct the search, and no update of logical internal file types *should* occur.

[2] Inquiry weights reported by Albrecht and Gaffney (1983) and the IBM Development Centre Productivity Measurement Guide (1984) are not quite equal to those reported by Dreger (1989). In this thesis Dreger's approach has been selected.

[3] In the following chapters the terms *passed files* and *shared files* are used.

1. Data or control information is *passed* from file X to file Y. X would receive both file and interface credit, and Y would receive interface credit only. Y is not updated.
2. Data or control information is *passed* from file Y to file X. Y would receive both file and interface credit, and X would receive interface credit only. X is not updated.
3. Data or control information is *shared* with file X and file Y. X and Y would each receive both file and interface credit.

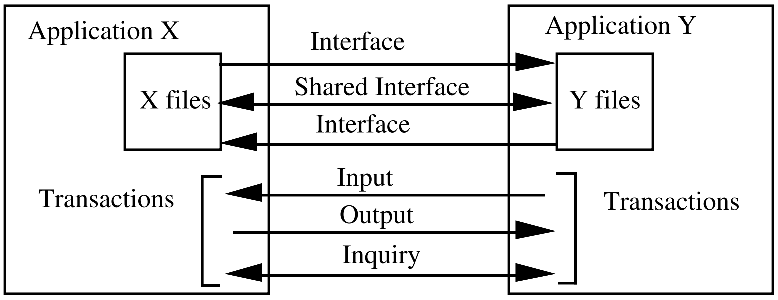


Figure 1.1. Connections between applications.

If applications are exchanging information by means of *transactions*, they will obtain input, output, and/or inquiry points - and possibly also interface points. A transaction file which is shared with another application may indicate an interface, depending on whether or not a *conversion* is required to match the record or file layout of the receiving application. If a transaction file is received from application Y, we would count one input for application X if no data conversion is required. Correspondingly, we would count one output if a transaction file is sent to application Y and no conversion is required. If a conversion is required within application X, we would count both input and interface, or output and interface for application X.

The *total unadjusted function point value* (TUFPV) is counted by totalling all the functions weighted by the corresponding complexity factors or weights:

(1.2) $\mathrm{TUFPV} = \sum F_i * w_i$.

The functions are grouped as shown in Appendix B. F_i in formula (1.2) is the number of functions in the *i*th group and w_i is the weight of the group.

The implementation of a logical system may vary depending on objectives and available hardware/software environment. Because of this, there are 14 environmental *adjusting fac-*

tors (Albrecht & Gaffney 1983) to count the *final function point value* (FFPV). This is derived using the following formula:

(1.3) $\mathrm{FFPV} = \mathrm{TUFPV} * (0.65 + \mathrm{TDI}/100)$

where TDI depicts total degree of influence determined by the sum of the adjusting factors of the hardware/software platform.

1.2.2. Minimum Structures for Calculating TUFPV

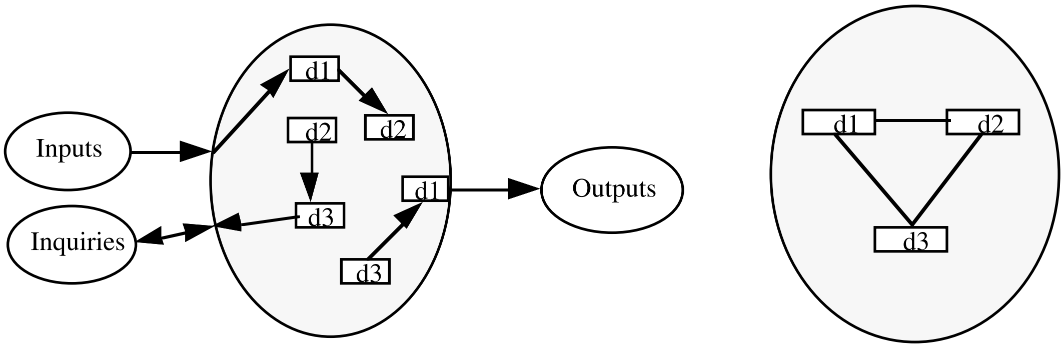
When we want to apply FPA to system descriptions, we have to understand the meaning of *transaction*[1]. This understanding is needed when we weight the *external data sets* (inputs, outputs, inquiries). In doing so we have to find those internal data sets (files) which are referenced by a specific transaction. This means that the description must include "facilities" which can bound the components inside the transaction, i.e. find the scope of a transaction. The most important of these facilities is the ability to *bound by direction*. This enables us to find the exact progress of an external event within a system description. If we had undirected edges in a specification, we could not bound the transactions, because we could not find the exact directed process paths which form the skeleton of a transaction.

The minimal structure satisfying the requirements to find the internal data sets referenced by external events can be seen as a *dynamic entity model*. This means that we have to find all the data sets inside the transaction, i.e. the data requirements in a form of *access paths* within a transaction. Figure 1.2 (a) shows the basic components needed to weight inputs, outputs, and inquiries using directed access paths to/from files *d1*, *d2*, and *d3*. The access paths correspond to the searching chains in dataflow and JSD specification diagrams.[2]

---

[1] We define the transactions within SA and JSD in chapters 2 and 3, corresponingly. See also Appendix C.

[2] See Algorithms 2.1, 2.2, 3.1, and 3.2 in chapters 2 and 3.



(a) Data structure for external data sets (b) Data structure for internal files

Figure 1.2. Minimal structures for weighting inputs, outputs, inquiries, and files.

The *logical internal files* of FPA all lie inside the system boundary. To weight the internal data sets, each logical file has to be identified and the relationships (undirected connections) have to be identified from one file to the other files. Thus the *static entity model* gives the minimal information for counting unadjusted function points for the logical internal files. Figure 1.2 (b) shows the data structure (with three files *d1*, *d2*, and *d3*) that must be described for FPA purposes. The static entity model may also contain *passed files* that are sent to or received from an external application (see also Figure 1.3).

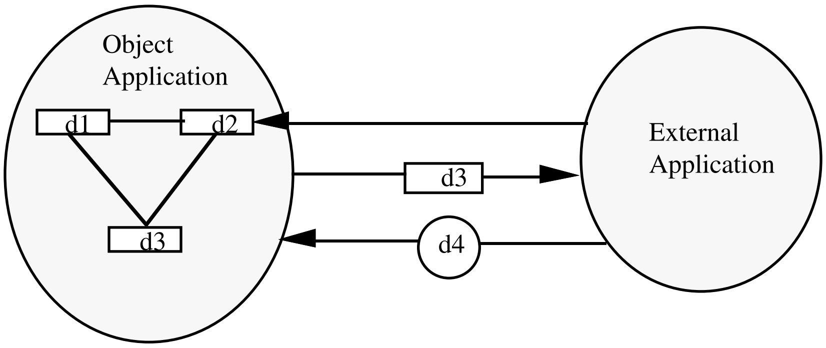


Figure 1.3. Minimal structures for weighting external interface files.

The *external interface file* can be a shared one, a passed file, or a transaction file. Figure 1.3 shows what possibilities exist between applications. File *d2* is a shared file, file *d3* is an exported passed file, and *d4* describes an imported transaction file. To count the unadjusted function points for external interfaces, the *directed external connections* between applications and *undirected connections between data sets* of a static entity model (for shared and passed files) have to be identified. Because the transaction files are either inputs, outputs, or inquiries, the structures presented in Figure 1.2(a) are also needed.

1.3. Approach Taken

The assumption of this thesis is, that Computer-Assisted Software Engineering (CASE) tools are needed in the software development process. It is also assumed that CASE-tools must support structured methods like Yourdon's (1989) Structured Analysis and Jackson's (1983) System Development method. Further, to automate the software measurement process, CASE-tools have to be expanded to contain the required measurement components. With this in mind, the goals and the outline of thesis are described below.

1.3.1. General Problem Formulation

The first step of this study is to show that it is possible to estimate software size automatically based on functional and static specifications of Yourdon's (1989) Structured Analysis (SA).[1] Here the concentration is on business applications, and technical and scientific applications are excluded. The measurement method used to measure SA descriptions is Albrecht's (1979) FPA[2].

The selection of FPA in this thesis is based on its popularity. For example, by analyzing data from a large survey of Dutch organizations, Heemstra and Kusters (1991) found that FPA was the most popular model used in cost estimation. FPA is also considered to be a more consistent a priori measure of software size than lines of code (Low & Jeffery 1990).

The second step is to show that the measurement and estimation can be independent of the development method. In addition to SA, the application of Albrecht's FPA will be extended to the specifications of Jackson's (1983) Systems Development (JSD).[3] To validate the methodical independence, transformation rules between the two development methods are also considered.

Until now, there have not been commonly accepted solutions for interpreting the functions of FPA for calculating TUFPV from a functional model of a system. The approach here is to seek a usable solution to this problem.[4]

---

[1] A short description of SA is given in section 2.1.

[2] FPA has also been extended to scientific and real-time systems (Reiffer 1990), but these features are excluded in this study.

[3] A short description of JSD is given in section 3.1.

[4] FPA is normally based on design specifications, e.g. a system flowchart and screen and form designs, but in this thesis FPA is used on the level of requirements specification. A CASE-tool, JoUCASE, has

Figure 1.4 shows the overall structure of this study. The final goal of this thesis is to prove the correctness of this structure. In Figure 1.4 the starting point is the SA and JSD specifications. Then FPA is applied to these specifications. This is indicated by measurement methods FPA(SA) and FPA(JSD). The metric value, TUFPV(SA) or TUFPV(JSD), from the measurement process corresponds to TUFPV of FPA. An additional aim is to show that there is a bi-directional function *f* which maps a JSD specification diagram, including process structure diagrams, to an SA dataflow diagram, and vice versa.

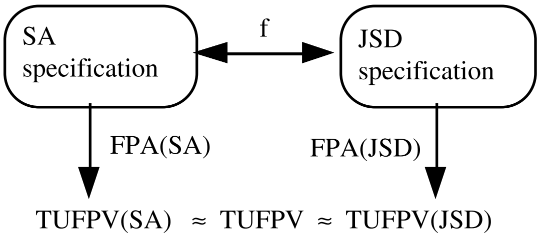


Figure 1.4. Structure of the thesis.

1.3.2. Thesis Outline

In Chapter 2 algorithms are developed to compute TUFPV(SA) automatically from SA specifications. When it is assumed that the specification is produced by a CASE-tool, the TUFPV(SA) computing can be done by giving the specification directly to the measurement component of the CASE-tool. These assumptions are also valid for JSD specifications in Chapter 3 where algorithms are developed to compute TUFPV(JSD) automatically from JSD specifications.

In Chapter 4 the equivalence between JSD and SA specifications will be derived. The purpose of the equivalence is to increase the credibility of the FPA algorithms in Chapters 2 and 3. Another, more general, purpose of the equivalence is to demonstrate the possibility of a method-independent software development environment by specifying transformation rules between different description techniques.

---

been developed to count TUFPV from SA specifications (Laamanen &Rask 1991). There are also other modified FPA automation approaches. LATURI-Tool performs a modified FPA size calculation from an association matrix (Nevalainen & Mäki 1992). The IEF CASE-tool is also reported to aspire to FPA automation (Mazzucco 1991).

Chapter 5 reports empirical studies conducted to test the validity of TUFPV computing from SA and JSD specifications. To test the validity of TUFPV(SA), two other well-known specification metrics are also used: DeMarco's (1982) Function Bang and Symons's (1988; 1991) Mark II metric[1]. The scope of the empirical studies consists of a literary case study, student projects, and experiments based on randomized material.

In Chapter 6 the major findings and restrictions of this thesis are considered and conclusions are drawn from the results.

2. APPLICATION OF ALBRECHT'S FPA TO SA SPECIFICATIONS

---

[1] Short descriptions of Function Bang and Mark II metrics are given in Appendices I and J.

To measure the complexity (size) of a system, one must trace the logical transactions triggered by events in the system environment[1]. This means that a model must be used from which each transaction and the measurable components of these transactions may be reliably traced and extracted. During formulation of the model, the level of abstraction for describing an object system must be chosen. Two basic ways to build the model are a top-down (from whole to parts) approach and a bottom-up (from parts to whole) approach. An intermediate form of these approaches is sometimes called a middle-out approach.

Currently, with CASE-tools, the system is often described at the specification level functionally by dataflow diagrams of Sructured Analysis method. Traditionally, the use of dataflow diagrams has involved the top-down approach (e.g. DeMarco 1978), but Yourdon (1989) has also proposed the middle-out approach. The static data model of a system is made by using an entity-relationship diagram. In this chapter algorithms for calculating the value of Albrecht's unadjusted function points[2] based on descriptions of Structured Analysis will be given. The skeleton of the measurement is the dataflow diagram but the entity-relationship diagram is also needed. The notation of dataflow and entity-relationship diagram techniques will be considered in the next section.

2.1. Components of Dataflow and Entity-Relationship Diagrams

Within *Structured Analysis* (SA) developed by DeMarco (1978) the system is described by a *dataflow diagram* (DFD) which consists of dataflows, processes, data stores and terminators. The set of symbols used by DeMarco is shown in Figure 2.1. Later on, this basic set has been revised and supplemented with continuous dataflows, control transformations, control flows and event stores (Ward & Mellor 1985; 1986, Woodman 1988, Yourdon 1989). These extensions, however, are not within the scope of this thesis.

A *process* in a DFD uses one or more incoming dataflows and produces one or more new outgoing dataflows as a result. Process descriptions contain texts which specify how the data is transformed. It is possible to indicate whether all incoming dataflows are always required, or all outgoing dataflows are always produced. This is done using an asterisk, *, for 'AND', and a plus sign inside a circle, ⊕, for 'OR' between two dataflows.

---

[1] See also Appendix C.

[2] This is called TUFPV(SA) in Figure 1.4.

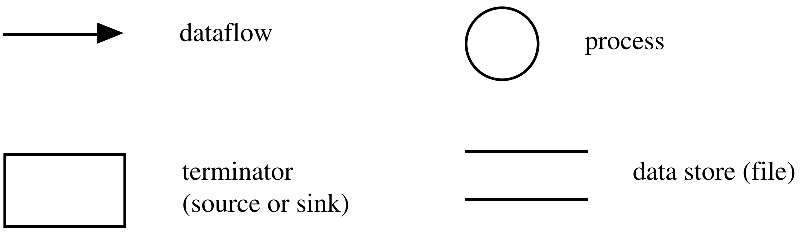
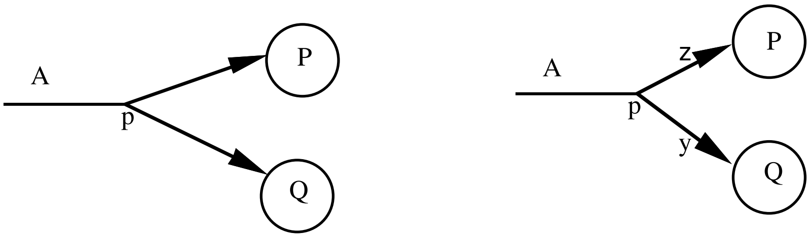


Figure 2.1. Basic symbols of the dataflow diagram.

Terminators are the origin (source) of the *input dataflows* or the destination (sink) of the *output dataflows*. They are located in the environment of the software system. Consequently, their behaviour is not modelled (Woodman 1988).

The *data store*, or file, is used to model a named collection of data packets. Data may not flow directly from a terminator to a file or from a file to a terminator. In both cases a process must transform the data. A file can be either retrieved or updated. When a file is updated, its data contents are either added, modified, or deleted.[1]

The input and output dataflow between a terminator and a process as well as the *internal dataflows* between process pairs are always named. A dataflow from/to a file is not named, however, if it represents the whole content of a file. We must remember that it is the data that are being modelled and not a physical file. It is also possible that a dataflow is *composite* (DeMarco 1979, Woodman 1988).



(a) A is used by both P and Q (b) z and y are components of A

Figure 2.2. Composite input dataflow.[2]

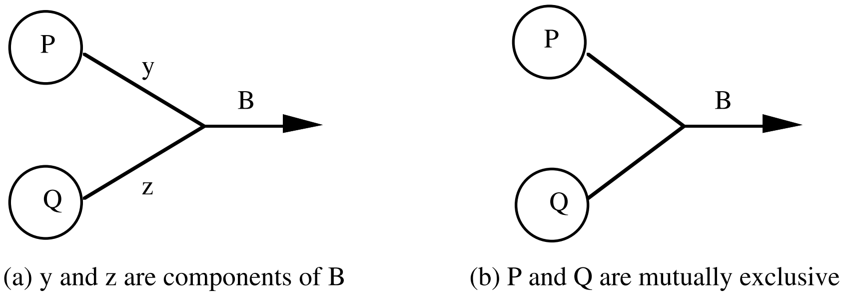
Figure 2.2(a) shows dataflow A going to two transformations - P and Q. Figure 2.2(b) shows dataflow A diverging: this implies that A is composed of *y* and *z*. Figure 2.3(a) im-

---

[1] Yourdon (1989) allows external data stores between a terminator and a context diagram. This principle is applied in Section 2.2.4.

[2] Point *p* is here called a divergence point.

plies that y and z are part of the composite dataflow B. Figure 2.3(b) shows two transformations - P and Q - both of which produce dataflow B, but not at a same time.



(a) y and z are components of B

(b) P and Q are mutually exclusive

Figure 2.3. Composite output dataflow.

The DFD depicts a functional description of a system, not a procedural one. The DFD, therefore, implies no ordering of processes except that a process which produces a dataflow for another must occur before the one that depends on it. The system is divided into parts by a *functional decomposition* refining and expanding each process into a DFD in a top-down way. DeMarco (1979) says that diagrams partitioned into approximately seven pieces are usually workable. Yourdon (1989) gives a middle-out method where the user transactions are first described each in turn and then the subdiagrams are connected - after possible refinement - into one DFD.

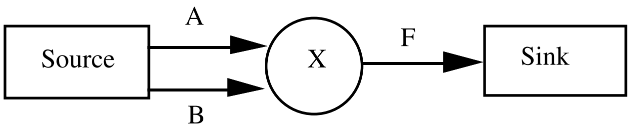


Figure 2.4. A context diagram.

Functional decomposition begins at the boundary between the software system and its environment. The diagram that depicts this division is called a *context diagram* and it constitutes the root of the hierarchy of DFDs representing a hierarchy of functions. A context diagram is a DFD which contains a single process that represents the entire system and the major sources of data and destinations for data in the environment. Figure 2.4 gives an example of a context diagram for system X with two input dataflows (A and B) and one output dataflow (F).

A process which is at the bottom of the refinement hierarchy is described as functionally primitive. The specifications of these *primitive processes* are often called mini-specifications.

An *entity-relationship diagram* (ER-diagram, ERD) is a graphical notation for modelling data. For ERDs there are no standard notations. When the notations of Yourdon (1989) are used, the following components may be found for ERDs: object types, relationships, associative object type indicators, and supertype/subtype indicators. Figure 2.5 illustrates these components.

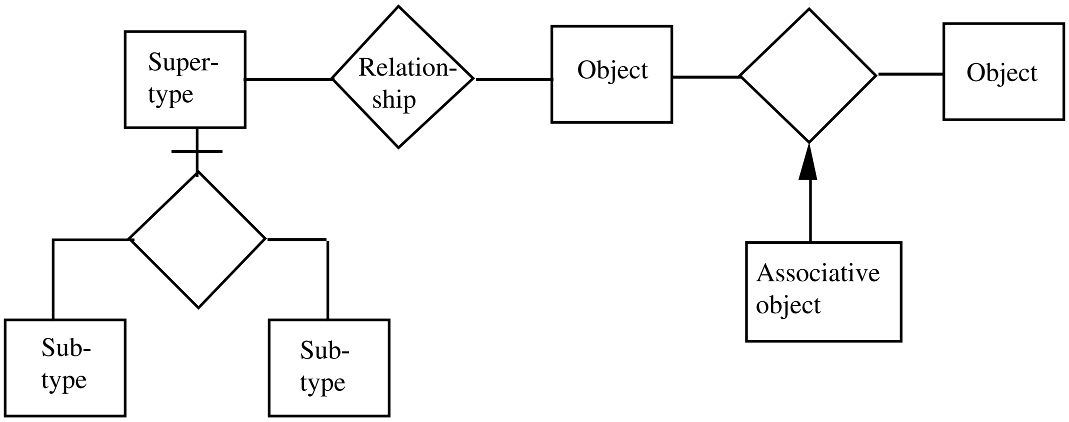


Figure 2.5. Components of an ERD.

The requirements of the data are recorded in a *data dictionary* which represents the structure of data within the system. The notations of Yourdon (1989) for a data dictionary are presented in Figure 2.6.

<table>
<tr><td>=</td><td>is composed of</td></tr>
<tr><td>+</td><td>and</td></tr>
<tr><td>( )</td><td>optional (may be present or absent)</td></tr>
<tr><td>[ ]</td><td>select one of several alternative choices</td></tr>
<tr><td>**</td><td>comment</td></tr>
<tr><td>@</td><td>identifier (key data item) for a data store[1]</td></tr>
<tr><td>|</td><td>separates alternative choices in the [ ] construct</td></tr>
</table>

Figure 2.6. Data dictionary notation.

The data flowing within a DFD must be valid for the system being modelled. A data dictionary with DFDs and ERDs is used to check the internal correctness of the specification. Cowan (1990) considers the consistency and completeness checks of DFDs and gives 17

---

[1] Based on Cowan (1990), keys used by processes to access data stores can be recorded as data items of dataflows and shown on the DFD. This principle is also followed here.

rules for this purpose. Yourdon (1989) also gives a list of balancing rules between and within Structured Analysis descriptions.[1]

2.2. Identification and Weighting of User Functions

In a DFD a *transaction* consists of elementary functions or primitive processes which are directly chained to each other via dataflows. Moreover, it contains all data stores and terminators that are connected to the functions defining a transaction (Shoval 1988, Shoval & Manor 1990). A user transaction is identified by an input dataflow or by an output dataflow. 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[2]. One of the primitive processes that constitute a transaction is the trigger, i.e. a function connected to the terminator that is the cause of an event.

Figure 2.7 represents two user transactions. In the transaction triggered by T, the trigger P1 is the starting function and the chain effect goes forward from P1 to P2. In the other transaction triggered by S, the trigger function P4 is the last function on its chain and cannot be executed before P3 is executed.

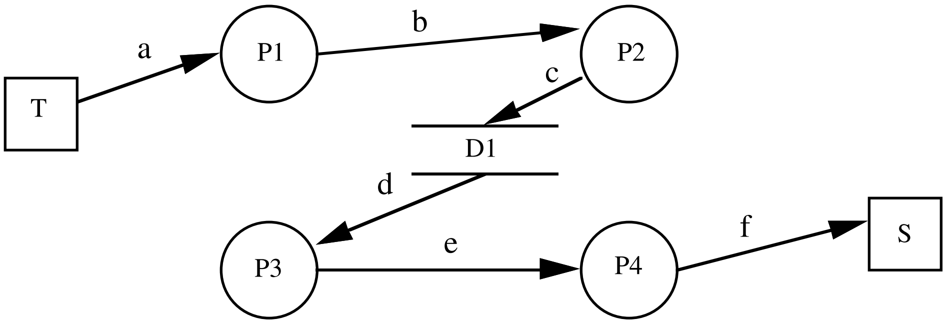


Figure 2.7. Two user transactions (Shoval 1988).

In the following it is shown how to compute TUFPV from SA descriptions by using the lowest level of the logical DFD system. To weight logical internal files and external interface files, an ER-diagram of SA is also needed. Each of the function types will be consid-

---

[1] The JoUCASE system that implements the algorithms of Section 2.2 supports most of these rules and a number of extra rules (Laamanen & Rask 1991).

[2] See also Section 2.2.1.

ered one after another. It will also be shown how functions can be identified in the dataflow diagram and how properties needed in the complexity classification can be identified[1]. One simple example of each function type will be given to illustrate the counting process. The rules without algorithms are given by Rask (1991a). The preliminary algorithms have also been published (Rask 1991b).[2]

2.2.1. External Input and Output Types

The *inputs* of a DFD originate from terminators (input dataflows) and the *outputs* flow to terminators (output dataflows). The *input data items* of the input dataflow structure can be taken from a data dictionary. The *files referenced* by an input transaction are the logical internal files corresponding to data stores of the dataflow diagram within the transaction. A referenced logical file is identified by the combination of a data store name and a key of a dataflow referencing the data store. The notation <d,key(f)> is used for logical files, where *d* is the name of the data store, and *f* is the name of the dataflow. If the dataflow is not named, the primary key of the data store is used to form a logical file[3].

At this stage the user interface is not usually designed[4]. Hence some data items may be missing. It is essential, however, that all input functions be identified in order to count correctly the value of the total unadjusted function points. The number of alternatives in an input dataflow also affects the number of unadjusted function points. It is not the same to have one input dataflow with three alternatives (for example *f* = [a|b|c]) as to have three separate input dataflows. Design decisions must recognize these situations to make the cost estimates reliable. By common consent the alternatives have to either be decomposed uniformly or be described with one dataflow.

In the case of composite input dataflows the system boundary should be placed as in Figure 2.8. In Figure 2.8(a) there are two inputs because two processes are needed to transform the composite input dataflow. In Figure 2.8(b) there is only one input (although there are two terminators) because only one process is needed to transform the composite input dataflow.

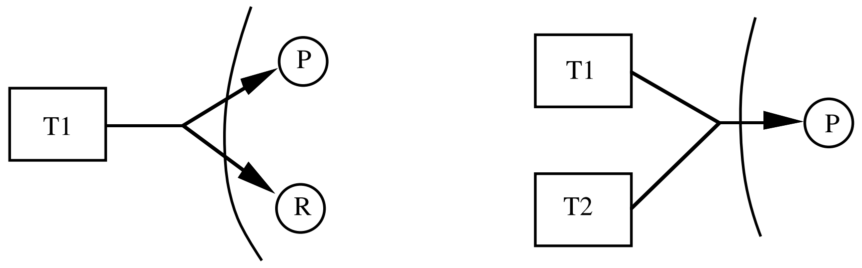
---

[1] The classification factors and weights for the functions are tabulated in Appendix B.

[2] The pseudocode notation of the algorithms is partly based on personal communication with Professor Martti Penttonen. The notation for the algorithms is summarized in Appendix H.

[3] See also Section 2.2.3.

[4] Appendix C contains a short discussion of user interface and application parts of a software system.



a) Two external inputs b) One external input

Figure 2.8. Counting composite input dataflows.

The above described principles for finding and weighting of external inputs from DFDs are formalized in Algorithm 2.1, which can be used to automate the counting process. This algorithm consists of two main procedures. One of these (weights_of_inputs_SA) weights each input and the other (find_logical_files_of_inputs) finds the logical files referenced by the inputs.

The complexity of the procedure weights_of_inputs_SA is $O(m)$ and the complexity of the procedure find_logical_files_of_inputs is $O(n)$, where the number of input dataflows is m and the number of dataflows in the processed dataflow diagram is n. The for-loop of the procedure weights_of_inputs is performed m times and in the worst case the procedure find_logical_ files_of_inputs inspects all dataflows per each input dataflow. Thus, the complexity of Algorithm 2.1 is $O(mn)$.

Table 2.1 lists the procedures used in Algorithm 2.1, the complexity of each procedure (excluding work done in called procedures), and the calling structure of these procedures. The parameters and local variables are explained within each procedure of Algorithm 2.1. Local functions and procedures, and special notations are explained in Appendix H.

Table 2.1. Procedures, complexity, and calling structure of Algorithm 2.1.

Procedure	Complexity	Calls
weights_of_inputs_SA	O(m)	find_logical_files_of_inputs
find_logical_files_of_inputs	O(n)	

m = number of input dataflows
n = number of all dataflows

Algorithm 2.1. Finding and weighting of inputs in SA.[1]

procedure weights_of_inputs_SA (input_set, input_weight_set):

---

[1] Algorithm 2.1 is a part of TUFPV(SA) in Figure 1.4.

```
-- parameter input_set imports the input dataflows
-- parameter input_weight_set exports the weights of the input dataflows
-- local variable m counts the number of input dataflows in input_set
-- local variable f denotes the input dataflow to be weighted
-- local variable t denotes the starting point (terminator) for f
-- local variable p denotes the ending point (primitive process) for f
-- local variable logical_file_set contains all logical internal files referenced by f
-- local variable n counts the number of logical internal files referenced by f
-- local variable s counts the number of data items of f
-- local variable w counts the weight of f
-- remark: at the beginning all dataflows are unmarked
-- remark: input_weight_set is initialized in the main program

m := |input_set|;
for all m input dataflows f=(t,p) with terminator t and primitive process p in input_set do
        logical_file_set : = Ø;
        find_logical_files_of_inputs (p, logical_file_set);
        n : = |logical_file_set|;
        s : = number_of_data_items(f);
        w : = input_weight(n, s);
        input_weight_set : = input_weight_set ∪ w.


procedure find_logical_files_of_inputs (v, logical_file_set):

-- parameter v imports the address of a primitive process or a divergence point within the transaction
-- parameter logical_file_set exports the logical internal files for the input dataflow selected in procedure
   weights_of_inputs
-- local variable f denotes a dataflow within the transaction
-- local variable v' denotes a primitive process, a data store, or a divergence point
-- local variable d denotes a data store

for all dataflows f=(v,v') and f=(v',v) do
        if f=(v,v') and v' is a divergence point and f is unmarked then
                mark(f);
                find_logical_files_of_inputs(v',logical_file_set)
        elsif f=(v,v') and v' is a primitive process and f is unmarked then
                mark (f);
                find_logical_files_of_inputs (v', logical_file_set);
        elsif v' is a data store d then
                if f is named then
                        if <d,key(f)> ∉ logical_file_set then
                                logical_file_set := logical_file_set ∪ <d,key(f)>;
                elsif <d,key(d)> ∉ logical_file_set then
                        logical_file_set := logical_file_set ∪ <d,key(d)>.
```

The classification principles for outputs are similar to those for inputs. The required *ouput data items* in the output dataflow structure can be found from the data dictionary. The *files referenced* by an output transaction are then the logical internal files of the data stores within the transaction. The referenced data stores are now found by traversing backwards along the process chain which forms the skeleton of the transaction. For composite output dataflows the system boundary has to be understood as presented for external inputs in Figure 2.8.

In most cases an output should also correspond to some input in a dataflow diagram. Only temporal events (Yourdon 1989) can be initiated without an input dataflow. An example of this situation was shown in Figure 2.7. In FPA the input is a part of the application as a user function and we must count each of these functions. This recommendation is also included in the counting principles of Symons's (1991) Mark II Function Point metric[1].

The possibility of missing data items before precise output design, the number of alternatives in an output dataflow, and the principles for composite output dataflows hold for external outputs as presented in connection with external inputs. For example, if all different error messages are counted as separate data elements, the complexity of the corresponding output may increase.

Algorithm 2.2 can be used to find and weight outputs from DFDs. The algorithm is similar to Algorithm 2.1 except for the direction of dataflows. In Algorithm 2.2 we go backwards through the dataflows, which connect the processes within the transaction, to find the logical files referenced by each output. The complexity of Algorithm 2.2 is also *O*(*mn*) where the number of output dataflows is *m* and the number of all dataflows in the processed dataflow diagram is *n*.

Table 2.2 lists the procedures used in Algorithm 2.2, the complexity of each procedure (excluding work done in called procedures), and the calling structure of these procedures. The parameters and local variables are explained within each procedure of Algorithm 2.2. Local functions and procedures, and special notations are explained in Appendix H.

Table 2.2. Procedures, complexity, and calling structure of Algorithm 2.2.

Procedure	Complexity	Calls
weights_of_outputs_SA	O(m)	find_logical_files_of_outputs
find_logical_files_of_outputs	O(n)	

m = number of output dataflows
n = number of all dataflows

Algorithm 2.2. Finding and weighting of outputs in SA.[2]

```
procedure weights_of_outputs_SA (ouput_set, output_weight_set):
```

---

[1] Based on Symons (1991), a system is a collection of logical transactions, with each transaction consisting of an input, process, and output component. The transaction is triggered by a unique event of interest to the user, or a need to retrieve information.

[2] Algorithm 2.2 is a part of TUFPV(SA) in Figure 1.4.

```
-- parameter output_set imports the output dataflows
-- parameter output_weight_set exports the weights of the output dataflows
-- local variable m counts the number of output dataflows in output_set
-- local variable f denotes the output dataflow to be weighted
-- local variable p denotes the starting point (primitive process) for f
-- local variable t denotes the ending point (terminator) for f
-- local variable logical_file_set contains all logical internal files referenced by f
-- local variable n counts the number of internal logical files referenced by f
-- local variable s counts the number of data items of f
-- local variable w counts the weight of f
-- remark: at the beginning all dataflows are unmarked
-- remark: output_weight_set is initialized in the main program

m := |output_set|;
for all m output dataflows f=(p,t) with primitive process p and terminator t in output_set do
        logical_file_set : = Ø;
        find_logical_files_of_outputs (p, logical_file_set);
        n : = |logical_file_set|;
        s : = number_of_data_items(f);
        w : = output_weight(n, s);
        output_weight_set : = output_weight_set ∪ w.
```

```
procedure find_logical_files_of_outputs (v, logical_file_set):

-- parameter v imports the address of a primitive process or a divergence point of the transaction
-- parameter logical_file_set exports the internal logical files for the output dataflow selected in procedure
   weights_of_outputs
-- local variable f denotes a dataflow within the transaction
-- local variable v' denotes a primitive process, a data store, or a divergence point
-- local variable d denotes a data store

for all dataflows f=(v',v) and f=(v,v') do
        if  f=(v',v) and v' is a divergence point and f is unmarked then
                mark(f);
                find_logical_files_of_outputs(v',logical_file_set)
        elsif  f=(v',v) and v' is a primitive process and f is unmarked then
                mark (f);
                find_logical_files_of_outputs (v', logical_file_set)
        elsif v' is a data store d then
                if f is named then
                        if  <d,key(f)> ∉ logical_file_set then
                                logical_file_set := logical_file_set  ∪ <d,key(f)>
                elsif <d,key(d)> ∉ logical_file_set then
                        logical_file_set := logical_file_set  ∪ <d,key(d)>.
```

Figure 2.9 gives an example of unique inputs and outputs in the context of two different transactions. Daily and weekly reports and the detail and total parts of the reports are separated[1]. The first transaction consists of one input (d from R), two outputs (d-details and d-totals to S), one process (P1), and one data store (D1). The data store is also contained in the second transaction.

In Figure 2.9 at least six unadjusted function points can be found for external inputs: there are two inputs (d, w) and the weight 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

---

[1] According to Dreger (1989) IFPUG (International Function Point Users Group) disagrees with the separation between the totals and the detail level.

weight of an output is at least four. The exact value of the unadjusted function points depends on the number of data items referenced, and this information can be obtained from the data dictionary. The complexity values derived above are the minimum values.

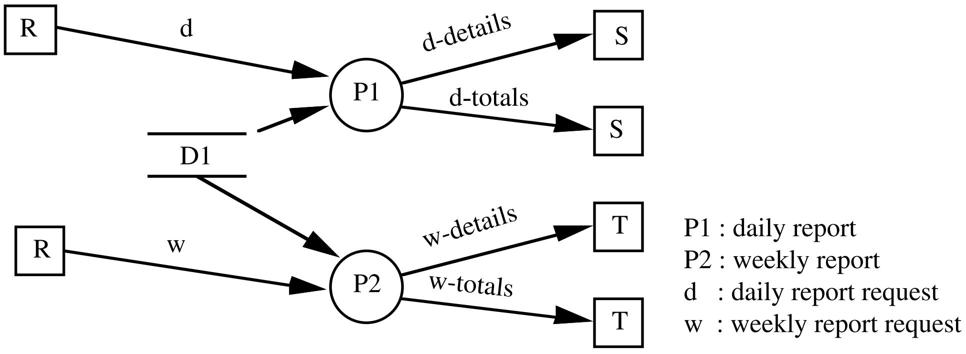


Figure 2.9. Input and output transactions.

2.2.2. External Inquiry Types

An *inquiry* includes both an input and an 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 from a dataflow (context) diagram, an inquiry dataflow should be represented by a *double-directed edge* (an arrowhead at both ends of the dataflow).[1] If an inquiry is depicted by two directed edges, it is difficult to make a distinction between inquiries and input and output types.

An inquiry can be decomposed into subinquiries and after the decomposition the input and output dataflows of the decomposed inquiry may be related to separate processes. It must, however, be possible to associate these separate dataflows to the upper level inquiry. A separate input or a separate output dataflow of an inquiry cannot be further divided into separate input or output dataflows. Although the inquiry cannot perform updates[2], it is often part of an update, as described in Figure 2.10.

---

[1] Although Yourdon (1989) also uses this double-directed edge, its meaning is different, because his theory does not separate the input, output and inquiry types from one another.

[2] See Section 1.2.

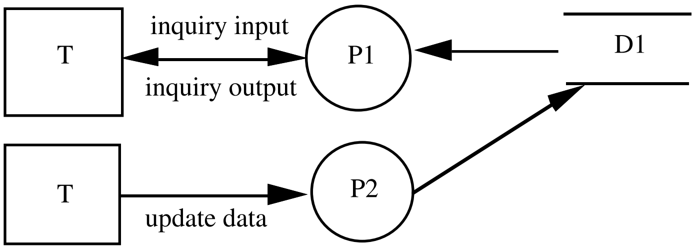


Figure 2.10. A typical inquiry as part of an update.

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 the worst (input or output) part has to be chosen for function point counting (Dreger 1989).[1]

The above described principles for finding and weighting of external inquiries from DFDs are formalized in Algorithm 2.3, which can be used to automate the counting process. The procedure weights_of_inquiries weights both an input part and an output part of an inquiry and then selects the most complex one. Because an inquiry consists of an input part and an output part, it is possible that the procedure weights_of inquiries_SA in Algorithm 2.3 applies the procedures find_logical_files_of_inputs and find_logical_files_of_outputs of Algorithms 2.1 and 2.2 to count the weights for input and output parts of an inquiry dataflow. Thus, the complexity of Algorithm 2.3 is $O(2mn) = O(mn)$ where the number of dataflows in the processed dataflow diagram is n and the number of inquiry dataflows is m.

Table 2.3 lists the procedures used in Algorithm 2.3, the complexity of each procedure (excluding work done in called procedures), and the calling structure of these procedures. The parameters and local variables are explained within each procedure of Algorithm 2.3. Local functions and procedures as well as special notations are explained in Appendix H.

Table 2.3. Procedures, complexity, and calling structure of Algorithm 2.3.

Procedure	Complexity	Calls
weights_of_inquiries_SA	O(m)	find_logical_files_of_inputs[2]

---

[1] When Dreger's (1989) approach is used, the possible weights for an inquiry are 3, 4, 5, 6, and 7 based on input and output weights. When original IBM' s approach (Development Centre Productivity Measurement Guide 1984) is used, the complexity of an inquiry is based on the complexities of input and output parts, but the possible weights are 3,4, and 6 based on input weights. Here Dreger's (1989) inquiry classification principles have been followed. In Rask et al. (1992b) the original weights were used, and results consistent with the results reported in this thesis were obtained.

[2] See Algorithm 2.1.

find_logical_files_of_outputs[1]

find_logical_files_of_inputs	O(n)
find_logical_files_of_outputs	O(n)

m = number of inquiry dataflows
n = number of all dataflows

Algorithm 2.3. Finding and weighting of inquiries in SA.[2]

```
procedure weights_of_inquiries_SA (inquiry_set, inquiry_weight_set):

-- parameter inquiry_set imports the inquiry dataflows
-- parameter inquiry_weight_set exports the weights of the inquiry dataflows
-- local variable m counts the number of inquiry dataflows in inquiry_set
-- local variable f denotes the inquiry dataflow to be weighted
-- local variable $f_I$ denotes the input part of f
-- local variable $f_O$ denotes the output part of f
-- local variable t denotes a terminator
-- local variable p denotes a primitive process
-- local variable logical_file_set contains all logical files referenced
-- local variable w counts the weight of f
-- local variable $n_I$ counts the number of internal logical files of $f_I$
-- local variable $s_I$ counts the number of data items of $f_I$
-- local variable $w_I$ counts the weight of $f_I$
-- local variable $n_O$ counts the number of internal logical files of $f_O$
-- local variable $s_O$ counts the number of data items of $f_O$
-- local variable $w_O$ counts the weight of $f_O$
-- remark: inquiry_weight_set is initialized in the main program

m := |inquiry_set|;
for all m inquiry dataflows f = [$f_I$,$f_O$] in inquiry_set do
        select input part $f_I$ = (t,p) from inquiry dataflow f;
        logical_file_set : = Ø;
        find_logical_files_of_inputs (p, logical_file_set);
        $n_I$ : = |logical_file_set|;
        $s_I$ : = number_of_data_items($f_I$);
        $w_I$ : = input_weight($n_I$, $s_I$);
        select output part $f_O$ = (p,t) from inquiry dataflow f;
        logical_file_set : = Ø;
        find_logical_files_of_outputs (p, logical_file_set);
        $n_O$ : = |logical_file_set|;
        $s_O$ : = number_of_data_items($f_O$);
        $w_O$ : = output_weight($n_O$, $s_O$);
        w : = max($w_I$, $w_O$);
        inquiry_weight_set : = inquiry_weight_set $\cup$ w.
```

As an example, Figure 2.10 describes two transactions: one for an inquiry and one for an update. The complexity value for the inquiry is at least four, since the weight of the inquiry input is at least three and the weight of the inquiry output is at least four. Thus, the

---

[1] See Algorithm 2.2.

[2] Algorithm 2.3 is a part of TUFPV(SA) in Figure 1.4.

worst case complexity is at least four. For the input of the update the complexity value is at least three.

2.2.3. Logical Internal File Types

In a DFD each data store corresponds to one or more logical internal files. Each data store contains a primary key, but it can also be referenced by other keys depending on the dataflow definitions referencing the data store. The number of different keys used (including the primary key) 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). If a dataflow is unnamed, we can infer that all the data items of the data store are referenced and the key of the dataflow is the primary key of the data store. For a logical internal file, the notation <d,key(f)> is used, where *d* means a data store and *f* means a dataflow referencing data store *d*.

If a data store contains different record formats, the referenced (named) dataflow should contain the name of one of these records. Thus, the *number of logical record formats* of a logical file is defined by the number of differently named dataflows referencing it. For a logical record, notation <d,f,key(f)> is used, where *d* means a data store and *f* is a dataflow referencing data store *d*. For each logical file the *number of referenced data items* can be found by examining dataflows (containing the same key) around the corresponding data store.

The relationships in which a data store (a logical internal file) participates may also increase the value of unadjusted function points. To identify the relationships between data stores, the *entity model* (ER-diagram) of the application must be available. This model directs the derivation of the DFD. This idea is also supported 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 of an ERD. Thus, for purposes of function point counting *the number of relationships* between data stores can be found directly from the entity model. In most cases 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 the name of a book, the relationships of these logical files corresponds to the relationships of the entity BOOK in an ERD.

The above described principles for finding and weighting of logical internal files based on a DFD and an ERD are formalized in Algorithm 2.4, which can be used to automate the counting process. The complexity of Algorithm 2.4 in the worst case is $O(p((p-1)+q+q^2))$

= $O(pq^2)$ where the number of internal data stores[1] in the processed dataflow diagram is p and the number of processes[2] is q. In the worst case each data store is connected to each process and there is a relationship between each pair of data stores. As it is reasonable to assume that $q>p$, we can say that the complexity in the worst case is $O(pq^2)$.

Table 2.4 lists the procedures used in Algorithm 2.4, the complexity of each procedure (excluding work done in called procedures), and the calling structure of these procedures. The parameters and local variables are explained within each procedure of Algorithm 2.4. Local functions and procedures as well as special notations are explained in Appendix H.

Table 2.4. Procedures, complexity, and calling structure of Algorithm 2.4.

Procedure	Complexity	Calls
weights_of_logical_files_SA	$O(pq^2+p^2-p)$	find_logical_records_and_files
find_logical_records_and_files	$O(q)$	

p = number of data stores
q = number of processes

Algorithm 2.4. Finding and weighting of logical internal files in SA.[3]

```
procedure weights_of_logical_files_SA (internal_data_store_set, file_weight_set):

-- parameter internal_data_store_set imports the data stores inside the system boundary
-- parameter file_weight_set exports the weights of the internal logical files
-- local variable p counts the number of data stores in internal_data_store_set
-- local variable d denotes the data store to be divided into logical files
-- local variable r counts the number of relationships for d
-- local variable logical_record_set contains all logical records of d
-- local variable logical_file_set contains all logical internal files of d
-- local variable l counts the number of logical record formats for <d,k>
-- local variable unique_data_item_set contains the unique data items collected from <d,c,k> of <d,k>
-- local variable n counts the maximum of r and l
-- local variable s counts the number of data items of <d,k>
-- local variable w counts the weight of <d,k>
-- remark: file_weight_set is initialized in the main program


p := |internal_data_store_set|;
for all p data stores d in internal_data_store_set do
        r : = number_of_relationships(d);
        logical_record_set : = Ø;
        logical_file_set := Ø;
```

---

[1] Passed files, which are weighted in Algorithm 2.5, are excluded from internal data stores.

[2] Shared file connections to external applications are excluded. Based on Dreger (1989), only those data items actually required by the application need be taken into account.

[3] Algorithm 2.4 is a part of TUFPV(SA) in Figure 1.4.

```
        find_logical_records_and_files (d, logical_record_set, logical_file_set);
        for all logical files <d,k> in logical_file_set do
                l := 0;
                unique_data_item_set := Ø;
                for all logical record formats <d,c,k> in logical_record_set do
                        if <d,c,k> is a logical record format of logical file <d,k> then
                                l := l+1;
                                unique_data_item_set := unique_data_item_set ∪
                                        data_items(<d,c,k>);
                n : = max(r,l);
                s : = |unique_data_item_set|;
                w : = file_weight(n, s);
                file_weight_set : = file_weight_set ∪ w.
```

```
procedure find_logical_records_and_files (d, logical_record_set, logical_file_set):

-- parameter d imports the data store to be processed
-- parameter logical_record_set exports the logical records of d
-- parameter logical_file_set exports the logical internal files of d
-- local variable f denotes a dataflow connected to d

for all dataflows f connected to d, excluding shared file connections to external applications do[1]
        if f is named then
                if <d,f,key(f)> ∉ logical_record_set then
                        logical_record_set := logical_record_set ∪ <d,f,key(f)>;
                if <d,key(f)> ∉ logical_file_set then
                        logical_file_set := logical_file_set ∪ <d,key(f)>
        else
                if <d,d,key(d)> ∉ logical_record_set then
                        logical_record_set := logical_record_set ∪ <d,d,key(d)>;
                if <d,key(d)> ∉ logical_file_set then
                        logical_file_set := logical_file_set ∪ <d,key(d)>.
```

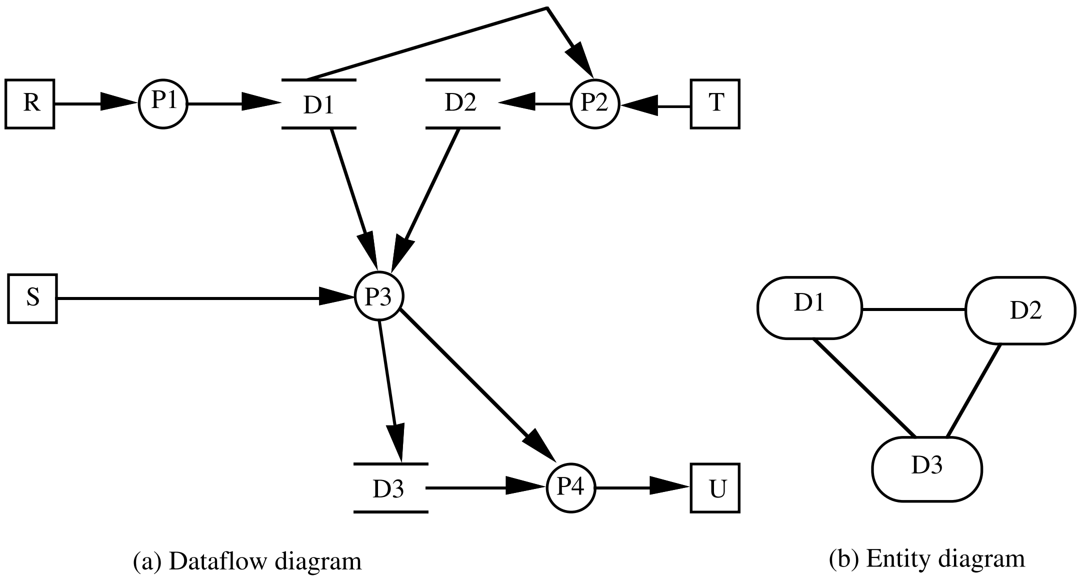
The more *normalized*[2] the data stores are, the more realistic the total number of unadjusted function points is. Accordingly, Gane and Sarson (1979) propose that the data base schema should be formed by dividing each data store in a DFD using normalization rules.

As an example, Figure 2.11(a) contains three transactions. If we know the corresponding entity model of Figure 2.11(b), we can conclude that there is a relationship between data stores D1 and D2, D1 and D3, and D2 and D3. If each data store has one key and the number of data items referenced is at most 19, the weight for each file is seven as each file has only two logical relationships.

---

[1] See also section 2.2.4.

[2] Normalization is discussed in any good database design book.

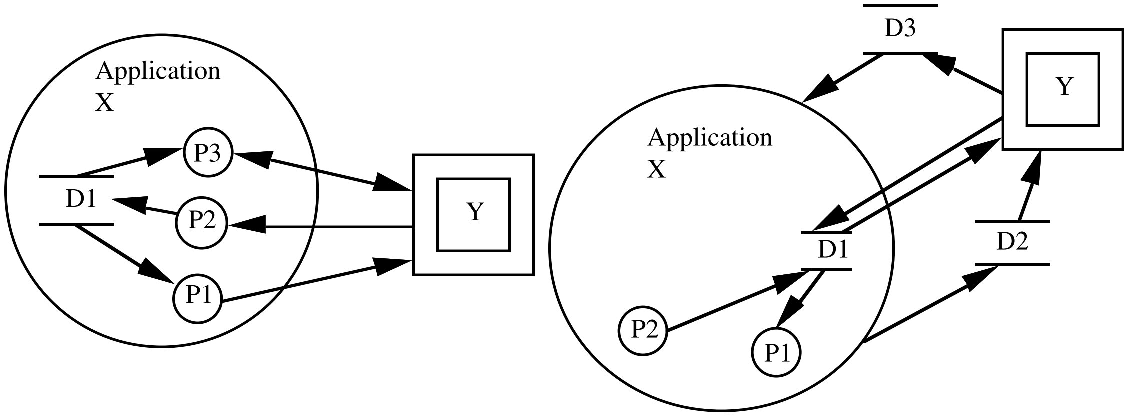


(a) Dataflow diagram (b) Entity diagram

Figure 2.11. Logical internal files and their relationships.

2.2.4. External Interface File Types

To distinguish an external application from other terminators of a DFD, the type[1] of each terminator could be specified on a repository only. For clarity, however, a special graphical symbol should be given for external applications. Therefore an external application will be described explicitly by using *double rectangles*. Accordingly, the interface files in a DFD are represented as in Figure 2.12.



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

Figure 2.12. Principles of an interface file type.

---

[1] The type of a terminator is either "normal" or "external application".

Figure 2.12(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 possible conversion of transaction files in a dataflow diagram. It is, however, possible to compare incoming and outgoing dataflows of a transaction process to determine the conversion. If the incoming dataflow of the process which writes the transaction file differs from the output dataflow, then it may be assumed that a conversion is made and one interface type may be counted in addition to one output type[1]. Correspondingly, if the outgoing dataflow of the process which reads the transaction file differs from the input dataflow, it may be assumed that a conversion is made and one interface type may be counted in addition to one input type. For inquiry transactions (input and output parts) the principles explained for input and output transactions are applied.

To record all the transaction types the number of types in a dataflow must be obtained. Different transaction types are described by a selection of alternatives, for example, *t*=[r|s] where *r* and *s* are transaction types. When the number of unadjusted function points is calculated, the number of relationships to other files is not counted, but the *number of logical record formats* of the transaction file is measured. Here there are two possibilities: each transaction type may be interpreted as a record format or each transaction type may be interpreted as a separate transaction. Of course, the interpretation influences the number of function points. If each transaction type is interpreted as a separate transaction[2], the transaction dataflow has to be divided into separate dataflows by *decomposing* the original transaction process into separate processes corresponding to the transaction types[3]; otherwise it is not possible to count the unadjusted function points. Accordingly, a transaction type may then contain different record formats, for example *r*=[c|d|e].

In Figure 2.12(b) the different interpretations for master files have been described. Data stores D2 and D3 are *passed files*[4]. Data store D3 is produced by application Y and used by application X. Data store D2 is produced by application X and used by application Y. Here D3 receives interface rank only and D2 receives both file and interface rank, although D2 is a passed file. It is, however, created by application X, and according to Dreger (1989), it should also receive a file rank. If the same passed file appears more than once in

---

[1] There is a difference between the dataflows if the incoming and outgoing dataflows have different names or the names of the decomposed outgoing dataflows do not match the names of the components of the incoming (composed) dataflow.

[2] This is recommended by Dreger (1989).

[3] This decomposition may also improve the quality and reliability of the system because each subtransaction has to be specified separately. When the counting process is automated the completeness of the specification can be tested.

[4] Yourdon (1989) uses the term "external store".

the context diagram, it should be counted only once as a file and as an interface[1]. Each passed file is understood to be one logical file and contains one record format.

Data store D1 in Figure 2.12(b) is a *shared file.* It is shared between applications X and Y. Both applications use and maintain D1 as though it were their internal file. To indicate an external shared file interface between the shared data store of the object system and the terminator that describes the external application, a direct dataflow connection is needed. In this situation the external application is interpreted as a process.

The number of external interface files depends on the number of different key data items referencing the shared data store. Correspondingly, each shared external interface file may contain different record formats. The relationships to other entities are defined from the ER-diagram, as with the logical internal files.[2]

The above described principles for finding and weighting of external interface files based on a DFD and an ERD are formalized in Algorithm 2.5, which can be used to automate the counting process. The main procedure is named weights_of_interface_SA. It calls the subprocedures that find and weight the interface inputs, interface outputs, interface inquiries, interface passed files, and interface shared files. The subprocedures also use the procedures of the previous algorithms to find logical files.

The complexity of Algorithm 2.5 in the worst case is $O(m_1n+m_2n+2m_3n+p_1(p-1)+p_2(p-1+m_4+m_4{}^2)) = O(n(m_1+m_2+m_3)+(p_1+p_2)(p-1)+p_2(m_4+m_4{}^2))$ where the number of data stores in the DFD is p, the number of all dataflows is n, the number of interface input dataflows is m_1, the number of interface output dataflows is m_2, the number of interface inquiry dataflows is m_3, the number of external data stores is p_1, the number of shared file connections to external applications is m_4, and the number of shared data stores is p_2.

Table 2.5 lists the procedures used in Algorithm 2.5, the complexity of each procedure (excluding work done in called procedures), and the calling structure of these procedures. The parameters and local variables are explained within each procedure of Algorithm 2.5. Local functions and procedures, and special notations are explained in Appendix H.

---

[1] This is also recommended in the IBM Development Centre Productivity Measurement Guide (1984).

[2] See also Section 2.2.3.

Table 2.5. Procedures, complexity, and calling structure of Algorithm 2.5.

Procedure	Complexity	Calls
weights_of_interfaces_SA	$O(5)$	weights_of_interface_inputs weights_of_interface_outputs weights_of_interface_inquiries weights_of_interface_passed_files weights_of_interface_shared_files
weights_of_interface_inputs	$O(m_1)$	find_logical_files_of_inputs[1]
weights_of_interface_outputs	$O(m_2)$	find_logical_files_of_outputs[2]
weights_of_interface_inquiries	$O(m_3)$	find_logical_files_of_inputs[1] find_logical_files_of_outputs[2]
weights_of_interface_passed_files	$O(p_1(p-1))$	
weights_of_interface_shared_files	$O(p_2(p-1+m_4^2))$	find_logical_records_and_files
find_logical_files_of_inputs	$O(n)$	
find_logical_files_of_outputs	$O(n)$	
find_logical_records_and_files	$O(m_4)$	

m_1 = number of interface input dataflows
m_2 = number of interface output dataflows
m_3 = number of interface inquiry dataflows
m_4 = number of shared file connections to external applications
n = number of all dataflows
p = number of data stores
p_1 = number of passed files (external data stores)
p_2 = number of shared data store connections

Algorithm 2.5. Finding and weighting of external interface files in SA.[3]

```
procedure weights_of_interfaces_SA(interface_input_set, interface_output_set, interface_inquiry_set,
passed_file_set, shared_file_set, input_weight_set, output_weight_set, inquiry_weight_set,
interface_file_weight_set, file_weight_set):

-- parameter interface_input_set imports the interface input dataflows
-- parameter interface_output_set imports the interface output dataflows
-- parameter interface_inquiry_set imports the interface inquiry dataflows
-- parameter passed_file_set imports the interface passed files (external data stores)
-- parameter shared_file_set imports the interface shared data store connections
-- parameter input_weight_set exports the weights of the interface input dataflows
-- parameter output_weight_set exports the weights of the interface output dataflows
-- parameter inquiry_weight_set exports the weights of the interface inquiry dataflows
-- parameter interface_file_weight_set exports the weights of the external interface files
-- parameter file_weight_set exports the weights of the logical internal files

weights_of_interface_inputs(interface_input_set, input_weight_set, interface_file_weight_set);
weights_of_interface_outputs(interface_output_set, output_weight_set, interface_file_weight_set);
weights_of_interface_inquiries(interface_inquiry_set, inquiry_weight_set, interface_file_weight_set);
weights_of_interface_passed_files(passed_file_set, interface_file_weight_set, file_weight_set);
weights_of_interface_shared_files(shared_file_set, interface_file_weight_set).
```

---

[1] See Algorithm 2.1.

[2] See Algorithm 2.2.

[3] Algorithm 2.5 is a part of TUFPV(SA) in Figure 1.4.

```
procedure weights_of_interface_inputs (interface_input_set, input_weight_set,
interface_file_weight_set):

-- parameter interface_input_set imports the interface input dataflows
-- parameter input_weight_set exports the weights of the interface input dataflows
-- parameter interface_file_weight_set exports the weights of the interface files
-- local variable $m_1$ counts the number of interface input dataflows in interface_input_set
-- local variable f denotes the interface input dataflow in interface_input_set
-- local variable e denotes an external application
-- local variable p denotes a primitive process
-- local variable logical_file_set contains all logical files referenced by f
-- local variable n counts the number of internal logical files referenced by f
-- local variable s counts the number of data items of f
-- local variable w counts the weight of f
-- remark: input_weight_set is initialized in the main program
-- remark: interface_file_weight_set is initialized in the main program

$m_1$ := |interface_input_set|;
for all $m_1$ interface input dataflows f=(e,p) with external application e and process p in
interface_input_set do
        logical_file_set : = Ø;
        find_logical_files_of_inputs(p,logical_file_set);
        n : = |logical_file_set|;
        s : = number_of_data_items(f);
        w : = input_weight(n,s);
        input_weight_set : = input_weight_set ∪ w;
        if f is converted  then
                n : = number_of_logical_record_formats(f);
                w : = interface_file_weight(n,s);
                interface_file_weight_set : = interface_file_weight_set ∪ w.
```

```
procedure weights_of_interface_outputs(interface_output_set, output_weight_set,
interface_file_weight_set):

-- parameter interface_output_set imports the interface output dataflows
-- parameter output_weight_set exports the weights of the interface output dataflows
-- parameter interface_file_weight_set exports the weights of the interface files
-- local variable $m_2$ counts the number of interface output dataflows in interface_output_set
-- local variable f denotes the interface output dataflow in interface_output_set
-- local variable e denotes an external application
-- local variable p denotes a primitive process
-- local variable logical_file_set contains all logical files referenced by f
-- local variable n counts the number of internal logical files or the number of logical record formats of f
-- local variable s counts the number of data items of f
-- local variable w counts the weight of f
-- remark: output_weight_set is initialized in the main program
-- remark: interface_file_weight_set is initialized in the main program

$m_2$ := |interface_output_set|
for all $m_2$ interface output dataflows f=(p,e) with process p and external application e in
interface_output_set do
        logical_file_set : = Ø;
        find_logical_files_of_outputs(p,logical_file_set);
        n : = |logical_file_set|;
        s : = number_of_data_items(f);
        w : = output_weight(n,s);
        output_weight_set : = output_weight_set ∪ w;
        if f is converted then
                n : = number_of_logical_record_formats(f);
                w : = interface_file_weight(n,s);
                interface_file_weight_set : = interface_file_weight_set ∪ w.
```

```
procedure weights_of_interface_inquiries(interface_inquiry_set, inquiry_weight_set,
```

```
interface_file_weight_set):

-- parameter interface_inquiry_set imports the inquiry dataflows
-- parameter inquiry_weight_set exports the weights of the inquiry dataflows
-- parameter interface_file_weight_set exports the weights of the interface files
-- local variable $m_3$ counts the number of interface inquiry dataflows in interface_inquiry_set
-- local variable f denotes the interface inquiry dataflow in interface_inquiry_set
-- local variable $f_I$ denotes the input part of f
-- local variable $f_O$ denotes the output part of f
-- local variable e denotes an external application
-- local variable p denotes a primitive process
-- local variable logical_file_set contains all logical files referenced
-- local variable $n_I$ counts the number of internal logical files for $f_I$
-- local variable $s_I$ counts the number of data items of $f_I$
-- local variable $w_I$ counts the weight of $f_I$
-- local variable w counts the interface file weight of $f_I$ or $f_O$
-- local variable n counts the number of internal logical files or the number of logical record formats of f
-- local variable $n_O$ counts the number of internal logical files for $f_O$
-- local variable $s_O$ counts the number of data items of $f_O$
-- local variable $w_O$ counts the weight of $f_O$
-- remark: interface_file_weight_set is initialized in the main program
-- remark: inquiry_weight_set is initialized in the main program

$m_3$ := |interface_inquiry_set|;
**for** all $m_3$ interface inquiry dataflows f = [$f_I$,$f_O$] in interface_inquiry_set **do**
        select input part $f_I$ = (e,p) from inquiry dataflow f;
        logical_file_set : = Ø;
        find_logical_files_of_inputs(p, logical_file_set);
        $n_I$ : = |logical_file_set|;
        $s_I$ : = number_of_data_items($f_I$);
        $w_I$ : = input_weight($n_I$, $s_I$);
        **if** $f_I$ is converted **then**
                n : = number_of_logical_record_formats($f_I$);
                w : = interface_file_weight(n,$s_I$);
                interface_file_weight_set : = interface_file_weight_set ∪ w;
        select output part $f_O$ = (p,e) from inquiry dataflow f;
        logical_file_set : = Ø;
        find_logical_files_of_outputs (p, logical_file_set);
        $n_O$ : = |logical_file_set|;
        $s_O$ : = number_data_items($f_O$);
        $w_O$ : = output_weight($n_O$, $s_O$);
        w : = max($w_I$, $w_O$);
        inquiry_weight_set : = inquiry_weight_set ∪ w;
        **if** $f_O$ is converted **then**
                n : = number_of_logical_record_formats($f_O$);
                w : = interface_file_weight(n,$s_O$);
                interface_file_weight_set : = interface_file_weight_set ∪ w.

**procedure** weights_of_interface_passed_files(passed_file_set, file_weight_set,
interface_file_weight_set):

-- parameter passed_file_set imports the interface passed files (external data stores)
-- parameter interface_file_weight_set exports the weights of the interface files
-- parameter file_weight_set exports the weights of the internal logical files
-- local variable $p_1$ counts the number of passed files in passed_file_set
-- local variable d denotes the passed file in passed_file_set
-- local variable r counts the number of relationships for d
-- local variable s counts the number of data items of d
-- local variable w counts the weight of d
-- remark: interface_file_weight_set is initialized in the main program
-- remark: file_weight_set is initialized in the main program
$p_1$ := |passed_file_set|;
```

```
for all $p_1$ passed files d in passed_file_set do
    r : = number_of_relationships(d);
    s : = number_of_data_items(d);
    w : = interface_file_weight(r, s);
    interface_file_weight_set : = interface_file_weight_set ∪ w;
    if d is sent to external application e then
        w : = file_weight(r,s);
        file_weight_set : = file_weight_set ∪ w.
```

```
procedure weights_of_interface_shared_files(shared_file_set, interface_file_weight_set):

-- parameter shared_file_set imports the interface shared files
-- parameter interface_file_weight_set exports the weights of the interface files
-- local variable $p_2$ counts the number of shared files in shared_file_set
-- local variable d denotes the shared file
-- local variable logical_record_set contains all logical records of d
-- local variable logical_file_set contains all logical files of d
-- local variable r counts the number of relationships for d
-- local variable l counts the number of logical record formats for <d,k>
-- local variable unique_data_item_set contains the unique data items collected from <d,c,k> of <d,k>
-- local variable n counts the maximum of r and l
-- local variable s counts the number of data items of d
-- local variable w counts the weight of <d,k>
-- remark: interface_file_weight_set is initialized in the main program

$p_2$ := |shared_file_set|;
for all $p_2$ shared files d in shared_file_set  do
    r : = number_of_relationships(d);
    logical_record_set : = Ø;
    logical_file_set := Ø;
    find_logical_records_and_files (d, logical_record_set, logical_file_set);
    for all logical files <d,k> in logical_file_set do
        l := 0;
        unique_data_item_set := Ø;
        for all logical record formats <d,c,k> in logical_record_set do
            if <d,c,k> is a logical record format of logical file <d,k> then
                l := l+1;
                unique_data_item_set := unique_data_item_set ∪ data_items(<d,c,k>);
        n : = max(r,l);
        s : = |unique_data_item_set|;
        w : = interface_file_weight(n, s);
        interface_file_weight_set : = interface_file_weight_set ∪ w.
```

```
procedure find_logical_records_and_files (d, logical_record_set, logical_file_set):

-- parameter d imports the data store to be processed
-- parameter logical_record_set exports the logical records of d
-- parameter logical_file_set exports the logical internal files of d
-- local variable f denotes a shared file connection
-- local variable e denotes an external application

for all dataflows f=(e,d) or f=(d,e) where e is an external application and d is a shared file do
    if f is named then
        if <d,f,key(f)> ∉ logical_record_set then
            logical_record_set := logical_record_set ∪ <d,f,key(f)>;
        if <d,key(f)> ∉ logical_file_set then
            logical_file_set := logical_file_set ∪ <d,key(f)>
    else
        if <d,d,key(d)> ∉ logical_record_set then
            logical_record_set := logical_record_set ∪ <d,d,key(d)>;
        if <d,key(d)> ∉ logical_file_set then
            logical_file_set := logical_file_set ∪ <d,key(d)>.
```

As an example, consider Figure 2.12. If a conversion is required in Figure 2.12(a) with each transaction file, there is one output, one input, one inquiry, and three interfaces; therefore, the number of unadjusted function points is at least 26. If there are no conversions, the number of unadjusted function points is at least 11. In Figure 2.12(b) we have an example of four master file connections. Two of them are shared file connections and two are passed files. Now the number of unadjusted function points for interfaces is at least 20, provided that the shared file connections reference different logical groups of user data. In addition, both D1 and D2 produces at least seven points as a file.

2.3. Summary

According to the principles presented in Section 2.2, TUFPV(SA) can be evaluated from SA specifications. This is the first, hence earliest possible, stage of software development process when SA is being used. Moreover, the presented method has been automated (Laamanen & Rask 1991) and can therefore be implemented in CASE-tools. Figure 2.13 summarizes the components needed to count TUFPV(SA).[1]

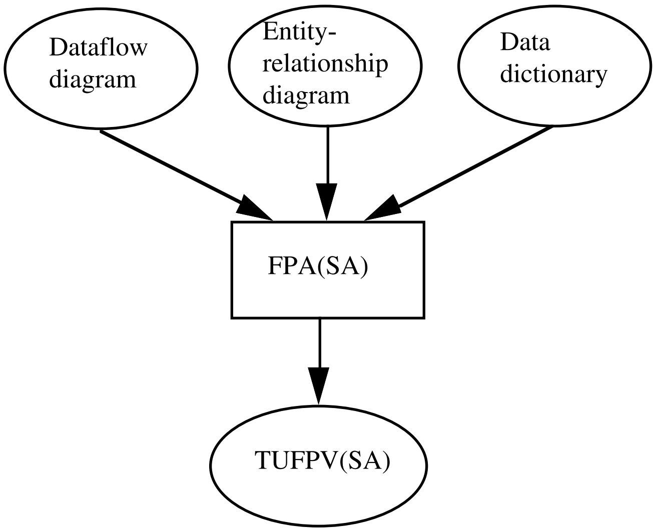


Figure 2.13. Components for counting unadjusted function points from SA.

Although the use of unadjusted function points in estimating product size has been criticized as being too complex (e.g. Banker et al. 1990b), the requirement to simplify the FPA method is not fully acceptable. There is no reason to simplify the method if one can auto-

---

[1] See also Figure 1.4.

mate the original method in an efficient way. This automation has been done by the previous algorithms.

In Chapter 5 the reliability of our method will be examined by means of empirical studies which compare it with other metrics and with results obtained by applying the methods of Chapters 3 and 4. In Chapter 3 algorithms will be formed to count unadjusted function points from JSD descriptions. By doing this an attempt will be made to try to show the generality of the FPA method. In Chapter 4 transformations between JSD and SA methodologies will be investigated. By doing this an attempt will be made to increase the credibility of the FPA algorithms developed in Chapters 2 and 3. In addition, a step is taken forward developing a method-independent environment for software development.

3. APPLICATION OF ALBRECHT'S FPA TO JSD SPECIFICATIONS

One alternative to the Structured Analysis (SA) method is the *Jackson System Development* (JSD) method. While SA starts with a functional description, JSD starts with entities, entity structures and communication between these entities. Afterwards the system is supplemented by functions. The result is a specification which consists of an upper-level network and lower-level tree descriptions of processes in the network.

To calculate function points all user functions must be found from the specification diagram. The exact automation of a TUFPV calculation from the JSD specification has not earlier been reported[1]. Before the algorithms are given which will enable this automation, the notation of JSD will be examined in Section 3.1.

3.1. Components of the JSD Specification Diagram

A JSD *specification diagram* (Cameron 1989, Jackson 1983) contains *processes* of a system and *connections* between processes and from processes to the real world. The processes may be connected to each other either by dataflows or by state vector connections. Figure 3.1 shows the basic symbols used.

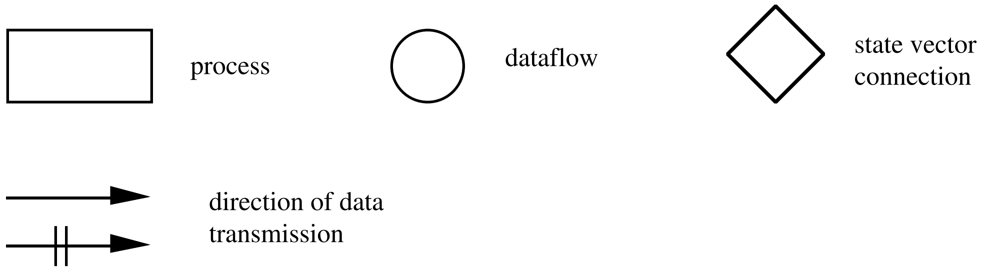


Figure 3.1. Basic symbols of the JSD specification diagram.

A *dataflow* is written by one process and it is read by another in the same order. A *state vector connection* means that one process reads some or all attributes of the state vector of another process. We assume that a function process does not contain a separate state vec-

---

[1] There is another approach for applying FPA within a JSD environment (Ratcliff & Rollo 1990). That approach, however, is a simplification of FPA. In particular, the concept of a logical internal file and the complexity classification principles are different from the approach used here.

tor.[1] We assume also that each dataflow and state vector connection is named. The *state vector* of a process may be interpreted as a data store of the process. In general, the real world entities are connected to the *model processes* by dataflows. The model processes are also connected to each other. The *function processses,* which produce outputs of a system, are often connected to the model processes by state vectors. Figure 3.2 shows two model processes (P and Q) and one function process (R); there is also a typical feedback (dataflow E) from function process R to model process Q.

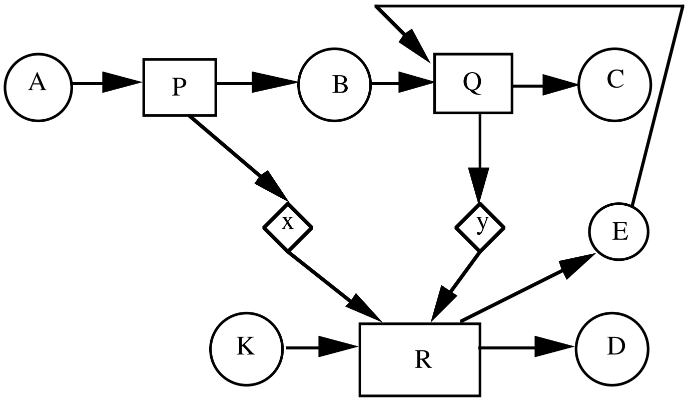


Figure 3.2. Example of a JSD specification diagram.

As with a SA dataflow diagram, it is possible to merge two or more input dataflows (Figure 3.3). Here dataflows D and E from real world entities P and Q are merged as an input to the model process R.

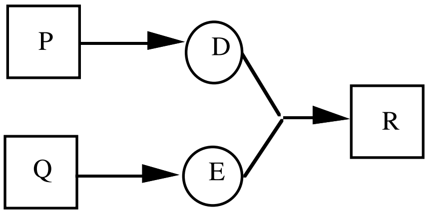


Figure 3.3. Example of a merged dataflow (Jackson 1983).

When a specification diagram is drawn, the structure of each model process is known as a *structure diagram* which consists of three component types: sequence, selection and itera-

---

[1] Alhough a function process could include a state vector, the data items of the state vector are either written to the output dataflow, or to a model process. If the receiver is a model process, the data items are included into its state vector.

tion.[1] The structure diagram of a model process P orders the actions P_i corresponding to the actions carried out by a real world entity. Actions P_i receive input records from the real world entity. The input records can be described by one input dataflow or by separate dataflows. The structure components (corresponding to the actions of a model process) of a function process show the structure of the output information. If, for example, process P in Figure 3.2 means a customer of a bank, its structure with actions INVEST, PAY_IN, WITHDRAW and TERMINATE, could be modelled as shown in Figure 3.4.

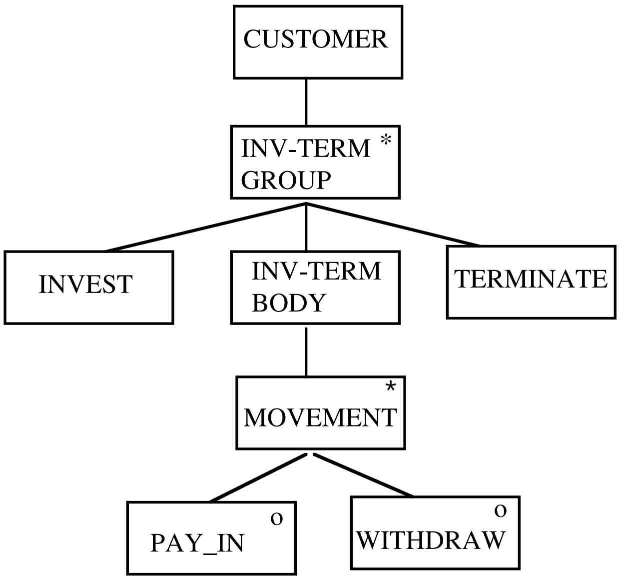


Figure 3.4. Example of a JSD structure diagram (Jackson 1983).

Because of the structure diagrams, a JSD specification diagram can also be said to depict a procedural description. However, JSD cannot reasonably be called either a top-down method or a bottom-up method (Cameron 1986). The network is not built by successive explosions of a process into a subnetwork of processes. It proceeds "middle-out" from an initial set of model processes.

The two-level structure of JSD descriptions makes possible two different interpretations for communication between processes. At the network level the processes communicate with each other. At the process level the actions of the separate processes communicate with each other. At the network level one can say that process P writes dataflow *f* to process Q. At the process level this means that the actions P_i of P write the records f_j of *f* to the actions Q_j of process Q. Correspondingly, the state vectors are inspected by actions of

---

[1] A selection type is denoted by "o" and an iteration type by "*"; otherwise it is a sequence type.

the inspecting processes. For example, action P_i of P can inspect the state vector QSV of model process Q via state vector connection *t*.[1]

3.2. Identification and Weighting of User Functions

A *transaction* in a specification diagram consists of a set of model processes and function processes which are directly chained to one another and to the real world via dataflows, and model processes which are connected to the chain via state vector connection. Accordingly, the state vector is read by one of the chained processes. A transaction is activated by an input dataflow.[2]

In Figure 3.2 there were two transactions. The first contains processes P and Q and dataflows A, B, and C. The other transaction contains all the processes, dataflows C, D, E and K, and state vector connections *x* and *y*.

In the following FPA counting rules will be applied to JSD specification diagrams.[3] To do this, the principles of the algorithms presented in Chapter 2 for SA descriptions will be utilized.[4] The transaction definition above is needed to find the referenced files for external inputs, outputs, and inquiries.[5]

3.2.1. External Input and Output Types

An *input* in a JSD specification is directed to a model process or to a function process[6] in the form of one or more dataflow records. Attributes of the corresponding input dataflow records are interpreted as *referenced data items*. To find the referenced files, the possible dataflow chains must be traversed forward within the transaction. The state vectors of the referenced model processes inside the transaction are the basis for referenced files.

Logical internal files of a referenced model process are determined by the keys referencing the model processes. Each incoming dataflow $f_j=(P_i,Q_j)$ received by an action Q_j of a

---

[1] The JSD discussion is continued in Chapter 4.

[2] This definition is derived from the corresponding transaction definition for DFDs in Section 2.2.

[3] Assume that the structure diagrams and the communication between actions at the process level are known.

[4] The pseudocode notation of the algorithms is summarized in Appendix H.

[5] All the complexity classification factors and weights for function types are tabulated in Appendix B.

[6] The function process in this case does not immediately provide a response to the user. Instead, it is scheduled to produce a report based on input data.

model process Q from an action P_i of a process P or from the real world is assumed to contain a key. The notation <Q,key(f_j)> is used for a referenced logical internal file. Other referenced logical internal files referenced by an input dataflow are those incoming state vector connections used by the chained processes of the input transaction. Each state vector connection contains a key. The notation <V,key(t)> is used for a referenced logical internal file when the inspected model process is V and the state vector connection is *t*.[1]

In Figure 3.5, for instance, inputs A (five data items) and K (one data item) both reference "files" CUSTOMER and ACCOUNT. If dataflow B did not connect the processes CUSTOMER and ACCOUNT, input A would reference only a "file" CUSTOMER. Accordingly, the total number of unadjusted function points for these two inputs is seven.[2]

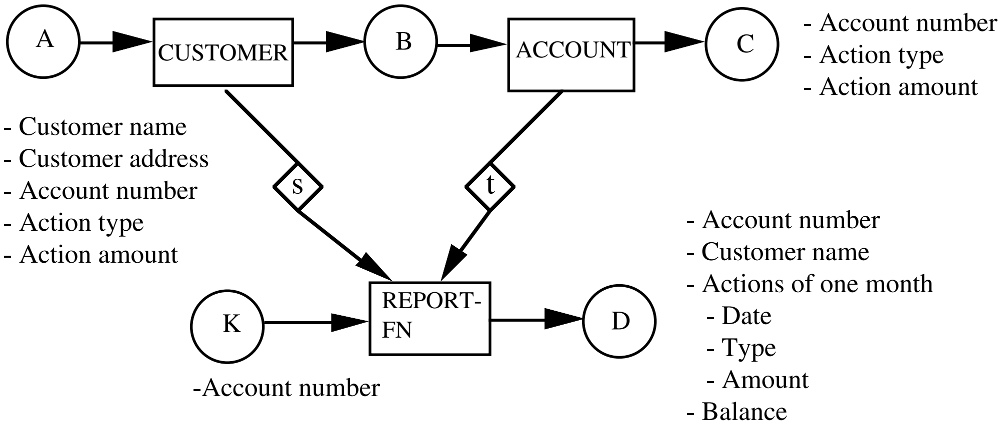


Figure 3.5. Example of JSD inputs and outputs.

The above described principles for finding and weighting of external inputs from JSD specification diagrams are formalized in Algorithm 3.1, which can be used to automate the counting process. The complexity of procedure weights_of_inputs_JSD is $O(m)$ and the complexity of procedure find_logical_files_of_inputs is $O(n)$ where the number of input dataflow records is m and the total number of dataflows and state vector connections in the processed specification diagram is n. The for-loops of the procedure weights_of_inputs_JSD go through all input records and in the worst case the procedure find_logical_files_of_inputs inspects all other dataflows and state vector connections per each input dataflow. Thus, the complexity of Algorithm 3.1 is $O(mn)$.

---

[1] See also Section 3.2.3.

[2] Here it is assumed that both of the input dataflows contain only one record. In practice, however, input dataflow A should contain four different records or it should be divided into four subdataflows corresponding to the actions in Figure 3.4. To find the exact number of logical internal files would require more information about process and dataflow structures.

Table 3.1 lists the procedures used in Algorithm 3.1, the complexity of each procedure (excluding work done in called procedures), and the calling structure of these procedures. The parameters and local variables are explained within each procedure of Algorithm 3.1. Local functions and procedures, and special notations are explained in Appendix H.

Table 3.1. Procedures, complexity, and calling structure of Algorithm 3.1.

Procedure	Complexity	Calls
weights_of_inputs_JSD	O(m)	find_logical_files_of_inputs
find_logical_files_of_inputs	O(n)	

m = number of input dataflow records
n = number of all dataflows and state vector connections

Algorithm 3.1. Finding and weighting of inputs in JSD.[1]

```
procedure weights_of_inputs_JSD (input_set, input_weight_set):

-- parameter input_set imports the input dataflows
-- parameter input_weight_set exports the weights of the input dataflows
-- local variable k counts the number of input dataflows in input_set
-- local variable f denotes the input dataflow to be weighted
-- local variable e denotes the starting point (external entity) for f
-- local variable p denotes the ending point (process) for f
-- local variable $f_i$ denotes a record of f
-- local variable $p_i$ denotes an action of p
-- local variable logical_file_set contains all logical internal files referenced by $f_i$
-- local variable n counts the number of internal logical files referenced by $f_i$
-- local variable s counts the number of data items of $f_i$
-- local variable w counts the weight of $f_i$
-- remark: at the beginning all dataflows are unmarked
-- remark: input_weight_set is initialized in the main program

k:= |input_set|;
for all k input dataflows f=(e,p) with external entity e and process p in input_set do
        for all separate input records $f_i$ =(e,$p_i$) do
                logical_file_set : = Ø;
                if f is connected to a model process p then
                        if <p,key($f_i$)> ∉ logical_file_set then
                                logical_file_set : = logical_file_set ∪ <p,key($f_i$)>;
                find_logical_files_of_inputs ($p_i$, logical_file_set);
                n : = |logical_file_set|;
                s : = number_of_data_items($f_i$);
                w : = input_weight(n, s);
                input_weight_set : = input_weight_set ∪ w.
```

---

[1] Algorithm 3.1 is a part of TUFPV(JSD) in Figure 1.4.

```
procedure find_logical_files_of_inputs ($p_i$, logical_file_set):

-- parameter $p_i$ imports the address of each process action of the transaction
-- parameter logical_file_set exports the internal logical files for the input dataflow
-- local variable $f_j$ denotes a dataflow within the transaction
-- local variable $q_j$ denotes a model or a function process action
-- local variable $p_i$ denotes a model or a function process action
-- local variable t denotes a state vector connection
-- local variable v denotes a model process in a state vector connection t
-- local variable p denotes the process with actions $p_i$
-- local variable q denotes the process with actions $q_j$

for all incoming state vector connections $t=(v,p_i)$ do
        if v is a model process and t is unmarked then
                mark(t);
                if <v, key(t)> $\notin$ logical_file_set then
                        logical_file_set : = logical_file_set $\cup$ <v,key(t)>;
for all outgoing dataflows $f_j=(p_i, q_j)$ do
        if q is a model process and $f_j$ is unmarked then
                mark($f_j$);
                if <q, key($f_j$)> $\notin$ logical_file_set then
                        logical_file_set : = logical_file_set $\cup$ <q, key($f_j$)>;
                find_logical_files_of_inputs ($q_j$, logical_file_set)
        elsif q is a function process and $f_j$ is unmarked then
                mark($f_j$);
                find_logical_files_of_inputs ($q_j$, logical_file_set).
```

With JSD each dataflow (record) leaving the application boundary should be counted as an *output*. The output dataflow is written either by a model process which contains an internal function or, in general, by a distinct function process. For example, the model process ACCOUNT in Figure 3.5 contains an internal function which produces account information into dataflow C. Often the function process is connected by incoming state vector connections to one or more model processes as shown by the function process REPORT-FN with incoming state vector connections *s* and *t* in Figure 3.5.

Attributes of the output dataflows (records) are interpreted as *referenced data items*. The state vectors of the referenced model processes are interpreted as *referenced files*. The keys which determine the logical internal files are found as presented in connection with the input types. But now we must traverse backwards along the process chains bounded by the output transaction. The key for a logical internal file <Q,key(f_j)> is found from the incoming dataflow $f_j=(P_i,Q_j)$ produced by an action P_i of process P and used by an action Q_j of process Q.

Algorithm 3.2 can be used to find and weight outputs from JSD specification diagrams. The algorithm is almost the same as Algorithm 3.1. The only exception is that in Algorithm 3.2 we must go backwards through the dataflows, i.e. go through incoming dataflows to the chained processes. Correspondingly, the complexity of Algorithm 3.2 is also $O(mn)$ where the number of output dataflow records is m and the total number of dataflows and state vector connections in the processed specification diagram is n.

Table 3.2 lists the procedures used in Algorithm 3.2, the complexity of each procedure (excluding work done in called procedures), and the calling structure of these procedures. The parameters and local variables are explained within each procedure of Algorithm 3.2. Local functions and procedures as well as special notations are explained in Appendix H.

Table 3.2. Procedures, complexity, and calling structure of Algorithm 3.2.

Procedure	Complexity	Calls
weights_of_outputs_JSD	O(m)	find_logical_files_of_outputs
find_logical_files_of_outputs	O(n)	

m = number of output dataflow records
n = number of all dataflows and state vector connections

Algorithm 3.2. Finding and weighting of outputs in JSD.[1]

```
procedure weights_of_outputs_JSD (output_set, output_weight_set):

-- parameter output_set imports the output dataflows
-- parameter output_weight_set exports the weights of the output dataflows
-- local variable k counts the number of output dataflows in output_set
-- local variable f denotes the output dataflow to be weighted
-- local variable q denotes a starting (process) point for f
-- local variable $q_j$ denotes an action of a model or a function process q
-- local variable e denotes the ending point (external entity) for f
-- local variable logical_file_set contains all logical files referenced by f
-- local variable n counts the number of internal logical files referenced by f
-- local variable s counts the number of data items of f
-- local variable w counts the weight of f
-- remark: at the beginning all dataflows are unmarked
-- remark: output_weight_set is initialized in the main program

k:= |output_set|;
for all k output dataflows f=(q,e) with process q and external entity e in output_set do
        for all separate output records $f_j$ =($q_j$,e) do
                logical_file_set : = Ø;
                find_logical_files_of_outputs ($q_j$, logical_file_set);
                n : = |logical_file_set|;
                s : = number_of_data_items($f_j$);
                w : = output_weight(n, s);
                output_weight_set : = output_weight_set ∪ w.


procedure find_logical_files_of_outputs ($q_j$, logical_file_set):
```

---

[1] Algorithm 3.2 is a part of TUFPV(JSD) in Figure 1.4.

```
-- parameter $q_j$ imports the address of each process action of the transaction
-- parameter logical_file_list exports the internal logical files for the output dataflow
-- local variable $f_j$ denotes a dataflow between two actions within the transaction
-- local variable $p_i$ denotes a model or a function process action in a dataflow connection $f_j$
-- local variable q denotes a process with actions $q_j$
-- local variable $q_j$ denotes a model or a function process action in a dataflow connection $f_j$
-- local variable t denotes a state vector connection
-- local variable v denotes a model process in a state vector connection t

for all incoming state vector connections $t=(v,q_j)$ do
        if v is a model process and t is unmarked then
                mark(t);
                if $<v, key(t)> \notin$ logical_file_set then
                        logical_file_set : = logical_file_set $\cup$ $<v,key(t)>$;
for all incoming dataflows $f_j=(p_i,q_j)$ do
        if q is a model process and $f_j$ is unmarked then
                mark($f_j$);
                if $<q, key(f_j)> \notin$ logical_file_set then
                        logical_file_set : = logical_file_set $\cup$ $<q, key(f_j)>$;
                find_logical_files_of_outputs ($p_i$, logical_file_set);
        elsif q is a function process and $f_j$ is unmarked then
                mark($f_j$);
                find_logical_files_of_outputs ($p_i$, logical_file_set).
```

As an example, in Figure 3.5 there are two external outputs which both reference to CUSTOMER and ACCOUNT "files". The number of data items in output dataflow C is three and the number of data items in output dataflow D is six. According to Dreger (1989), the balance of output dataflow D should be counted as a unique output, because it requires different processing logic.[1] This has been taken into account by counting separate output dataflow records. On this basis there are three unique outputs. Assuming that the number of logical internal files for both of the model processes is one, the number of unadjusted function points for outputs will be 12.

As with Structured Analysis, some data items can be missing, due to the later design of the user interface, but it is essential that all input and output functions can be found while calculating the value of unadjusted function points for purposes of estimating project cost.

3.2.2. External Inquiry Types

When function points are counted from a JSD specification diagram, an *inquiry* function cannot contain a feedback dataflow to a model process. Such a situation would be interpreted as an update[2], not an inquiry. If needed, however, it is possible to connect separate

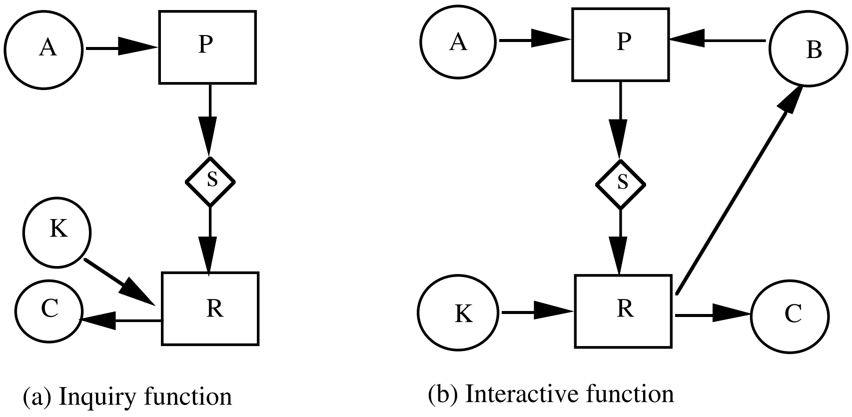
---

[1] See also footnote 2 on page 30.

[2] Uppdating is possible for the input type in FPA. See Section 3.2.1.

function processes by a dataflow because it is assumed that a function process does not contain a separate state vector.

Figure 3.6(a) presents a typical JSD inquiry function (R) with state vector connection (s) to a model process (P). In Figure 3.6(b) we have an "interactive" function (R) with an additional feedback dataflow connection (B) to a model process (P).



(a) Inquiry function (b) Interactive function

Figure 3.6. Inquiry and interactive functions.

As with SA, an inquiry dataflow must be presented by an *inquiry symbol* (Figure 3.7). If an inquiry consists of two different (input and output) dataflows as in Figure 3.6(a), it is difficult to differentiate inquiry types from input and output types.

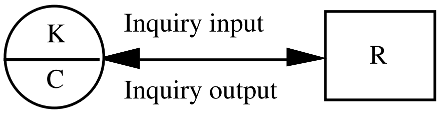


Figure 3.7. Proposal for an inquiry symbol.

To find the *file references* and the number of *data items referenced*, the inquiry dataflow is divided into separate input and output parts, which are then weighted as presented with input and output types, with two restrictions. An inquiry transaction cannot update a state vector of a model process; and based on the idea of inquiry functions, it is reasonable to allow only one inquiry input record and one inquiry output record per inquiry transaction.

The above described principles for finding and weighting of external inquiries from JSD specification diagrams are formalized in Algorithm 3.3, which can be used to automate the counting process. The complexity of Algorithm 3.3 is $O(2mn) = O(mn)$ where the total number of dataflows and state vector connections in the processed specification diagram is *n*, and the number of inquiry dataflows is *m*.

Table 3.3 lists the procedures used in Algorithm 3.3, the complexity of each procedure (excluding work done in called procedures), and the calling structure of these procedures. The parameters and local variables are explained within each procedure of Algorithm 3.3. Local functions and procedures as well as special notations are explained in Appendix H.

Table 3.3. Procedures, complexity, and calling structure of Algorithm 3.3.

Procedure	Complexity	Calls
weights_of_inquiries_JSD	O(m)	find_logical_files_of_inputs[1] find_logical_files_of_outputs[2]
find_logical_files_of_inputs	O(n)	
find_logical_files_of_outputs	O(n)	

m = number of inquiry dataflows
n = number of all dataflows and state vector connections

Algorithm 3.3. Finding and weighting of inquiries in JSD.[3]

```
procedure weights_of_inquiries_JSD (inquiry_set, inquiry_weight_set):

-- parameter inquiry_set imports the inquiry dataflows
-- parameter inquiry_weight_set exports the weights of the inquiry dataflows
-- local variable logical_file_set contains all logical files referenced
-- local variable m counts the number of inquiry dataflows in inquiry_set
-- local variable f denotes the inquiry dataflow to be weighted
-- local variable $f_I$ denotes the input part of f
-- local variable $f_O$ denotes the output part of f
-- local variable e denotes an external entity
-- local variable p denotes a process
-- local variable w counts the weight of f
-- local variable $n_I$ counts the number of internal logical files for $f_I$
-- local variable $s_I$ counts the number of data items of $f_I$
-- local variable $w_I$ counts the weight of $f_I$
-- local variable $n_O$ counts the number of internal logical files for $f_O$
-- local variable $s_O$ counts the number of data items of $f_O$
-- local variable $w_O$ counts the weight of $f_O$
-- remark: inquiry_weight_set is initialized in the main program


m := |inquiry_set|;
for all m inquiry dataflows f = [$f_I$,$f_O$] in inquiry_set do
```

---

[1] See Algorithm 3.1.

[2] See Algorithm 3.2.

[3] Algorithm 3.3 is a part of TUFPV(JSD) in Figure 1.4. See also footnote 2 on page 32.

```
select input part $f_I = (e,p)$ from inquiry dataflow f with external entity e and process p;
logical_file_set : = $\emptyset$;
find_logical_files_of_inputs (p, logical_file_set)
$n_I$ : = |logical_file_set|;
$s_I$ : = number_of_data_items($f_I$);
$w_I$ : = input_weight($n_I$, $s_I$);
select output part $f_O = (p,e)$ from inquiry dataflow f with process p and external entity e;
logical_file_set : = $\emptyset$;
find_logical_files_of_outputs (p, logical_file_set)
$n_O$ : = |logical_file_set|;
$s_O$ : = number_of_data_items($f_O$);
$w_O$ : = output_weight($n_O$, $s_O$);
w : = max($w_I$, $w_O$);
inquiry_weight_set : = inquiry_weight_set $\cup$ w.
```

As an example, the number of unadjusted function points for the inquiry in Figure 3.6(a) is at least four when the weights recommended by Dreger (1989) are used.

3.2.3. Logical Internal File Types

An internal file type is represented in a JSD specification diagram by the state vector of a model process. A state vector is normally updated by the incoming dataflows and used by the outgoing state vector connections to other function or model processes. Keys by which the data of a model process are accessed must be contained in these dataflows and state vector connections. Accordingly, the number of keys used is equal to the number of *logical internal files* included in each model process.

For a logical internal file the notation <P,key(x)> is used, where P is a model process and *x* is either a dataflow, a state vector of process P, or an outgoing state vector connection for P[1]. Each logical internal file contains one or more *logical record formats*. These are determined by the name and key of incoming dataflows or outgoing state vector connections. For the logical record formats the notation <P,x,key(x)> is used, where P and *x* are defined as above.

In FPA the *relationships* between files affect the number of unadjusted function points of file types. Thus, as discussed in Section 2.2.3, the entity model of the application must be known. This model should direct the naming of JSD entities, i.e. model processes, because thereby the relationships can be found with ease in the entity model.

---

[1] Shared file connections to external applications are excluded. Based on Dreger (1989), only those data items actually required by the application have to be taken into account. See also Section 2.2.3.

Cameron (1988) gives an alternative interpretation for the relationships by claiming that a JSD model contains entities and attributes explicitly, and relationships implicitly. In Cameron's approach, each foreign key is reinterpreted as a relationship. A foreign key is an attribute of one entity which can act as an identifier for another entity. A JSD model, therefore, contains a simple and well-structured data model.

The number of *referenced data items* can be found for each logical internal file as a union of the data items for the corresponding logical records.

The above described principles for finding and weighting of logical internal files from JSD specification diagrams and corresponding ER-diagrams are formalized in Algorithm 3.4, which can be used to automate the counting process. Part of the information it uses is from an ER-diagram. The complexity of Algorithm 3.4 in the worst case is $O(z_1((z_1-1)+z_2+(z_2)^2)) = O(z_1(z_2)^2)$, where the number of "internal" model processes[1] in the specification diagram is z_1 and the number of actions is z_2. In the worst case all actions of the specification per model process are processed and there is a relationship between each model process pair. As it is reasonable to assume that $z_2 > z_1$, the complexity can be said to be in the worst case $O(z_1(z_2)^2)$.

Table 3.4 lists the procedures used in Algorithm 3.4, the complexity of each procedure (excluding work done in called procedures), and the calling structure of these procedures. The parameters and local variables are explained within each procedure of Algorithm 3.4. Local functions and procedures as well as special notations are explained in Appendix H.

Table 3.4. Procedures, complexity, and calling structure of Algorithm 3.4.

Procedure	Complexity	Calls
weights_of_logical_files_JSD	$O(z_1(z_2)^2+(z_1)^2-z_1))$	find_logical_records_and_files
find_logical_records_and_files	$O(z_2)$	

z_1 = number of model processes
z_2 = number of all actions

Algorithm 3.4. Finding and weighting of logical internal files in JSD.[2]

---

[1] The passed file model processes are excluded here. They are explained in Section 3.2.4 and weighted by Algorithm 3.5.

[2] Algorithm 3.4 is a part of TUFPV(JSD) in Figure 1.4.

```
procedure weights_of_logical_files_JSD(model_process_set, file_weight_set):

-- parameter model_process_set imports the model processes excluding "passed files" (see Section 3.2.4)
-- parameter file_weight_set exports the weights of the internal logical files
-- local variable $z_1$ counts the number of model processes in model_process_set
-- local variable q denotes the model process to be divided into logical files
-- local variable r counts the number of relationships for q
-- local variable logical_record_set contains all logical records of q
-- local variable logical_file_set contains all logical internal files of q
-- local variable l counts the number of logical record formats for a logical file <q,k>
-- local variable unique_data_item_set contains the unique data items collected from <q,c,k> of <q,k>
-- local variable n counts the maximum of r and l
-- local variable s counts the number of data items of a logical file <q,k>
-- local variable w counts the weight of a logical file <q,k>
-- remark: file_weight_set is initialized in the main program

$z_1$ := |model_process_set|;
for all $z_1$ "internal" model processes q in model_process_set do
        r : = number_of_relationships(q);
        logical_record_set : = Ø;
        logical_file_set : = Ø;
        find_logical_records_and_files(q, logical_record_set, logical_file_set);
        for all logical files <q,k> in logical_file_set do
                l : = 0;
                unique_data_item_set := Ø;
                for all logical record formats <q,c,k> in logical_record_set do
                        if <q,c,k> is a logical record format of logical file <q,k> then
                                l := l+1;
                                unique_data_item_set := unique_data_item_set ∪ data_items(<q,c,k>);
                n : = max(r,l);
                s : = |unique_data_item_set|;
                w : = file_weight(n, s);
                file_weight_set : = file_weight_set ∪ w.
```

```
procedure find_logical_records_and_files(q, logical_record_set, logical_file_set):

-- parameter q imports the address of the model process in the specification diagram
-- parameter logical_record_set exports the logical records for q
-- parameter logical_file_set exports the logical files for q
-- local variable $f_j$ denotes a dataflow from action $p_i$ to action $q_j$
-- local variable $p_i$ denotes an action of p
-- local variable $q_j$ denotes an action of q
-- local variable $v_k$ denotes an action of process v
-- local variable t denotes a state vector connection from q
-- local variable e denotes an external application

for all incoming dataflows $f_j=(p_i,q_j)$, excluding the shared file connections to external applications do
        if $<q,f_j,key(f_j)> \notin$ logical_record_set then
                logical_record_set : = logical_record_set ∪ $<q,f_j,key(f_j)>$;
        if $<q,key(f_j)> \notin$ logical_file_set then
                logical_file_set : = logical_file_set ∪ $<q,key(f_j)>$;
for all outgoing state vector connections $t=(q,v_k)$, excluding the shared file connections to external
applications do
        if <q,t,key(t)> ∉ logical_record_set then
                logical_record_set : = logical_record_set ∪ <q,t,key(t)>;
        if <q,key(t)> ∉ logical_file_set then
                logical_file_set : = logical_file_set ∪ <q,key(t)>.
```

As an example, in Figure 3.8 three logical (internal) files are connected to the model process P. Each file has a different key. The model process Q operates only on one logical internal file. Thus, there are four different logical internal files for these two model processes.

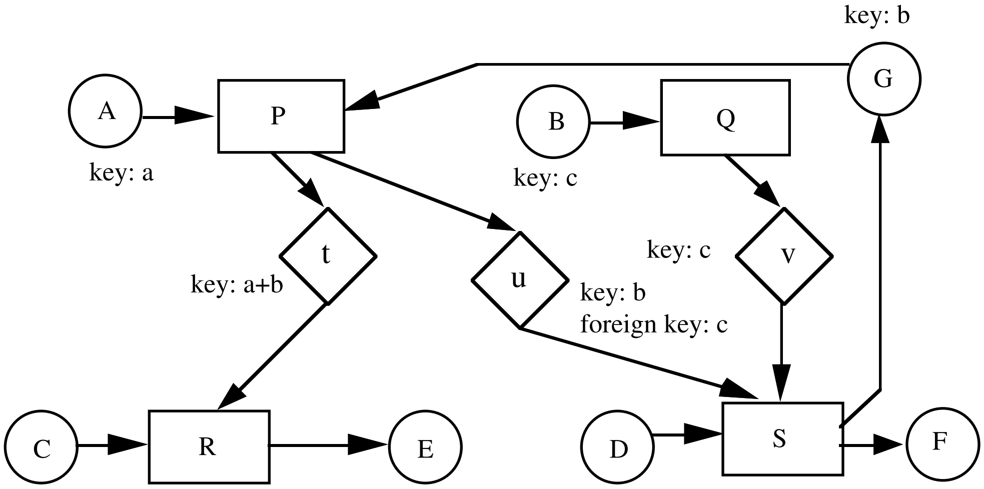


Figure 3.8. Logical internal files in a specification diagram.

In Figure 3.8 there is also one logical relationship between processes P and Q based on the state vector connections *u* and *v*, which form a relationship by means of the foreign key *c*. This relationship is connected to a part of P having an attribute *b* as the key. Hence the number of unadjusted function points for the logical files is at least 28 when the number of referenced data items for the logical files is not known.

3.2.4. External Interface File Types

A *transaction file* is presented in a JSD specification diagram by a dataflow between applications. It is not possible to describe explicitly the conversion of transaction files in a specification diagram without exact process descriptions. It is, however, possible to handle the conversion in the same way as with dataflow diagrams in Section 2.5 by comparing the input and output dataflows of the transaction process that processes the transaction dataflow.

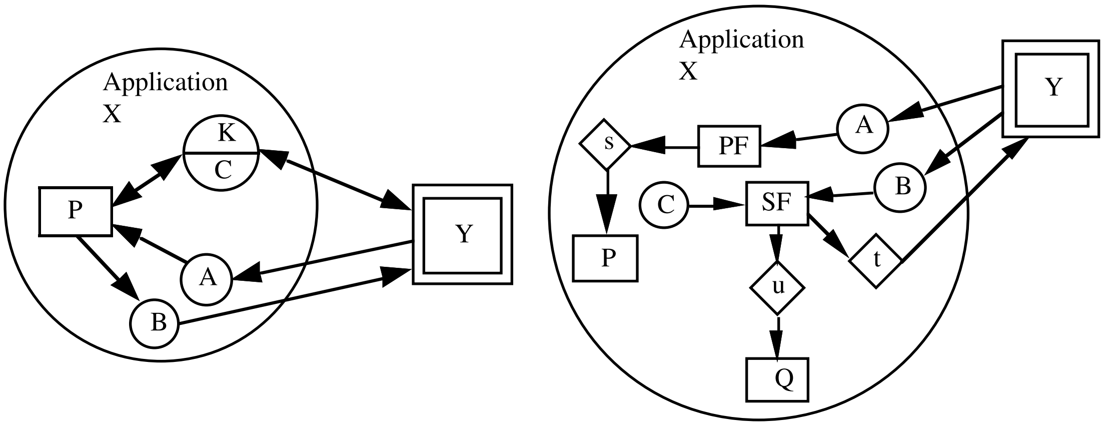
A *shared master file* is presented in a JSD specification diagram by a model process with an outgoing state vector connection to the external application and with an incoming dataflow from the external application. Both applications can use and update the shared master file.

To find a concept equivalent to the notion of external data store in SA, it is necessary to describe a *master file passed* from an external application by a combination of a dataflow and a special model process that receives the dataflow and serves as a "data store" with its state vector. For a master file passed to the external application we could use a corresponding model process and dataflow combination. It is not reasonable in JSD to present an external "data store" explicitly at the level of the specification diagram, because it would require a new symbol and would also be against the philosophy of JSD.

Thus, we have four types of dataflows in JSD: "normal" dataflows, transaction dataflows, passed file dataflows, and dataflows that update shared files. The types of dataflows must be given in the data dictionary.[1]

For an external application the same symbol, *double rectangle*, will be applied, as was proposed for a dataflow diagram in Section 2.2.4. Consequently, the interface file types can be represented in a specification diagram as illustrated in Figure 3.9.

Figure 3.9(a) contains three transaction file connections between application X and the external application Y. The external application Y writes dataflow A to the model process P, which in turn writes dataflow B as a transaction file to the external application Y. The third transaction is an inquiry dataflow connection.



(a) Transaction file interface

(b) Shared file (SF) and passed file (PF) interfaces

Figure 3.9. Principles of interfaces.

---

[1] In JSD it is also possible to inspect and maintain the state vector of an "external" model process. In this thesis, however, an external application is interpreted more broadly.

Figure 3.9(b) shows one passed file connection between the external application Y and the object application X, on the other hand, and one shared file, on the other. The combination of special model process PF with dataflow A is interpreted as a passed master file. The state vector of process PF could contain, for example, account numbers sent by the external application Y. The process P then uses these account numbers.

The shared file SF in Figure 3.9(b) is used by the process Q via a state vector connection u and by the external application Y via a state vector connection t. The shared file SF is also updated by both applications (dataflows B and C).

The above described principles for finding and weighting of external interface files from JSD specification diagrams and corresponding ER-diagrams are formalized in Algorithm 3.5, which can be used to automate the counting process. The main procedure is named weights_of_interface_JSD. It calls subprocedures that find and weight the interface inputs, interface outputs, interface inquiries, interface external files, and interface shared files. The subprocedures also use the procedures of the previous algorithms to find logical files.

The complexity of Algorithm 3.5 in the worst case is $O(m_1n+m_2n+2m_3n+t_1(z_1-1)+t_2(z_1-1+m_4+m_4^2)) = O(n(m_1+m_2+m_3)+z_1(t_1+t_2)+t_2(m_4+m_4^2)-t_1-t_2)$ where the number of model processes in the specification diagram is z_1, the number of all dataflows and state vector connections is n, 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 passed file connections is t_1, the number of shared file connections to external applications is m_4, and the number of shared model processes is t_2.

Table 3.5 lists the procedures used in Algorithm 3.5, the complexity of each procedure (excluding work done in called procedures), and the calling structure of these procedures. The parameters and local variables are explained within each procedure of Algorithm 3.5. Local functions and procedures as well as special notations are explained in Appendix H.

Table 3.5. Procedures, complexity, and calling structure of Algorithm 3.5.

Procedure	Complexity	Calls
weights_of_interfaces_JSD	O(5)	weights_of_interface_inputs weights_of_interface_outputs weights_of_interface_inquiries weights_of_interface_passed_files weights_of_interface_shared_files
weights_of_interface_inputs	$O(m_1)$	find_logical_files_of_inputs[1]
weights_of_interface_outputs	$O(m_2)$	find_logical_files_of_outputs[2]
weights_of_interface_inquiries	$O(m_3)$	find_logical_files_of_inputs[1] find_logical_files_of_outputs[2]
weights_of_interface_passed_files	$O(t_1(z_1-1))$	
weights_of_interface_shared_files	$O(t_2(z_1-1+(m_4)^2))$	find_logical_records_and_files
find_logical_files_of_inputs	O(n)	
find_logical_files_of_outputs	O(n)	
find_logical_records_and_files	$O(m_4)$	

m_1 = number of interface input dataflow records
m_2 = number of interface output dataflow records
m_3 = number of interface inquiry dataflows
m_4 = number of shared file interface connections
n = number of all dataflows and state vector connections
t_1 = number of passed model process connections
t_2 = number of shared model process connections
z_1 = number of model processes

Algorithm 3.5. Finding and weighting of external interface files in JSD.[3]

```
procedure weights_of_interfaces_JSD(interface_input_set, interface_output_set, interface_inquiry_set,
external_file_set, shared_file_set, input_weight_set,output_weight_set, inquiry_weight_set,
interface_file_weight_set, file_weight_set):

-- parameter interface_input_set imports the interface input dataflows
-- parameter input_weight_set exports the weights of the interface input dataflows
-- parameter interface_output_set imports the interface output dataflows
-- parameter output_weight_set exports the weights of the interface output dataflows
-- parameter interface_inquiry_set imports the interface inquiry dataflows
-- parameter inquiry_weight_set exports the weights of the interface inquiry dataflows
-- parameter passed_file_set imports the interface passed file connections
-- parameter interface_file_weight_set exports the weights of the interface files
-- parameter shared_file_set imports the interface shared file connections
-- parameter file_weight_set exports the weights of the internal logical files

weights_of_interface_inputs(interface_input_set, input_weight_set, interface_file_weight_set);
weights_of_interface_outputs(interface_output_set, output_weight_set, interface_file_weight_set);
weights_of_interface_inquiries(interface_inquiry_set, inquiry_weight_set, interface_file_weight_set);
weights_of_interface_passed_files(passed_file_set, file_weight_set, interface_file_weight_set);
weights_of_interface_shared_files(shared_file_set, interface_file_weight_set).


procedure weights_of_interface_inputs (interface_input_set, input_weight_set,
```

---

[1] See Algorithm 3.1.

[2] See Algorithm 3.2.

[3] Algorithm 3.5 is a part of TUFPV(JSD) in Figure 1.4.

```
interface_file_weight_set):

-- parameter interface_input_set imports the interface input dataflows
-- parameter input_weight_set exports the weights of the interface input dataflows
-- parameter interface_file_weight_set exports the weights of the transaction files
-- local variable $k_1$ counts the number of interface input dataflows in interface_input_set
-- local variable f denotes the interface input dataflow in interface_input_set
-- local variable e denotes an external application
-- local variable p denotes a process
-- local variable $f_i$ denotes a record of f
-- local variable $p_i$ denotes an action of p
-- local variable logical_file_set contains all logical internal files referenced by $f_i$
-- local variable n counts the number of logical internal files referenced by $f_i$
-- local variable s counts the number of data items of $f_i$
-- local variable w counts the weight of $f_i$
-- remark: input_weight_set is initialized in the main program
-- remark: interface_file_weight_set is initialized in the main program

$k_1$ := |interface_input_set|;
for all $k_1$ interface input dataflows f=(e,p) with external application e and process p in
interface_input_set do
        for all separate interface input records $f_i$=(e,$p_i$) do
                logical_file_set : = Ø;
                if f is connected to a model process p then
                        logical_file_set : = logical_file_set ∪ <p,key($f_i$)>;
                find_logical_files_of_inputs ($p_i$, logical_file_set);
                n : = |logical_file_set|;
                s : = number_of_data_items(f);
                w : = input_weight(n, s);
                input_weight_set : = input_weight_set ∪ w;
                if $f_i$ is converted then
                        n : = number_of_logical_record_formats($f_i$);
                        w : = interface_file_weight(n,s);
                        interface_file_weight_set : = interface_file_weight_set ∪ w.
```

```
procedure weights_of_interface_outputs(interface_output_set, output_weight_set,
interface_file_weight_set):

-- parameter interface_output_set imports the interface output dataflows
-- parameter output_weight_set exports the weights of the interface output dataflows
-- parameter interface_file_weight_set exports the weights of the transaction files
-- local variable $k_2$ counts the number of interface output dataflows in interface_output_set
-- local variable f denotes the interface output dataflow in interface_output_set
-- local variable $f_j$ denotes a record of f
-- local variable e denotes an external application
-- local variable q denotes a process
-- local variable $q_j$ denotes an action of q
-- local variable logical_file_set 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 $f_j$
-- local variable w counts the weight of $f_j$
-- remark: output_weight_set is initialized in the main program
-- remark: interface_file_weight_set is initialized in the main program

$k_2$ := |interface_output_set|;
for all $k_2$ interface output dataflows f=(q,e) with external application e and process q in
interface_output_set do


        for all separate output records $f_j$=($q_j$,e) do
                logical_file_set : = Ø;
```

```
        find_logical_files_of_outputs ($q_j$, logical_file_set);
        n : = |logical_file_set|;
        s : = number_of_data_items($f_j$);
        w : = output_weight(n, s);
        output_weight_set : = output_weight_set ∪ w;
        if $f_j$ is converted then
            n : = number_of_logical_record_formats($f_j$);
            w : = interface_file_weight(n,s);
            interface_file_weight_set : = interface_file_weight_set ∪ w.
```

```
procedure weights_of_interface_inquiries(interface_inquiry_set, inquiry_weight_set,
interface_file_weight_set):

-- parameter interface_inquiry_set imports the inquiry dataflows
-- parameter inquiry_weight_set exports the weights of the inquiry dataflows
-- parameter interface_file_weight_set exports the weights of the transaction files
-- local variable $k_3$ counts the number of interface inquiry dataflows in interface_inquiry_set
-- local variable f denotes the interface inquiry dataflow in interface_inquiry_set
-- local variable $f_I$ denotes the input part of f
-- local variable $f_O$ denotes the ouput part of f
-- local variable e denotes an external application
-- local variable p denotes a process
-- local variable logical_file_set contains all logical files referenced
-- local variable w counts the weight of f
-- local variable $n_I$ counts the number of internal logical files for $f_I$
-- local variable $s_I$ counts the number of data items of $f_I$
-- local variable $w_I$ counts the weight of $f_I$
-- local variable $n_O$ counts the number of internal logical files for $f_O$
-- local variable $s_O$ counts the number of data items of $f_O$
-- local variable $w_O$ counts the weight of $f_O$
-- remark: inquiry_weight_set is initialized in the main program
-- remark: interface_file_weight_set is initialized in the main program

$k_3$ := |interface_inquiry_set|;
for all $k_3$ interface inquiry dataflows f = [$f_I$,$f_O$] in interface_inquiry_set do
    select input part $f_I$ = (e,p) from f with external application e and process p;
    logical_file_set : = Ø;
    find_logical_files_of_inputs (p, logical_file_set);
    $n_I$ : = |logical_file_set|;
    $s_I$ : = number_of_data_items($f_I$);
    $w_I$ : = input_weight($n_I$, $s_I$);
    if $f_I$ is converted then
        n : = number_of_logical_record_formats($f_I$);
        w : = interface_file_weight(n,$s_I$);
        interface_file_weight_set : = interface_file_weight_set ∪ w;
    select output part $f_O$ = (p,e) from inquiry dataflow f with process p and external application e;
    logical_file_set : = Ø;
    find_logical_files_of_outputs (p, logical_file_set);
    $n_O$ : = |logical_file_set|;
    $s_O$ : = number_of_data_items($f_O$);
    $w_O$ : = output_weight($n_O$, $s_O$);
    w : = max($w_I$, $w_O$);
    inquiry_weight_set : = inquiry_weight_set ∪ w;
    if $f_O$ is converted then
        n : = number_of_logical_record_formats($f_O$);
        w : = interface_file_weight(n,$s_O$);
        interface_file_weight_set : = interface_file_weight_set ∪ w.
```

```
procedure weights_of_interface_passed_files(passed_file_set, file_weight_set,
interface_file_weight_set):
```

```
-- parameter passed_file_set imports the interface passed file connections
-- parameter interface_file_weight_set exports the weights of the interface files
-- parameter file_weight_set exports the weights of the logical internal files
-- local variable $t_1$ counts the number of passed files in passed_file_set
-- local variable c denotes the passed file connection
-- local variable p denotes a passed file (model process) in c
-- local variable e denotes an external application in c
-- local variable f denotes a dataflow between p and e in c
-- local variable r counts the number of relationships for p
-- local variable s counts the number of data items of p
-- local variable w counts the weight of p
-- remark: interface_file_weight_set is initialized in the main program
-- remark: file_weight_set is initialized in the main program

$t_1$ := |passed_file_set|;
for all $t_1$ passed file connections c=(p,f,e) or c=(e,f,p), where p is a model process corresponding to the
passed file, e is an external application and f is a dataflow between p and e in passed_file_set do
        r : = number_of_relationships(p);
        s : = number_of_data_items(p);
        w : = interface_file_weight(r, s);
        interface_file_weight_set : = interface_file_weight_set ∪ w;
        if c=(p,f,e) (i.e. p is sent to external application e) then
                w : = file_weight(r,s);
                file_weight_set : = file_weight_set ∪ w.


procedure weights_of_interface_shared_files(shared_file_set, interface_file_weight_set):

-- parameter shared_file_set imports the interface shared file connections
-- parameter interface_file_weight_set exports the weights of the interface files
-- local variable $t_2$ counts the number of shared files in shared_file_set
-- local variable q denotes the shared file in c
-- local variable r counts the number of relationships for q
-- local variable logical_record_set contains all logical records of q
-- local variable logical_file_set contains all logical files of q
-- local variable l counts the number of logical record formats for the logical file determined by key(c)
-- local variable n counts the maximum of r and l
-- local variable s counts the number of data items of the shared file q
-- local variable w counts the weight of the logical file <q,key(c)>
-- remark: interface_file_weight_set is initialized in the main program

$t_2$ := |shared_model_process_set|;
for all $t_2$ shared model processes q in shared_model_process_set do
        r : = number_of_relationships(q);
        logical_record_set : = Ø;
        logical_file_set : = Ø;
        find_logical_records_and_files(q, logical_record_set, logical_file_set);
        for all logical files <q,k> in logical_file_set do
                l : = 0;
                unique_data_item_set := Ø;
                for all logical record formats <q,c,k> in logical_record_set do
                        if <q,c,k> is a logical record format of logical file <q,k> then
                                l := l+1;
                                unique_data_item_set := unique_data_item_set ∪ data_items(<q,c,k>);
                n : = max(r,l);
                s : = |unique_data_item_set|;
                w : = interface_file_weight(n, s);
                interface_file_weight_set : = interface_file_weight_set ∪ w.

procedure find_logical_records_and_files(q, logical_record_set, logical_file_set):

-- parameter q imports the address of the shared model process in the specification diagram
```

```
-- parameter logical_record_set exports the logical records for q
-- parameter logical_file_set exports the logical files for q
-- local variable e denotes an external application
-- local variable $q_j$ denotes an action of q
-- local variable $f_j$ denotes a dataflow from e to $q_j$
-- local variable t denotes a state vector connection from q to e

for all incoming dataflows $f_j=(e,q_j)$ from external applications e do
        if $<q,f_j,key(f_j)> \notin$ logical_record_set then
                logical_record_set : = logical_record_set $\cup <q,f_j,key(f_j)>$;
        if $<q,key(f_j)> \notin$ logical_file_set then
                logical_file_set : = logical_file_set $\cup <q,key(f_j)>$;
for all outgoing state vector connections t=(q,e) to external applications e do
        if $<q,t,key(t)> \notin$ logical_record_set then
                logical_record_set : = logical_record_set $\cup <q,t,key(t)>$;
        if $<q,key(t)> \notin$ logical_file_set then
                logical_file_set : = logical_file_set $\cup <q,key(t)>$.
```

As an example, the minimum number of unadjusted function points for the transaction files in Figure 3.9(a) is 11 assuming that no conversions are needed. For the passed file and shared file in Figure 3.9(b) at least 15 unadjusted function points as interfaces can be calculated, provided that B and *t* reference different logical groups of user data.

3.3. Summary

In this chapter it has been shown that FPA can also be applied to JSD specification diagrams. The principles are quite similar to SA descriptions when the model processes in JSD are interpreted to correspond to data stores in SA.

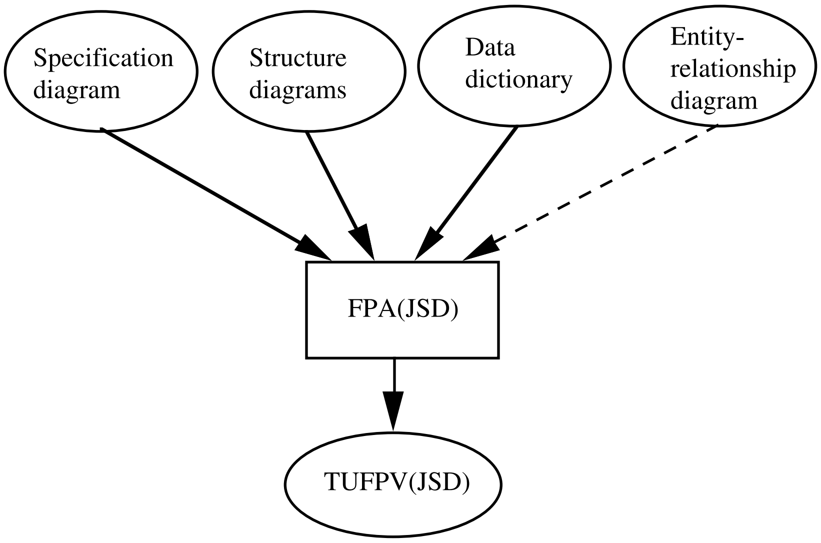


Figure 3.10. Components for counting unadjusted function points in JSD.

The interpretation of the interfaces in JSD are not as clear as in SA. In particular, the difference between transaction files and passed master files between applications is not easy to recognize. Figure 3.10 summarizes the components needed to automate the counting process.[1] The entity-relationship diagram has been used in these algorithms, although, as discussed in Section 3.2.3, this is not obligatory. This optionality is shown by a broken line in Figure 3.10.

Although the JSD method is not currently popular in CASE-tools, it is much referenced in the literature and probably will become more common in the future. This is why JSD was selected as another object method for FPA in this thesis. The FPA algorithms for JSD have not yet been implemented but the graphical interface for JSD has been programmed (Hyttinen 1990). This makes the implementation possible in the near future.

4. METHODICAL EQUIVALENCE

---

[1] See also Figure 1.4.

During the system analysis phase, a logical system description can be formed by the Jackson System Development method (JSD), by Structured Analysis (SA), or by some other method. If two or more of these methods are used, or if it is desirable to move from a method to another, the common features of these methods must be understood. This understanding can provide a basis for transforming descriptions from one method to another (Rask 1990b).

Two methods are said to be *equivalent* if they can be used to describe the same basic functions. If the corresponding basic components of one method can be described by using another method, it is possible to convert the component descriptions from the first method to the other. In Section 4.1 the equivalence between JSD and SA is shown. The purpose of showing the equivalence between these two methods is to increase the credibility of the FPA algorithms introduced in Chapters 2 and 3 and to help achieve method-independent environments for software development.

4.1. Transformation between JSD and SA

There are certain similarities between JSD specification diagrams and dataflow diagrams if the component types (sequence, selection and iteration) of JSD structure diagrams and the "conditionality" (* and ⊕) of SA dataflows are excluded.[1] The state vector of a JSD model process is described by a data store in a dataflow diagram. The dataflows in these two methods correspond to each other. If the actions in JSD processes are not taken into account, the JSD processes can be interpreted as SA processes.

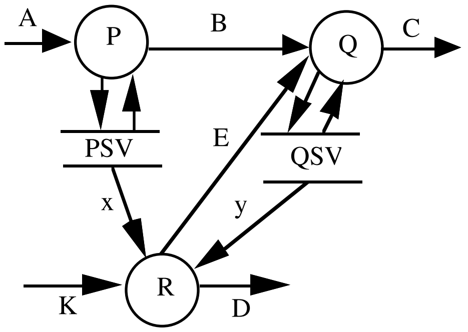


Figure 4.1. Example of a dataflow diagram after transformation.

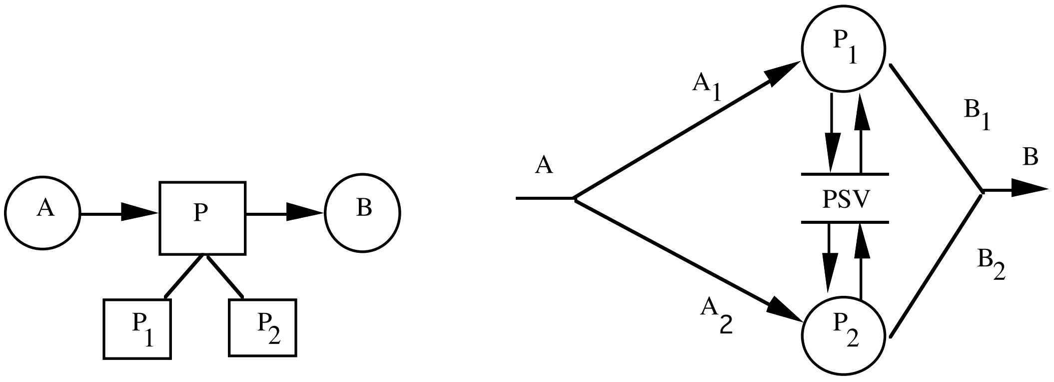
---

[1] See Sections 2.1 and 3.1.

Figure 4.1 gives an example of transforming the JSD specification diagram of Figure 3.2 into a dataflow diagram. In the following two sections, transformation from JSD to SA and transformation from SA to JSD will be considered.[1]

4.1.1. Transformation from JSD to SA

An *action* of a structure diagram in the JSD method can be transformed to a *functional primitive* in a dataflow diagram of the SA method. After this transformation the dataflow in the resulting dataflow diagram must be divided between the processes corresponding to the actions in a JSD structure diagram. Figure 4.2 gives an example of interpreting a specification diagram (which contains a structure diagram) as a dataflow diagram.



(a) JSD diagram: one model process (P) with two actions (P_1 and P_2).

(b) SA dataflow diagram: two processes (P_1, P_2) and one data store corresponding to the JSD state vector.

Figure 4.2. Transformation of a JSD specification to an SA dataflow diagram.

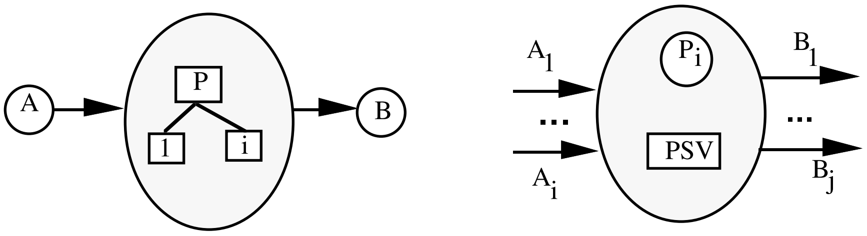
The JSD diagram in Figure 4.2(a) contains one model process (P), an input dataflow A, an output dataflow B, and two actions of P. In the corresponding SA dataflow diagram (Figure 4.2(b)) the input dataflow A has been diverged into two branches A_1 and A_2, and it is assumed that both P_1 and P_2 write and read the data store PSV (the original state vector of the model process P). Finally, the decomposed output dataflow has been combined into one dataflow B supposing that both SA processes, P_1 and P_2, write an output record.

It is also possible that the dataflow A in Figure 4.2 is not diverged during the transformation, because it can be decomposed into separate records. Thus one incoming dataflow A_i per SA process P_i can be drawn. In practice, some of the JSD actions (SA processes) P_i

---

[1] The notations are partly based on personal communication with Associate Professor Jorma Sajaniemi.

do not write a record (dataflow). Accordingly, the number of incoming dataflows can differ from the number of outgoing dataflows. It is possible to examine which of the processes P_i should write and which only read the data store PSV. This examination can be based on the structure text of each model process. In general, each input and output dataflow of a model process can be interpreted as described in Figure 4.3.



(a) JSD component

(b) SA component

Figure 4.3. Basic structures of external input/output transformations for a model process.

When two model processes P and Q are connected to one dataflow B, the process decomposition principle shown in Figure 4.3 can be used. The divided dataflows B_j from processes P_i must, however, be connected to processes Q_k. Figure 4.4 shows the result of this transformation. Here it is assumed that a JSD process P has i actions which receive h input records, and j of the actions produce a record into dataflow B. These j records are consumed by process Q producing l records into dataflow C.

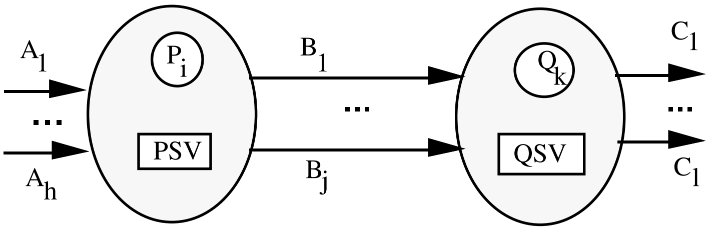


Figure 4.4. Transformation of model processes connected with one dataflow.

It is possible that a JSD process P has more than one incoming dataflow or more than one outgoing dataflow. For example, when there is a feedback dataflow E from model process Q back to model process P, the previous model in Figure 4.4 acquires the form presented in Figure 4.5. The feedback dataflow E references the data store PSV of the receiving SA component. An extra process P_{i+1} may be needed to manage this new dataflow. If none of the Q_k processes produces the feedback dataflow E, an extra producer Q_{k+1} is also needed.

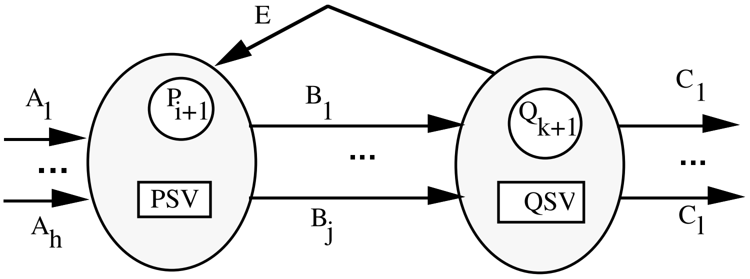
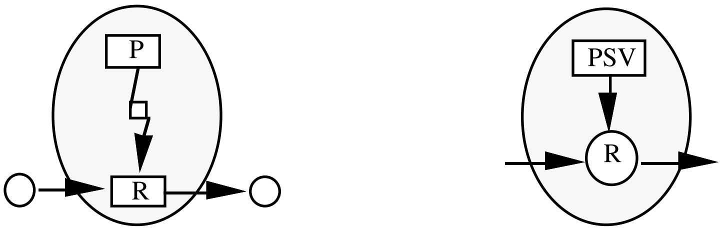


Figure 4.5. Feedback between model processes.

If there is a state vector connection between two model processes, i.e. from a model process P to a model process Q, the state vector is replaced by a dataflow from the data store PSV to an action process that corresponds to the "reading" action in the JSD diagram. In this situation no separate connection is established to data store QSV[1].

Connections between two function processes, and between a function process and a model process, are treated similarly to that depicted for model processes. When needed, a function process can be decomposed into separate actions. A function process should not, however, contain its own state vector[2]. As can be seen in Figure 4.6, where the connection of a function process to a model process via a state vector connection is generalized, these cases are equivalent. The state vector connection in JSD is replaced by a dataflow in SA, and the external inputs and outputs are equivalent.



(a) JSD component

(b) SA component

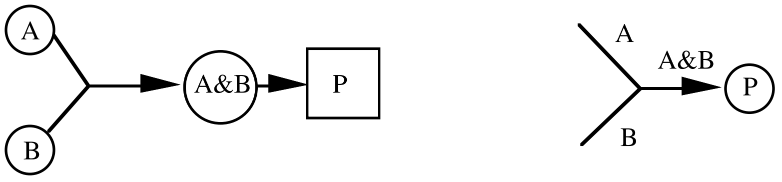
Figure 4.6. Connection of a function process to a model process.

In JSD it is also possible to merge two or more dataflows. This is equivalent to the principles shown in Figures 2.3 and 3.3. Figure 4.7 summarizes the equivalence of the principles, indicating how to merge dataflows A and B into one combined dataflow A&B.

---

[1] In Algorithm 3.4 this possibility is also excluded by excluding incoming state vector connections when the logical records and files were defined. It is assumed that the state vector of a model process is uppdated by incoming dataflows.

[2] See also Section 3.2.



(a) Merging in JSD (B) Merging in SA

Figure 4.7. Merging the dataflows.

Now the basic component transformations from JSD to SA have been illustrated. When the component transformations are known, these components can be *combined* to obtain more complicated structures. This is illustrated in Figure 4.8 which generalizes Figure 4.1.

The dataflow diagram that takes into account the structure of processes in a specification diagram can be understood as a *refinement* of a dataflow diagram which corresponds to a specification diagram without process structures. An example of the top level transformation was depicted in Figure 4.1.

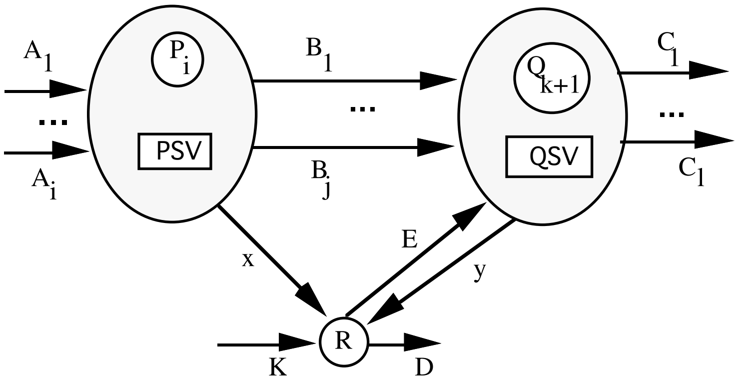


Figure 4.8. Generalization of Figure 4.1.

Algorithm 4.1 gathers together the principles for transforming a JSD specification into a dataflow diagram of SA. The main procedure transform_JSD_to_SA first performs the basic transformations and then calls for the specific transformation procedures. Clearly, the complexity of Algorithm 4.1 is polynomial. To use the algorithm, the communication between processes at the action level has to be known.

Algorithm 4.1. Transformation from JSD to SA.[1]

```
procedure transform_JSD_to_SA(JSD, SA):

-- parameter JSD imports the JSD specification diagram including actions
-- parameter SA exports tha dataflow diagram of Structured Analysis
-- remark: JSD and SA structures are understood to be global for the procedures below
-- remark: the meaning of each local variable is implicitly clear

for all external entities E in JSD do
        introduce a terminator T in SA;
for all processes P in JSD do
        for all actions $P_i$ of P in JSD do
                introduce a functional primitive, an action process, $P_i$ in SA;
        if P is a model process then
                introduce a data store PSV in SA;
for all pairs (A,P) where P is a process and A is an input dataflow from an external entity in JSD do
        manage_inputs(A,P);
for all pairs (P,B) where P is a process and B is an outgoing dataflow in JSD do
        manage_outputs(P,B);
for all state vector connections S between process P and process R in JSD do
        substitute S by a dataflow from the data store PSV to the required action
        process $R_i$ of R in SA.

procedure manage_inputs(A,P):

-- parameter A imports the current input dataflow used by the process P
-- parameter P imports the current model or function process using the input dataflow A

for all input records $A_i = (E,P_i)$ in JSD do
        form an input dataflow $A_i = (E,P_i)$ for the action process $P_i$ in SA;
        if P is a model process then
                if $A_i$ updates PSV then
                        connect $P_i$ into the data store PSV.

procedure manage_outputs(P,B):

-- parameter P imports the current process producing the outgoing dataflow B
-- parameter B imports the current outgoing dataflow produced by the process P

for all actions $P_i$ of P in JSD do
        for all output records $B_j$ of $P_i$ in JSD do
                if the destination of $B_j$ is an external entity E then
                        form an output dataflow $B_j = (P_i,E)$ in SA
                elsif the destination of $B_j$ is a model process Q then
                        form an incoming dataflow $B_j = (P_i,Q_k)$ for the action process $Q_k$ in SA;
                        if $B_j$ updates QSV then
                                connect $Q_k$ into the data store QSV
                elsif the destination of $B_j$ is a function process F then
                        form an incoming dataflow $B_j = (P_i,F_k)$ for the action process $F_k$ in SA;
                if $B_j$ uses PSV then
                        connect the data store PSV to $P_i$.
```

4.1.2. Transformation from SA to JSD

---

[1] Algorithm 4.1 does not explicitly deal with the features of FPA.

The opposite transformation - from a SA dataflow diagram to a JSD specification diagram - is more informal, demanding more application knowledge. It is not necessarily known which entities are carrying out the actions described in a dataflow diagram; in addition, SA-actions do not always correspond to JSD-actions - they may also be JSD-functions. Therefore, it is possible to draw different JSD specification diagrams from one dataflow diagram, if not enough is known about the application. An example of this is given in Figures 4.9 and 4.10.

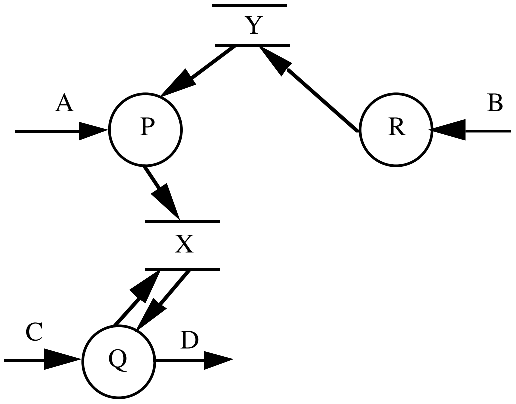


Figure 4.9. Example of a dataflow diagram.

In Figure 4.10 two interpretations of Figure 4.9 are demonstrated. The first interpretation has two model processes (X and Y) corresponding to the data stores in Figure 4.9. The model process X has two actions (P and Q) connected with it, and the model process Y has one action (R).

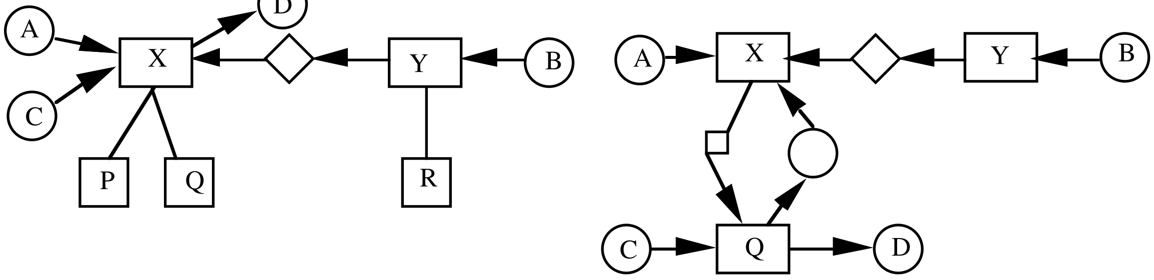


Figure 4.10. Two interpretations of the dataflow diagram in Figure 4.9.

The second interpretation in Figure 4.10 also has two model processes (X and Y), but action Q has been replaced by a function process Q. In this example nothing is known about the application or the meaning of the processes in Figure 4.9. Accordingly, it is impossible to say which of the solutions in Figure 4.10 is correct. However, it is possible to do the transformation, when we have sufficient knowledge of the application.

4.2. Summary

In this chapter it has been shown that a JSD specification can be interpreted as an SA specification and, with some restrictions, vice versa. The transformation from JSD to SA can be done algorithmically. The transformation from SA to JSD requires more knowledge. With current knowledge engineering technology it is not impossible to formalize all of this knowledge so that the human analyst can be completely freed from intervening in the transformation process.

The purpose of the transformations in this thesis is to increase the credibility of the FPA algorithms introduced in Chapters 2 and 3. In the following chapter the structure presented in Figure 1.4 will be further verified.

In general, to increase the productivity of software projects, it should be possible to transform dynamically from one method to another. This transformation facility should be one of the functions of a CASE system. Current CASE-tools do not, however, support this process.

5. VALIDATION OF THE FPA ALGORITHMS AND THE TRANSFORMATION RULES

In Chapters 2 and 3 algorithms to calculate the value of Albrecht's (1979) unadjusted function points from SA and JSD descriptions were given. In Chapter 4 an attempt was made to improve the credibility of Algorithms 2.1-2.5 and Algorithms 3.1.-3.5 by showing the methodical equivalence between SA and JSD. In this chapter the usability of the algorithms is further analyzed by reporting on empirical experiments.

In Section 5.1 the results given by Algorithms 2.1-2.5 are verified. This is made by analyzing correlations between the results of Algorithms 2.1-2.5 and two other specification level metrics: DeMarco's (1982) Function Bang metric[1] and Symons's (1988; 1991) Mark II metric[2]. It is shown that the results of the analysis are in line with our general knowledge of the characteristics of the metrics. The experiment is carried out on two different test materials.

In Section 5.2 Algorithms 3.1-3.5 are verified in two experiments. It is now assumed that the results given by Algorithms 2.1-2.5 for SA descriptions in Section 5.1 are correct. The SA test material is transformed by hand to JSD and Algorithms 3.1-3.5 are applied to the JSD specifications. The experiment is then repeated with another test material, but now the JSD specifications are produced and measured first; after that corresponding SA descriptions are produced and measured. The purpose of the analysis is to indicate that there is no significant difference between the results given by Algorithms 2.1-2.5 and those by Algorithms 3.1-3.5.

Section 5.3 shows that the transformation of Algorithm 4.1 preserves TUFPV. It is now assumed that the results of Algorithms 2.1-2.5 and Algorithms 3.1-3.5 are correct. First the JSD test material produced in the second experiment of Section 5.2 is measured. Then Algorithm 4.1 is used to transform the JSD specification into the corresponding SA specification. After the transformation the SA specification is measured. Together with the results of the other experiments, the TUFPV-preserving nature of the transformation gives evidence for the "soundness" of the structure of this study introduced by Figure 1.4 in Chapter 1.

5.1. Validation of FPA Algorithms for SA Descriptions

In this section it is shown that the Algorithms 2.1-2.5 are consistent with the Symons's Mark II and DeMarco's Function Bang metrics. Consistency means that the two metrics

---

[1] A short description of Function Bang metric is given in Appendix I.

[2] A short description of Mark II metric is given in Appendix J.

correlate with the results of Algorithms 2.1-2.5 in the same way as they would correlate with TUFPV. To do this, two experiments are carried out. The object system of the first experiment is a case study from the literature which contains a small set of SA descriptions. The case study provides well-known test material. The object system of the second experiment consists of a set of randomized SA descriptions. This provides more versatile material compared with the literature case study.

5.1.1. A Preliminary Test Using a Well-known Test Material

The goal of this experiment is to show that the algorithms of Chapter 2 measure the value of Albrecht's unadjusted function points (TUFPV) correctly. This is proved by comparing the results with the values of Mark II and Function Bang metrics. As the characteristics of each metric are known, it is also known how the values of the metrics should correlate.

Research design

The object system to be measured is The Yourdon Press Case Study (Yourdon 1989), which contains dataflow diagrams, an entity-relationship diagram, a data dictionary, and process descriptions.[1]

By definition, when DeMarco's Function Bang is calculated, only the dataflow diagram is used. When Albrecht's Function Point Analysis is carried out by the algorithms of Chapter 2 the dataflow diagram and the entity-relationship diagram are used. By definition, the Mark II metric uses the entity-relationship diagram, but the dataflow diagram is also needed when he transactions are analyzed. The data items are counted from the data dictionary presented in the case study.

To compare the metrics the separate eight processes on the top level have been interpreted as separate systems:

1. Process Orders,
2. Manage Customers,
3. Produce Accounting Reports,
4. Manage Printers,

---

[1] In the Case Study descriptions there are some defects, which can be found by comparing the diagrams and the process descriptions. An attempt has been made to correct the dataflow and ER-diagrams before measurements. For example, the description of Process Orders (process 1.1) has been changed as shown in Appendix D. The corrected ER-diagram is shown in Appendix L.

5. Create New Book Record,
6. Interact with Marketing,
7. Interact with Warehouses, and
8. Interact with Authors.

In this study the files on the top level are not interpreted as interfaces, but as internal files of each process referenced. If the files had been interpreted as interfaces, the number of unadjusted function points would have increased enormously. For example, file BOOKS inside subsystem 1 (Process Orders) would take part in 19 interfaces with the other subsystems, and the number of unadjusted function points for these interfaces would be 95. This is unreasonable when compared with the values in Table 1.1. The normal minimality principle in systems development is applied, which aims to minimize the number of interfaces at each level of abstraction.

This experiment can be thought of as a sample of a possible system space. The only problem is the size of the sample, but this is not unusual in computer science experiments (e.g. Symons 1988, Bourque & Côté 1991). The problem with a small sample may be a test case that significantly differs from the other test cases and affects the result by decreasing the correlation.

Table 5.1. Properties of the object subsystems.

Sub-system	Functional Primitives	Data stores	Input flows	Output flows	Inquiry flows	Process connections
1.	7	8	3	7	0	5
2.	9	12	7	8	3	0
3.	8	12	1	8	0	0
4.	4	2	5	7	0	0
5.	1	1	4	3	0	0
6.	4	5	3	3	1	0
7.	6	9	5	7	0	0
8.	3	7	3	4	0	0
Mean	5.3	7.0	3.9	5.9	0.5	0.6
s	2.5	3.9	1.7	2.0	1.0	1.7

Table 5.1 contains some properties of the subsystems: the number of functional primitives, the number of data stores, the number of input, output and inquiry dataflows, and the number of process pairs connected with a dataflow. In general, there are more data stores than functional primitives within subsystems. Only the first subsystem contains process chains, i.e. process pairs connected with a dataflow.

Measurement and results

TUFPV(SA)[1] is calculated by Algorithms 2.1-2.4[2] using the dataflow diagrams, the data dictionary and the entity-relationship diagram. Appendix E contains an example of data gathering from process 1.1 (Process Orders). Column 2 in Table 5.2 shows the calculated TUFPV(SA) values for all eight subsystems.

Table 5.2. Summary of SA measurements for a case taken from the literature.[3]

Subsystem	TUFPV(SA)	Function Bang	UFP	UFP1	UFP2
1.	101	926.5	108	101.3	102.9
2.	161	554.3	117	113.3	112.5
3.	131	789.0	100	84.5	76.7
4.	57	94.7	43	35.3	35.6
5.	31	37.5	16	15.5	15.9
6.	62	191.3	31	30.1	28.9
7.	116	491.5	87	89.5	87.9
8.	75	298.6	41	39.6	40.0
Total	734	3283.4	543	509.1	500.4
Mean	91.75	422.93	67.88	63.64	62.55
s	40.43	303.10	36.76	35.02	34.48

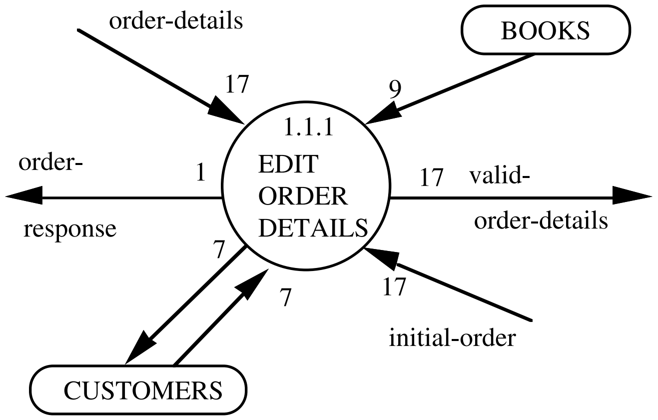


Figure 5.1. Data tokens around a process.

The number of data items for Function Bang calculations are counted from the dataflow diagrams and the data dictionary. For example, the number of data items for process 1.1.1 (Edit Order Details) is 75; Figure 5.1 shows the number of data items for each dataflow. Column 3 in Table 5.2 contains Function Bang values for the subsystems.

---

[1] See Figure 1.4.

[2] Algorithm 2.5 is not needed, because the case study does not contain any interfaces.

[3] The values were produced by the JoUCASE system described in Laamanen & Rask (1991). The metric values were calculated also manually and the results were found to be in line with the results in Table 5.2.

Mark II metric is based on an entity model. To find the logical file references for inputs and outputs in a dataflow diagram, the definition of a user transaction given in Section 2.2 is used. To count the number of input and output data elements, the corresponding data elements in the data dictionary are summarized. Each input and output is processed only once. Each inquiry contains one separate input and one separate output.[1] The entity reference is calculated each time during one transaction when it is referenced either by an input or an output. For example, in Figure 5.2 there are 49 entity references, four outputs which contain six data items, and three inputs which contain 37 data items.

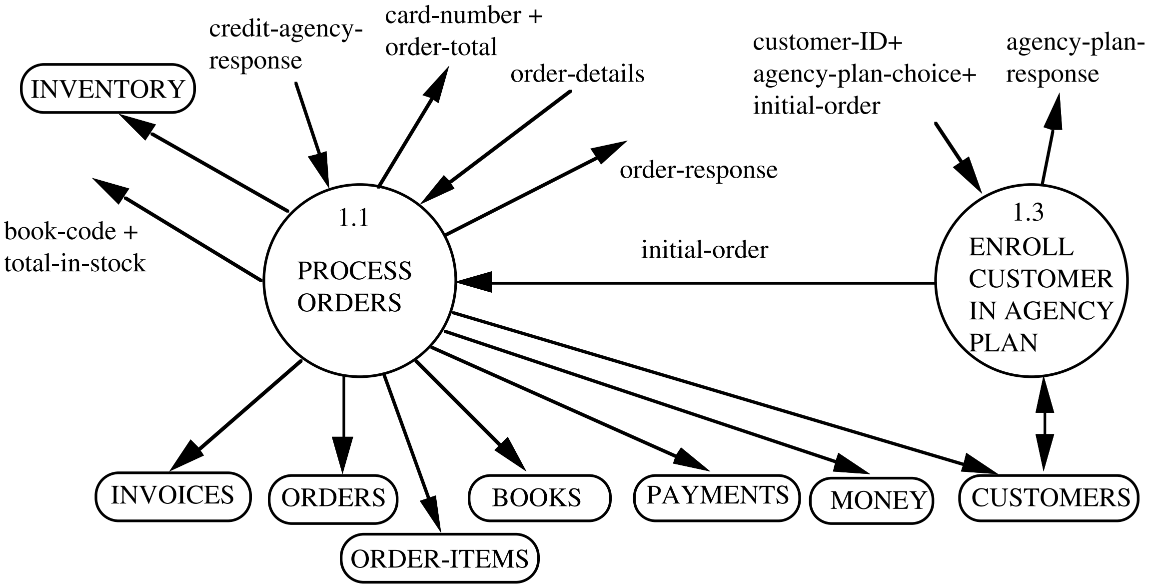


Figure 5.2. Transactions in counting Mark II.

The Mark II values are presented in columns 4-6 of Table 5.2. Column 4 (UFP) is obtained without weights by formula (J.1). Columns 5 (UFP1) and 6 (UFP2) are obtained by formulas (J.2) and (J.3), respectively.[2] The Mark II values are near TUFPV(SA). The largest size values are obtained by the Function Bang metric.

The first line in Table 5.3 shows the correlation coefficients between the results given by Algorithms 2.1-2.4 (TUFPV(SA)) and the other metrics. The second line contains correlations between unweighted Mark II (UFP) and Function Bang. The last two lines show correlations between weighted Mark II metrics (UFP1 and UFP2) and Function Bang.

---

[1] These interpretations correspond to the transformation rules of Symons (1991).

[2] See the Mark II formulas in Appendix J.

Table 5.3. Correlations between the metrics.

	Function Bang	Function Bang[1]	UFP	UFP1	UFP2
TUFPV(SA)	0.75	0.90	0.93	0.93	0.91
UFP	0.89	0.90			
UFP1	0.87	0.86			
UFP2	0.85	0.82			

Discussion

As can be seen in Table 5.3, TUFPV(SA) is closely correlated with Mark II. This is in agreement with the result of Symons (1988)[2]. In his experiment, Symons calibrated Mark II metric by "pegging" the Mark II scale to Albrecht's scale. The calibration was possible because the metrics correlated well for the small system sizes in that experiment. As a result, Symons formed the formula (J.2) in Appendix J.

The same correlation (0.93) between TUFPV(SA) and UFP, and between TUFPV(SA) and UFP1 can be explained by the high internal complexity of the subsystems, as depicted in Table 5.4 by the quotient F/(I+O). A lower correlation (0.91) between TUFPV(SA) and UFP2 is explained by the variation in the number of input and output data items (external complexity), which together with the different weights of UFP1 and UFP2 has an effect on the results.

A better correlation between TUFPV(SA) and Function Bang can be obtained if the subsystem that contains process chains is excluded. This is obvious, because process chains have a greater effect on the value of Function Bang than on TUFPV.

Based on Symons (1988; 1991), Mark II indicates the internal complexity of an application better than TUFPV does. The correlations in the first column of Table 5.3 show the same thing: the correlation between Mark II and Function Bang is greater than the correlation between TUFPV(SA) and Function Bang, when the subsystem with process chains is included.

---

[1] This column is obtained with seven subsystems. Subsystem 1, which contains process chains, has been excluded.

[2] Symons's (1988) observation cannot be generalized. See also Symons (1991).

Table 5.4. Internal and external complexity indicated by Mark II.

	Subsystem									
	1	2	3	4	5	6	7	8	Mean	s
File references (F)	45	52	36	14	7	14	43	18	28.63	16.14
Inputs+Outputs (I+O)	10	18	9	12	7	8	12	7	10.38	3.43
F/(I+O)	4.50	2.89	4.00	1.17	1.00	1.75	3.58	2.57	2.68	1.22
Input data items	37	29	1	15	6	4	16	13	15.13	11.63
Output data items	26	36	63	14	3	13	28	10	24.13	17.83

The above discussion supports the claim that the algorithms of Chapter 2 measure the original TUFPV.

5.1.2. An Extended Test Using Random SA Test Material

The goal of this extended test is to further support the correctness of Algorithms 2.1-2.5. To do this two experiments are carried out. The test material used is more versatile than that in the preliminary experiment described in the previous section. The first experiment compares the results produced by Algorithms 2.1-2.5 with the values of Function Bang metric. The second compares the results of Algorithms 2.1-2.5 with the values of Mark II metric.

5.1.2.1 Comparison of Algorithms 2.1-2.5 with Function Bang

Research design

To implement the Function Point algorithms presented in Chapter 2 and to calculate the value of DeMarco's Function Bang metric, a prototype CASE system - JoUCASE - was built (Laamanen & Rask 1991). Later on the prototype system was augmented so that it could also calculate the Mark II metric. This prototype measurement system gave a possibility to analyze the relationships among the three metrics.

To get a versatile test material, a "simulation" of real systems was carried out by producing SA descriptions with a randomized data dictionary for each DFD to be measured. Thus it became possible to form different system classes based on the characters of the metrics. Figure 5.3 shows the phases of the data generation and measurement process.

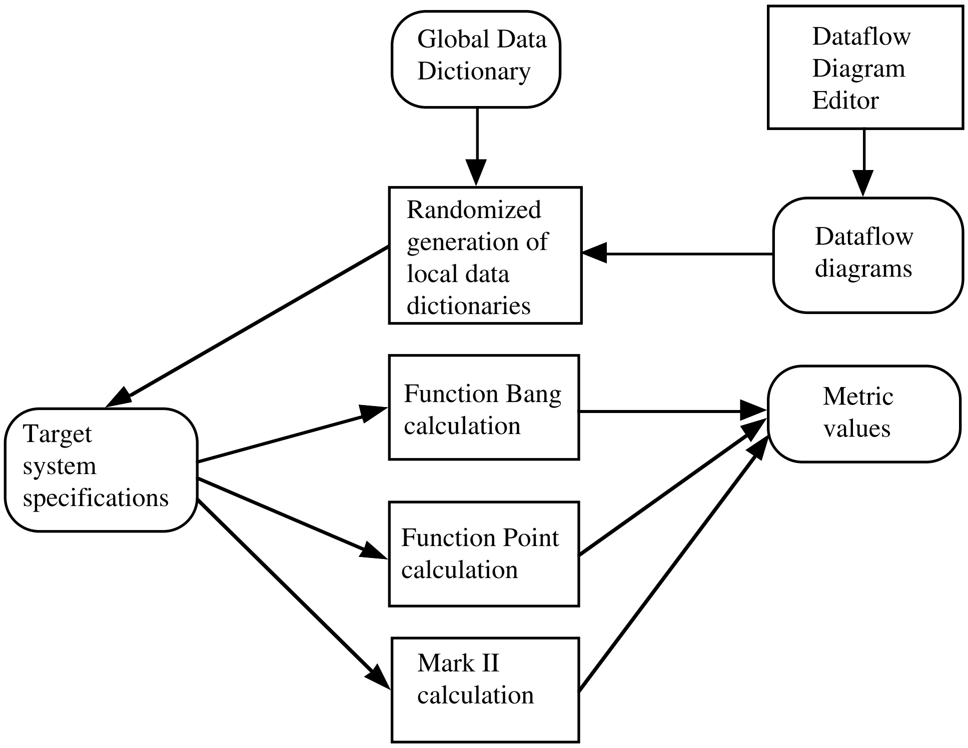


Figure 5.3. Data generation and measurement process.[1]

The global data dictionary, which is common to all SA specifications, was generated taking into account the distribution of file sizes as presented in FPA[2]. The upper limit of the size of each data specification is 69 primitive data items. The global data dictionary contains these 69 different primitive data item names and their different combinations.

The basic dataflow diagrams in our study contain inputs, outputs, internal files (data stores), and unchained primitive processes (Rask & Laamanen 1991). The number of processes is from one to seven[3] and the number of data stores does not exceed the number of processes[4]. Each data store is updated by one or more input dataflows, and each data store is used by one or more input/output dataflow combinations. Within these limits the number of dataflows surrounding the data stores and surrounding the processes is

---

[1] Algorithms 2.1-2.5 and Mark II metric are compared in Section 5.1.2.2.

[2] See Appendix B.

[3] The upper limit for the number of processes is constrained by the adopted distribution of file relationships in FPA (see Appendix B). The limit seven comes also from DeMarco's (1978) observation that a system specification is more manageable if the number of pieces after each decomposition does not exceed seven. A more detailed analysis is given in Rask et al. (1992a; 1992b).

[4] This restriction corresponds to the author's experiences of SA descriptions.

minimized. This restricts the number of possible (equal) system combinations and produces "minimal" DFDs.

For a single process only one minimal DFD can be drawn. Figure 5.4 shows the possible basic dataflow diagrams containing two processes. When there are three processes and one data store, it is possible that either two processes update the data store and one process uses it, or one process updates the data store and two processes use it. When all possible combinations are formed based on these principles, the total number of *Basic diagrams* increases to 113.

Each data store of a target system specification contains 1-69 primitive data items, one of which is a primary key. One data store may contain several logical internal files, depending on the number of separate access keys used; if the dataflow that references a file is not named, the access key is supposed to be the same as the primary key of the data store. Each output dataflow contains 1-24 primitive data items and each input dataflow contains 1-19 primitive data items. These upper limits are based on the distributions of data items referenced in FPA[1]. The number of file relationships is produced randomly. The upper limit here is the number of data stores minus one[1].

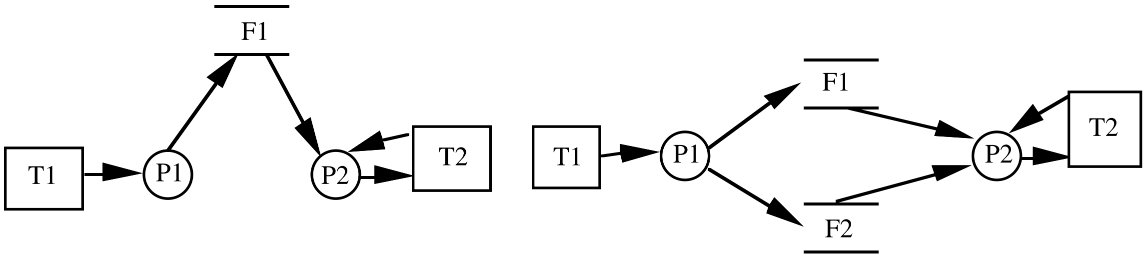


Figure 5.4. Two examples of test dataflow diagrams.

In addition to calculating metrics from the Basic diagrams, the effect of inquiries, interfaces, naming of file references, and process chains on metrics is tested. The three first extensions are based on the principles of Albrecht's FPA. Process chains are used to determine how the growth of the function-strong features, defined by DeMarco (1982), affect the correlation between the two metrics.

To measure the effect of inquiries, each input/output combination of the Basic diagrams is replaced by an external inquiry symbol. According to the principles of FPA, an external in-

---

[1] According to Yourdon (1989), it is not very common to have parallel relationships between two entity types.

quiry cannot update a data store. Thus, there are 112 *External inquiry diagrams*. To see the effect of process chains, the primitive processes of each basic diagram is doubled, which produces 113 *Process chain diagrams*. To explain the effect of interfaces, one instance of each interface type (transaction file, shared file, and external file) is added into the Basic diagrams. This produces 113 *External interface diagrams*.

Table 5.5 summarizes the properties of the generated dataflow diagrams in the form of mean and standard deviation for each diagram type. Compared with the properties of subsystems in Table 5.1, the average number of data stores is now lower. Thus the subsystems in the preliminary experiment were more data intensive.

Table 5.5. Properties of the dataflow diagrams.

	Functional Primitives		Data stores		Input flows		Output flows		Inquiry flows		Process connections	
Diagram type	Mean	s	Mean	s	Mean	s	Mean	s	Mean	s	Mean	s
Basic diagrams	5.7	1.4	3.4	1.8	5.7	1.4	2.9	1.5	0.0	0.0	0.0	0.0
Ext. inquiries	5.8	1.3	3.4	1.8	2.9	1.5	0.0	0.0	2.9	1.5	0.0	0.0
Process chains	11.4	2.7	3.4	1.8	5.7	1.4	2.9	1.5	0.0	0.0	8.6	2.5
Ext. interfaces	7.7	1.4	4.4	1.8	6.7	1.4	2.9	1.5	0.0	0.0	1.0	0.0

Naming of file references may affect the number of internal logical files in FPA. To analyze this effect, 0% and 100% limits[1] are used for named file references, within all four diagram groups. This doubles the number of target system specifications.

To form the local (random) data dictionaries, both a uniform and a normal distribution is used. In the first experiment random numbers are produced that are uniformly distributed into a range [0,1]. In the second experiment a transformation is used:

$$Z = \sqrt{-2 \ln U_1} \cos(2\pi U_2) \tag{5.1}$$

to obtain normally N(0,1) distributed random numbers from uniformly distributed random numbers. Here U_1 and U_2 are uniformly distributed into the range [0,1]. The normally N(0,1) distributed random numbers were then transformed into the range [0,1] using a mean of 0.5 and standard deviations of 0.15 and 0.25.[2]

---

[1] Either all file references are unnamed or all file references are named.

[2] These principles (three random distributions) triple the number of diagrams to be analyzed.

By using normal distribution an attempt is made to reduce the effect of extreme values, for example, to avoid situations in which a data store contains only one data item. With varying standard deviations the shape of the normal distribution can be further affected.

With a standard deviation of 0.15 (i.e. normal distribution N(0.5,0.0225)) an attempt was made to get all the random numbers into a range [0,1]. With the standard deviation of 0.25 (i.e. normal distribution N(0.5,0.0625)) a part of the random numbers have been allowed to remain outside the range [0,1], which causes the number of extreme values to increase, thus nearing uniform distribution.

Measurement and results

The results of the experiments were analyzed by calculating correlation coefficients and by carrying out a regression analysis.[1] The experiments were repeated 10 times and the correlation r_i for each repetition i was calculated for each test group. The following is a summary of the results.

Table 5.6. Mean correlations per diagram type: TUFPV(SA) vs. Function Bang.

		Mean correlation $\bar{r}$		
Diagram type	Named file reference %	U[0,1]	N(0.5,0.0225)	N(0.5,0.0625)
Basic diagrams	100 %	0.80	0.93	0.85
	0 %	0.80	0.90	0.85
External inquiries	100 %	0.80	0.90	0.84
	0 %	0.78	0.88	0.84
Process chains	100 %	0.71	0.83	0.80
	0%	0.70	0.80	0.75
External interfaces	100 %	0.65	0.82	0.72
	0 %	0.71	0.85	0.77

Table 5.6 contains the mean correlation $\bar{r}$ per diagram type for each distribution. Table 5.7 contains a summary of the correlations derived from the whole test material. Table 5.8 shows the regression equations for N(0.5,0.0225) distribution when the percentage of named file references is 100 (all file references are named).

Table 5.7. Total correlations per distribution: TUFPV(SA) vs. Function Bang.

Named file reference %	U[0,1]	N(0.5,0.0225)	N(0.5,0.0625)

---

[1] The statistical methods are discussed, for example, by Conte et al. (1986).

100 %	0.39	0.38	0.40
0 %	0.51	0.51	0.51
all together	0.29	0.27	0.29

Discussion

The Basic diagrams have the highest correlations. Transformation of external input/output pairs into external inquiries reduces the correlation slightly. This reduction is caused by the changes in the TUFPV(SA) values. Moreover, the reduction is greater if new processes are added into the Basic diagrams, because the resulting process chains increase the Function Bang values but have no effect on the TUFPV(SA) values. This causes the scatter to be more random. The addition of external interfaces into the Basic diagrams also reduces the correlation. The addition of transaction files and shared files increases the values of the Function Bang metric more than TUFPV(SA) values causing the scatter to be more random.

The naming of file references may increase the correlation in all groups except External interfaces. The naming of file references always increases TUFPV(SA), because the number of logical internal files is also increased. The effect on Function Bang is the opposite, because the unnamed file references have the same number of data items as the referenced data stores. This is always greater than (or equal to) the number of data items in a named file reference. In the present experiment the number of external interfaces remains constant regardless of the size of the dataflow diagram. With small diagrams the interfaces cause a greater variation between TUFPV(SA) and Function Bang values when the data store references are named. This explains the reduction in the correlation with External interfaces.

The correlations with normal N(0.5,0.0225) distribution follow a pattern similar to that for correlations with a uniform distribution, but they are constantly higher. This can be explained by the distribution used. When a normal distribution is used, there are not as many extreme random number values as with uniform distribution.

The normal N(0.5,0.0625) distribution with a greater standard deviation slopes a little more gently than the normal N(0.5,0.0225) distribution. The effect of this change in the form of the distribution can be seen in the correlations. The correlations for N(0.5,0.0625) distribution are between the correlations of the uniform distribution and the N(0.5,0.0225) distribution. In other respects the correlations of N(0.5,0.0625) distribution follow approximately the direction of the correlations of the other two distributions within each group.

Table 5.7 shows the total correlation when the percentage of named file references is 100 (all file references are named), the total correlation when the percentage of named file references is 0 (all file references are unnamed), and the total correlation calculated for all the test material used. The total correlations do not mean that the correlation between TUFPV(SA) and Function Bang values, in general, is about 0.28 (an average of the total correlations in the last row of Table 5.7). Instead, the correlation depends on the type of system. This argument is based on the poor total correlations and the high correlations within system types. The naming of file references emphasizes differences between the system types decreasing the total correlation.

Table 5.8 shows the regression equations for N(0.5,0.0225) distribution when all file references are named. The correlation and regression coefficient for External inquiries correspond closely to the correlation and regression coefficient of the Basic diagrams, although the intercept of the regression lines on the BANG axis varies slightly. Also the regression coefficient for External interfaces is near to the regression coeffiecient of the Basic diagrams. For Process chains, however, the relationship to Basic diagrams is not so clear. In all, the regression equations explain the poor total correlation in Table 5.7.

Table 5.8. Results of Regression Analysis: TUFPV(SA) vs. Function Bang.

Diagram Type	n	r	Regression equation	s_e[1]
Basic diagrams	1130	0.93	BANG = 46.73 + 7.06 * TUFPV(SA)	90.82
External inquiries	1120[2]	0.90	BANG = 67.61 + 7.72 * TUFPV(SA)	108.79
Process chains	1130	0.83	BANG = 712.56 + 18.45 * TUFPV(SA)	392.46
External interfaces	1130	0.82	BANG = 366.46 + 7.00 * TUFPV(SA)	165.79

The standard deviations s_e (Baxter 1976) for the errors e_i in BANG terms have also been calculated. According to these values, the deviations from the regression line are greatest with Process chains, whereas the standard deviations for Basic diagrams and External inquiries are smaller and not far from each other. The External interfaces are between these extremes. No special DFD structure has been found within the diagram types that could specifically cause the deviations.

The different extensions have been used to find out how different properties of dataflow diagrams affect the correlation between the metrics. With respect to all the extension types, an attempt has been made to identify the most obvious changes in order to show the effect

---

[1] Standard deviation $s_e = s_{BANG}\sqrt{(1 - r^2)}$.

[2] For inquiries a DFD with only one process cannot be drawn.

of each extension. It is clear, however, that the magnitude of each extension affects the intercept point and regression coefficient of a regression line. If, for example, the number of processes is doubled, the values of TUFPV(SA) do not change, but the Function Bang values almost triple when the data store references are named. In Table 5.8 the corresponding changes in regression coefficient and the intercept point of the regression equation can be observed.

The results of this experiment are consistent with the results in Section 5.1.1. The correlations of Process chains correspond to the correlation 0.75 in Table 5.3. When the subsystem containing process chains in the preliminary experiment were excluded, the correlation was 0.90. This clearly corresponds to the correlations for N(0.5,0225) distribution in Table 5.6, excluding Process chains and External interfaces.[1]

5.1.2.2. Comparison of Algorithms 2.1-2.5 with Mark II

Research design

In this experiment the testing environment was the same that was described in the previous section when Algorithms 2.1-2.5 were compared with DeMarco's Function Bang metric. The test material containing SA specifications with process chains was, however, excluded because process chains do not affect the metric values of Algorithms 2.1-2.5 and Mark II.

Depending on their direction, the external interfaces of FPA are interpreted as inputs or outputs[2]. For example, the interfaces with transactions in Figure 2.12(a) contain two inputs and two outputs, and the interfaces in Figure 2.12(b) contain two inputs and two outputs.

In the following, TUFPV(SA) and unweighted Mark II metric UFP are considered first. Then the same consideration is carried out for TUFPV(SA) and the weighted Mark II metrics UFP1 and UFP2.[3]

Measurement and results: TUFPV(SA) vs. unweighted Mark II

---

[1] The experiment was repeated using the limits 8-14 and 15-21 for the number of processes in Basic diagrams (Rask et al. 1992b). The generation of dataflow diagrams in these extended experiments was automated and a random sample of 113 target system specifications was taken. The results from these experiments were in line with the results reported here.

[2] This interpretation corresponds to the transformation rules of Symons (1991).

[3] See formulas (J.1)-(J.3) in Appendix J.

As in the previous section, the experiments were repeated 10 times and the correlation r_i was calculated for each repetition per test group. Table 5.9 contains the mean correlation $\bar{r}$ per diagram type for each distribution. Table 5.10 contains the summary of the correlations based on the whole test material. Table 5.11 contains the regression equations for N(0.5,0.0225) distribution when all file references are named.

Table 5.9. Mean correlations per diagram type: TUFPV(SA) vs. unweighted Mark II (UFP).

Diagram type	Named file reference %	Mean correlation $\bar{r}$ U[0,1]	N(0.5,0.0225)	N(0.5,0.0625)
Basic diagrams	100 %	0.66	0.68	0.66
	0 %	0.60	0.64	0.61
External inquiries	100 %	0.56	0.61	0.59
	0 %	0.47	0.52	0.50
External interfaces	100 %	0.40	0.53	0.47
	0 %	0.38	0.52	0.44

Table 5.10. Total correlations per distribution: TUFPV(SA) vs. unweighted Mark II (UFP).

Named file reference %	U[0,1]	N(0.5,0.0225)	N(0.5,0.0625)
100 %	0.67	0.73	0.72
0 %	0.67	0.74	0.71
all together	0.60	0.65	0.64

Discussion: TUFPV(SA) vs. unweighted Mark II

The correlations in Table 5.9 are clearly lower than those in Table 5.6. This was expected because the number of input and output elements greatly affects the value of UFP. The variation in the number of input and output elements easily causes an opposite deviation compared with the corresponding TUFPV(SA) value. The correlations between the three distributions follow approximately the pattern of the correlations in Table 5.6: the best correlations are with the normal N(0.5,0.0225) distribution and the worst correlations are with the uniform distribution.

The correlation for External inquiries is much lower than that for Basic diagrams because with UFP an inquiry means one input and one output; with FPA we have either one input or one output. The interfaces inrease, in particular, the value of UFP because they are interpreted as inputs and outputs. Thus, interfaces reduce the correlation.

The naming of file references increases the correlation in all groups. This can be explained by the fact that the naming of file references has no effect on UFP values. The naming, however, always increases the value of TUFPV(SA), because the number of logical internal files is also increased. Thus, TUFPV(SA) gets closer to UFP values.

Table 5.10 shows the total correlations. When they are compared with the results in Table 5.7, they are seen to be much higher. The effect of the diagram types is not as strong as with TUFPV(SA) and Function Bang. The diagram type independence can also be seen from the regression equations of Table 5.11 which shows regression equations for N(0.5,0.0225) distribution when all file references are named. For all the equations the regression coefficients are about the same.

Table 5.11. Results of Regression Analysis: TUFPV(SA) vs. unweighted Mark II (UFP).

Diagram Type	n	r	Regression equation	s_e[1]
Basic diagrams	1130	0.68	UFP = 41.68 + 0.69 * TUFPV(SA)	23.90
External inquiries	1120[2]	0.61	UFP = 51.41 + 0.65 * TUFPV(SA)	24.68
External interfaces	1130	0.53	UFP = 130.40 + 0.69 * TUFPV(SA)	37.92

According to the standard deviations s_e for the errors e_i in UFP values, the deviations from the regression line are greatest with External interfaces, whereas the standard deviations for Basic diagrams and External inquiries are smaller and near to each other. The effect of the input and output sizes can also be seen from the intercept of the regression lines on the UFP axis. The External interfaces have the greatest intercept point.

The results in Table 5.9 are also lower than those in the preliminary experiment shown in Table 5.3. This is explained by the lower internal complexity of the randomized descriptions, which is indicated by the quotient F/(I+O) in Table 5.12 compared with Table 5.4 in Section 5.1.1. The randomized descriptions are not as data intensive as the subsystems in the preliminary experiment.

Table 5.12. Internal complexity indicated by Mark II.

---

[1] Standard deviation $s_e = s_{UFP}\sqrt{(1 - r^2)}$.

[2] For inquiries a DFD with only one process cannot be drawn.

	File references (F)		Inputs+Outputs (I+O)		F/(I+O)	
Diagram type	Mean	s	Mean	s	Mean	s
Basic diagrams	12.55	4.29	8.57	2.46	1.48	0.41
External inquiries	12.63	4.20	8.61	2.39	1.49	0.41
External interfaces	15.63	4.28	11.57	2.46	1.36	0.29

Measurement and results: TUFPV(SA) vs. weighted Mark II

When the weighted Mark II metrics UFP1 and UFP2 suggested by Symons (1988; 1991) using the formulas (J.2) and (J.3) in Appendix J are used, the following two correlation tables 5.13 and 5.14 are obtained. Tables 5.15 and 5.16 show the regression equations for N(0.5,0.0225) distribution when all file references are named.

Table 5.13. Mean correlations per diagram type: TUFPV(SA) vs. Mark II with weights.

		Mean correlation $\bar{r}$					
Diagram type	Named file reference %	U[0,1] UFP1	U[0,1] UFP2	N(0.5,0.0225) UFP1	N(0.5,0.0225) UFP2	N(0.5,0.0625) UFP1	N(0.5,0.0625) UFP2
Basic diagrams	100 %	0.80	0.80	0.83	0.84	0.81	0.82
	0 %	0.76	0.75	0.79	0.79	0.76	0.76
External inquiries	100 %	0.72	0.74	0.77	0.80	0.76	0.77
	0 %	0.66	0.66	0.70	0.72	0.68	0.69
External interfaces	100 %	0.52	0.46	0.68	0.63	0.60	0.54
	0 %	0.51	0.44	0.66	0.62	0.58	0.52

Table 5.14. Total correlations per distribution: TUFPV(SA) vs. Mark II with weights.

Named file reference %	U[0,1] UFP1	U[0,1] UFP2	N(0.5,0.0225) UFP1	N(0.5,0.0225) UFP2	N(0.5,0.0625) UFP1	N(0.5,0.0625) UFP2
100 %	0.75	0.72	0.82	0.81	0.80	0.78
0 %	0.75	0.72	0.81	0.80	0.78	0.77
all together	0.68	0.66	0.73	0.72	0.71	0.70

Discussion: TUFPV(SA) vs. weighted Mark II

The results show that the correlations are improved by reducing the effect of inputs and outputs and by increasing the effect of file references. The effect of the weights is greatest with the uniform distribution. The changes with the normal distributions are the same.

From Tables 5.13 and 5.14 it can also be seen that there is no significant difference between the weighted Mark II metrics UFP1 and UFP2. The slight difference is explained by the different weights of UFP1 and UFP2 and the diagram types which cause variation in input and output numbers.

The correlation between the results produced by Algorithms 2.1-2.5 and Mark II in the preliminary experiment in Section 5.1.1 was slightly higher than the results in this section. This is explained by the data intensiveness mentioned in connection with the results of unweighted Mark II.

The regression equations shown in Tables 5.15 and 5.16 are more gently sloping than the regression equations for unweighted Mark II in Table 5.11. This is caused by the weights of UFP1 and UFP2, which diminish the difference between TUFPV and Mark II value.

Table 5.15. Results of Regression Analysis: TUFPV(SA) vs. UFP1

Diagram Type	n	r	Regression equation	s_e[1]
Basic diagrams	1130	0.83	UFP1 = 18.39 + 0.45 * TUFPV(SA)	9.71
External inquiries	1120[2]	0.77	UFP1 = 22.97 + 0.45 * TUFPV(SA)	10.86
External interfaces	1130	0.68	UFP1 = 55.01 + 0.44 * TUFPV(SA)	15.95

Table 5.16. Results of Regression Analysis: TUFPV(SA) vs. UFP2

Diagram Type	n	r	Regression equation	s_e[3]
Basic diagrams	1130	0.84	UFP2 = 21.26 + 0.46 * TUFPV(SA)	9.48
External inquiries	1120[2]	0.80	UFP2 = 25.31 + 0.46 * TUFPV(SA)	10.14
External interfaces	1130	0.63	UFP2 = 57.92 + 0.44 * TUFPV(SA)	18.50

In summary, the results of this section together with those of Section 5.1.2.1 show that the behaviour of TUFPV(SA), as computed by Algorithms 2.1-2.5, comes up to expectations.

5.2. Validation of FPA Algorithms for JSD Descriptions

---

[1] Standard deviation $s_e = s_{UFP1}\sqrt{(1 - r^2)}$.

[2] For inquiries a DFD cannot be drawn with only one process.

[3] Standard deviation $s_e = s_{UFP2}\sqrt{(1 - r^2)}$.

This section shows the correctness of the FPA algorithms in Chapter 3. To do this two experiments are carried out and the results (TUFPV(SA) and TUFPV(JSD)) computed from SA and JSD specifications are compared by using manual transformations between the methods. Assuming that Algorithms 2.1-2.5 are correct, it can be concluded that the algorithms of Chapter 3 are correct. In the experiments it is shown that the values produced by Algorithms 2.1-2.4 and Algorithms 3.1-3.4[1] from the corresponding SA and JSD descriptions correlate well. To compensate for possible human errors, the transformations were carried out in two directions: from SA to JSD and from JSD to SA.

5.2.1. Application of Algorithms 3.1-3.4 to JSD after Transformation from SA

The goal of this experiment is to show that, after converting an SA description into a corresponding JSD description, measurement of the JSD descriptions corresponds to the results obtained from the initial SA descriptions. This implies that Algorithms 3.1-3.4 produce correct results.

Research design

The object system of the first experiment is The Yourdon Press Case Study (Yourdon 1989) described in Section 5.1.1. But now the SA description is first transformed manually into a JSD description. An example of the transformation is given in Appedix F, which shows the transformed form of the dataflow diagram in Appendix D. Here some processes of the dataflow diagram have been grouped under three model processes.

Measurement and results

Algorithms 3.1-3.4 were applied manually to the JSD specification obtained from the corresponding SA specification. Appendix G contains the FPA data sheets corresponding to the JSD specification in Appendix F.

Table 5.17. Summary of JSD measurements.

Subsystem	TUFPV(JSD)[2]
1.	107
2.	157

---

[1] Algorithm 3.5 is not needed because the case study does not contain any interfaces.

[2] See Figure 1.4.

3.	131
4.	57
5.	31
6.	62
7.	119
8.	75
Total	739

Table 5.17 contains the results given by Algorithms 3.1-3.4 for all eight processes. There is only a small difference between the SA description (Table 5.2) and the JSD specification in TUFPV values. Only the sizes of three subsystems (1, 3 and 7) differ slightly.

The value of the correlation coefficient between TUFPV(SA) computed from SA description (Table 5.2) and TUFPV(JSD) computed from the JSD specification (Table 5.17) is 0.998.

Discussion

The high correlation between the results computed from SA and JSD specifications demonstrates that the algorithms in Chapter 3 yield the correct TUFPV.

5.2.2. Application of Algorithms 3.1-3.4 to JSD before Transformation to SA

The goal of this experiment is to show that after converting a JSD description manually into a corresponding SA description the measurement of the JSD description corresponds to results obtained from the initial SA description. This gives further evidence for the correctness of Algorithms 3.1-3.4[1].

Research design

The object system of this experiment consists of systems made as laboratory assignments by computer science students with the JSD method. First the TUFPV(JSD) values were calculated from these JSD descriptions by the algorithms in Chapter 3. Then the students transformed the JSD descriptions, manually and independently of the algorithms of Chapter 4, into SA descriptions, which were then measured by the FPA algorithms of

---

[1] Algorithm 3.5 is not needed because the case study does not contain any interfaces.

Chapter 2. Finally, the correlation between the TUFPVs of JSD and SA descriptions was solved.

The case material of the laboratory exercises are not "real" systems. The sizes of the object systems shown in Table 5.18 are rather small compared with the sizes in Table 1.1. The students received a short one page task description (one assignment per student), which was then clarified during conversations with the instructor. The subjects of the laboratory exercises were:[1]

1. Photo shop
2. Raw material purchases
3. Advance invoices
4. Course arrangements
5. Patient prescriptions
6. Offer calculations
7. Raw material purchases
8. Car rental
9. Spare part sales

These laboratory exercises were solved with the JSD method. The descriptions contained no external interfaces and only a few inquiries.

After the JSD laboratory exercises were finished, the corresponding SA descriptions based on Yourdon's method were made by the same students. SA descriptions were made without knowledge of the transformation principles described in Chapter 4.

Measurement and results

There are three possible interpretations of the JSD descriptions, which are here called JSD-1, JSD-2, and JSD-3. The interpretations are based on the idea of Jackson (1983) to separate the attributes of each action of each process into separate groups or records. It is then possible to interpret these attribute subgroups as separate dataflows as discussed in Chapters 3 and 4. In the interpretation of JSD-1 the attribute subgroups are not separated into different dataflows. Table 5.18 shows the JSD-1 results from JSD descriptions used in systems design laboratory exercises.[2]

---

[1] The task descriptions for exercises 2 and 7 were the same.

[2] Interpretation JSD-1 simplifies Algorithms 3.1-3.5 and 4.1.

In interpretation JSD-2 attribute subgroups are separated into different dataflows; files referenced by each subdataflow are the same as in the original JSD-1 dataflow. The complexity of an input or an output subdataflow may be lower than the complexity of the original dataflow because each subdataflow contains fewer data items, but the sum of the complexities of the subdataflows may be greater than the complexity of a single (parent) dataflow. In interpretation JSD-3 the real file references of each subdataflow were sought. Thus the number of unadjusted function points in interpretation JSD-3 may be lower than in interpretation JSD-2.[1]

Table 5.18. Values from JSD laboratory exercises: TUFPV(JSD-1).

OP = Outputs IP = Inputs
IQ = Inquiries FI = Files
IF = Interfaces

Work Number	OP	IP	IQ	FI	IF	Total
1.	20	21	0	21	0	62
2.	17	27	0	49	0	93
3.	42	16	14	28	0	100
4.	10	25	0	35	0	70
5.	20	28	0	35	0	83
6.	19	32	0	35	0	86
7.	27	37	0	35	0	99
8.	20	17	4	28	0	69
9.	21	22	4	28	0	75
Total	196	225	22	294	0	737
Mean						81.9
s						13.8

Table 5.19 contains the number of unadjusted function points (inputs and total) in interpretations JSD-2 and JSD-3.[2] Comparison with the results shown in Table 5.18 reveals that the change (increase) is greatest in experiment JSD-2. The standard deviation is, however, greatest with interpretation JSD-3.

Table 5.19. Values from JSD laboratory exercises: TUFPV(JSD-2) & TUFPV(JSD-3).

	TUFPV(JSD-2)		TUFPV(JSD-3)	
Work Number	IP	Total	IP	Total
1.	65	106	49	90
2.	60	126	51	117
3.	64	148	62	146
4.	61	106	46	91
5.	80	135	64	119
6.	78	132	68	122
7.	84	146	71	133
8.	43	95	37	89

---

[1] Interpretations JSD-2 and JSD-3 were applied in Algorithms 3.1-3.5 and 4.1.

[2] Only the changed input points (IP) and the total number of unadjusted function points are shown.

9.	40	93	31	84
Total	575	1087	479	991
Mean		120.8		110.1
s		21.2		22.3

After the students had solved their assignment using JSD, they also used Yourdon's Structured Method to describe the same system. The FPA values were calculated from these SA descriptions by Algorithms 2.1-2.4. These results are shown in Table 5.20, and the resulting correlations are shown in Table 5.21.

Table 5.20. Values from SA exercises: TUFPV(SA).

Work Number	OP	IP	IQ	FI	IF	Total
1.	21	58	0	35	0	114
2.	17	30	0	28	0	75
3.	42	16	14	28	0	100
4.	10	20	0	28	0	58
5.	20	15	0	35	0	70
6.	19	35	0	35	0	89
7.	27	35	0	35	0	97
8.	20	17	4	28	0	69
9.	19	22	4	28	0	73
Total	195	248	22	280	0	745
Mean						82.8
s						18.2

Discussion

The first row in Table 5.21 shows the correlation coefficients of TUFPVs between JSD descriptions and SA results. The values are rather small compared with the result in Section 5.2.1. There is one major reason for this difference: the number of input dataflows. If in some cases separate processes are described in a dataflow diagram for the actions of JSD model processes and in some projects they are not, the changes in the counted FPA values will not follow in the same direction.

In the experiments, JSD laboratory assignment number one and the corresponding SA exercise differ from the others in this respect.[1] If this difference is not taken into account, i.e. laboratory exercise number 1 is excluded, the standard deviation of the SA exercises decreases from 18.2 in Table 5.20 to 14.8. The second row in Table 5.21 shows a clear im-

---

[1] The accuracy of the diverged laboratory assignment was better than the accuracy of the the other eight cases.

provement in correlations when the sample size is eight laboratory exercises. Experiment JSD-1 now has the best correlation, 0.86. This is caused by the decrease in standard deviations of both SA and JSD descriptions. The standard deviation of JSD-1 descriptions decreases from 13.8 in Table 5.18 to 12.4. On the other hand, the standard deviations of JSD-2 and JSD-3 descriptions increase, JSD-2 from 21.2 to 21.9 and JSD-3 from 22.3 to 22.4. This variation explains the lower correlations compared with JSD-1.

Table 5.21. Correlation between TUFPV(SA) and TUFPV(JSD): independent descriptions.

	TUFPV(JSD-1)	TUFPV(JSD-2)	TUFPV(JSD-3)
TUFPV(SA_9)	0.20	0.39	0.38
TUFPV(SA_8)[1]	0.86	0.76	0.84

As mentioned above, the SA descriptions were made by the students without knowledge of the transformation rules described in Chapter 4. This also caused other differences in the students' assignments, in addition to the number of input dataflows. These differences may also have caused differences in unadjusted function point values. The possible defect factors are the following:

- Dataflow connections in a JSD specification diagram may cause differences in the number of file references. Sometimes it can be better to use state vector connections instead of dataflow connections.
- There may be more data stores in a dataflow diagram than model processes in the JSD specification diagram. Students may have normalized the SA descriptions more than the JSD description.
- It is easier to take into account updating (changing and deleting) in JSD analysis than in SA analysis. This may cause differences in the number of input dataflows.
- If two model processes in a JSD specification are connected with a dataflow connection and these model processes have a common input dataflow, at least one input may be lost. This input is, however, more complex than the corresponding SA inputs, because it has more data items.

In this section the applicability of Function Point Analysis with several SA and JSD descriptions has been considered. We have seen that the descriptions must be examined care-

---

[1] Does not contain laboratory exercise number 1.

fully to get a reliable size estimate for the software system. If there are different working practices in separate projects, a common productivity model cannot be used to predict the cost of the systems. But if both the development method and the size estimation method are combined and understood, the results improve considerably. When the interpretation alternatives between SA and JSD are considered, it can be supposed that the greatest differences originate in inputs and files. It must, however, be remembered that these differences emerge mainly when the correspondences between JSD and SA are not known well enough.

5.3. Properties of Algorithmic Transformation

The goal of this last experiment is to show that Algorithm 4.1 preserves TUFPV complexity when a JSD description is transformed into the corresponding SA description.

Research design

In this experiment the tranformation principles given in Chapter 4 for the systems design laboratory assignments discussed in the previous section will be used and the number of unadjusted function points of JSD and SA descriptions will be compared.

Measurement and results

Table 5.22 summarizes TUFPV(SA) counted from the results of the algorithmic transformation from JSD to SA for the students' laboratory assignments. These results were obtained by the algorithms developed in Chapter 2.

The correlations between the results in Table 5.22 and the interpretations JSD-1, JSD-2, and JSD-3 in Tables 5.18 and 5.19 are summarized in Table 5.23.

Table 5.22. Values from JSD to SA transformations: TUFPV(SA).

Work Number	OP	IP	IQ	FI	IF	Total
1.	20	49	0	21	0	90
2.	17	51	0	42	0	110
3.	42	62	14	28	0	146
4.	10	46	0	35	0	91
5.	20	64	0	35	0	119
6.	19	68	0	35	0	122

7.	27	71	0	35	0	133
8.	20	37	4	28	0	89
9.	19	31	4	28	0	82
Total	194	479	22	287	0	982
Mean						109.1
s						22.4

Discussion

Compared with Table 5.18, the total values in Table 5.22 are higher and, in general, near to the values of interpretation JSD-3 in Table 5.19.

In Table 5.23 the correspondence between the sizes of JSD and SA descriptions is very good even with the interpretation JSD-1. The highest correlation is with JSD-3, which is consistent with the high correlation between SA and JSD sizes derived in Section 5.2.1.

Table 5.23. Correlation between TUFPV(SA) and TUFPV(JSD): direct transformation.

	TUFPV(JSD-1)	TUFPV(JSD-2)	TUFPV(JSD-3)
TUFPV(SA)	0.89	0.98	0.99

The results of this experiment show that Algorithm 4.1 preserves TUFPV. Together with the results of the other experiments this points out the "soundness" of the structure for this study introduced in Chapter 1.

5.4. Summary of the Results

With the experiments reported in this chapter an attempt was made to show that it is possible to calculate TUFPV from the SA descriptions and from the JSD specification as proposed in Figure 1.4 in Section 1.3.1.

The experiments in Sections 5.1.1 and 5.1.2 confirmed that TUFPV(SA), produced by Algorithms 2.1-2.5, measures TUFPV. The conclusion was based on knowledge of the behaviour of TUFPV and two other well-known metrics, Function Bang and Mark II. The experiments used well-known SA material from the literature and more versatile random SA material.

The experiments in Sections 5.2.1 and 5.2.2 confirmed that TUFPV(JSD), produced by Algorithms 3.1-3.4[1], measures TUFPV. This was based on the assumption that Algorithms 2.1-2.5 are correct. Two experiments were carried out. The first of these computed TUFPV(JSD) from the case material in the literature used in Section 5.1.1, after transforming the SA descriptions into JSD. The second computed TUFPV(JSD) from students' JSD laboratory assignments and TUFPV(SA) from the corresponding SA descriptions after transformation[2].

Finally, the experiment in Section 5.3 confirmed that Algorithm 4.1 preserves TUFPV, provided that Algorithms 2.1-2.5 and Algorithms 3.1-3.5 are correct. Here Algorithm 4.1 was used to transform students' JSD laboratory assignments into SA before computing TUFPV(SA).

6. SUMMARY AND CONCLUSIONS

This thesis has concentrated on measurement of software size. In this study the size measurement of business applications has been shifted into the requirements specification level. Another aspect of the study has been the application of metrics to different structured specifications produced by CASE-tools, i.e. automation of the size measurement in CASE environments that support structured methods. Within this aspect transformations between the development methods have also been examined. The major applied method of measurement is Albrecht's (1979) Function Point Analysis (FPA). The development methods used have been Yourdon's (1989) Structured Analysis (SA), and Jackson's (1983) System Development (JSD) method.

In Chapter 2 algorithms were developed for calculating Albrecht's unadjusted function points from SA descriptions. These algorithms have also been implemented (Laamanen & Rask 1991) and thus the applicability of the algorithms as a part of a CASE-tool has been proved. The reasons for selecting Albrecht's FPA and Yourdon's SA were the following:

---

[1] Our test material did not contain external interfaces to be measured by Algorithm 3.5.

[2] Transformation here means respecification, not an algorithmic transformation.

both are well-known and generally used methods, but up to now there has not been a well-developed FPA algorithm for SA.

In Chapter 3 FPA algorithms were developed that can be applied in connection with Jackson's JSD method. This adaptation pointed out the generality of Albrecht's FPA. It can be applied with different methods, provided that the capacity of the development method is sufficient to describe all user functions as defined in FPA. Thus, FPA could be part of development practice at the requirements specification level independently of the development method used.

In Chapter 4 the concept of methodical equivalence between SA and JSD was explored. The transformation of the basic components in JSD to those in SA, and vice versa, was investigated. It was shown that a JSD description (specification diagram with model process structures) can be transformed automatically into an SA description (dataflow diagram). Because the JSD method permits two-level descriptions (network and process levels) two-level SA dataflow diagrams can also be formed. The transformation in the opposite direction, from SA to JSD, demands expert knowledge. Here the main difficulty lies in grouping of the primitive processes of SA into JSD actions in structure diagrams.

To verify the results above, in Chapter 5 empirical experiments were carried out using a well-known case study, students' laboratory assignments, and randomly generated data simulating real data. The randomly generated test material contained versatile data for testing all types of FPA functions. In the experiments it was found that (1) the FPA algorithms of Chapters 3 and 4 produce statistically[1] correct results and the same results within different system descriptions (SA and JSD) if the descriptions are prepared accurately enough; and (2) after the transformation from SA specification to JSD specification and vice versa, the measurement produces statistically the same result as before the transformation. These results prove the "soundness" of the structure of this thesis as defined by the problem formulation in Section 1.3.1.

In all, the use of Albrecht's FPA has been made easier, earlier and more general during software development. The process of measuring product size has been totally automated, and it has been made possible to compute FPA from two equivalent descriptions of the analysis phase. The results, together with the implemented testing environment, provide many possibilities for future research. It is now possible to develop and study local cost

---

[1] As statistical methods, correlations and regression analysis were applied.

estimation environments, to generalize FPA further by building it into new methods, and to explore the possibilities for more general methodical equivalence and measures.[1]

REFERENCES:

Adamov R. & Bauman P. 1987. *Literature Review on Software Metrics.* Institut für Informatik der Universität Zürich, Nr. 87.03, October.

Adamov R. & Richter L. 1990. A Proposal for Measuring the Structural Complexity of Programs. *Journal of Systems Software*, 12, 55-70.

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 R. D., Kauffman R. J., and Kumar R. 1990a. *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.

---

[1] First step is taken in Hakkarainen et al. (1993).

Banker R., Kauffman R. J., and Kumar R. 1990b. *Output Measurement Metrics in an Object-Oriented Computer Aided Software Engineering (CASE) Environment: Critique, Evaluation and Proposal.* Center for Research on Information Systems, New York University, Information Systems Department, Leonard N. Stern School of Business.

Basili V. R. 1981. Resource Models. *Programming Productivity: Issues for The Eighties*, Jones C. (Ed.), 110-115, IEEE.

Baxter R. S. 1976. *Computer and Statistical Techniques for Planners*. Methuen & Co Ltd.

Behrens C. A. 1983. Measuring the Productivity of Computer Systems Development Activities with Function Points. *IEEE Transactions on Software Engineering,* SE-9(6), November, 648-652.

Boehm B. W. 1987. Improving Software Productivity. *Computer,* September, 43-57.

Bourque P. & Côté V. 1991. An Experiment in Software Sizing with Structured Analysis Metrics. *Journal of Systems Software*, 15, 159-172.

Cameron J. R. 1986. An Overview of JSD, *IEEE Transactions on Software Engineering*, SE-12(2), February, 222-240.

Cameron J. R. 1988. The modelling phase of JSD, *Information and Software Technology*, 30(6), July/August, 373-383.

Cameron J. R. 1989. *JSP & JSD: the Jackson Approach to Software Development*, IEEE Computer Society Press.

Conte S. D., Dunsmore H. E., and Shen V. Y. 1986. *Software Engineering Metrics and Models*, The Benjamin/ Cummings Publishing Company, Inc.

Cowan J. B. 1990. Quality assurance potential of analyst/designer workbenches, *Information and software technology*, 32(1), January/February, 46-52.

Curtis B., Sheppard S. B., Milliman P., Borst M. A., and Love T. 1979. Measuring the Psychological Complexity of Software maintenance Tasks with the Halstead and McCabe Metrics, *IEEE Transactions on Software Engineering*, SE-5(2), March, 96-104.

DeMarco T. 1978. *Structured analysis and system 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. & Sarson T. 1979. *Structured systems analysis: tools and techniques*, Prentice-Hall.

Hakkarainen J., Laamanen P., and Rask R. 1993. Neural Networks in Specification Level Software Size Estimation. To be published.

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

Harrison W. 1990a. *Insights on Improving the Maintenance Process Through Software Measurement.* Department of Computer Science, Portland State University School of Engineering and Applied Sciences, TR 90-4, March.

Harrison W. 1990b. *Using Metrics to Allocate Testing Resources in a Resource Constrained Environment*. Department of Computer Science, Portland State University School of Engineering and Applied Sciences, TR 90-5, April.

Heemstra F. J. & Kusters R. J. 1991. Function point analysis: evaluation of a software cost estimation model. *Eur. J. Inf. Systs.*, 1(4), 229-237.

Henry S. & Kafura D. 1981. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, SE-7(5), September, 510-518.

Hunter R. 1990. Lecture 1: Software Measurement. *Software Tools 1990*: *The Practical Use of Software Metrics*. Wembley Conference Centre, London, 12-14 June, Blenheim Online.

Hyttinen T. 1990. *Verkkojenkäsittelyohjelman graafinen käyttöliittymä.* Erikoislaboratoriotyö, Joensuun yliopisto, Tietojenkäsittelyopin laitos.

IEEE Standard 982.2 1988. Standard Dictionary of Measures to Produce Reliable Software. *Software Engineering Standards*. Third edition. IEEE.

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

Jones C. 1986. *Programming Productivity*. McGraw-Hill, New York.

Kitchenham B. 1990a. Measuring Software Development. *Software Reliability Handbook* (Ed. P. Rook). Elsevier Science Publishers Ltd.

Kitchenham B. 1990b. Session 4: Software Project Cost Estimation. *Software Tools 1990*: *The Practical Use of Software Metrics*. Wembley Conference Centre, London, 12-14 June, Blenheim Online.

Laamanen P. 1991. *Tietovirtakaavion piirto- ja analysointiohjelma*. Erikois-laboratoriotyö, Joensuun yliopisto, Tietojenkäsittelyopin laitos.

Laamanen P. & Rask R. 1991. *A Prototype System for Automating the Analysis of SA Descriptions.* Report A-1991-2. University of Joensuu, Department of Computer Science.

Londeix B. 1987. *Cost Estimation for Software Developmen*t. Addison-Wesley.

Low C. G. & Jeffery D. R. 1990. Function Points in the Estimation and Evaluation of the Software Process, *IEEE Transactions on Software Engineering*, 16(1), January, 64-71.

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

McCabe T. J. & Butler C. W. 1989. Design Complexity Measurement and Testing, *Communications of the ACM*, 32(12), December, 1415-1425.

Mazzucco F. 1991. IEF - Automatic Function Point Count. *International Function Point Users Group Conference Proceedings*, Spring 1991, 168-181.

Nevalainen R. & Mäki H. 1992. An Empirical Software Cost Estimating Tool (LATURI-Tool). J. Leponiemi (ed.): *NordDATA'92 Proceedings*, 819-324.

Nunamaker J. F. & Chen M. 1989. Software Productivity: A Framework of Study and an Approach to Reusable Components. B. D. Shriver (ed.): *Proceedings of the 22nd Annual Hawaii International Conference on System Science*, vol. II, 959-968.

Pfleeger S. L. 1991. Model of Software Effort and Productivity. *Information and Software Technology,* 33(3), 224-231.

Poel van der K. G. & Schach S. R. 1983. A Software Metric for Cost Estimation and Efficiency Measurement in Data Processing Systems Development, *The Journal of Systems Software*, 3, 187-191.

Rask R. 1985. *Ohjelmamoduulin sisäisen ja ulkoisen monimutkaisuuden mittaus*. Lisensiaatintyö, Jyväskylän yliopisto.

Rask R. 1990a. 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. 1990b. One view of the transformation of JSD to SA, *Fourth International Workshop on Computer-Aided Software Engineering*, Advance Working Papers, (Ed. R. J. Norman & R. Van Ghent), 248-249, Irvine, California, 5-8 December.

Rask R. 1991a. 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.

Rask R. 1991b. *Algorithms for counting unadjsuted Function Points from dataflow diagrams.* Report A-1991-1. University of Joensuu, Department of Computer Science.

Rask R., Laamanen P., and Lyytinen K. 1992a. A Comparison of Albrecht's Function Point and Symons's Mark II Metrics. To be published.

Rask R., Laamanen P., and Lyytinen K. 1992b. Simulation and Comparison of Albrecht's Function Point and DeMarco's Function Bang Metrics in a CASE Environment. To be published.

Ratcliff B. and Rollo A. L. 1990. Adapting Function Point Analysis to Jackson System Development. *Software Engineering Journal*, January, 79-84.

Reiffer D. J. 1990. Asset-R: A Function Point Sizing Tool for Scientific and Real-Time Systems. *J. Systems Software*, 11, 159-171.

Shepperd M. 1988. An evaluation of software product metrics. *Information and software technology*, 30(3), 177-188.

Shepperd M. & Ince D. 1989. Metrics, outlier analysis and the software design process. *Information and Software Technology*, 31(2), March, 91-98.

Shoval P. 1988. ADISSA: Architectual design of information systems based on structured analysis, *Information Systems*, 13(2), 193-210.

Shoval P. & Manor O. 1990. Software engineering tools supporting ADISSA methodology for systems analysis and design, *Information and Software Technology*, 32(5), 357-369.

Symons C. R. 1988. Function point analysis: difficulties and improvements. *IEEE Transactions on software engineering*, 14(1), 2-11.

Symons C. R. 1991. Software Sizing and Estimating, Mk II FPA (Function Point Analysis). John Wiley & Sons.

Ward P. T. & Mellor S. J. 1985. *Structured Development for Real-Time Systems*, Volumes 1&2. Yourdon Press.

Ward P. T. & Mellor S. J. 1986. *Structured Development for Real-Time Systems*, Volume 3. Yourdon Press.

Woodman M. 1988. Yourdon dataflow diagrams: a tool for disciplined requirements analysis, *Information and software technology*, 30(9), November, 515-534.

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

Yu W. D., Smith P, and Huang S. T. 1990. Software Productivity Measurements. *AT&T Technical Journal,* May/June, 110-120.

APPENDIX A - A Model for Measuring and Controlling Software Productivity.

Productivity is generally defined as (Boehm 1987, Yu et al. 1990, Pfleeger 1991):

$$\text{Productivity} = \frac{\text{Outputs produced by the process}}{\text{Inputs consumed by the process}} \tag{A.1}$$

The inputs can represent the amount of work effort required to produce the final product, while the outputs represent the work product itself. Productivity improvement can be stated as a reduction in unit cost or as an increase in output per unit cost (Behrens 1983).

Productivity analysis is useful only if the software metrics are good, especially with regard to software size. The use of size metrics is possible only if they are examined in a large environment. Consequently, a model is needed to describe the process of measuring and analyzing software development.

The measurement object is a software system. Measurement must be possible in different development phases (DeMarco 1982, Banker et al. 1990a). As the system descriptions are refined, the metric values must be improved. In Figure A.1 the development phases - analysis, design, and implementation - are represented by the layered description. On each layer the same type of structure is applied, but different product metrics.

This thesis concentrates on measuring the analysis phase product, the foremost level in Figure A.1. The other levels are hidden, but they contain the same estimation phases (indicated by rectangles) and data sets (indicated by ovals). Only the descriptions of work products and size metrics used are different. The potential metrics for each development phase are summarized in Appendix K.

To estimate the size of a software system, a suitable product metric is applied to systems descriptions, i.e. to the work product. The purpose of the estimation of software size is to provide a basis for the estimation of software cost, i.e. the required work effort. The cost estimation function is a process metric. To obtain a reliable result, both estimates, product metric and process metric, need application knowledge in a suitable form.

The broken lines in Figure A.1 imply feedback in the process of productivity analysis.[1] By using historical size and work effort data, the cost equation can be changed when necessary.

---

[1] Kitchenham (1990b) regards feedback as the most important concept in cost estimation.

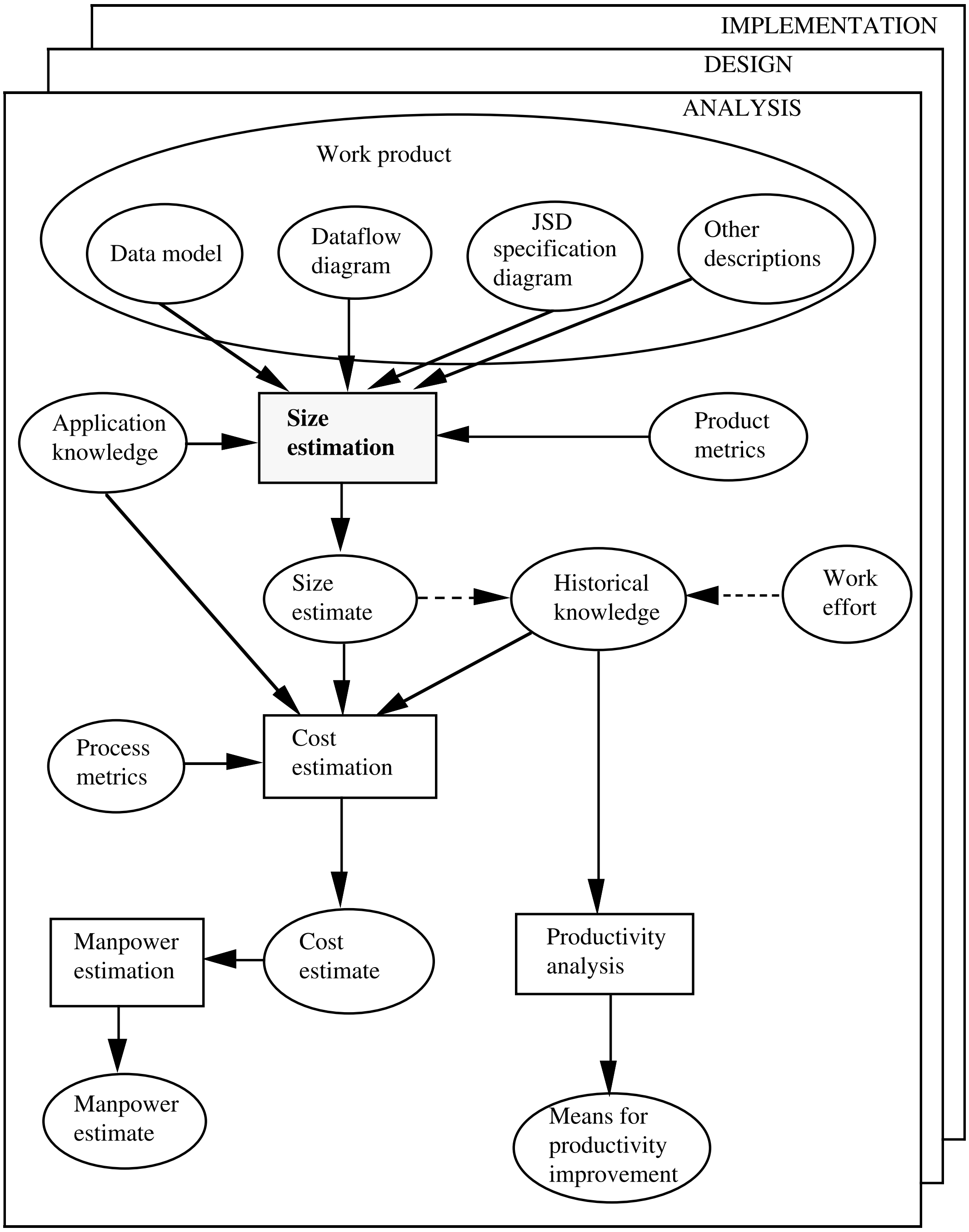


Figure A.1. Estimation of software size, cost and productivity.

IBM's Development Centre Productivity Measurement Guide (1984) regards the work effort reporting system as the most important and difficult aspect in the analysis of productivity. If the historical data of work effort and work product are not accurate and complete, the productivity calculations can be difficult to execute and the estimates may be unreliable. Otherwise, the estimation results can be analyzed to select and apply the best method and tool to decrease the costs in future projects. Figure A.2 simplifies the feedback connections of Figure A.1.

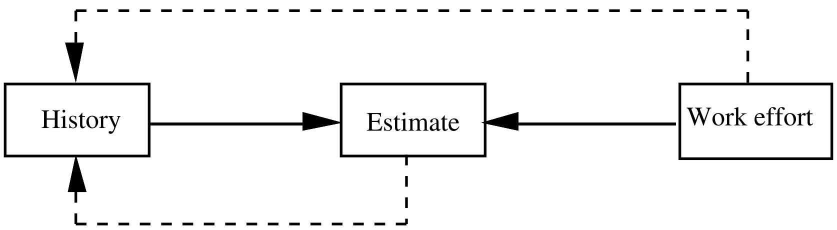


Figure A.2. Principles of a control and estimation system.

When there is an accepted size metric P, the software development costs C can be estimated in person-months by local linear or non-linear equations as suggested by DeMarco (1982), Conte et al. (1986), and Londeix (1987) as follows:

(A.2) $C = K + M * P$

(A.3) $C = K + M * P^E$

where E, K and M are environment-dependent constants.

From the cost estimate the manpower of the project tasks can be further specified and the tasks scheduled. The manpower estimation of a project could be based on the Rayleigh's distribution as a function of time *t* (Basili 1981, Londeix 1987):

(A.4) $m(t) = 2 * C * a * t * \exp(-a * t^2)$

where C denotes costs in man-months throughout the life of the software, including its maintenance (DeMarco 1982), and *a* is a design parameter which affects the form of the resulting distribution. This shape parameter *a* depends upon the point in time, i.e. the development time t_d, at which *m*(t) reaches its maximum as defined by the function:

(A.5) $a = 1/2t_d^2$.

APPENDIX B - FPA Detail and Summary Sheets.

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	Complexity Class	Number of Functions (F_i)		Complexity Factor, or weight (w_i)
OUTPUTS	Simple	F_1	*	4
	Average	F_2	*	5
	Complex	F_3	*	7
INQUIRIES	Simple	F_4	*	4
	Average	F_5	*	5

	Complex	F_6	*	6
	Complex	F_7	*	7
INPUTS	Simple	F_8	*	3
	Average	F_9	*	4
	Complex	F_{10}	*	6
FILES	Simple	F_{11}	*	7
	Average	F_{12}	*	10
	Complex	F_{13}	*	15
INTERFACES	Simple	F_{14}	*	5
	Average	F_{15}	*	7
	Complex	F_{16}	*	10

TUFPV = $\Sigma F_i * w_i$

APPENDIX C - Software System.

A *software system* serves a certain domain in the real world by implementing desired user functions via a user interface. Figure C.1 illustrates the software system containing an application part, a user interface layer, and input-output channels between the software system and the real world.The *application* part contains the automated user procedures which are instances of the real world transactions. A transaction in the software system is initiated by some event in the environment. The application part thus consists of the descriptions of a transaction set to implement the functions required by users. For example, in an invoicing system the user may request a dunning letter concerning the unpaid invoices. This triggers a transaction within the application part to find the unpaid invoice data and to produce the dunning letters. To implement this and other transactions, the software development phases are gone through and the required descriptions are produced.

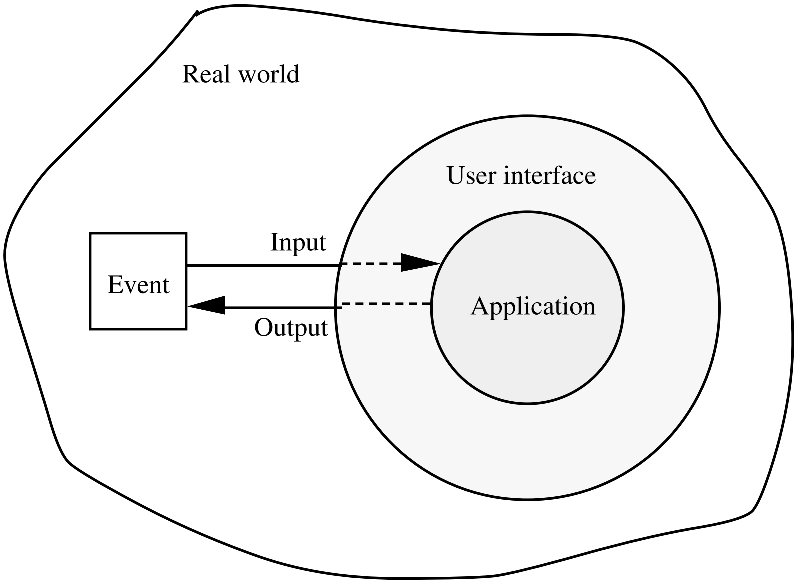
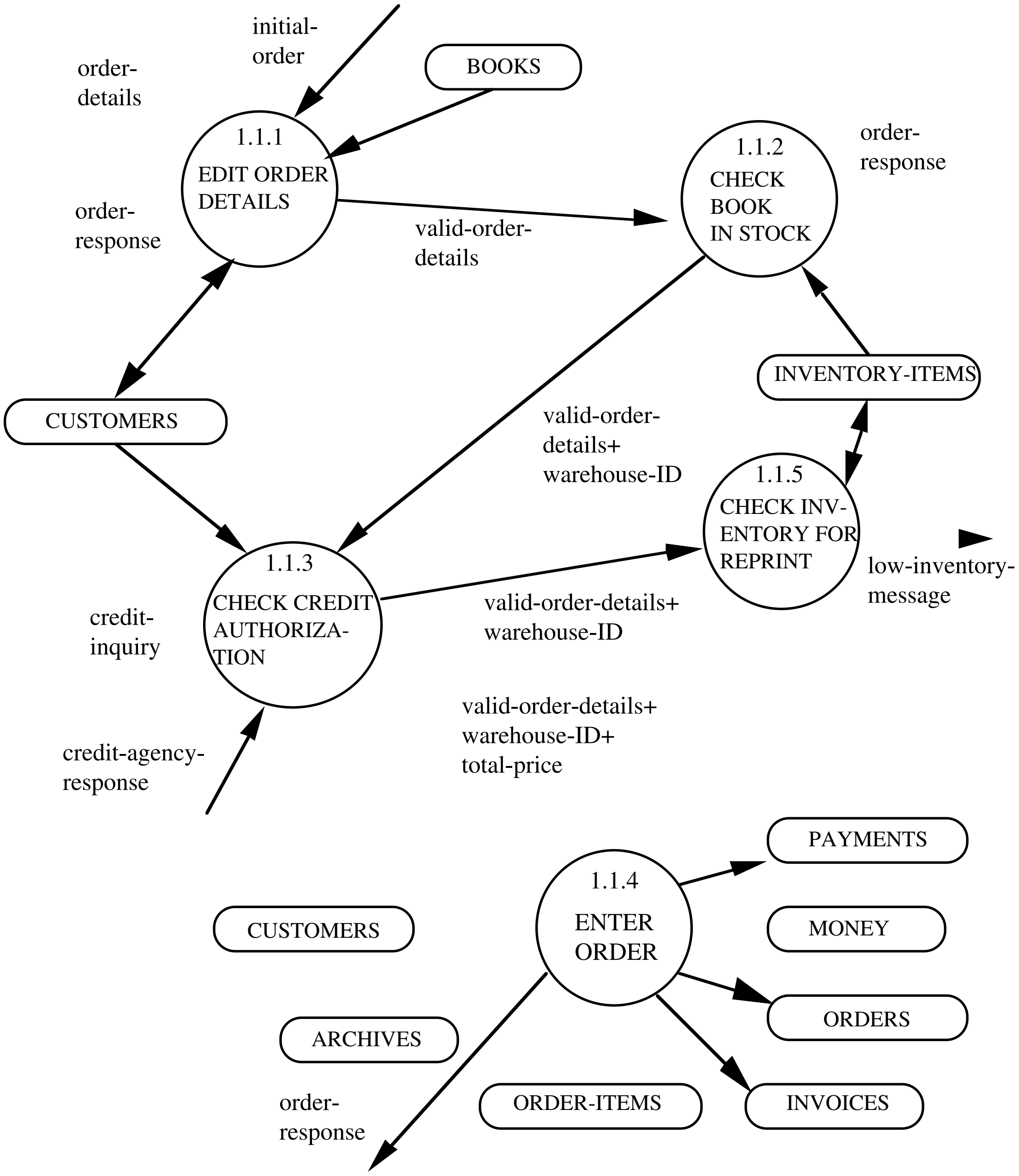


Figure C.1. Model for a software system.

The *user interface* part of the software system caters for human factors, ergonomics, and checks. For example, there may be pulldown menus, pop-up menus, and special input and inquiry screens to provide the services needed by the users. To ascertain of the validity of the data, different checking rules have to be designed and programmed. The implementation of all these components requires us to take into account human factors, i.e. mental and physical ergonomics. The implementation of an ergonomic user interface thus requires an additional effort.

APPENDIX D - Example of a Dataflow Diagram


initial-order
order-details
BOOKS
1.1.1
EDIT ORDER DETAILS
1.1.2
CHECK BOOK IN STOCK
order-response
order-response
valid-order-details
CUSTOMERS
INVENTORY-ITEMS
valid-order-details+ warehouse-ID
1.1.5
CHECK INV-ENTORY FOR REPRINT
1.1.3
low-inventory-message
CHECK CREDIT AUTHORIZA-TION
credit-inquiry
valid-order-details+ warehouse-ID
valid-order-details+ warehouse-ID+ total-price
credit-agency-response
PAYMENTS
1.1.4
ENTER ORDER
CUSTOMERS
MONEY
ORDERS
ARCHIVES
order-response
ORDER-ITEMS
INVOICES