Yourdon dataflow diagrams:
a tool for disciplined requirements analysis

MARK WOODMAN

Abstract: During the past two decades many informal methods for requirements analysis and specification have been proposed. The majority of these claim to be 'structured' and have a graphical notation as a central component. The simplicity of some of these notations has made them popular with analysts and acceptable to their customers. However, vagueness in early descriptions of the syntax and semantics of the notations has allowed an unhelpful degree of flexibility in their application and interpretation, and has diminished their usefulness. Dataflow diagrams, as promoted by the Yourdon organization, have suffered in this way. This tutorial introduces the notation and describes how it has been used and how it should be used.

Keywords: software engineering, dataflow diagrams, Yourdon

Dataflow diagams (of one sort or another) have probably become the most commonly used graphical notation in software development. You will see dataflow diagrams in many situations, in many media–on the backs of envelopes, as colleagues discuss problems on a train, and as part of system specifications produced by high-cost CASE tools. The fundamental concept of dataflow diagrams is that they depict *data* and *process*. In particular, they facilitate the description of how processing elements of a system should transform data entering the system into data which is output by the system. Figure 1 depicts how the system *Sys* transforms data which comes from sources outside the system (*Source 1* and *Source 2*) into data which is output from the system (to *Sink 1*, *Sink 2* and *Sink 3*). The input data is represented by the lines labelled *a*, *b*, and *c*; the output data is represented by *x* and *y*.

The simplicity of this style of diagram has been both its strength and weakness: it can be learned easily and can apparently be understood easily; but its simplicity suggests a lack of expressive power, which encourages extensions to the notation, and disagreement in interpretation. (The latter phenomenon can thus make comprehension apparent rather than actual.)

This paper is a tutorial on the disciplined use of dataflow diagrams. It takes the view that Yourdon-style dataflow diagrams are only really useful if their syntax and semantics are understood and adhered to. The paper first reviews the original dataflow diagram notation and how it was suggested it be used in requirements analysis, and then illustrates how both the notation and method have been refined and expanded–resulting in the possibility of using them in a disciplinary way during analysis.

Before describing Yourdon dataflow diagrams we need to establish a common vocabulary and to set the bounds of the activities being discussed. First, let us assume a simple phase-oriented model of software development. Second, let us restrict our domain of discourse to the software *system* which a customer might require; we shall assume that the *environment* in which the system is to exist has been decided. We shall also assume that the customer has carried out appropriate feasibility studies and has considered the financial aspects of producing the software. In our model of the development process a customer's

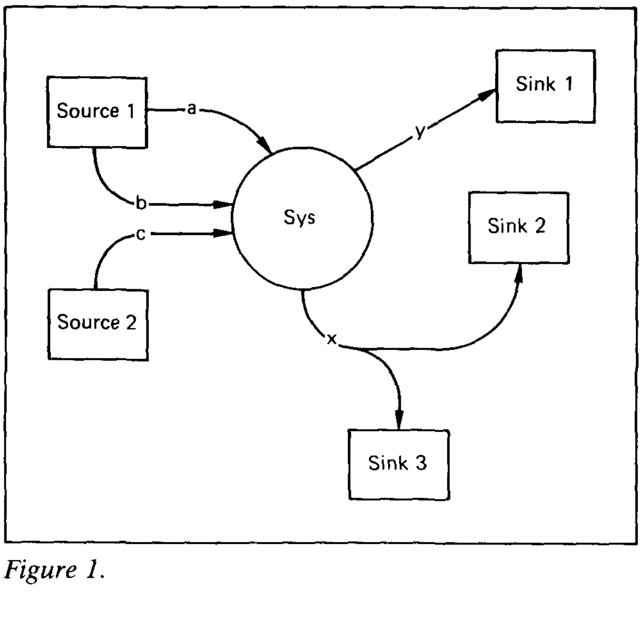


Figure 1.

Open University, Computing Department, Walton Hall, Milton Keynes, MK7 6AA, UK

statement of requirements is the first input to the process which we are interested in. The developer must carry out the activity of requirements analysis using the statement of requirements to produce a system specification.

The system specification is a precise, unambiguous, consistent and complete definition of the system to be built which (in theory at least) can form the basis of a contract between customer and developer. It is also the input to the system design phase of the development process (which precedes coding, testing, etc.).

In this paper we are primarily concerned with the use of the dataflow diagram notation as a tool for requirements analysis. However, dataflow diagrams are also used during the system design phase and their use in this way will be outlined.

Requirements analysis and dataflow diagrams

In this section the relationship between Yourdon dataflow diagrams and requirements analysis is established and the basic symbols of the notation are introduced. The use of functional decomposition as an analysis technique which facilitates the production of dataflow diagrams is then discussed.

Structured analysis

The term 'structured analysis' first came to prominence in 1975 when Ross and Schoman published their Structured Analysis and Design Technique (SADT)[1] which included an early form of dataflow diagram called an activity diagram. At about that time the consultancy company Yourdon Inc. were promoting the ideas of 'structured design'[2]. Within Yourdon Inc., work was also progressing on tools and techniques for requirements analysis (more usually called 'systems analysis' then) and in 1977 two ex-Yourdon employees, Gane and Sarson, published a seminal book which suggested a dataflow-based approach to modelling system requirements and a style of dataflow diagram to support their ideas[3]. It should be noted that Gane and Sarson's diagrams contain all the elements of the early Yourdon-style diagrams.

In the following year, three books from the Yourdon press served to establish the Yourdon style of dataflow diagram: DeMarco and Weinberg published books on structured systems analysis[4,5] and Yourdon and Constantine[6] published the second edition of *Structured Design* which showed software developers how they might proceed from requirements analysis (with Yourdon dataflow diagrams) to system design.

When these ideas and notations were first advocated, they were perceived as a radically new and improved way to approach requirements analysis. The techniques were new because they relied on a graphical notation rather than pure English. They were improvements because they allowed developers to get away from the 'Victorian novel' style of specification documents, and to move towards ones which were more precise and easily understood. However, the proponents of the new methods failed to explain exactly the conceptual basis of their technique and how their graphical notation should be used as a tool of the technique. In particular, the way in which diagrams were to be drawn was not well-specified. Both these factors have led many practitioners to build their own development technique around a notation, or around an extended version of one.

It must be stressed that the pioneers of software engineering cited above are not as culpable as the above implies; at the time when their work was being published noone had sufficient experience to recognize the weaknesses in the published descriptions of dataflow diagrams and their use. The most comprehensive description of technique and notation until recently was DeMarco's[4]. We shall therefore refer to the earlier form of Yourdon dataflow diagram as the DeMarco dataflow diagram.

To some extent, the need to abstract the description of a system away from its realization – its implementation – was recognized and advocated by proponents of all the techniques. DeMarco made this strategy clear by prescribing four modelling phases. DeMarco defines seven component activities in structured analysis. This discussion is restricted to the first four. Because of the common need to computerize manual or semi-automatic systems at the time, the first two assumed an existing predecessor to the system required by a customer. The phases suggested by DeMarco are as follows:

- model current physical environment
- model current logical environment
- model new logical environment
- model new physical environment

DeMarco's analysis begins with the system's environment; this was needed in order to determine how the existing system (i.e. 'current', in DeMarco's nomenclature) should be modelled. If there is no existing system, work should start immediately on modelling the new logical environment. The terms 'physical' and 'logical' are used to describe the implementation-dependent and implementation-independent models of the system.

Having arrived at this logical model, the man-machine boundary can be identified, and the computer implementation specified. The final product is a structured specification which contains a levelled set of dataflow diagrams supported by a set of data and process definitions; data definitions are provided in a data dictionary; process specifications may be expressed in structured English, decision tables, decision trees, or a combination of all three.

The following discussion concentrates on the dataflow diagram notation and its use, since it is here that most changes have occurred.

Dataflow diagrams

A dataflow diagram consists of a number of graphical symbols (mostly circles and rectangles) connected by

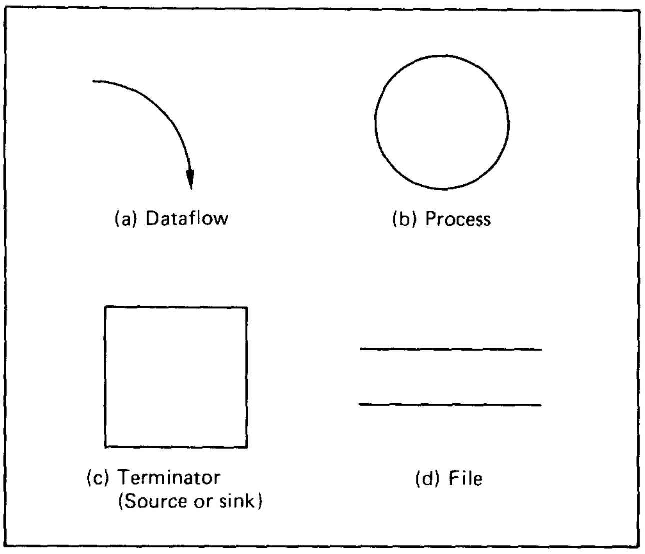


Figure 2.

labelled, directed lines which represent data 'flowing' through the system. Data 'flows' from one circle to another, with each one using some or all of its input data to produce its output. The set of symbols used by DeMarco is shown in Figure 2.

A circle represents a process that uses data which 'arrive' at the circle and produces new data as a result. Processes must contain text which suggests how the data is transformed, and must be numbered. A process can have more than one input, and more than one output dataflow. Therefore, it can be useful to annotate these flows to indicate whether all input flows are always required, or all output flows are always produced. This is done using a star symbol,*, for 'AND', and a plus sign inside a circle, ⊕, for 'OR', as shown in Figure 3. Figure 3 (a) denotes the requirement that *both* x and y are required by the process.

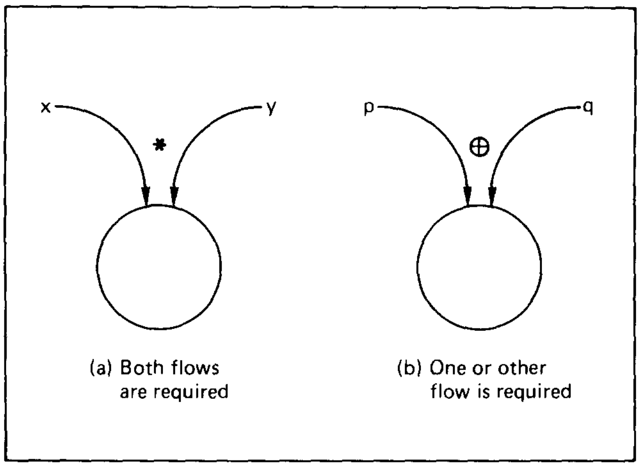
Figure 3 (b) denotes the requirement that *either* p or q are required by the process. (DeMarco was not in favour of this inclusion of procedural information in dataflow diagrams.)

Rectangles represent terminators which are the origin (called a source) or the destination (called a sink) of dataflows. Terminators exist in the environment in which the software is to exist and so their behaviour must be taken as read and not modelled.

Parallel, horizontal lines depict a file for data within the system. A file is intended to be a temporary repository of data, and should not be thought of in terms of disc files or databases. Note that data may not flow directly from a terminator to a file, or from a file to a terminator; in either case a process must transform the data.

Figure 4 shows a simple dataflow diagram, which uses the symbols just described. It models a system which inputs names and stores them. When all names have been input, they are sorted and restored, and finally printed.

The rectangle on the left in Figure 4, labelled *Personnel clerk* is a source terminator; it represents the


x
y
*
p
q
⊕
(a) Both flows are required
(b) One or other flow is required
Figure 3.

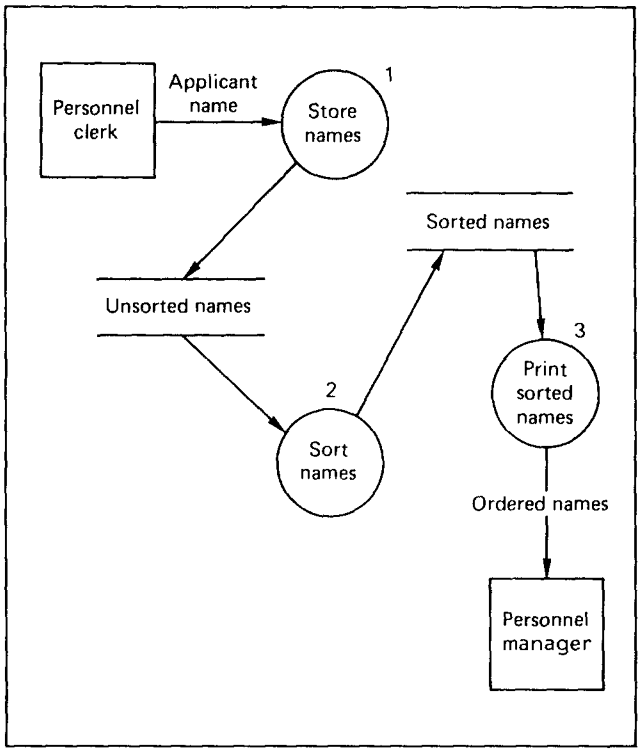


Figure 4.

source of the sequence of names implied in the last paragraph. The rectangle on the right represents the *Personnel manager* – the ultimate destination of the sorted names. The dataflow labelled *Applicant name* represents the names to be input to the system, and the dataflow labelled *Ordered names* represents the sorted list of output names.

There are three processes in Figure 4: *Store names, Sort names,* and *Print sorted names*, which are numbered 1, 2, and 3 respectively.

Two files are used: *Unsorted names* is needed because all names must be input before *Sort names* can be used to order them. Similarly, sorting must have been completed before and the names must have been temporarily stored in *Sorted names* before they can be printed.

Note the different conventions for labelling dataflows to and from files. Whereas dataflows between terminators and processes are labelled, a dataflow to or from a file is not labelled if the dataflow represents the whole of a data item stored in the file.

Figure 4 thus represents a model of a system whose function is to input, sort and print the names of applicants. The reader is supposed to imagine *Applicant name* flowing, as if water in a pipe, into something like a tank (the file *Unsorted names*) which, when it is full releases them to be 'processed' by *Sort names*, and so on. The vagueness of this description is a virtue by which implementation bias is excluded.

A crucial characteristic of dataflow diagrams is that they depict a *functional* description of a system and not a *procedural* one. (At least, dataflow diagrams should model how a system behaves, not how it should be implemented; if this is not the case in a particular diagram, then its validity is suspect.) Dataflow diagrams, therefore, imply no ordering of processes except that a process which produces a dataflow for another must occur before the one which depends on it. Thus in Figure 5 *M* must transform *i* to *j* before *N* transforms *j* to *k*.

To further illustrate the point about order consider a stepwise description of a system to make carrot soup:

1 get the vegetables from the vegatable rack
2 wash the vegetables
3 prepare the onions
4 prepare the carrots
5 fry the onions
6 add water to the pot
7 add carrots to the pot
8 add seasoning
9 cook
10 serve in tureen

To some extent the above recipe captures the essence of making carrot soup; it certainly leaves out implementation details of what appliance should be used to cook the mix, for instance. However, it unnecessarily defines that onions be prepared before carrots, and precludes an implementation which overlaps the frying of the onions with preparing the carrots. The dataflow diagram which models soup making has none of these faults. It is given in Figure 6.

In this Figure the only ordering defined is that the vegetables be washed and sorted before they can be prepared, that onions be prepared before frying and that onions and carrots are ready before cooking. The preparation of carrots, frying of the onions, and adding of water and seasoning can all proceed in parallel.

Note that the model of a system to make soup contains procedural information: the * symbol dictates that *both Clean onions* and *Clean carrots* emerge when the *Wash and sort veg* process has completed. While this might conform to common practice when making soup, the conjunction of the ingredients prior to cooking raises the problem of how the dataflow diagram guarantees that all ingredients are ready for cooking at the same time. This point shall be discussed later.

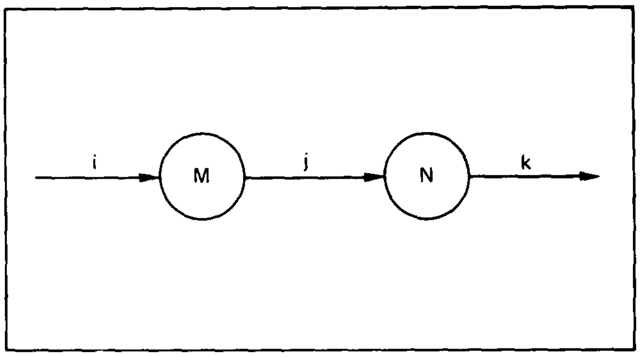


Figure 5.

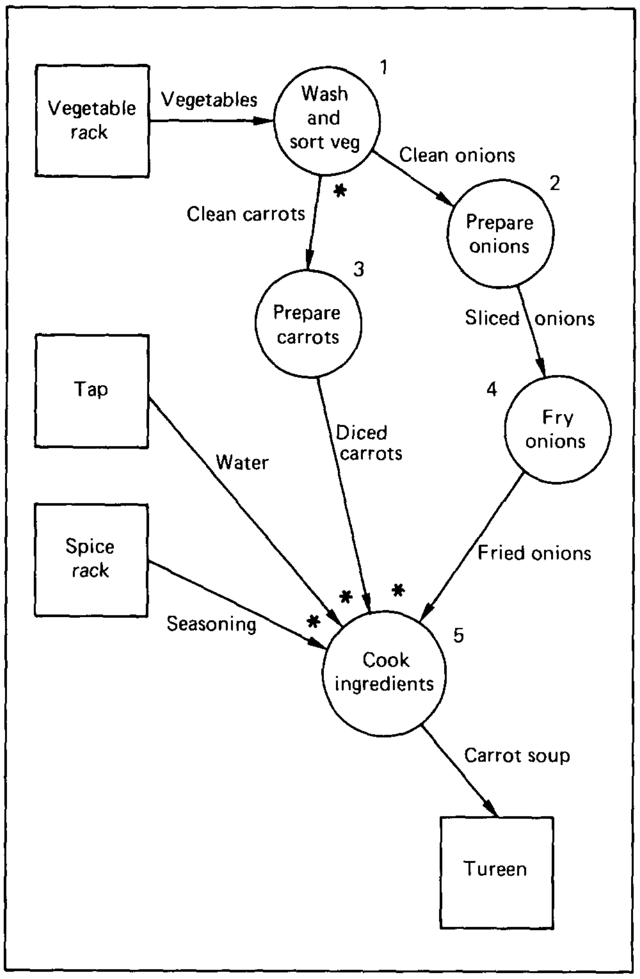


Figure 6.

Functional decomposition and dataflow diagrams

In early versions of the Yourdon Structured Analysis method, functional decomposition was promoted as the means by which a system should be modelled: from a few major functions identified by the customer in a statement of requirements, the developer must discover and specify the subfunctions which the software will have to provide. This is done by repeatedly decompos-

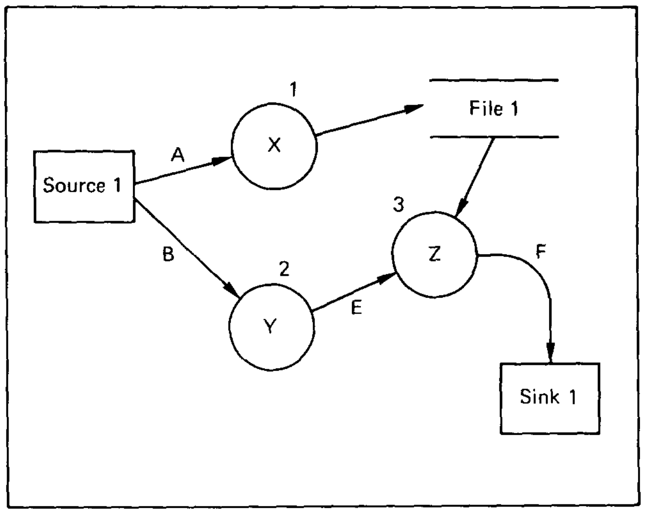


Figure 7. Diagram 0: XYZ system

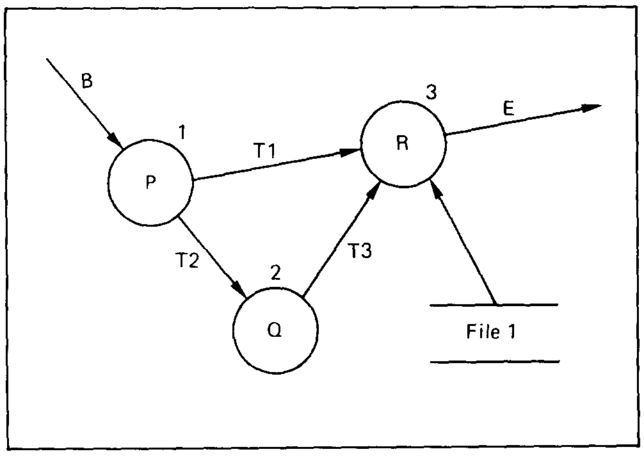


Figure 8. Diagram 2: Y

ing identified functions until a simple description will complete the specification.

In dataflow diagrams functional decomposition is depicted by a refinement of a process. Successive levels of detail about the behaviour of a system being modelled are built up by expanding each process into a dataflow diagram. For example, Figure 8 is a refinement of process 2 in Figure 7. Figure 8 shows what process *Y* is supposed to do: it takes the dataflow *B* and transforms it into *E* using processes *P*, *Q* and *R* and data from *File 1*.

There are several noteworthy aspects of how Figure 8 is a refinement of Figure 7. First, notice that the number of the diagram in Figure 7 is the same as the number on the process *Y* in *Diagram 0* (Figure 7). This facilitates cross-referencing between diagrams: the process *P* in *Diagram 2* (Figure 8) may be referred to as process 2.1.

Second, note that the diagram caption is, by convention, the same as the process name. The caption of *Diagram 2* is thus *Y*, because it represents the process *Y* of *Diagram 0*.

The third point is that there must be consistency between the refined process and its refinement: the same number of flows must be input to or output from the refinement.

Of course, dataflow names must be preserved by the refinement.

In a refinement, the objects (in the previous level) from which input has arrived, and the objects (in the previous level) to which outputs are bound, are omitted. This reduces the complexity of the refinement diagram. You can see from Figure 8 that *Source 1* and *Z* (which appear in Figure 7) are omitted. Inputs are usually shown as coming from 'nowhere' (as *B* is in Figure 8), and outputs are shown as going to 'nowhere' (as *E* is in Figure 8).

Finally, note that a file can appear in several places throughout a hierarchy (e.g. *File 1* in Figures 7 and 8).

Functional decomposition (and dataflow diagram refinement) begins at the boundary between the software system and its environment. The diagram which depicts this division is called a context diagram and constitutes the root of a hierarchy of dataflow diagrams which represents a hierarchy of functions required of the system. A context diagram is a dataflow diagram which contains a single process that represents the *entire* system and the major sources of data and destinations for data in the environment. (Indeed sources and sinks usually only appear in the context diagram.) The process in the context diagram is usually labelled by a noun (such as the name of the system) rather than a verb describing a transformation. Figure 9 shows the context of the *XYZ System* which is refined in Figures 7 and 8.

Thus, to produce a model of a system, the function of main process of the context diagram should be decomposed and the circle which represents it should be refined into a diagram whose processes are refined, and so on. Repeated decomposition and process refinement results in a hierarchy of dataflow diagrams. Such as hierarchy is called a levelled set by DeMarco[4].

Figure 10 shows how a hierarchy of diagrams are related by refinement; it shows the relationship between Figures 9, 7 and 8 with the source and sink of Figure 7 in the context diagram at the top.

A process which is at the bottom of the refinement hierarchy is described as functionally primitive. The specifications of primitive processes are often called mini-specifications. These are written in structured English or using pre- and post-conditions[7].

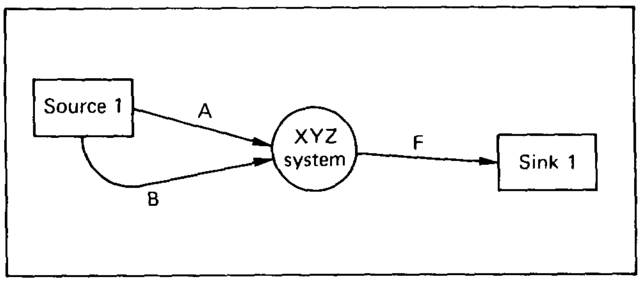


Figure 9.

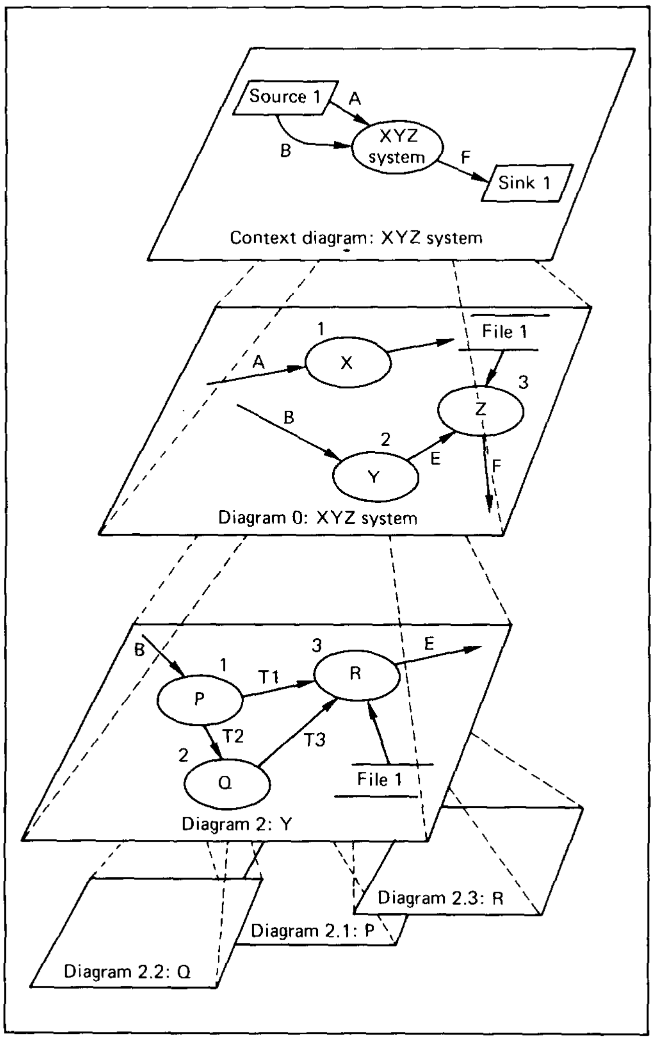


Figure 10.

Data dictionaries

Up to now the question of what is a dataflow has been ignored. It is easier to answer this by examining what a system needs in order to provide the software functions required of it: it needs *valid* input data to be *available*. In dataflow diagrams the functions are modelled by a network of processes; if the data required to provide a function is to be made available to the processes modelling it, a path must exist from the environment to the relevant processes. Data can then enter the system from the environment and 'flow' through it, being transformed in some way by any processes in its path.

To make any sense, the data flowing within a diagram must be valid for the system being modelled. The requirements of the data are therefore recorded in a data dictionary. Space does not permit even a cursory discussion of the uses of a data dictionary, but it is important to recognize the need to record the structure of data within the system. The following example provides some idea of how this is done by defining the dataflow *Applicants name* used in Figure 4:

Applicants Name = Name + Address + (Department)
Name = Surname + First Name
Address = Street Address + Town + [Post Code County]

The above definitions reveal that an *Applicants name* is more than just a name! It includes an *Address* and optionally (denoted by the parentheses) a *Department name*. A *Name* is composed of a *Surname* and a *First name* , and an *Address* is composed of a *Street address, Town* and *either* a *Post code* or *County* (choices are within brackets and are separated by vertical bars).

Definitions such as those above are included in a data dictionary with simple structures being described in natural language terms.

Problems with DeMarco dataflow diagrams

There is a relationship between a software development technique and any notation it uses. If the notation is lacking–either in expressive power or conceptual clarity–the practioner may be unable to record the results of his or her analysis or may become confused when attempting to carry out analysis using the notation. If the technique is deficient, deficiencies in the notation may be missed. In this manner, problems with Yourdon Structured Analysis and dataflow diagrams have been difficult to separate.

The first problem is that neither the need for the separation of technology dependence and technology independence nor the means to achieve it was made clear. The terms 'logical' and 'physical' have many connotations and of themselves are not sufficient for the analyst to judge what aspects of requirements may be deferred until system design. The provision of 'processes' and 'files' in the notation compounds the problem. Indeed it can be argued that the notion of a physical model is misplaced in analysis and is properly part of the system design.

However, the biggest problem with the technique is the absolute reliance on functional decomposition. Functional decomposition is, by definition, concerned with the functions, or processing, which a system must provide. Concentration on just one aspect of systems, coupled with an implication that the process of functional decomposition is a steady, nonbacktracking progression from context diagram to primitive processes can cause the analyst to ignore vital information concerning the data which is to be processed or concerning events in time.

This problem with the early version of the Yourdon method has meant that its notations do not adequately deal with all aspects of systems: the data processing aspect may be described by dataflow diagrams, but time-related processing and the relationship between data items stored in the system are not included in the method nor provided for by the notation. In particular there is a lack of expression for realtime systems. At a time when data processing systems were not expected to respond to realtime events, this may not have been a problem, but such distinctions are hardly tenable today. For example, there is no way to distinguish between continuous and discrete dataflows, nor of depicting events, and no way to express control considerations.

In the next section the new Yourdon method is outlined; the differences between it and the older version will be highlighted.

Yourdon Structured Method

In this section the current Yourdon approach – Yourdon Structured Method (YSM) – is outlined and its present use of dataflow diagrams is described. Much of the earlier approach is still used; in most cases the technique has been refined and placed in a different context, and the graphical notation has been extended and augmented.

Overview of YSM

The first obvious change from the early approach is the emphasis in YSM on modelling the *behaviour* of a required system. DeMarco described how dataflow diagrams can be used in a system specification to document the *functions* of a system.

The second change is the introduction of new graphical notations which are associated with particular aspects of modelling.

The third change is the clearer distinction between modelling phases: YSM is divided into three distinct phases. The first is the feasibility study, which includes the study of any current system and its environment. The second is essential modelling: the aim of this phase is to describe the *essence* of a software system in terms of how the required system must behave and what data it must store. Essential modelling was derived from 'Essential Systems Analysis'[8] which provides many of the solutions to the problems with the old version of the Yourdon method discussed above. Its major contribution in this area is the clarification of the question of technology dependence which leads to a new way of constructing dataflow diagrams. (This is discussed later.)

In general software engineering terms, essential modelling corresponds to requirements analysis. However, an essential model corresponds only to the new logical model of the original technique. The essential model is expressed in the customer's terms and describes:

- the context in which the system is to exist (i.e. the boundary between the system and its environment);
- the behaviour of the system to be constructed.

YSM identifies three dimensions of a system which must be described by an essential model. The first is the *processing* behaviour of the system–how the system uses its inputs to produce its outputs; dataflow diagrams are used to represent processing behaviour. The second is the structure and use of *data* in the system; a data dictionary (as described above) and a set of entity relationship diagrams (outlined later) describe the data dimension of a system. The third dimension is the *dynamic* behaviour–how events in time affect behaviour; this dimension is modelled by extending dataflow diagrams to represent control and by specifying control behaviour using state transition diagrams.

An essential model should not assume a particular technology (it should not even assume a digital computer) and the technology which it models should be considered to be 'perfect'. That is, the technology can process any data in the system instantaneously and storage is infinitely large and infinitely fast. This perfect technology allows the semantics of dataflow diagrams to be defined, and thus to be used with increased discipline.

The third phase of YSM is implementation modelling: it aims to incorporate those aspects of the customer's statement of requirements which are dependent on a particular technology, with the essential model. Thus an implementation model is an elaboration of an essential model, and is expressed more in terms of how the software is to be implemented. Dataflow diagrams are used in implementation modelling; those which are part of an essential model are transformed by introducing technology-related flows and processes.

In general software engineering terms implementation modelling corresponds to system design. It also corresponds to the derivation of the new physical model in the DeMarco method, which is a design rather than requirements analysis activity.

Representing data, process and control

When first encountered, many of the changes to Yourdon dataflow diagrams appear to be slight changes of syntax and nomenclature. While such changes have been made, they are not gratuitous but serve to emphasise the separation of essential and implementation modelling and to allow the semantics of dataflow diagrams to be more succinctly described. To some degree the notation has also been extended to better cope with modelling realtime systems.

The most important change in nomenclature is that the term 'process' is now denegrated in favour of 'transformation'. The term 'process' is ubiquitous in computing and its particular meaning in dataflow diagrams is difficult to sustain; it has an obvious implementation bias. The term 'transformation' emphasises the modelling activity. (Consequently, dataflow diagrams are now also known as transformation schemas.)

The second change is that 'file' is similarly denegrated in favour of 'data store'. This emphasizes that a store is a single data item or a set of items which have 'come to rest' in a system and that a store is *not* used to change data, but to hold changed data.

The revised set of symbols for dataflow diagrams is shown in Figure 11.

Most of the symbols are the same as those for DeMarco dataflow diagrams, but note that two types of data flow are now distinguished: discrete dataflows, which arrive at their destination at discrete intervals in time, and continuous dataflows which are always available at their destination. These are represented by

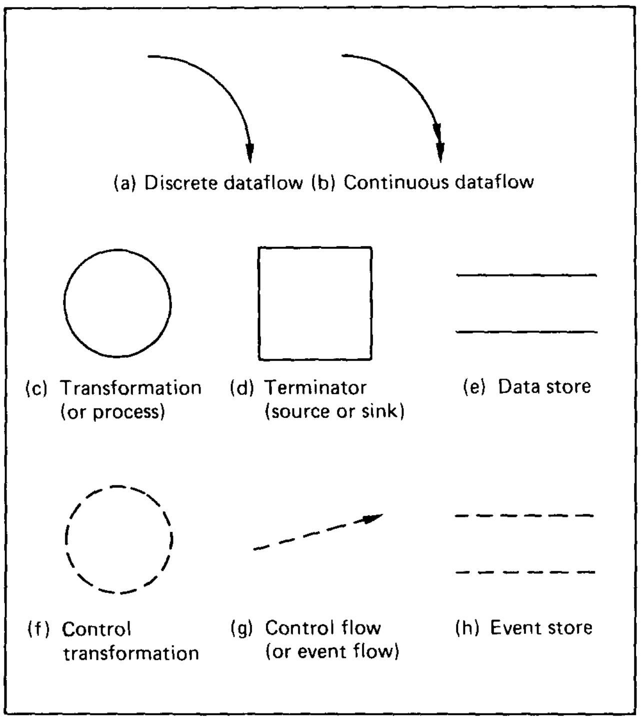

(a) Discrete dataflow
(b) Continuous dataflow
(c) Transformation (or process)
(d) Terminator (source or sink)
(e) Data store
(f) Control transformation
(g) Control flow (or event flow)
(h) Event store

Figure 11.

a line with a single arrow head and one with a double head respectively, as shown in Figures 11(a) and (b). Also, Figures 11(f), (g) and (h) are control transformations, control flows (or event flows) and event stores respectively. They are used to model aspects of systems and are discussed later.

Dataflows and control flows

In order to produce a viable model of a system using YSM, the correct types of flows must be used. This means that the distinction between dataflows (discrete or continuous) and control flows must be maintained, and that discrete and continuous dataflows must not be confused.

To understand flows the fundamental concept of an event must be understood. An *event* is a change in the environment of a system which leads to a set of actions by the system. Such a set is called a *response*. Consider, for example, an automatic cash point system at a bank. An event might be a customer requesting cash; the response would either be an amount of cash or a message indicating that no cash would be dispensed.

Events are represented in a context diagram either by dataflows or control flows which cross the boundary between the system and its environment. A control flow is used to represent an event which has no data associated with it: only its occurrence is of interest. An example of such an event is the pressing of a start button on a machine. (Because control flows are needed to model events without data, control transformations which transform them are also required.) On the other hand, a request for cash is an event which both occurs and has data associated with it–the amount

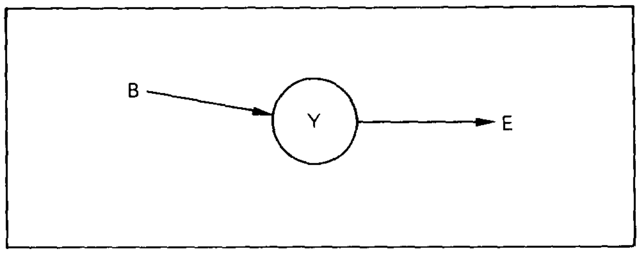


Figure 12.

of cash requested. This type of event is represented by a dataflow.

A dataflow which has crossed the system boundary and is connected to a transformation therefore models the beginning of a network which transmits the occurrence of the event and of the associated data. In Figure 10 if the event represented in the context diagram by *B* occurs, then the *B*'s data will be passed to *Y* where its occurrence will cause its data to be transformed by *P* into *T1* and *T2*, whose occurrence will prompt *R* and *Q*, and so on until *E* and then the response *F* is produced.

For the fragment reproduced in Figure 12, this can be interpreted as: '*Y* is waiting for an occurrence of *B*-data arriving; such an occurrence prompts *Y* to transform the data of *B* into *E*.' exists only from time to time and so *B*-data is defined to be a *discrete* dataflow.

The permanent readiness of *Y* may not be appropriate. It may be desirable to prevent *Y* transforming *B* to *E*. For example, in the automatic cash point systems, the failure of the cash dispenser should cause the system to stop. Control flows are used to model this type of control of a transformation (see the section later on controlling transformations for details).

Some events happen in sequence so frequently that it is useful to consider them to be continuous in time; continuous dataflows (with a double-headed arrow) are used to depict these events. Typically, continuous flows are used to represent physical characteristics such as temperature and voltage. By distinguishing between

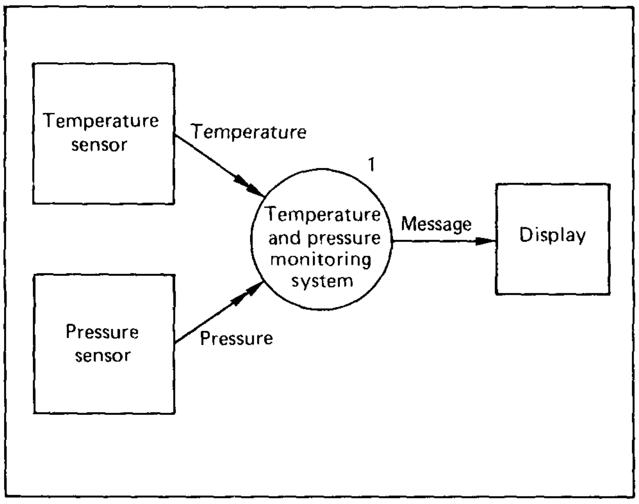
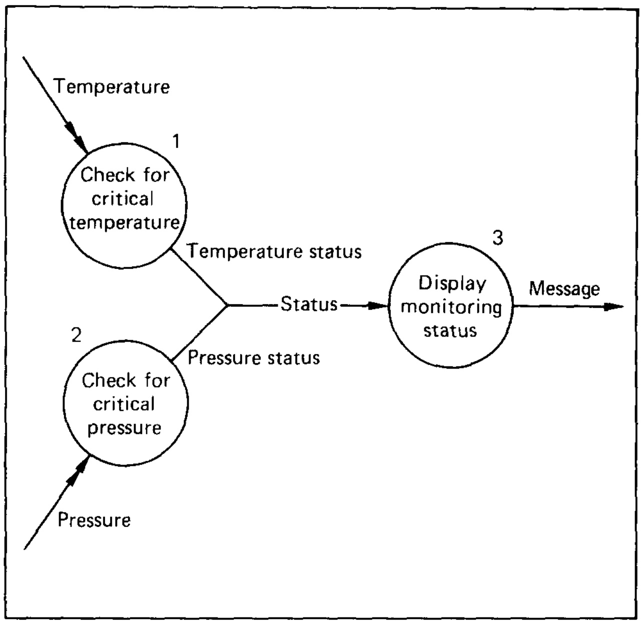


Figure 13.



Temperature
1
Check for critical temperature
Temperature status
3
Display monitoring status
Message
Status
2
Pressure status
Check for critical pressure
Pressure

Figure 14.

discrete and continuous dataflows an opportunity is created to verify the model by analysing the way the different types of flows are used (see below for the rules) and comparing them to the events in the environment.

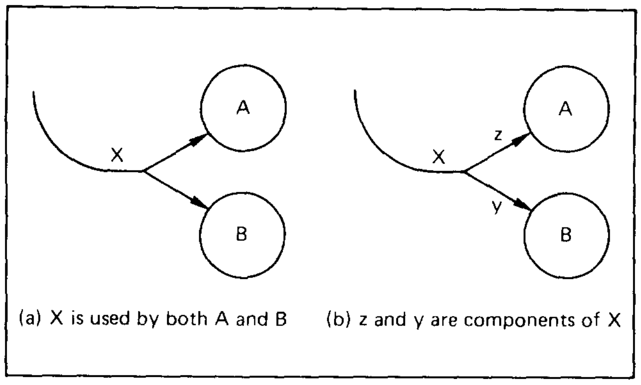
As an example of continuous dataflows, consider Figure 13, which is the context diagram for a plant monitoring system.

In Figure 13 both the input dataflows *Temperature* and *Pressure* are depicted as continuous flows. This reflects the nature of the data event in the system's environment and is dictated by the specification of the environment. The output *Message* is discrete because it is produced occasionally only. The importance of properly identifying the nature of dataflows: the syntax rules take account of these (see later) and prevent nonsensical combinations of dataflow. The refinement of the main transformation is given in Figure 14.

Note that the flows *Temperature status* and *Pressure status* have been combined into a single composite data flow called *Status*. The latter flow would be defined in the data dictionary to be composed of either a temperature or pressure status. This device obviates the need for the 'AND' and 'OR' symbols.

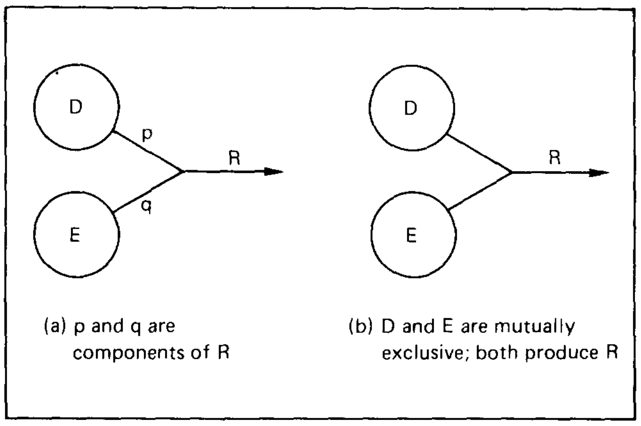
The rules for composite flows are now given. Consider Figures 15 and 16. Both diagrams depict composite dataflows. Figure 15(a) shows the flow *X* going to two transformations–*A* and *B*. Figure 15(b) shows *X* diverging: it implies that *X* is composed of *y* and *z*. Figure 15(b) could be shown as *z* and *y* flowing from the same source terminator or transformation into *A* and *B*, but their relationship as part of *X* would be lost.

Figure 16(a) implies that *p* and *q* are part of the composite dataflow *R*. Figure 16(b) shows two transformations–*D* and *E*–which both produce the dataflow *R*, but not at the same time: under some



A
X
B
A
z
X
y
B
(a) X is used by both A and B
(b) z and y are components of X

Figure 15.



D
p
R
q
E
D
R
E
(a) p and q are components of R
(b) D and E are mutually exclusive; both produce R

Figure 16.

conditions, *D* will produce *R*, under others *E* will produce *R*.

The diagrammatic styles shown in Figures 15 and 16 are not merely shorthand devices. They both simplify diagrams (by eliminating the need for transformations which just combine or split dataflows) and make explicit the composition nature of some dataflows.

Interpreting dataflow diagrams

When interpreting dataflow diagrams, with a view to analysing a system's behaviour, the types of flows which occur are important. The type of the data determines how a transformation should respond to the data and whether data needs to be stored in a system. To help an analyst check the consistency of a dataflow diagram in an essential model we need to define how transformations behave with different types of input and output dataflows.

Transformations which operate on continuous flows (such as in Figure 17) are straightforward: since continuous dataflows represent things which exist (i.e. have defined values at every instant in time), these transformations accept input and produce output continuously.

Continuous flows and 'continuous transformations' are common when modelling realtime systems.

The interpretation of how a transformation responds to a discrete flow is more subtle. The transformations can be thought of as being in a state of readiness, waiting for a data value to arrive. An arrival of a value immediately activates the transformation, which processes the value instantaneously, generating a value for an output flow.

If the input to a transformation is discrete, the output *must* be discrete. This is because the transformation can only be active at the instant the discrete input flow arrives and so can only produce an output at that point. Discrete dataflows occur most often in modelling data processing software and often correspond to the notion of a 'transaction'; in fact, the notation introduced above only supports discrete dataflows.

Interpreting a dataflow model is a little more complex when more than one input flow is attached to a transformation. Again, more than one continuous flow as input to a transformation is not a problem: since all continuous flows always have a value, a transformation can continuously operate on the flows. But, how, for instance, should Figure 18 be interpreted?

The transformation *T* is permanently active because of the continuous flow *c*. However, *T* cannot complete its processing, and thus produce *O*, until it has a value of the discrete flow *d*. The interpretation of Figure 18 is that (unless values of *c* are being stored 'inside' *T*), only the value which *c* has at the instant that *d* has a value is used with *d* to produce *O*.

Consider a system which tracks the position of a satellite. The satellite transmits its position continuously, but the ground station only needs to be able to check the position occasionally – by displaying the position and time on a VDU screen. The transmitted *Position* is therefore modelled as a continuous flow into a transformation which displays the time and position when a discrete flow, *Time*, is input. This is depicted by Figure 19.

However, look at Figure 20. It is ambiguous if the interpretation rules given above are applied. In fact, there are two ambiguities in Figure 20: the first concerns the inputs to the transformation, while the second concerns the outputs:

1 Given the simple rules above *either d1* or *d2* will activate *T2*. But another interpretation of the diagram is that *both d1* and *d2* are needed to produce the outputs. If the latter is the correct interpretation, then how is it guaranteed that *d1* and *d2* arrive at the same time? Or is the first to arrive stored pending the arrival of the second?
2 It is not clear whether both outputs, *O1* and *O2*, are produced by the transformation when it is activated or whether only one might be produced. Further interpretation rules are required.

This type of situation arises from the need to synchronize the transformation of the dataflows (remember the question about synchronization posed after Figure 6). This type of transformation shall be defined as synchronous data transformation for which rules are defined that eliminate the ambiguities discussed above. For a synchronous data transformation, only *one* discrete input is allowed; if the input is a composite dataflow, all components must have a value. Only discrete outputs may exist. If more than one exists, they are to be interpreted as mutually exclusive alternatives.

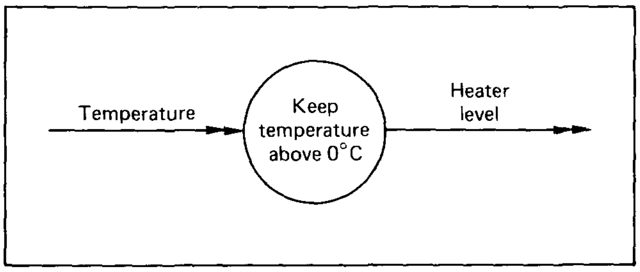


Figure 17.

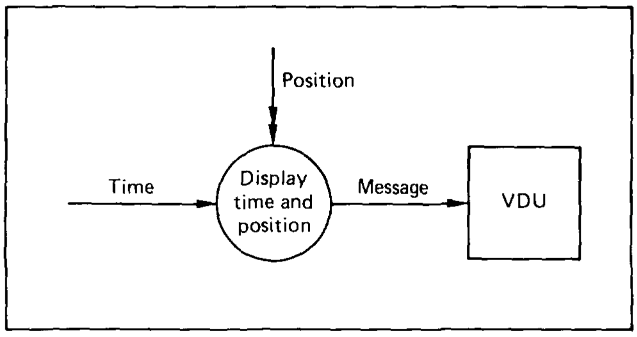


Figure 19.

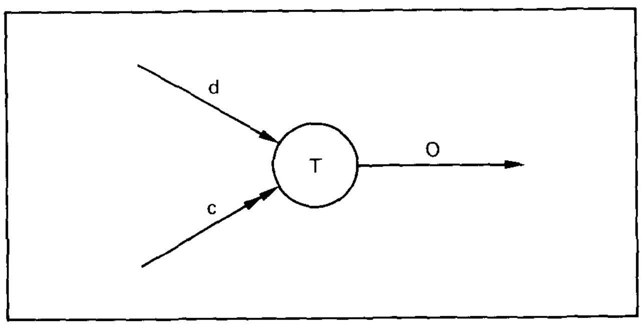


Figure 18.

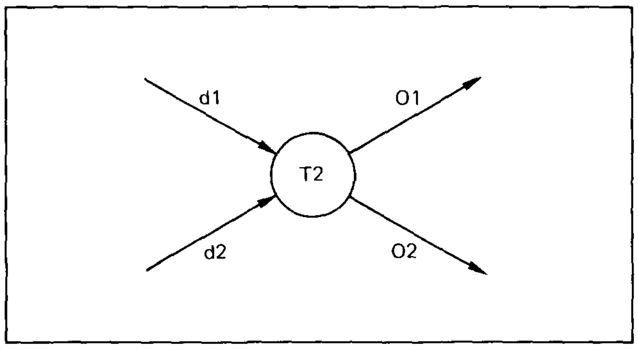


Figure 20.

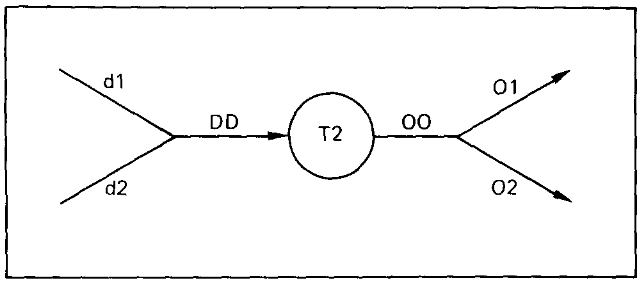


Figure 21.

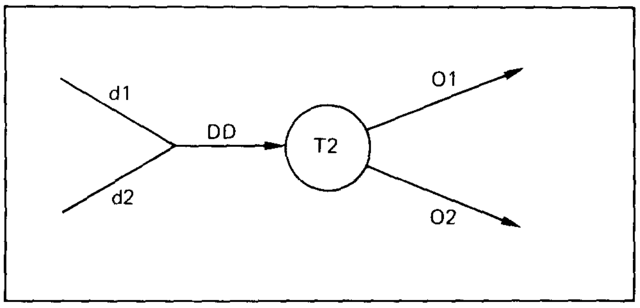


Figure 22.

Therefore, to show that both *d1* and *d2* are needed by *T2* to produce *both O1* and *O2*, one must combine the inputs into *DD*, say, and the outputs into *OO*, as shown in Figure 21. If *O1* and *O2* are alternative outputs, then Figure 22 depicts this.

Note that the rules for synchronous data transformations, only apply to labelled *discrete* dataflows; they do not apply to flows from stores.

This clarification of the use of discrete dataflows allows the procedural operators to be dropped from the notation. Therefore, returning to the soup example used earlier (Figure 4), it is now possible to redraw it using these conventions so as to simplify and clarify the diagram (see Figure 23).

Note one of the interpretations of the ambiguous Figure 20 can be made explicit, as in Figure 24, by using a data store. Data stores introduce the notion of a time delay into the interpretation of dataflow diagrams: rather than combining flows into a composite dataflow which will activate a transformation, they can be combined in a store from which a transformation can extract any component.

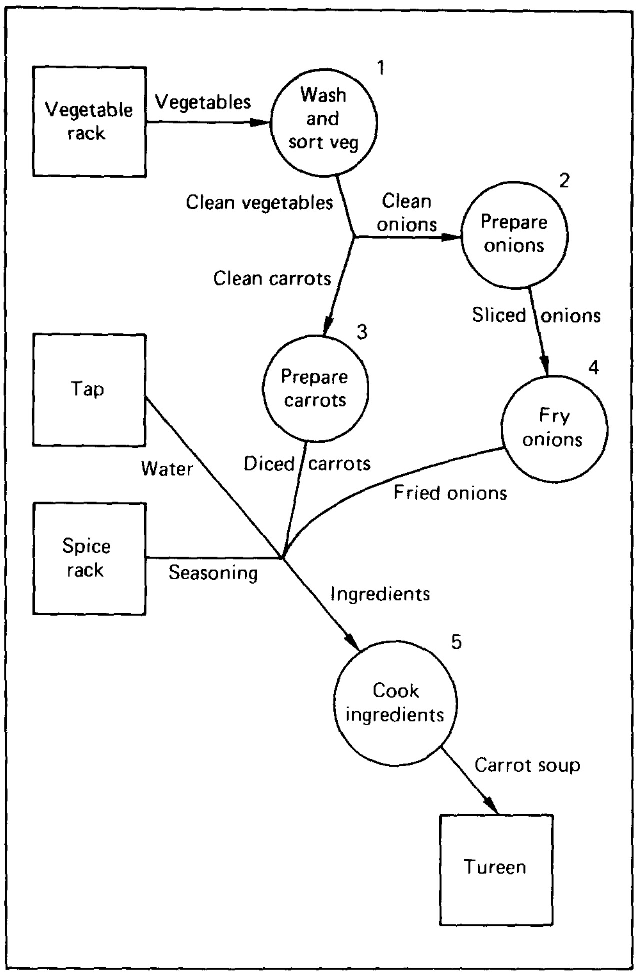


Figure 23.

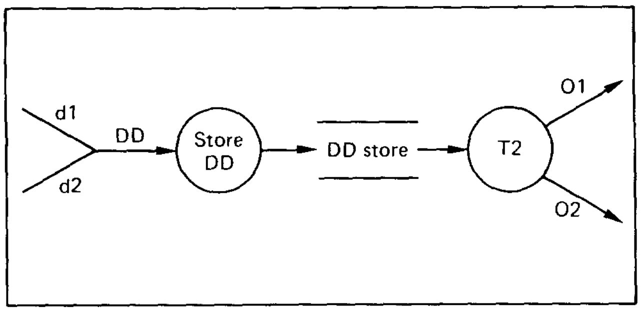


Figure 24.

Controlling transformations

As a part of an essential model a dataflow diagram should have no implementation bias and can be regarded as a specification for machines whose technology is perfect. While this allows us to disregard details of how the system's behaviour might vary due to processing time, we cannot ignore behavioural changes caused by events which occur in real time. For example, in essential modelling we should ignore the need for buffering due to the different processing speeds of interacting devices, but we cannot ignore the fact that automatic pilot software should stop controlling an aircraft if the pilot switches it off!

If we re-examine the dataflow diagram which modelled the making of carrot soup (last seen in Figure 23), it should be noticed that as long as ingredients arrive at the transformation to *Cook ingredients*, they will be cooked: the model does not reflect the behaviour which would be required of such a system–that ingredients are ignored (i.e. not cooked) if the *Cook ingredients* transformation is not ready. There is no element of control, no dynamic aspect, in the model. What is needed is a pair of *control flows* which apply the heat to the pot, or remove it. In other words

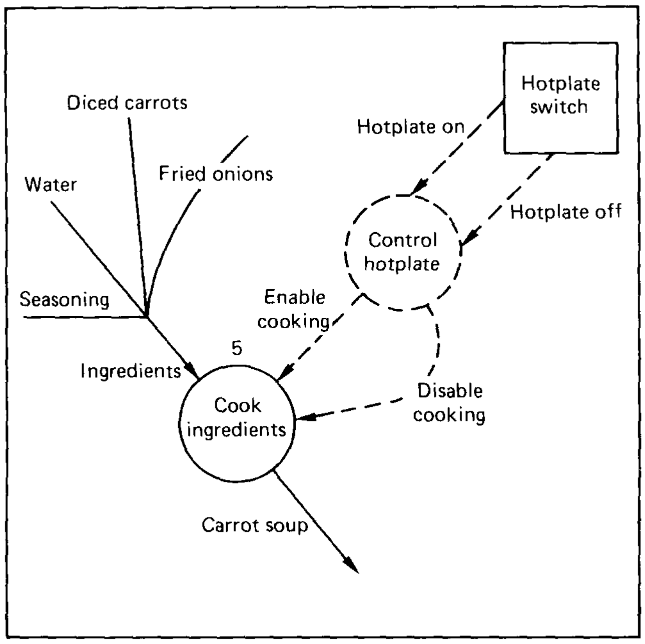


Figure 25.

control flows should be introduced to allow the transformation *Cook ingredients* to be activated (or to 'enable' it) and to prevent *Cook ingredients* from being activated (or to 'disable' it). This is depicted in Figure 25, which is a fragment of Figure 23 with the *Control hot-plate* control transformation and *Hot-plate switch* terminator added.

As the term implies the purpose of a control flow is to control a transformation not, as is the case of a dataflow, to provide the means by which a transformation models the provision of a function. In Figure 25 *Enable cooking* and *Disable cooking* are used to control the transformation of *Ingredients* into *Carrot soup*. If the *Cook ingredients* transformation is 'enabled', the transformation will occur. Having been enabled, if *Cook ingredients* is 'disabled', the transformation will no longer accept *Ingredients* and will not produce any *Carrot soup*.

The control flows *Hot-plate on* and *Hot-plate off* both come from the *Hot-plate switch* terminator; i.e. they both come from the system's environment. These external controls are then transformed to internal controls–the control flows *Enable cooking* and *Disable cooking*–which allow or disallow the cooking of the soup.

Note that the control transformation which 'enables' and 'disables' *Cook ingredients* cannot be dispensed with; that is the *Enable cooking* and *Disable cooking* control flows cannot simply come from the *Hot-plate switch*. This is not just a matter of a syntax rule which insists that control flows from the environment go via a control transformation. As we shall see later, a control transformation is needed in order to interpret the control flow from the environment. (For instance, *Control hot-plate* would have to 'know' what to do if two *Hot-plate on* flows occurred in succession.)

The crucial difference between a discrete dataflow and a control flow is that the former both has an occurrence (at some point in time) and has *content*. Control flows occur, but have no content, and it is their occurrence which is significant. The need for control flows arises from the common situation (particularly in realtime systems) in which a system must be signalled that something has happened, or a command has been issued. For example, switching a machine on or off will be of significance to the controlling software.

Another need to model the control of a data transformation arises when a transformation must be activated for an instant only (rather than for some period, as in the control of cooking soup). A typical situation is modelled in Figure 26. The fragment of dataflow diagram in that Figure represents part of an electronic till system: at the end of the day the till system will print summaries of the transactions made and stored throughout the day. The *Print summaries* transformation is not activated until the key is inserted and turned to a certain position. This is modelled by the input control flow *Print summaries command*. The *Till control* control transformation interprets this input and 'sends' a control flow – *Trigger print summaries* – to *Print summaries* to activate it. On activation, *Print summaries* instantly prints the day's totals (remember that in an essential model transformations 'execute' instantly).

The words 'enable', 'disable' and 'trigger' which were used above have been chosen intentionally. They are reserved for use with control flows and have special meanings. To enable a transformation means to allow it

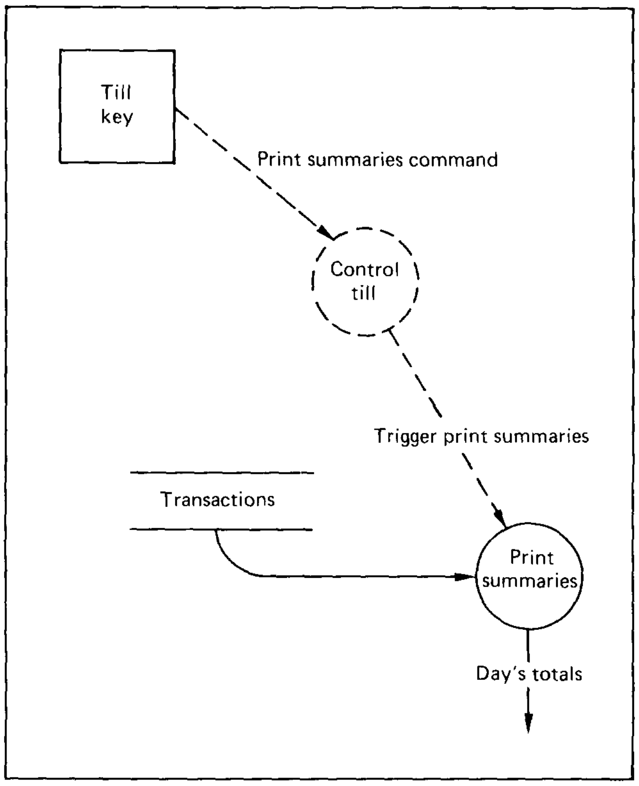


Figure 26.

to be activated by a dataflow. That is, to enable a transformation is to cause the transformation to be ready to operate on its input data until it is disabled.

To disable a transformation is to prevent it from being activated, thus preventing the transformation of input data. When a transformation is not active, all data flowing to it is ignored; it is *not* queued up for later use. Enabling and disabling control flows should be paired. (As a notional shorthand an 'Enable/Disable' label may be used on a single control flow if the same control transformation is the source of the control flows.)

Consider, for example, an electronic arcade game. The context diagram of the system which implements the game shows how commands from the game machine's various buttons will be transformed into actions on the machine's screen. However, arcade games are not free: a coin slot would usually start such a game, and a timer would halt it. These requirements are modelled by the *Enable game* and *Disable game* control flows shown in Figure 27.

To trigger a transformation is to activate it in such a way that it deactivates itself as soon as it has finished its task. For example, consider a system which continuously monitors the temperature of a device. The average temperature is to be calculated by storing measured temperatures at fixed periods in time. A transformation *Store temperature* can be used to model the recording functions of the system. The temporal aspect of the requirements can be modelled by a *Clock* terminator: the *Clock* signals the *Control monitoring* transformation that the required time period has elapsed by the control flow *Tick*. *Control monitoring* will decide if the occurrence of *Tick* means that another temperature should be stored. If so, it will output *Trigger store temperature*. (How such decision making is modelled is described in the section entitled implementation modelling.) Figure 28 shows this model.

The examples looked at so far have involved a control transformation transforming control flows from the environment into Enable/Disable or Trigger control flows. Often the initial control originates within the model of the system – within a data transformation. The next example illustrates how a data transformation can generate a control flow which leads to the triggering of others.

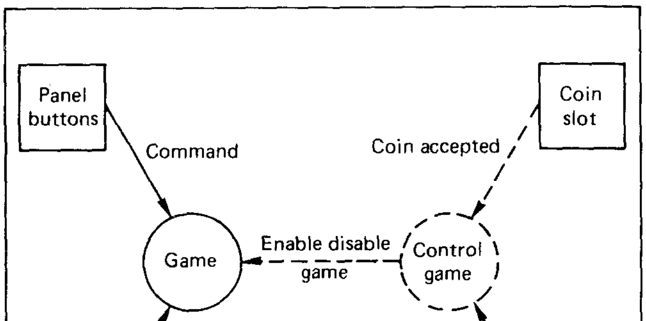


Figure 27.

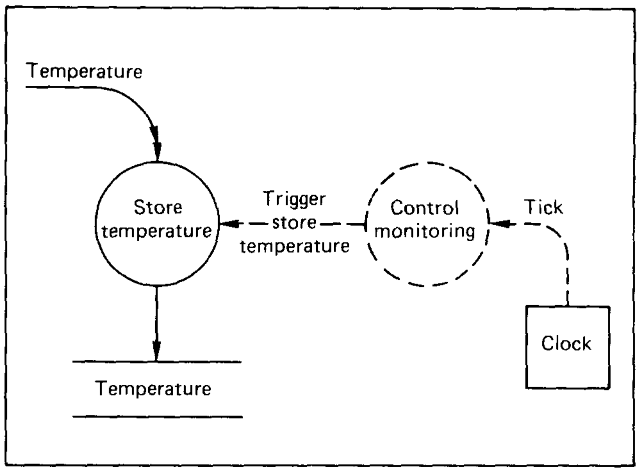


Figure 28.

Consider the system to monitor temperature and pressure shown in Figure 13. If the statement of requirements is revised to now specify that the chemical plant be shut down if either the temperature or pressure becomes unsafe:

. . . In the event of a critical level being reached the system must be shut down immediately. The shut down process will involve interacting with the shift supervisor through his/her console. ...The status of all commands entered during shut down must be shown on the supervisor's console. . .

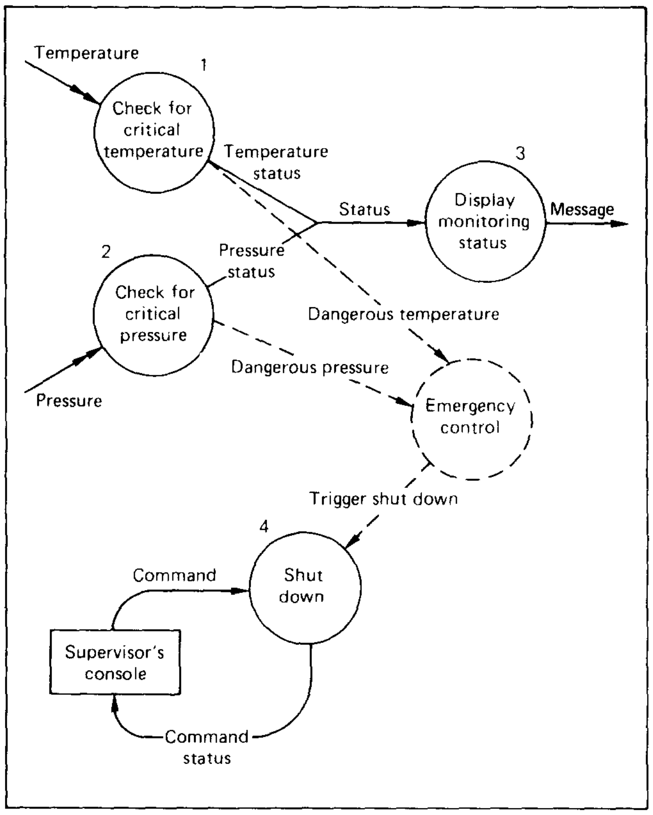


Figure 29.

The diagram in Figure 13 must be revised as shown in Figure 29. First, a transformation, *Shut down*, must be introduced which takes *Command* input from the *Supervisor's console* and outputs *Command status*. Second, control flows to activate (or 'trigger') the new transformation must be output from the transformations which check the temperature and pressure. A control flow which is labelled by 'Enable', 'Disable' or 'Trigger' is called a prompt.

The need for different types of prompt arises from the different types of input dataflows which dataflow diagrams allow, and the way in which these different types activate transformations. Recall from previous section on interpreting dataflow diagrams that a continuous flow keeps a transformation active, a discrete dataflow activates a transformation when an input value occurs, and that a data store makes data permanently available. It is because of this variety that there is a variety of ways in which transformations may be controlled.

If an input to a transformation is a discrete dataflow, Enable/Disable prompts may be used to control its output: if a transformation *T1* (see Figure 30(a)) has been enabled, the arrival of a value of *d* will cause the generation of *w*. A trigger should *not* be applied to such a transformation because synchronization between the occurrence of the trigger and the arrival of the discrete data value cannot be guaranteed.

A trigger should be used to activate a transformation whose only input is a flow from a store, as in Figure 30(b). The occurrence of the trigger activates the transformation *T2* and allows it to draw upon some data stored in *S* in order to produce *x*.

Both triggers and enable/disable prompts may be applied to transformations which have continuous input flows. If a transformation *T3* (Figure 30(c)) is triggered, it will use the value of *c1* at the point in time at which the trigger occurred to instantaneously produce *y*.

Figure 30(d) shows how the transformation *T4* can be enabled to allow values of *c2* to produce *z*. Disabling *T4* causes values of *c2* to be discarded and *z* to be undefined. This permits the selection of values of *c2* over a period of time–as delimited by the occurrence of the enable and disable prompts.

Note that in dataflow diagrams which do not include explicitly enabled transformations, the transformations are considered to be permanently enabled.

However, not all types of flow and transformation are compatible. The Yourdon technique allows the following possibilities:

1 Control flows are allowed as prompts to data transformations, as depicted in Figures 25, 26, 27 etc.
2 Control flows are allowed as prompts to control transformations, as depicted by Figure 31.
3 Within a system a control flow may be output by either a control transformation or a data transformation, shown in Figures 32(a) and (b) respectively.

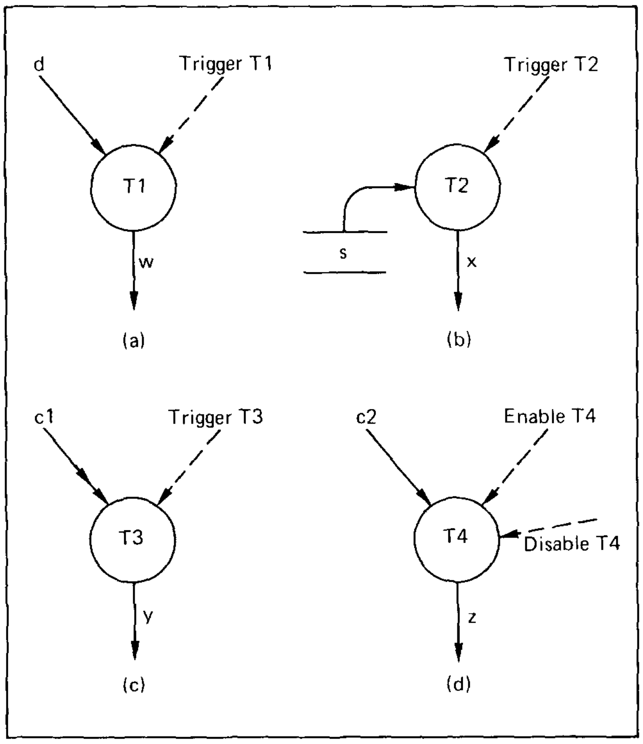


Figure 30.

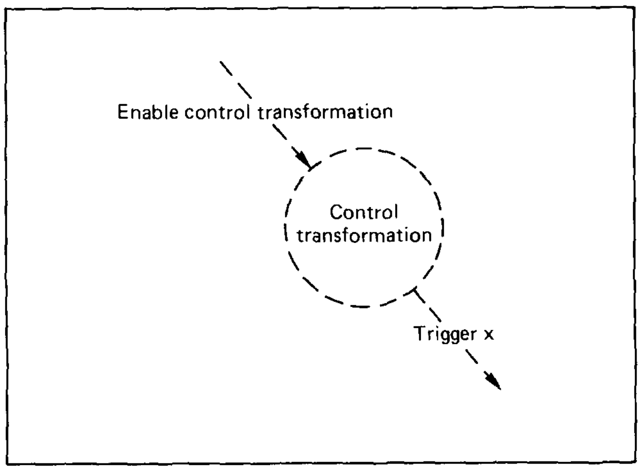


Figure 31.

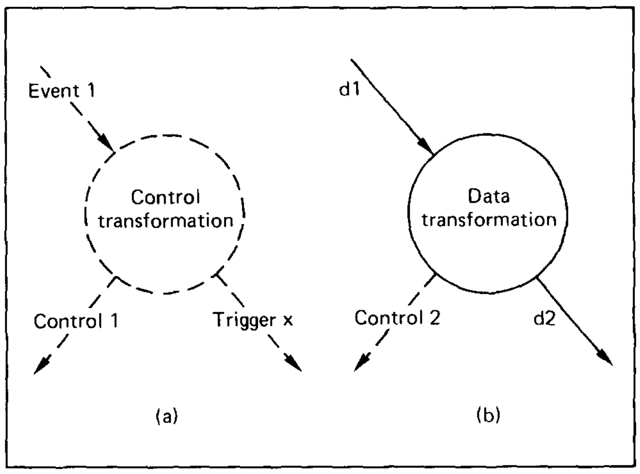


Figure 32.

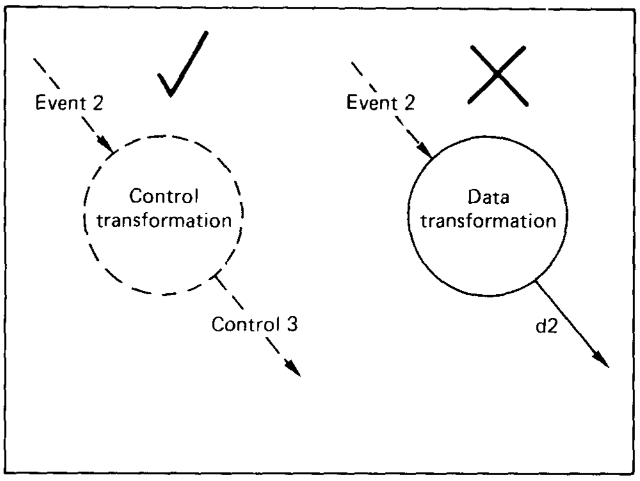


Figure 33.

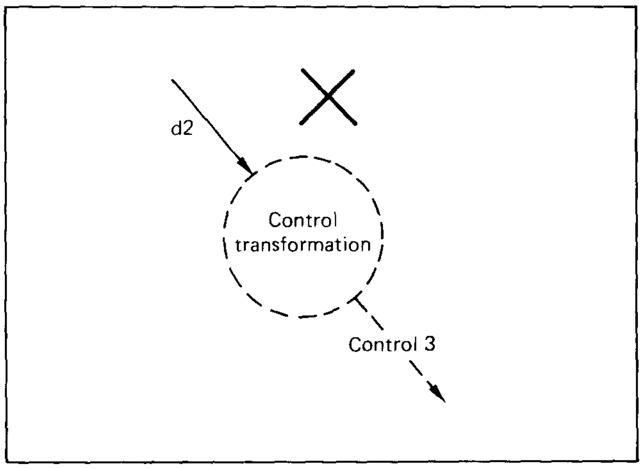


Figure 34.

4 Control flows (other than prompts) may be input to control transformations, but not to data transformations. See Figure 33.
5 Data flows may not be input to control transformations. See Figure 34.

Summary of dataflow diagram syntax

The syntax rules of dataflow diagrams are summarized below:

1 All dataflow lines must be labelled with unique dataflow names, unless they are to or from a store (in which case no label need be used).
2 The transformations must be numbered relative to a diagram.
3 A diagram must be numbered and include an explanatory caption.
4 A transformation may not originate data.
5 Data flowing to or from a data store must come from or go to only a transformation.
6 Data flowing to a sink or from a source may only come from or go to a transformation.
7 Only one discrete flow may be input to a transformation (except in the context diagram); this restriction ensures synchronous data transformation and the removal of ambiguity from the interpretation of transformation activation by discrete flows.
8 If a transformation has more than one discrete output flow, they should be interpreted as alternatives.
9 Data stores may introduce a time delay in the model; dataflows to and from a store are discrete and are only labelled if a component of the stored data is being used.
10 Consistency between the levels of a functional hierarchy of dataflow diagrams must be ensured: the number, names and types of flows must be preserved between a transformation and its refinement.

From essential modelling to implementation modelling

This final section outlines the other activities in YSM by looking briefly at how essential modelling and implementation modelling is carried out. The way in which control transformations are specified and the way in which the relationships between stored data are modelled are also discussed by way of completing the discussion of YSM.

Essential modelling

Essential modelling is concerned with producing a model of the behaviour of a system. The model comprises a number of notations, both textual and graphical. These notations include dataflow diagrams, state transition diagrams, entity relationship diagrams, natural language and data dictionaries. Like the DeMarco approach, essential modelling begins with a system's environment; unlike that approach essential modelling has to do with behaviour rather than function. Therefore the starting point is not the primary functions of the system, but the events in the environment to which the system must respond.

To begin with, an event list is constructed. This is often a table such as that shown below:

Event	Name of flows	Response	Name of flows
Punter plays game	*Command (D)*	*Change screen*	*Screen actions (D)*
	Coin Accepted (C)	*Start Game*	*Enable Game*
Time runs out	*Time expired (C)*	*Finish Game*	*Disable Game*

The table shows an event list for the arcade game whose dataflow diagram is given in Figure 27. The event and response plus the flow names and their types (e.g. D = discrete) are enumerated in the event list.

From the event list a preliminary dataflow diagram which represents the behaviour of the system is constructed. This is done by considering each event and modelling how the system should respond to it. In parallel, the control transformations are specified and the data relationships are modelled (see below). All these activities can effect each other and cause revisions. The preliminary dataflow diagram is then restructured into a levelled set (of the type described earlier).

The main difference between this activity and the old way in which a model of a system was created is that the levelling happens last in YSM but was used as modelling progressed in the old method.

Specifying control transformation

The modelling tool for control transformations is the state transition diagram. A state transition diagram specifies how a control transformation is to take account of its input control flows and how it is to output control flows. The effect which input control flows have depends on the 'state' of the system, as represented by the state transition model: input control flows may change the 'state' of the system and may cause control flows to be output. And, as we have seen, the output control flows have an effect on the dataflow model of system behaviour.

A system's state is a mode of system behaviour which is externally observable. That is, if the system's behaviour was monitored, each state would be distinguishable. A state also included a system's potential response to an input (i.e. to a control flow). This means that if, when behaving in a certain way, a system were capable of responding to an event occurring at different times in, say, two different ways, then two states must exist for that mode of behaviour. For example, many compact disc players can be programmed to play tracks from a CD either sequentially or in an order chosen by the listener. When a track is playing it is in one of two states depending how the CD player has been set up: the first state will respond to the occurrence of the end of the track by moving to the next in the sequence stored on the CD; the second state will respond to the end of the track by playing the next listener-programmed track.

The state of a system in Yourdon modelling does not include *how* system should behave if the state changes: that is modelled by the dataflow diagrams. However, a change of state allows a control flow to be output in order to effect the behaviour represented by a dataflow diagram.

In a Yourdon model, either an event or the achievement of some condition causes a system to change from one state to another. Such events or conditions are modelled by input control flows to a control transformation. The act of changing from one state to another is called a state transition. As the system is in transition between states an action may be taken. Such actions are modelled by output control flows. In this way a control transformation can effect the behaviour specified by the dataflow model – by responding to input control flows with state transitions which cause output control flows.

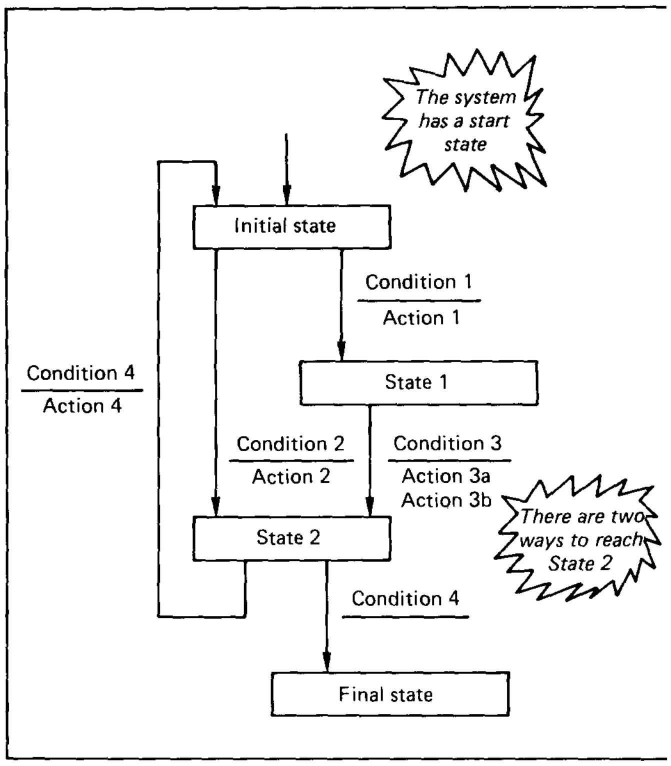


Figure 35.

The lexicon of state transition diagrams is given in Figure 35. States are represented by rectangles; the *Initial state, State 1 State 2* and *Final state* are the states of the system known as the control transformation which the state transition diagrams specifies. (These need not be all the states of a system.)

Transitions are represented by directed lines; these are usually straight and meet each other at right angles. Thus, there are two possible transitions from the *Initial state*: one to *State 1* and the second to *State 2*. There are no transitions from the *Final state*. The transition lines are labelled by the conditions which have to hold in order to change states, and what actions must happen if such a transition occurs. The conditions are shown above a horizontal line and any actions are shown beneath the horizontal line. For example, Figure 35 specifies two ways in which to change from the *Initial state*: if *Condition 1* occurs, *Action 1* is taken, and the system moves to *State 1*; if *Condition 2* occurs, *Action 2* is taken, and the system moves to *State*.

By convention the initial state is shown with an incoming transition line from 'nowhere'. This transition can be labelled by conditions or actions as just described. A system's final state is recognizable by the absence of any transition line leaving the state.

The input control flows to a control transformation must always appear as conditions for state transitions in the model which specifies the transformation. Similarly, all the output flows should appear as transition actions. However, not all conditions need cause an action: they may simply cause a change of state. In such a situation, nothing appears below the line separating conditions and actions in a transition label. There is an example of this in Figure 35: if *Condition 4* occurs when the system is in *State 2*, a transition occurs to change the system to its *Final state*, but no action is specified.

This is not the only way in which control transformation differ from data transformations. A control transformation has, as its name suggests, a controlling or management rôle in an essential model. Because a control transformation can 'start' or 'stop' a data transformation by issuing prompts, its specification must include references to the data transformations under its control. A control transformation is thus inextricably bound to its data transformations. On the other hand, dataflows may be modelled, and eventually specified, independently of other transformations.

Consider the system to sort a list of names which was introduced earlier and in Figure 4. The control of the system may be introduced into the model of that system's behaviour by adding a control transformation as shown in Figure 36. That Figure represented the triggering of the sorting process by the detection of the end of the sequence of names, and the triggering of the printing process by the detection of the completion of the sorting process. Figure 36 shows a *Sort system control* transformation which 'manages' the system's behaviour by modelling its states and responding to conditions which arise. It manages the behaviour of the system by knowing about the system's data transformations and its states and by responding to conditions which occur as input control flows with output triggers

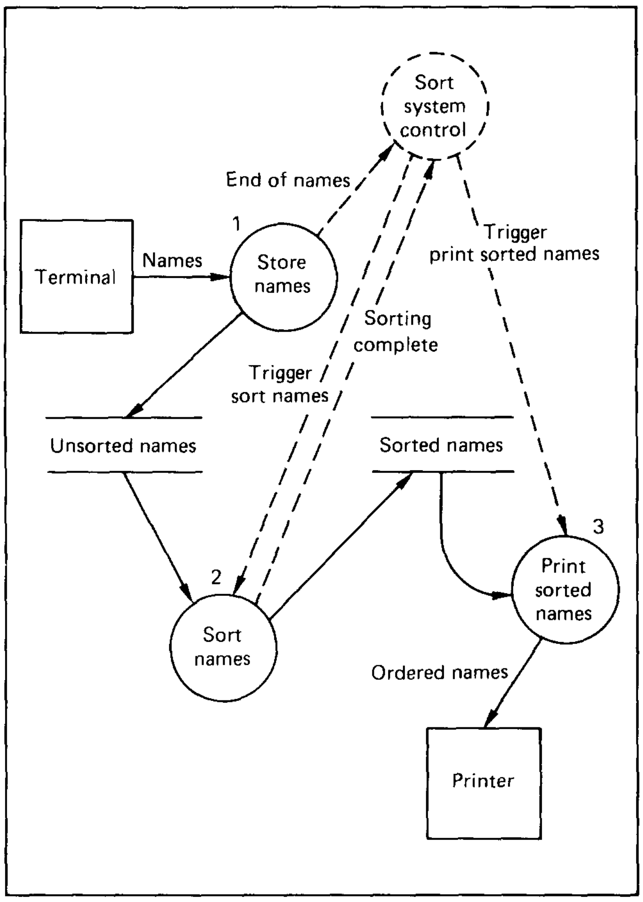


Figure 36.

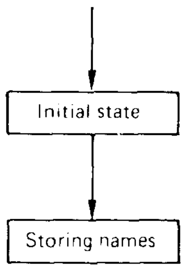
which model changes in the behaviour of the system. The states of the system are given below:

- initial state
- storing names
- sorting names
- printing names
- final state

The conditions which change states are listed below. The states which they cause a change from (the 'current' state) and to (the 'next' state) are given, together with any action which is to be taken. The fragment of state transition diagram which models these is then given:

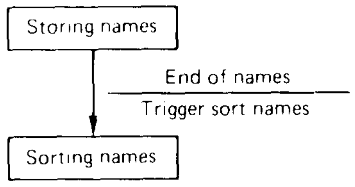
current state:	*initial state*
condition:	none
action:	none
next state:	*storing names*

is depicted as:



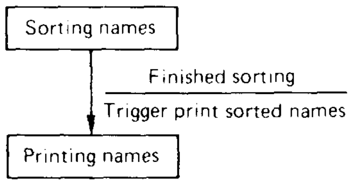
current state:	*storing names*
condition:	*end of names*
action:	*trigger sort names*
next state:	*sorting names*

is depicted as:



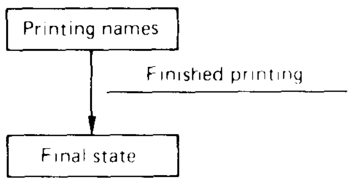
current state:	*sorting names*
condition:	*finished sorting*
action:	*trigger print sorted names*
next state:	*printing names*

is depicted as:



current state:	*printing names*
condition:	*finished printing*
action:	none
next state:	*final state*

is depicted as:



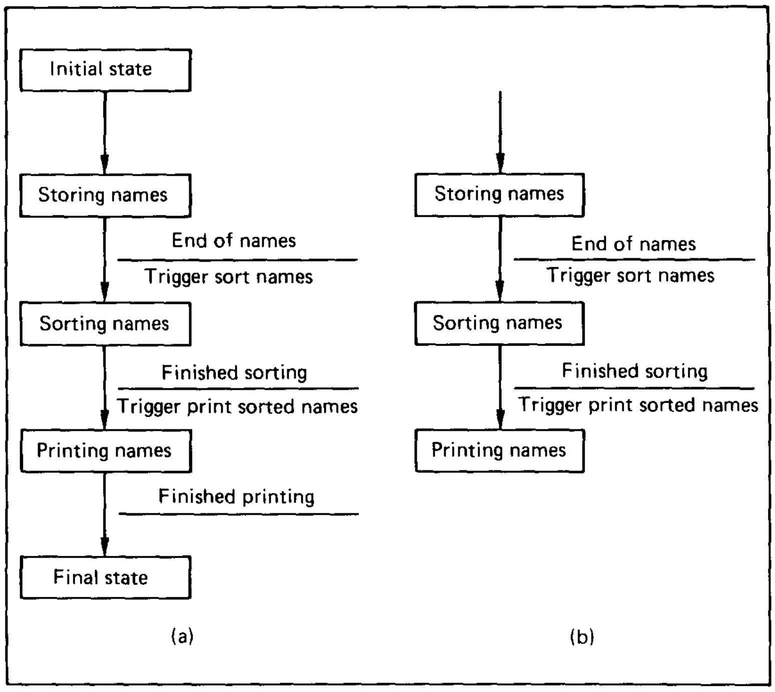


Figure 37.

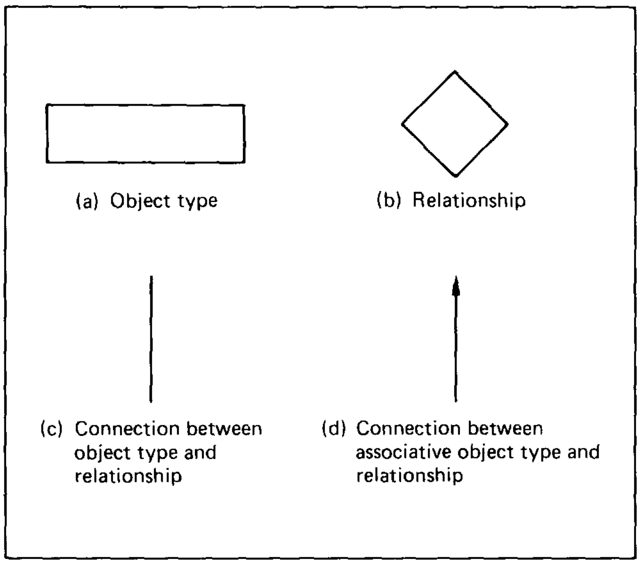

(a) Object type
(b) Relationship
(c) Connection between object type and relationship
(d) Connection between associative object type and relationship

Figure 38.

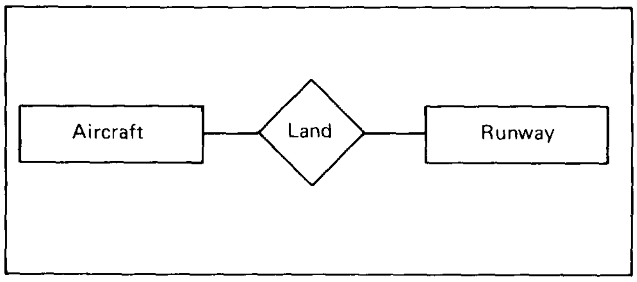


Figure 39.

Collecting these fragments together gives Figure 37(a). But, you can see from that Figure that *initial state* and *final state* are redundant, and could not be observed externally as being different from *storing names* and *printing names*. *Initial state* and *final state*, and the condition which causes the transition to *final state* can be dropped completely. The final state transition diagram is given Figure 37(b).

Modelling data

Just as dataflows must be given some meaning in order to strengthen an essential model, the entities of the customer's world must be modelled. In particular the relationships between the entities should be established. Entity-relationship diagrams are used in this part of the essential model[7,9]. The symbols of entity-relationship diagrams are shown in Figure 38.

Consider a system for reporting on the usage of runways at an airport. The entities in a system like this include runways and aircraft. These can be modelled by the two entity types *Aircraft* and *Runway* which are related by the act of landing, represented by *Land*. These types and relationships are shown in Figure 39.

Data modelling is a field of study in its own right and is not further discussed here. The important point to note is that the rest of the essential modelling techniques must produce parts of the model which are consistent with the data model.

Implementation modelling

Implementation modelling is the activity which marks the beginning of system design. Its purpose is to make an essential model more concrete by considering the impact of real, finite-capacity technology, and performance, size, equipment constraints. While an essential model is independent of an implementation, an implementation model is an implementation-oriented model of a system's behaviour. An implementation model is represented by the same notations used in an essential model (dataflow diagrams, state transition diagrams, etc.) plus the structure chart notation which is used to show the structure of modules of code[6].

Since an implementation model depends on real technology it may not be viable for economic or technical reasons, and so may have to be abandoned.

As well as being derived from the essential model, rather than the statement of requirements, an implementation model has two other important characteristics. First, its focus is the technology which is to be used by the system to solve a problem. The essential model's focus is the problem itself. Second, the implementation model is intended to facilitate communication within the developer's organization, rather than between the developer and the customer.

Implementation modelling is carried out in a number of phases:

1 Model the physical processors involved: this involves changing the dataflow diagrams and state transition diagrams of the essential model so that groups of transformations are allocated to be implemented by particular devices.
2 Model the software environment in which the system is to exist: often, the functions required of a system are provided by some part of the environment such as the operating system. The dataflow diagrams are changed to show this. Also, the entity relationship diagrams can be used to determine the use of any database management systems which are available.
3 Model the structure of the software to be produced: this involves transforming the dataflow diagrams into a tree structure (represented by a structure chart) which represents the architecture of the software to be constructed.

At each stage of the transformation of the essential model to the physical, an attempt is made to avoid distorting the essential model.

In essence, implementation modelling is the activity which bridges the gap between specification and system design.

Conclusions

Graphical notations underpin most of the informal approaches to software requirements analysis. The Yourdon dataflow diagram notation is a good example of the genre. It has an intuitively simple syntax and semantics which encourages use (as well as extension and misuse). In this paper both the original dataflow diagram conventions and more recent rules (based on the conventions) were given.

The dataflow diagrams which are now used in the essential and implementation modelling phases of YSM have a precise syntax and semantics, which were given. Both models consist of a number of documents which describe different aspects of the system. These aspects are listed below:

- how data is to be processed if the software is to provide the functionality which the customer requires,
- how the processing is to be controlled (by both external and internal events),
- what the data is and how data is related.

In conclusion, this paper argues that the dataflow diagram is a useful notation which can be used with some discipline. The vagueness of the early analysis approaches which used dataflow diagrams did encourage casual use. The current Yourdon method now means that if the syntax of dataflow diagrams is adhered to, sufficient semantics can be ascribed to a set of diagrams for the purpose of informal analysis and reasoning.

References

1 **Ross, D T and Schoman, K E Jr.** 'Structured analysis for requirements' Definition, *IEEE Trans. Software Eng.* Vol SE-3, No 1, pp 6–15, (January 1977)
2 **Stevens, W P, Myers, G J and Constantine, L L** 'Structured design' *IBM Syst. J.* Vol 13 No 2, pp 115–39 (May 1974)
3 **Gane, C and Sarson, T** *Structured Systems Analysis: Tools and Techniques* Prentice-Hall, Englewood Cliffs, NJ, USA (1979)
4 **DeMarco, T** *Structured Analysis and System Specifcation* Yourdon Press, New York, USA (1978)
5 **Weinberg, V** *Structured Analysis* Yourdon Press, New York, USA (1978)
6 **Yourdon, E and Constantine, L L** *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, 2nd Ed., Yourdon Press, New York, USA 1978
7 **Ward, P T and Mellor, S J** *Structured Development for Real-time Systems Volume 1: Introduction and Tools* Yourdon Press, Englewood Cliffs, NJ, USA (1985)
8 **McMenamin, S M and Palmer, J F** *Essential Systems Analysis* Yourdon Press, New York, USA (1984)
9 **Ward, P T and Mellor, S J** *Structured Development for Real-time Systems Volume 2: Essential Modelling Techniques* Yourdon Press, Englewood Cliffs, NJ, USA (1985)

Further reading

Mellor, S J and Ward, P T *Structured Development for Real-time Systems Volume 3: Implementation Modelling Techniques* Yourdon Press, Englewood Cliffs, NJ, USA (1986)
Myers, G J *Composite/Structured Design* Van Nostrand Reinhold, New York, USA (1978)