A Prototype System for Automating Measurement and Verification of SA Descriptions

Petteri Laamanen and Raimo Rask

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

October 20, 1991

ABSTRACT

The size of a software system provides a basis for the estimation of software cost during software development. Hence, it is important to measure the system size reliably as early as possible. Two best known specification level metrics are Albrecht's Function Points and DeMarco's Function Bang which, however, lack automated estimation support. We demonstrate how the calculation of these metrics can be automated and describe a prototype assistant tool, which calculates these metrics from Structured Analysis (SA) descriptions. To assure the reliability of estimation, the SA descriptions must be verified before applying the automated metrics. In our prototype system, we have implemented a set of rules for automated checking of completeness and consistency of a structured specification.

Keywords: automation, CASE, estimation, software size metrics, structured analysis, specification verification.

1. INTRODUCTION

JoYCASE is a prototype Computer Aided Software Engineering (CASE) tool supporting graphical creation and automated analyzing of Structured Analysis (DeMarco 1978, Yourdon 1989) descriptions. It is able to count automatically two widely used software size estimators, *Function Bang* (DeMarco 1978) and *Unadjusted Function Points* (Albrecht 1979, Dreger 1990), from the specifications. We have implemented JoYCASE using Turbo Pascal 5.5 environment. The components of JoYCASE are shown in Figure 1.

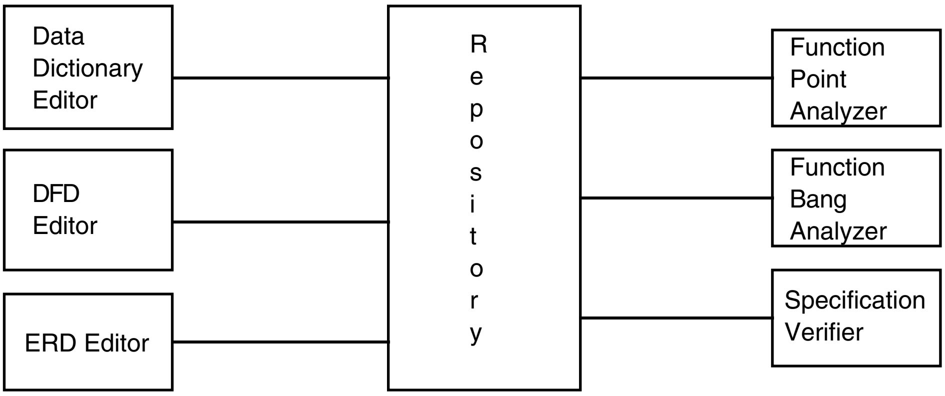

Figure 1. Components of JoYCASE.

A *Data Dictionary Editor* is a text editor which can be used to form a global data dictionary and local data dictionaries. The global data dictionary contains the common data specifications that can be used to form the local data dictionaries. The global data dictionary is updated by the new specifications of the local dataflow diagrams (DFDs) and entity-relationship diagrams (ERDs) of each local object system. The purpose of the global data dictionary is to guarantee the consistency of the data specifications. To specify the data items in the data dictionaries we use the notation used by Yourdon (1989). Currently, the data dictionary editor is not an integrated part of JoYCASE, instead the analyst must quit JoYCASE to create the data dictionary specifications.

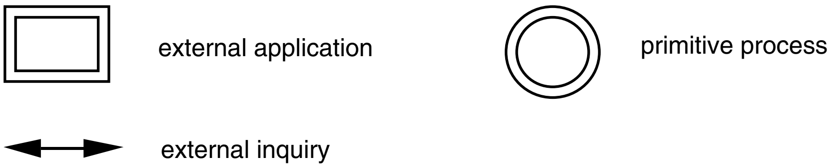

Figure 2. New symbols for dataflow diagrams

A *DFD Editor* and an *ERD Editor* are graphical diagram editors that use the notations suggested by Yourdon (1989) with three notable exceptions. We use unique symbols to describe an external application, a primitive process, and an inquiry which make for us possible to count software sizes. The new symbols of dataflow diagrams are shown in Figure 2. The graphic user interface is explained in chapter 2.

A *Function Point Analyzer* is a module that counts Albrecht's (1979) Unadjusted Function Point value from DFDs and ERDs. We do not give here the exact algorithms to count function points from these Structured Analysis descriptions but we demonstrate the counting procedure by an example in chapter 3. The precise algorithms to count the Function Point value are given by Rask (1991).

A *Function Bang Analyzer* calculates the value of DeMarco's (1982) Function Bang metric. JoYCASE system interprets each elementary or self-defining data item as a token in DeMarco's terminology. We are not trying to find out whether a data item can be further divided within a primitive. We will demonstrate the counting procedure of Function Bang in chapter 4.

A *Specification Verifier* checks the correctness of data dictionaries, DFDs and ERDs. Cowan (1990) considers the consistency and completeness checks of dataflow diagrams and gives 17 rules for the purpose. Also Yourdon (1989) gives a list of balancing rules between and within Structured Analysis descriptions. We have implemented a part of these rules into the JoYCASE system. In addition we have implemented the specific rules needed by the Function Point Analyzer. In chapter 5 we demonstrate the various consistency and completeness checks of our Specification Verifier.

A *Repository* includes the global data dictionary and the local data dictionaries. In addition, the repository contains all the DFD and ERD descriptions. Each dictionary has its own text file and each diagram has its own binary file for nodes and edges. In this paper we have used the repository as a common name for all these different files. The internal representation for data structures is given shortly in chapter 6.

2. GRAPHIC USER INTERFACE

The user interface of the JoYCASE system is graphic. The general layout of the JoYCASE screen is shown in Figure 3. The screen is divided into five permanent windows and five temporal windows. The temporal windows appear onto the permanent windows when they are activated. The temporal windows are shown shadowed in Figure 3. Figure 4 gives a complementary example of a JoYCASE system screen to illustrate the discussion below.

A *Diagram Drawing Area* is used by the system analyst to create a new or to edit an old DFD or ERD. Activation of any symbol of the currently displayed diagram is done by the mouse. After activating the symbol, the analyst can choose an operation from the *Diagram Drawing Tools Menu* to be executed on that symbol. The available tools are shown as icons. What icons are shown, depends on the diagram editor - DFD or ERD editor - the analyst is using. In Figure 4 we have the DFD editor tool icons.

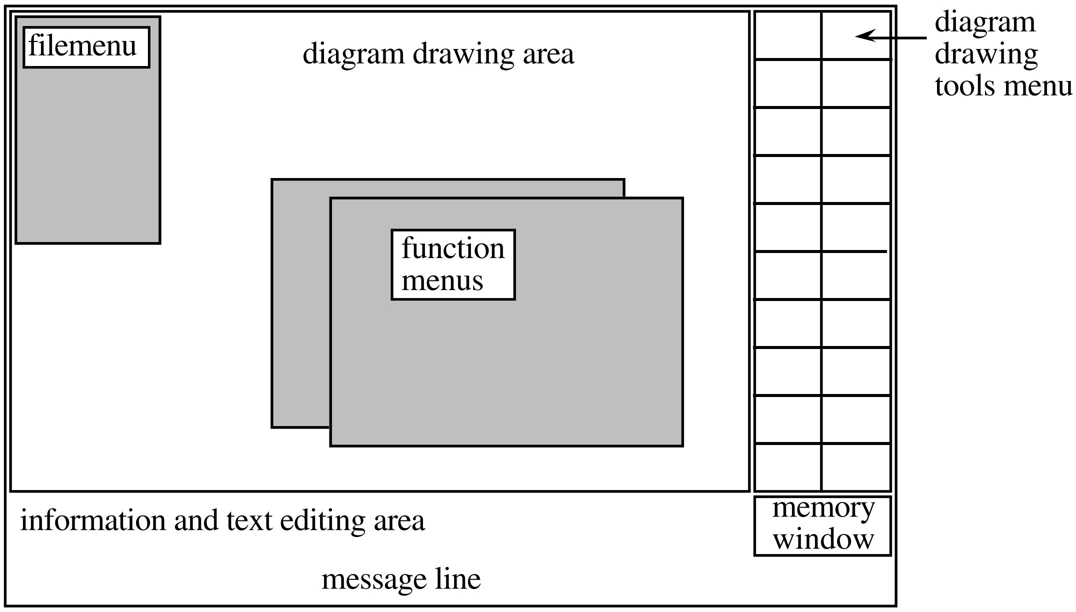

Figure 3. The general layout of the JoYCASE screen.

An *Information and Text Editing Area* is for showing general information about the current specification and system status. Any input of text information is displayed in this area. When not entering text information, the area shows the current date, name of the current specification, available disk space of the working disk drive, and the current working directory. A *Message Line* is meant for displaying all kinds of prompts and error messages for the analyst. A *Memory Window* shows the currently available free main storage space in bytes. The displayed amount is updated by each operation that causes either allocation or deallocation of memory.

Function Menus are for displaying the general functions that are available for the analyst. These menus appear in the middle of the screen when they are needed. The *Main Menu* of JoYCASE system is shown in Figure 4. In addition, there are submenus for Disc functions, Analyzing (estimation and verifying) and Reporting (estimator summaries). For Disc functions we have an additional File Menu (see Figure 3) for displaying all available specifications in the current working directory. We will consider the user interface of the main functions in the following sections.

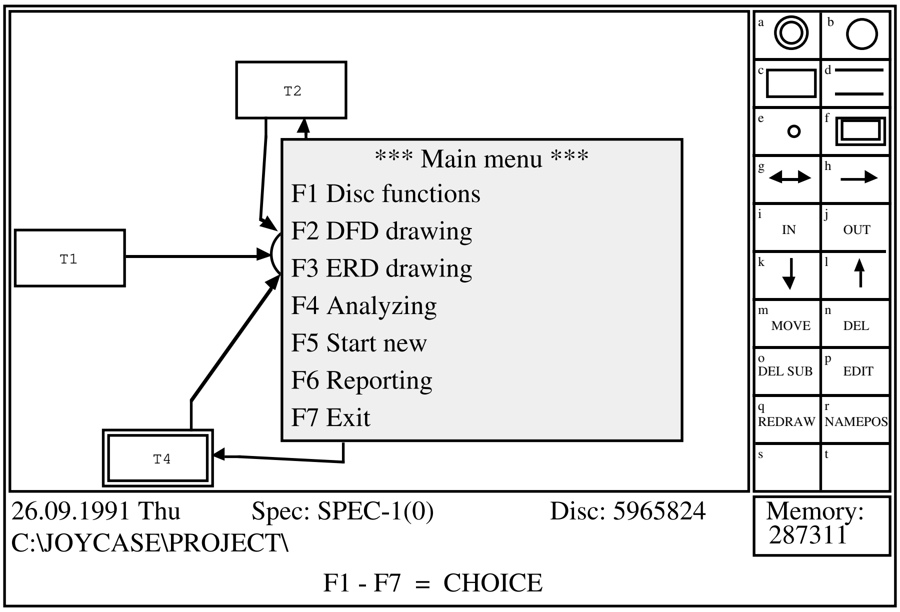

Figure 4. An example of a JoYCASE system screen.

2.1. Drawing of Dataflow Diagrams

The DFD editor is activated through the Main Menu in Figure 4 by choosing the function *F2 — DFD drawing*. Dataflow diagrams construct a hierarchy of diagrams, but only one subdiagram at a time can be shown and edited in the Diagram Drawing Area. To create a subdiagram for a process or to move to an existing subdiagram of a process the analyst must first activate the process and then choose the tool labeled by character *k* from the Diagram Drawing Tools Menu. This causes the current subdiagram to disappear from the screen, and the new subdiagram (maybe empty, if it was just to be created) to be drawn onto the screen. From now on, all tools apply to the new subdiagram displayed in the diagram drawing area. To move back from subdiagram to its parent diagram the analyst must use tool *l*.

The tools labeled *a-j* are for inserting new symbols to the diagram. The flow insertion tools (*g-j*) are also used for setting the parent—child relations (balancing) between dataflows of the current subdiagram and the dataflows entering or leaving the parent process of the current subdiagram. The tool labeled *e* is for inserting a dataflow diverging point that is presented by a small circle. The diverging point can be activated as any other symbol in the dataflow diagram and dataflows can enter to or leave from it.

The tools labeled *m-r* are for maintenance of an existing diagram. By tool *m* the analyst is enabled to move a node from one position to another with the dataflows entering or leaving that node moving with the node. With tool *n* the analyst can delete an activated symbol from the current subdiagram. Tool *o* deletes the subdiagram of the active process. Tool *p* is for editing the labels of the current subdiagram symbols and tool *q* causes the entire subdiagram to be redrawn onto the Diagram Drawing Area. With tool *r* the analyst can change the position of the name of an active dataflow if the relative position of the name was changed when moving nodes from place to another.

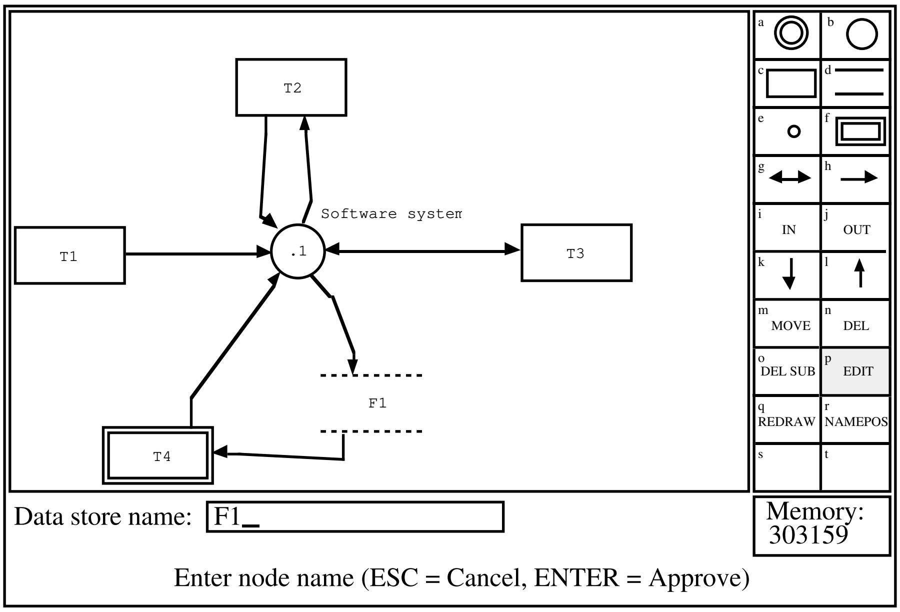

Figure 5. A sample screen of the DFD editor.

The size of the edited diagram is not limited by the size of the Diagram Drawing Area. The analyst can *scroll* the diagram drawing area *vertically* by PageUp and PageDown keys and *horizontally* by simultaneously pressing the Control key and the Left- or RightArrow key. The analyst can also *zoom* the size of the diagram symbols by the Insert (*enlarging*) and Delete (*reducing*) keys.

An example of a DFD editor screen is shown in Figure 5, where the analyst has activated a data store node F1 (here activation is highlighted by a hashed line, although the highlighting is actually illustrated by red colour) and he/she desires to edit the name of the data store (the chosen edit tool p is highlighted here by shadowing the corresponding icon box, actually the highlighting is done by drawing a rectangle around the icon box with yellow colour). The editing of the data store name is performed in a text editing field which appears onto the Information and Text Editing Area.

2.2. Drawing of Entity-Relationship Diagrams

The ERD editor is activated through the Main Menu function *F3 — ERD drawing*. If the specification already contains an ERD, then this is brought into the Diagram Drawing Area. The analyst can now insert new symbols to the diagram or edit the old ones.

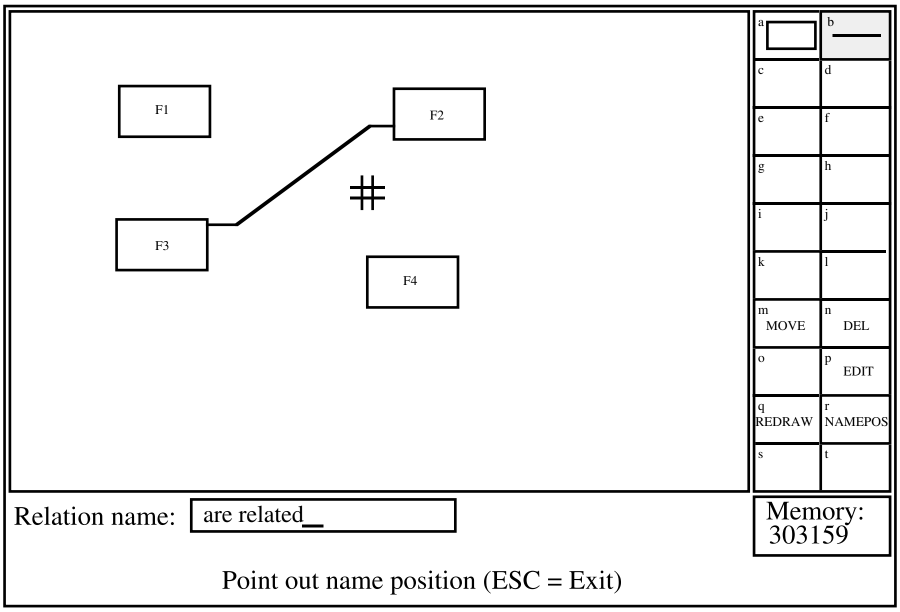

Figure 6. A sample screen of the ERD editor.

The tool icons differ from the ones in the DFD editor (see Figure 6). Tools labeled *a* and *b* are for inserting new symbols into the diagram. Tool *a* is for inserting new entities and tool *b* for inserting new relationships between entities. Tools *m, n, p, q* and *r* have the same meaning as in the DFD editor. Also the ERD editor also enables the scrolling, enlargement, and reduction of the current ERD. A notable difference between the ERD editor and the DFD editor is that the ERD editor does not support hierarchical ERDs.

In Figure 6 there is a sample screen of the ERD editor. The analyst has chosen the tool *b* for inserting a new relationship to the ERD and has pointed out the entities (F2 and F3) between which the relationship was to be inserted. The analyst has also entered the name, in the Information and Text Editing Area, for the relationship. Now the analyst is prompted to point out the position for the name of the inserted new relationship. The analyst can move the cross mouse cursor to a suitable place and then set the name position by pressing the mouse button.

2.3. Disc Functions

The Disc Functions Menu is activated through a Main Menu choice *F1 — Disc functions*. The analyst can choose between five disc functions: saving, renaming, retrieving or erasing a specification, and changing the working directory and disc drive. Renaming and saving apply to the current specification only. Retrieving and erasing can be applied to any specification that is in the working directory. If the analyst chooses the retrieval or the erasing function, then the File Menu appears into the top left corner of the Diagram Drawing Area. The File Menu is illustrated by Figure 7. The analyst can scroll the names of the available specifications with the arrow keys and choose the

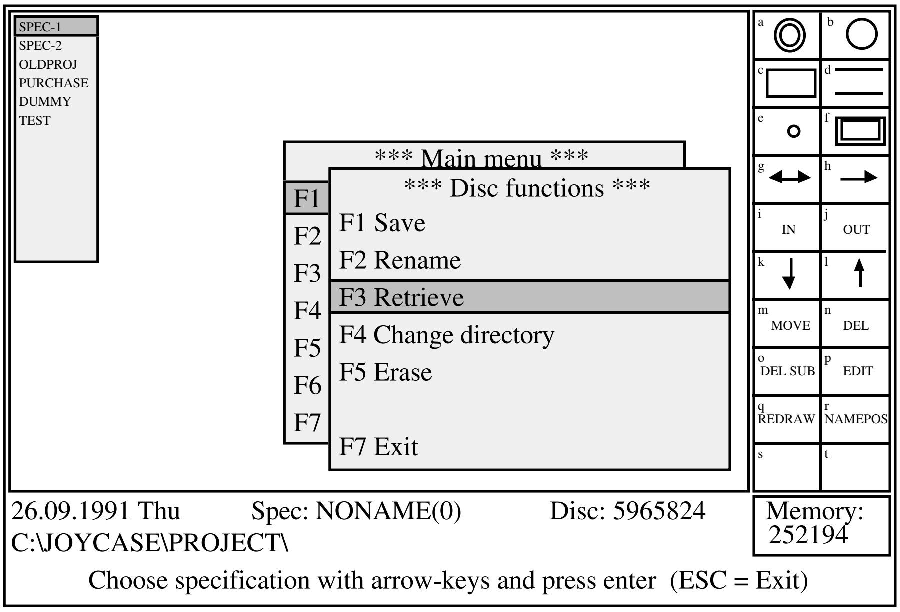

Figure 7. A sample screen with the Disc Functions Menu.

desired specification. The retrieval function includes - in addition to reading the chosen specification (DFD, ERD and data dictionary) into main storage - checking of the data dictionary. If any data dictionary errors are detected, the corresponding error messages are stored into an external ASCII text file. Regardless of the validity of the data dictionary, the JoYCASE reads the ERD and the DFD into main storage and draws the DFD into the Diagram Drawing Area.

2.4. Analyzing Specifications

The JoYCASE system is able to automatically estimate the size of the software that is going to be derived from the created structured specification. The estimators are collected to the *Analyzing Menu* which is opened with the Main Menu choice *F4 — Analyzing*. The implemented estimators are DeMarco's (1982) Function Bang and Albrecht's (1979) Function Points. Each time an estimation function is called by the analyst, the JoYCASE system first checks the validity

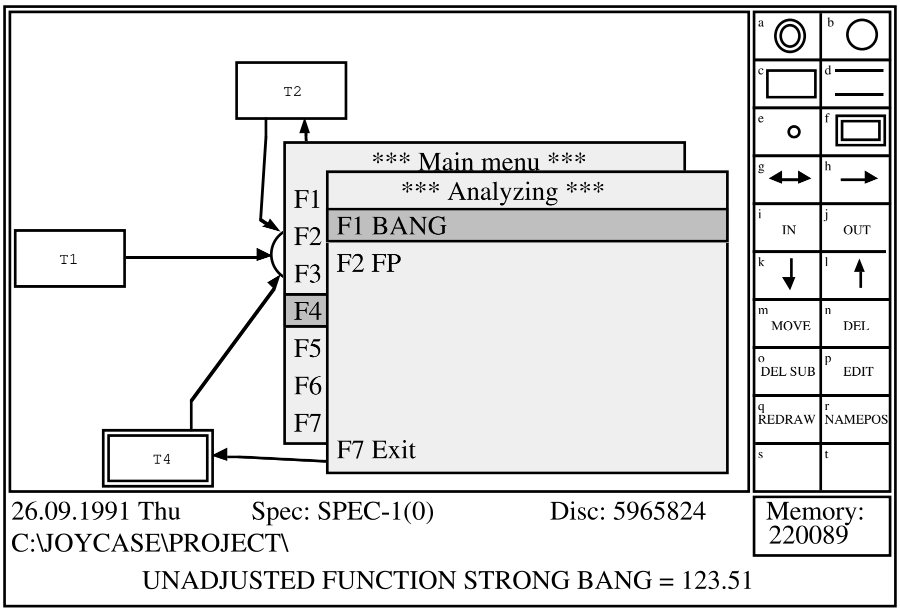

Figure 8. A sample screen with the Analyzing Menu.

of the specification to be estimated. If any errors are found, the corresponding error messages are written into an external ASCII file and the chosen estimator is not counted. Currently, the only way to explore the external error file is to quit JoYCASE and explore the errors on the operating system level. If the specification is found to be valid, then the JoYCASE system prompts the analyst, whether to print a summary of the calculation. Whatever the answer is, the estimator is counted, and the final result is displayed on the Message Line at the bottom of the screen. This is illustrated in Figure 8.

2.5. Reporting

The JoYCASE system supports limited automated documentation of the specifications. It is possible to print out DFDs and ERDs directly from the screen, or to use some graphics screen capture utility to save the screens into files, the contents of which can be manipulated later on and used as part of documentation created with different tools. From the *Reporting Menu* activated through the Main Menu choice *F6 — Reporting* the analyst can choose either to print out summaries of the results given by the available estimators or tables of the nodes and edges in the currently active diagram (DFD or ERD). The contents of the Reporting Menu is shown in Figure 9 where the analyst has chosen the diagram contents printing utility.

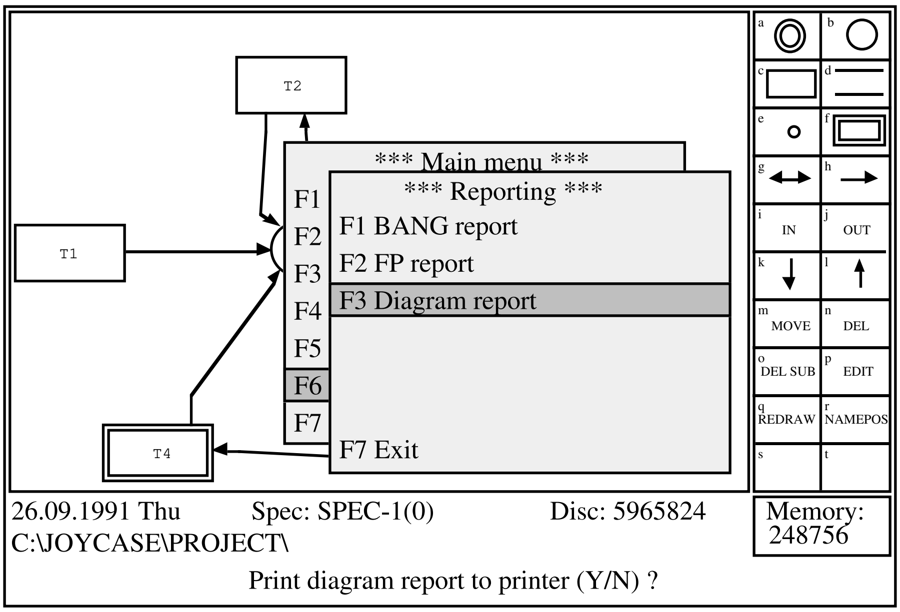

Figure 9. A sample screen with the Reporting Menu.

The reports on the consistency and completeness of the specification are created automatically. As described in section 2.3 the data dictionary is checked, and the possible error messages stored into an external ASCII file, when the specification is retrieved. The rest of the specification is thoroughly checked when the analyst chooses to estimate the size of the current specification or to print an estimator report. In chapter 4 we will declare all the checks the JoY-CASE system performs on the created structured specifications (data dictionaries, DFDs and ERDs).

3. FUNCTION POINT ANALYZER

A widely applied measurement method is Albrecht's (1979) *Function Point Analysis* (FPA). FPA measures the size of an application from an end-user perspective. A *function point* is defined as one end-user business function. The 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.

The functions can be organized into the following five groups (Albrecht & Gaffney 1983, Dreger 1989): external input types, external output types, external inquiry types, logical internal file types and external interface file types.

To analyze these functions, we first identify all unique functions and classify them into the groups above. Then we weight each function by its level of complexity: simple, average, or complex. The sum of these weights is called a *total of unadjusted function points*.

The classification and weighting of inputs, outputs, and inquiries depends on the number of files referenced and on the number of data items referenced. Correspondingly, the classification and weighting of logical internal files and external interface files depends on the data items referenced and on the logical record formats of the files or logical relationships between the files. Exact detail sheets (Dreger 1989) for function classification and weighting are found in the Appendix.

In the following we will apply the algorithms given by Rask (1991) to illustrate the operation of the Function Point Analyzer of our prototype system. For this purpose we have Figures 10, 11 and 12 and Table 3 that contain a structured specification produced by the editors of the JoYCASE system. In Figure 10 we have the context diagram of our sample specification. The context diagram is decomposed into five subprocesses in Figure 11. The definitions of the dataflows and data stores are shown in the data dictionary of Table 3. The ERD of our example is shown in Figure 12. The results given by the Function Point Analyzer are shown in Table 5.

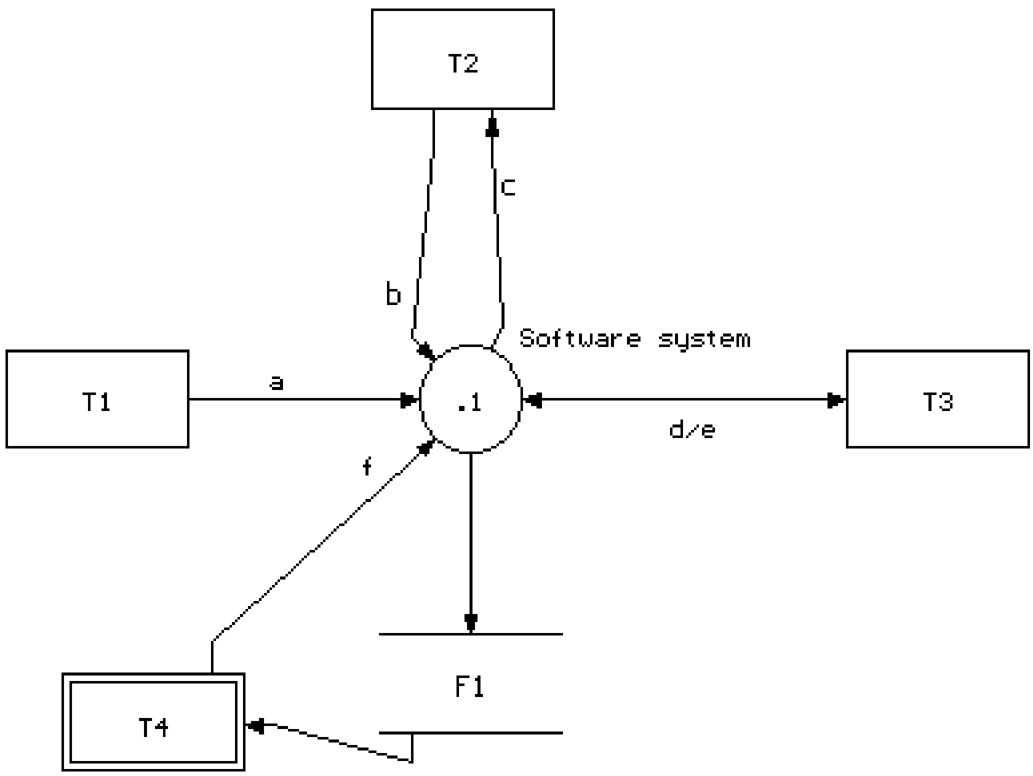

Figure 10. Context diagram of the sample specification.

For logical files we use a notation <F,K> where F means a data store and K means a key by which the data store F is referenced. A new logical record format of a certain logical file is found, if the referencing flow has the same key as the logical file, and the name of the referencing flow differs from the names of the other flows referencing to the same logical file. The unnamed dataflows are interpreted to have the same contents as the file they are referencing.

Figure 10 shows that outside the system boundary we have four terminators T1,...,T4 and an external data store F1. There is one external input *a* from terminator T1, one external input *b* from terminator T2 and one external input *f* from external application T4 into the system.

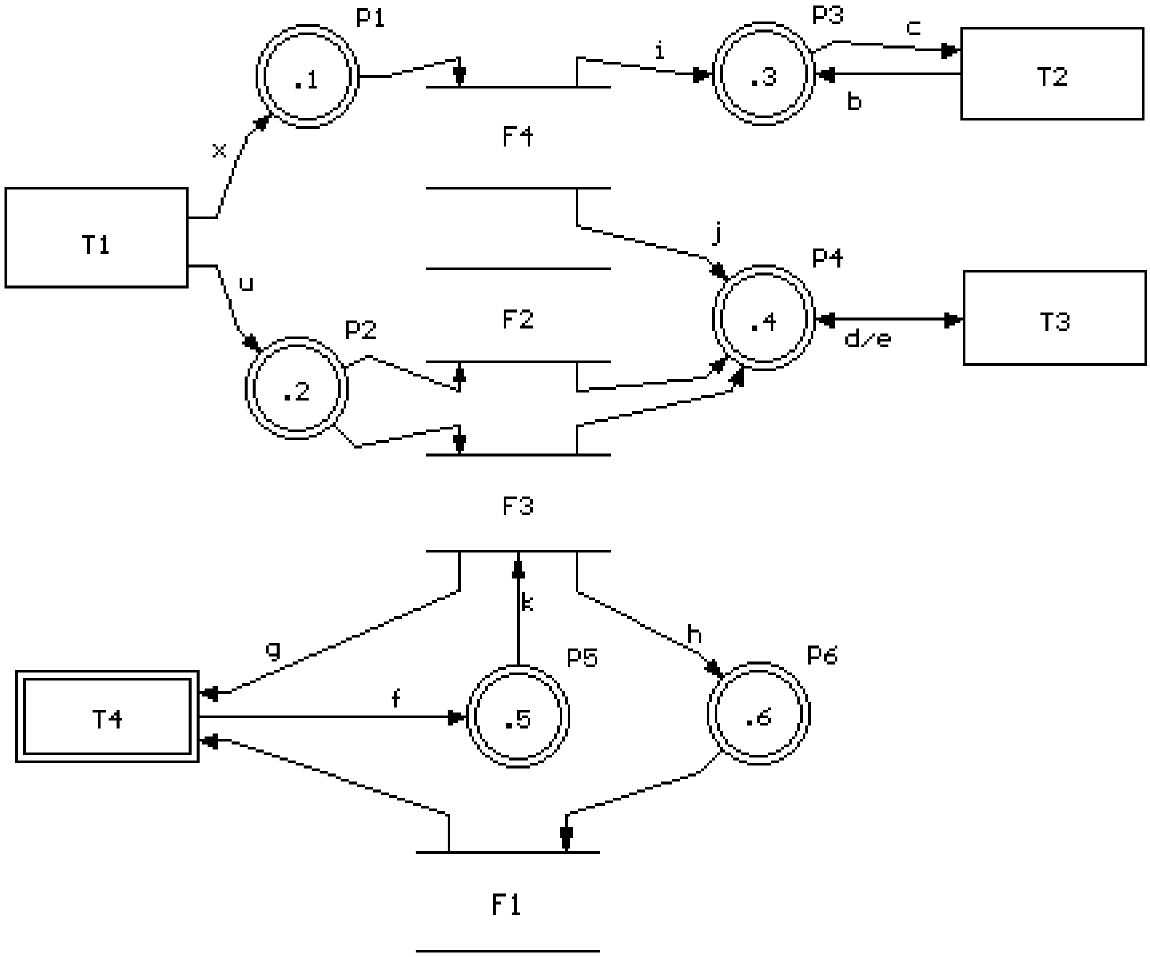

Figure 11. Decomposition of the context diagram.

In our example there are four *external inputs* at the lowest level DFD. Input flow *x* references one logical file <F4,x1>, flow *u* two logical files <F2,z1> and <F3,y1>, flow *b* one logical file <F4,x1> and finally flow *f* one logical file <F3,y1>. The sizes of input dataflows *x*, *u*, *b* and *f* can be counted from the data dictionary of Table 3; the calculation gives the sizes to be 5, 9, 2 and 3 elementary data items. Based on the sheets in the Appendix the unadjusted function points for these input dataflows will be 3, 4, 3 and 3 correspondingly (3 = a simple function, 4 = an average function).

The only *external output* dataflow *c* references only a logical file <F4,x1> and from the data dictionary we find its size to be four elementary data items. Thus the unadjusted function points for that output dataflow will be four (simple).

In the context diagram of Figure 11 there is one *external inquiry* between the software system and terminator T3. The inquiry contains an input part *d* and an output part *e*. From Figure 12 and Table 3 we can see that the inquiry references to logical files <F2, z1>, <F3,y1>, and <F4,x2>. The data dictionary of Table 3 tells the size of the input part *d* to be one and the size of the output part *e* to be six elementary data items. The complexity of the input part is thus four (average) and the complexity of the output part is five (average). The final value of unadjusted function points for the inquiry is five (the maximum of the input and output part complexities).

External application T4 has three *external interface* connections. Data store F3 is a shared interface file, data store F1 is an external interface file and dataflow *f* is an input to the system in form of a transaction file.

Table 1. The data dictionary of the sample specification.

```
a   = [x|u];                                      f   = t1 + t2 + t3;
u   = y + z;                                      k   = @y1+y4+y5+y6;
F4  = x;                                          g   = @y1+y2+y3+y4+y5+y6;
F3  = y;                                          x1-range = r1 + r2;
F2  = z;                                          i   = @x1 + x2 + x5;
F1  = h;                                          j   = @x2 + x3 + x4;
h   = @y1 + y2 + y6;                              b   = x1-range;
x   = @x1 + x2 + x3 + x4 + x5;                    c   = x1 + x2 + x5 + x-sum;
y   = @y1 + y2 + y3 + y4 + y5 + y6;               d   = x2;
z   = @z1 + z2 + z3;                              e   = x2+x3+x4+y3+y4+y6;
x-sum = **;
x1 = **;                x2 = **;                  x3 = **;
x4 = **;                x5 = **;
y1 = **;                y2 = **;                  y3 = **;
y4 = **;                y5 = **;                  y6 = **;
z1 = **;                z2 = **;                  z3 = **;
t1 = **;                t2 = **;                  t3 = **;
r1 = **;                r2 = **;
```

From the ERD of Figure 12 we can gather, that the *shared interface file* F3 has four relationships and from the DFD of Figures 11 and 12 we can find the number of logical record formats of the shared file to be four. The number of logical record formats for the interface is the number of logical record formats of the logical file that has the same key as the dataflow, that references to the shared file. In our example the key of the referencing flow *g* is *y1*. Data store

F3 has only one logical file which has the key *y1*, and there are four logical record formats for that logical file <F3,y1>. Three of the referencing dataflows (g, k, and h) are named and two have no name. Thus there is a total of four logical record formats in this interface. Now we have counted both the number of relationships and logical record formats for the external interface of the shared file F3 to be four; the maximum of these two stays four. The number of data items the external application T4 accesses from file F3 by dataflow *g* is six. The number of interface points for the shared file F3 will thus be five (simple).

Another interface is counted for the *external interface file* F1. It has only one logical record format (it is referenced only by one key, the primary key of the file) but from the ERD of Figure 12 we count the number of relationships to be two. Because the maximum of these weighting values is two, and the number of referenced data items of the external interface file F1 is three, the number of unadjusted function points for this interface will be five (simple).

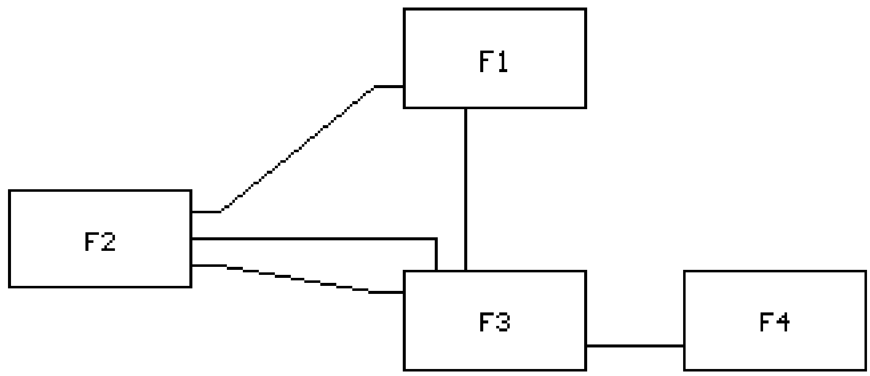

Figure 12. ERD of the sample specification.

The third and last interface will be counted for the external input *f*, which is sent to the system in form of a *transaction file*. Interface points for a transaction file are counted only if it is converted. A *conversion* is made, if the input flow of the transaction process does not balance with the output flows of the transaction process. They do balance, if the input flow is, say A, and the label A is defined as A = [B|C|D] or A = B+C+D in the data dictionary, and the output flows leaving the transaction process are labeled with some combination of B, C and D. Should any other labels, including blank labels, be used for the output flows, then our prototype system interprets that the transaction file has been converted.

The transaction process P5 in Figure 11 converts the transaction input flow *f* into flow *k*. That is why we need to count the interface points for this input flow. From the data dictionary definition (f = t1 + t2 + t3) we find out, that flow *f* (transaction file) has only one logical record format and the number of elementary data items in flow *f* is three. Hereby, the amount of function points for this interface will be five (simple). In any case no file relationships can be found for transaction files.

Finally, we have to count the *logical internal files* of our sample system. By examining the DFD and the data dictionary, we can gather the logical files in the system. First we must find out what data stores of the DFD are such that their logical files should be counted as internal. Data stores F2 and F4 are clearly internal, because only the system references to them. Data store F3 is shared because it is referenced by an external application but it should be counted as internal because it is also referenced by the system. Data store F1 is completely external, but since it is sent as an output to an external application, it should be counted as an internal file (Dreger 1990). For these reasons we should count all logical files from our sample specification. The logical files of our sample specification, their keys and logical record formats are shown in Table 4.

Table 2. Logical files of the sample specification.

Data Store	Access Key	Logical Record Formats
F1	y1	F1
F2	z1	F2
F3	y1	F3 k h g
F4	x2	j
F4	x1	F4 i

As already explained before, the logical files of a certain data store are determined by the different access keys used in the dataflows that reference to the data stores and the name of the referenced data store. A certain logical file can itself contain a number of logical record formats. These logical record formats are gathered from the dataflows that reference to the data store that is under examination. If two dataflows refer to the same data store, and they have the same key, then they refer to the same logical file. But if the names of these two dataflows differ from each other, then we have got two logical record formats for the logical file, that is identified by the common key of the referencing dataflows. For any logical file <F,K> the number of relationships is the same as the number of relationships of the physical data store F. The size of a certain logical file is the number of elements in the union between sets of elementary data items; the sets are composed of the elementary data items of the logical record formats of the corresponding logical file.

Table 3. Summary of the FPA.

Business Function	Number	Complexity		Factor		TOTALS Line	Group
OUTPUTS	1	SIMPLE	*	4	=	4	
	0	AVERAGE	*	5	=	0	
	0	COMPLEX	*	7	=	0	
						4	4
INPUTS	3	SIMPLE	*	3	=	9	
	1	AVERAGE	*	4	=	4	
	0	COMPLEX	*	6	=	0	
						13	13
INQUIRIES	0	COMPLEX	*	6	=	0	
	1	SIMPLEX	*	4	=	4	
	0	AVERAGE	*	5	=	0	
	0	COMPLEX	*	7	=	0	
						4	4
FILES	5	SIMPLE	*	7	=	35	
	0	AVERAGE	*	10	=	0	
	0	COMPLEX	*	15	=	0	
						35	35
INPUTS	3	SIMPLE	*	5	=	15	
	0	AVERAGE	*	7	=	0	
	0	COMPLEX	*	10	=	0	
						15	15
TOTAL UNADJUSTED FUNCTION POINTS =							**71**

In our example, the external interface data store F1 has only an unnamed dataflow referencing to it. Hereby, the data store F1 has only one logical file <F1,y1> and the logical file has just one logical record format (F1). Because the maximum of the number of the relationships (two) and the number of logical record formats (one) is two, and the size of the logical file is three (F1 = @y1 + y2 + y6), we get the unadjusted function points for logical file <F1,y1> to be seven (simple).

In the same manner we can count the function points for the logical file <F2,z1> to be seven (F2 has three relationships, the logical file has just one logical record format, and the size of the logical file is three). Data store F3 has as well only one logical file <F3,y1>. Now there are, however, a total of four logical record formats in the logical file. Because the number of relationships of F3 is four, and the size of <F3,y1> is determined by the logical record formats to be six (the number of elementary data items in the union of the elementary components of the record formats F3, k, h and g), we can conclude that the number of function points is seven (simple). Data store F4 has two logical files <F4,x1> and <F4,x2>. The number of relationships for these files is one. Logical file <F4,x1> has two (F4 and i) and <F4,x2> one logical record format (j). After counting the sizes of the files we can conclude that the number of function points for both files is seven (simple).

Table 5 summarizes the results of the function point counting process. The total value of unadjusted function points in our example is equal to 71.

4. FUNCTION BANG ANALYZER

The other software size estimator that is supported by the JoYCASE system is Tom DeMarco's (1982) *Function Bang* metric. The implementation of the Bang metric is much easier than that of the Function Point metric, due to the simplicity of the counting principles. We use the sample structured specification of chapter 3 to illustrate the counting of Bang metric. The Bang metric is based on counting the sizes of each primitive process of the DFD. The size of a certain functional primitive is defined as the number of elementary data items leaving or entering the primitive. The size of the *i*th functional primitive is calculated by the equation

$$\text{Size}_i = 0.5 * \text{Weight}_i * n_i * \log_2(n_i),$$

where n_i is the number of elementary data items found from the boundary of the *i*th primitive (we interpret a data item to be elementary, if it is no further divided into subcomponents in the data dictionary). Weight_i is the weighting factor of the primitive. The weighting factor is based on the character of the primitive. DeMarco (1982) gives 14 classes into which the primitives can belong. Having counted the sizes for all primitives in the manner described above, the Bang is finally obtained as the sum of the sizes.

Table 4. Summary of Function Bang.

Primitive	Weighting factor	Data items in the boundary	Size
P1	1.0	10	16.61
P2	1.0	18	37.53
P3	1.0	9	14.26
P4	1.0	18	37.53
P5	1.0	7	9.83
P6	1.0	6	7.75
TOTAL FUNCTION BANG =			**123.51**

In our sample specification all functional primitives are found from the sub-diagram of figure 11. The weighting factor $Weight_i$ is 1.0 for all primitives. As an example, the primitive P1 has two dataflows connected to it. The number of elementary data items in the boundary of primitive P1 is 5+5 = 10, because dataflow x is defined to include five elementary data items (x = @x1 + x2 + x3 + x4 + x5) and the other dataflow (nameless) is referencing to the data store F4, and the contents of F4 is defined to be x (which has five elementary data items). Now we count the final size for the primitive P1 to be 16.61. The sizes of all primitives and the total Function Bang are shown in Table 6.

5. SPECIFICATION VERIFIER

The validity of the structured specification must be guarantied, before any attempt to estimate software size is done. DeMarco (1978) gives some elementary validity rules for structured specifications. Yourdon (1989) adds some new rules to the one's of DeMarco, and Cowan (1990) gives a set of exact rules for the correctness of a DFD. Most of these rules, and a number of extra rules, are supported by the Specification Verifier of the JoYCASE system. The internal validity of the data dictionary is checked each time a specification is retrieved from disc storage. The data dictionary must follow the notation suggested by Yourdon (1989) with the exception that we add a semi-colon after each definition as illustrated by Table 3. The checking of a data dictionary covers tracing of

[DD-1] intermediately and immediately recursive decompositions,
[DD-2] undefined data items that are used in decompositions,
[DD-3] duplicate definitions, and

[DD-4] the checking of the syntactic correctness of the whole data dictionary.

Were any errors to be found during the checking of the data dictionary, the JoYCASE system reports on the errors by writing the error messages to an external ASCII text file. If the data dictionary is found to be defective, it is not read into memory (the analyst is informed about this) though the other parts of the specification would be read, any way. Estimation or verification of the rest of the specification would be impossible, due to the lack of the data dictionary.

The DFD and ERD are checked every time the analyst attempts to count an estimate of the structured specification. If the specification passes the verification (provided that the data dictionary has been verified to be correct by the time, the whole specification was read into memory), the JoYCASE system counts the value of the desired estimator. Otherwise no estimation is performed, and the error messages (arised by the deficiencies in the specification) are written into an external ASCII text file.

The Specification Verifier makes four checks on the ERD of the structured specification:

[ERD-1] each entity must be named,
[ERD-2] each entity shown in the ERD must involve at least one relation,
[ERD-3] each entity shown in the ERD must have a corresponding data store on the DFD.

The DFD checks can be split into 26 different classes:

[DFD-1] Each subdiagram must contain at least one process.
[DFD-2] Dataflows entering or leaving a specific subdiagram, must balance with the dataflows entering or leaving the process, that is the parent of the subdiagram. The parent-child relation between the dataflow connected to the parent process and the dataflows entering or leaving the subdiagram is pointed out by the analyst at the time of the creation of the flows entering or leaving the subdiagram. This relation may be changed later, as the specification is altered. If no refinement of a dataflow is needed, when moving to a more lower level DFD, there may be exactly one child or many children for the

higher level dataflow. Any way, the lower level flows (i.e. the flows in subdiagram) must have exactly the same name as the upper level flow. The dataflows are allowed to be refined, when the process they are connected to, is exploded to a subdiagram. For example, a dataflow A = [B|C|D] which is input to the parent process P1, may be refined to two or more different flows, that enter the subdiagram of P1. The names of these flows must be any label from the set {A,B,C,D}. Refinement of an external inquiry differs from the one of ordinary inputs and outputs. The inquiry may be refined to subqueries or it may be split to separate input and output parts. Both of these two ways of inquiry refinement may not be used simultaneously. Another restriction is, that a separate output or input part of an inquiry may not be further refined.

[DFD-3] The diagram must be connected, i.e. no separate nodes are allowed.

[DFD-4] Each data store and dataflow name used in the hierarchy of DFDs must be included into the data dictionary.

[DFD-5] Each data store, terminator (including external applications) and process (including functional primitives) of the DFD must be named.

[DFD-6] Each dataflow of the DFD must be named. An exception is allowed on dataflows that reference data stores. If no name is given to such a dataflow, then this is interpreted as if the dataflow had exactly the same contents as the data store it is referencing. Normally dataflows have only one name, but inquiries, that are composed of an input and an output part, must have names for both of the parts.

[DFD-7] Each process must have a unique name.

[DFD-8] Each non-primitive process must be exploded to a subdiagram.

[DFD-9] Each process and primitive must have at least one input and one output dataflow.

[DFD-10] A terminator may be connected directly only to a process or a primitive. An external application may, however, be connected directly to an external or shared data store, too.

[DFD-11] A data store may not be directly connected to another data store.

[DFD-12] A branch of a combined or diverged dataflow may not reference to data store. A branch of a combined or diverged dataflow may not be directly further combined or diverged.

[DFD-13] A two way dataflow may not be be part of a combining or diverging dataflow.

[DFD-14] Diverging dataflows must involve exactly one input flow and two or more output flows that balance. The output flows must either have the same name as the input flow or the names of the output flows must form the decomposition of the input flow. For example, if the input flow A is defined as A = B+C+D, then the branching flows must have exactly the names B,C and D. A combination of dataflows must involve at least two input flows, that together form the output flow. As with a diverging flow, the branches that combine to the output flow, must either have the same name as the combination flow or they must form the decomposition of the resulting flow.

[DFD-15] The system (that is specified by the DFD) must write and read each internal data store.

[DFD-16] An external data store must be shown in the context diagram outside the system boundary. It should be shown only once: duplicate appearances will be interpreted as new external data stores.

[DFD-17] An external data store must have exactly two dataflows connected to it. The other must write it and the other must read it. The other referencer of an external data store must be an external application and the other the system itself. Neither of the referencing dataflows should be named. At the deeper DFD levels no inconsistent references to the external data store should be made. An inconsistency exists, if an external data store, that is sent to an external application, say T1, is written by T1 or it is referenced by external application other than T1 or the system reads that external data store.

[DFD-18] The system must write to a shared data store. Each shared data store should be read or written by one or more external applications.

[DFD-19] If a process P1 references to data store F1, then there must exist such process(es) in the subdiagram of process P1, that reference to data store F1 exactly in the same way, as P1 does.

[DFD-20] All components of a data store must be read and written.

[DFD-21] A dataflow, that references to a data store, must contain a key.

[DFD-22] A dataflow, that references to a data store F1, should contain only such data items that are included to data store F1.

[DFD-23] The definition of a data store label must contain the primary key for that data store.

[DFD-24] The definition of a data store label must not contain duplicate elementary data items. For example, if a data store, say F1, has a definition F1 = @A + B + C, where A is elementary (no further refined) and B = D + E and C = E + F (D, E and F are elementary), then this is an error, because elementary data item E is included twice into data store F1. Why do we need such a restriction? If we had a dataflow G, that references to the data store F1, and the data dictionary has the entry G = @A + E then we would not be able to know, which of the two E's in F1 is referred. That is why we do not allow duplicate elementary data items within a data store.

[DFD-25] The input part of an external inquiry is not allowed to write to any data stores.

[DFD-26] The outputs and the inputs (sent to and coming from terminators) of the system must not reference to external data stores.

6. ELEMENTARY DATA STRUCTURES

The two kind of diagrams - DFD and ERD - we need in our structured specification, is represented by a common net data structure, that consists of nodes (processes, terminators, entities etc.) and edges (dataflows or relationships). The basic data structure, that is used to construct more complicated data structures, is a generic*bidirectional linear list* provided with an identifier node. Length of all the linear list instances, that are needed in the data structures of the automated structured specification, will rarely exceed 10. If we need direct access to certain critical nodes of a long list, then we arrange this by saving pointers to these critical nodes.

The data dictionary is stored as an ASCII text file. When a specification is read into main storage, the data dictionary is checked and transformed to a suitable representation. The representation is chosen to be such, that the searching operation for a certain data definition - the most common operation - is made effective. The data dictionary is a list of definitions. Each definition consists of the data item name and its decomposition. The decomposition itself is a list of components. The components can be any of those suggested by Yourdon (1989). If the component happens to be a data item label (or a key label), then we provide it with a pointer to its definition in the data dictionary. In such a way we guarantee direct access to the definitions of the data item labels used in decompositions of other data items.

The record types used for representation of a hierarchical net, separate nodes, and edges are shown in Figure 13. The JoYCASE system uses these record formats to implement the DFD hierarchy as well as the ERDs. The internal representation for both DFDs and ERDs is the same, but the corresponding editors provide us with a different set of tools for editing the diagrams.

NetType	EdgeType	NodeType
LevelPath	Kind	Kind
ParentNode	Net	Name
ParentNet	Parent	Number
Edges	Childs	ProcNo
Nodes	ParentNum	Net
InFlows	Name	SubNet
OutFlows	QueryName	ComingEdges
NodeCount	StartNode	GoingEdges
ProcCount	EndNode	Class
	StartLine	SpotX,SpotY
	EndLine	Lines
	X1,Y1	
	X2,Y2	
	NameX,NameY	

Figure 13. Fields of the records used to represent a hierarchy of nets, edges and nodes.

NetType is for storing information of a specific subdiagram of the DFD hierarchy. *LevelPath* is a unique name for each subdiagram. It is derived from the numbers of those processes, that are the parents of the subdiagram. LevelPath of the context diagram is always 0, LevelPath of the subdiagram for the process number 1 of the context diagram is 0.1, LevelPath of the subdiagram for process number 3 of the subdiagram with LevelPath 0.1 is 0.1.3 and so on. *ParentNode* stores a pointer to the process, a refinement of which the described subdiagram is. *ParentNet* is for storing a pointer to the DFD of ParentNode. *Edges* is a list of all edges (dataflows) in the subdiagram. *Nodes* is a list of all nodes existing in the subdiagram. *InFlows* and *OutFlows* are for storing pointers to those dataflows, that either enter or leave the subdiagram. By collecting the boundary dataflows, we ease dataflow balancing checking and estimation of the specification. *NodeCount* tells the current number of nodes in the subdiagram, for numbering new nodes. The identifying number is necessary when restoring the diagram back into main storage from disc storage. The links within the dynamic data structure are reconstructed by using the unique node numbers. *ProcCount* tells the current number of processes in the subdiagram. An up to date numbering of processes is crucial, because we usually refer to possible errors found in the DFD hierarchy by LevelPath of the subdiagrams.

A record of type *EdgeType* contains all information about a specific edge in the net. The field *Kind* tells the class, into which the edge belongs (is it an inquiry, normal dataflow, etc). *Net* is pointer to the subdiagram, in which the edge exists. *Parent* is a pointer to the parent of the edge, if one exists. This pointer makes it easier to traverse the DFD hierarchy when we, for example, search the data stores referenced within an input transaction. *Childs* is a list containing pointers to all edges, that refine the currently described edge in the next deeper net level. Thus we have a pointer from a parent to its child (refinement) and vice versa. This makes the traversing of the net hierarchy, as well as balancing verification, effective. *Name* is the label for the edge (if the edge is an inquiry, then it contains the name of the input part of the inquiry). *QueryName* contains the name of the output part of an inquiry; if the edge is not an inquiry, then this field is unused. *StartNode* and *EndNode* are for storing pointers to nodes, from which the edge starts and to which the edge ends. Depending on the class of the edge, either StartNode or EndNode may be missing. For example, dataflows entering a subdiagram do not necessarily have a Start–Node (though the parent flow of the entering dataflow may have a StartNode). *StartLine* and *EndLine* help the graphical representation of the edges. They contain the specific edge anchoring points of StartNode and EndNode. These anchoring points are used for counting the exact screen coordinates for the start point (*X1,Y1*) and end point (*X2,Y2*) of the edge. Depending on the kind of a node, there are 10-16 anchoring points for edges. These points are located at the border of the graphical symbol of the node. *NameX* and *NameY* are used for storing the exact screen coordinates for the name of the edge.

NodeType is a record type to describe a specific node in the net (diagram). *Kind* tells the class of the node. There are several classes of different nodes. In DFDs we have terminators, processes, data stores and a specific node for diverging or combining dataflows. In ERDs we have entities. The Kind field determines the way of handling, which varies from node class to another. *Name* contains the name of the node. *Number* is the number of the node in its subnet. The nodes must be uniquely identified by their numbers. This is necessary when storing and restoring the nets to and from disk storage. Process nodes (including functional primitives) have an extra numbering fashion. *ProcNo* tells the up to date ordinal number of the process in its subdiagram. So, if one process should be removed, the numbers of the other processes in that subdiagram will be immediately updated. This is necessary because the identification of subdiagrams is based on the numbering of processes. *Net* is a pointer to the net, in which the node exists. If the node is exploded into a subnet, then a pointer to the subnet is stored into field *SubNet* (only non-primitive process nodes of DFDs can have subdiagrams). To ease the manipulation of the net, the node is provided with lists that contain pointers to those edges entering (*ComingEdges*) and leaving (*GoingEdges*) it. If the nature of an edge is bidirectional, then a pointer to it is stored to both Going- and ComingEdges of the node, from which the edge starts. If the node is a functional primitive, then its primitive class (DeMarco 1982) is stored into field *Class*. The graphical representation of the net requires three more fields to the node record. *SpotX* and *SpotY* tell the exact screen coordinates for the centre point of the graphical symbol of the node. *Lines* contain information about the state of each edge anchoring point of the node.

7. SUMMARY AND FUTURE PLANS

The JoYCASE system is a prototype CASE-tool. The main purpose for building the system was to implement the FPA algorithms given by Rask (1991) and the Function Bang given by DeMarco (1982). A secondary aim was the design of the user interface. The JoYCASE system lacks, as has been revealed, an integrated data dictionary editor as well as a possibility to correct specification errors immediately. Also more advanced generation of documentation based on the structured analysis descriptions should be supported. We are very interested in improving these currently inadequate features of the JoYCASE system.

As we implemented the Function Bang and FPA algorithms into the JoYCASE system, we saw a possibility to measure the correlation between these two metrics. For this purpose we have planned to build a test environment. We have also laid plans for implementation of a number of other well known software size estimators, such as Mark II (Symons 1989). Another point of interest is the specification method itself. We have planned to enlarge our prototype system to also support an object-oriented specification method. One more interest, which is raised by the large number of various software specification methods and descriptions, is an automated assistant tool for transforming the descriptions of one method to those of another. Along with these research branches we are interested in formalizing the notations used. The formalization is a general trend also in current professional literature (e.g. Coleman & al. 1991, Falkenberg & al. 1991, Kung 1991).

ACKNOWLEDGEMENTS

We are indepted to the North Karelian Department of the Finnish Culture Foundation for the grant that enabled the implementation of our prototype system.

REFERENCES

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

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

Coleman D. L. & al. 1991. Deliberations on Kung's Process Interface Modeling, *The Journal of Systems and Software*, 15, 193-198.

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

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

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

Dreger J. B. 1989. *Function Point Analysis*. Prentice-Hall.

Falkenberg E. D. et al. 1991. Understanding Process Structure Diagrams, *Information Systems*, 16(4), 417-428.

Kung C. 1991. Process Interface Modeling and Consistency Checking, *The Journal of Systems and Software*, 15, 185-191.

Rask R. 1991. *Algorithms for Counting Unadjusted Function Points from Data–flow Diagrams.* Report A-1991-1. University of Joensuu, Department of Computer Science.

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

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