characteristics can simplify some multi-tasking problems as we illustrated when considering the transfer of the controller parameters. Simplifications of this sort are part of the art of engineering; however, they must be used with care and must be *documented* – in particular the conditions for which the simplification is valid must be clearly stated.

EXERCISES

7.1 The standard input routines in languages such as FORTRAN, Pascal and BASIC cannot be used within a timed loop to obtain information from the keyboard. This is also true of Modula-2. Why can't we use the standard Modula-2 routines?

7.2 A plant operating in a remote location is controlled by an embedded computer control system. The plant operates in two modes referred to as Amode and Bmode. The control algorithm for Amode is of the form

$$m(n) = Ae(n) + Be(n-1) + Ce(n-2) + Dm(n-1) + Em(n-2)$$

and for Bmode

$$m(n) = K_1e(n) + K_2e(n-1) + K_3e(n-2)$$

where $e(n) = R - c(n)$
R = set point
$c(n)$ = the measured output of the plant at interval n.

The change-over from Amode to Bmode is to be made when $c(n) > ChangeA$ for five successive readings. The change-over from Bmode to Amode is to be made when $c(n) < ChangeB$ for five successive readings. The parameters *ChangeA* and *ChangeB* and the set point R can all be changed from a central station.

A change to the value of R requires a change in the values of *ChangeA* and *ChangeB*. The controller parameters A, B, C, D, E and K_1, K_2 and K_3 also need changing. They must be changed as the set $\{A, B, C, D, E, K_1, K_2, K_3\}$ and not as individual elements. The data transmission link to the remote station has a slow transmission speed and is subject to frequent bursts of interference. You can assume that the data transmission system support software contains error checking software and organises retransmission of erroneous data.

Discuss the problems of designing the software for the embedded computer system and discuss possible ways of dealing with the slow and unreliable data transmission system.

7.3 What is the principal difference between a pool and a channel? Explain why you would use (a) a pool and (b) a channel.

8

Real-time System Development Methodologies – 1

This chapter begins with an overview of the general approach now being adopted in the specification, design and construction of complex real-time systems, followed by a brief description of some of the standard methodologies. The Yourdon methodologies are then described in detail. The aims of the chapter are to:

- Show how specification, design and implementation can be considered as a process of *modelling*.
- Describe the major methodologies.
- Provide a more detailed understanding of one methodology, the Yourdon methodology.

8.1 INTRODUCTION

The production of robust, reliable software of high quality for real-time computer control applications is a difficult task which requires the application of engineering methods. During the last ten years increasing emphasis has been placed on formalising the specification, design and construction of such software, and several methodologies are now extant. The major ones are shown in Table 8.1. All of the methodologies address the problem in three distinct phases. The production of a *logical* or *abstract* model – the process of *specification*; the development of an *implementation* model for a *virtual machine* from the logical model – the process of *design*; and the construction of software for the virtual machine together with the implementation of the virtual machine on a physical system – the process of *implementation*. These phases, although differently named, correspond to the phases of development generally recognised in software engineering texts. Their relationship to each other is shown in Figure 8.1.

Abstract model: the equivalent of a requirements specification, it is the result of the requirements capture and analysis phase.

Implementation model: this is the equivalent of the system design; it is the product of the design stages – architectural design and the detail design.

Table 8.1 Summary of design methodologies

MASCOT	Design, construction, operation and test tools	Jackson and Simpson (1975)
CORE	Controlled Requirements Expression	British Aerospace, Systems Designers (1979)
PAISLey	Specification and simulation tools	Zave (1982)
DARTS	Design and Analysis of Real-Time Systems	Gomaa (1984)
JSD	Jackson System Development	Jackson (1983)
Yourdon (a)	Structured design and development of real-time systems	Ward and Mellor (1986)
Yourdon (b)	Strategies for real-time system specification	Hatley and Pirbhai (1988)
HOOD	Hierarchical Object-Oriented Design	CISI Ingenierie, CRA A/S Matra Aerospace (1989)

Implementation: the process of mapping the implementation model onto the physical hardware, identifying the software modules and coding them.

Although there is a logical progression from abstract model to implementation model to implemented software, and although three separate and distinct artifacts – abstract model, implementation model, and deliverable system – are produced, the phases overlap in time. The phases overlap because complex systems are best handled by a hierarchical approach: determination of the detail of the lower levels in the hierarchy of the logical model must be based on knowledge of higher-level design decisions, and similarly the lower-level design decisions must be based on the higher-level implementation decisions. Another way of expressing this is to say that the higher-level design decisions determine the requirements specification for the lower levels in the system.

All the methodologies require the support of CASE (Computer-Aided Software Engineering) tools for their effective use. Without such tools the methods become too laborious for use on large systems and many of their benefits in terms of enforcing consistency are lost. The number and range of CASE tools available is growing rapidly and some simple ones are now available for PCs (McClure, 1989).

We will consider MASCOT and the two Yourdon methodologies in some detail in later sections and also discuss the use of PAISLey. An introduction to CORE, JSD and HOOD can be found in Cooling (1991, Chapter 10).

CORE (Mullery, 1979) is specifically designed for the requirements capture and analysis phase of the development, that is the construction of the requirements specification. Subsequent design and implementation has to use other methodologies. It is an attempt to find an approach that will reduce the amount of information that the customer 'forgot to tell one about' – of course this will never be reduced to zero.

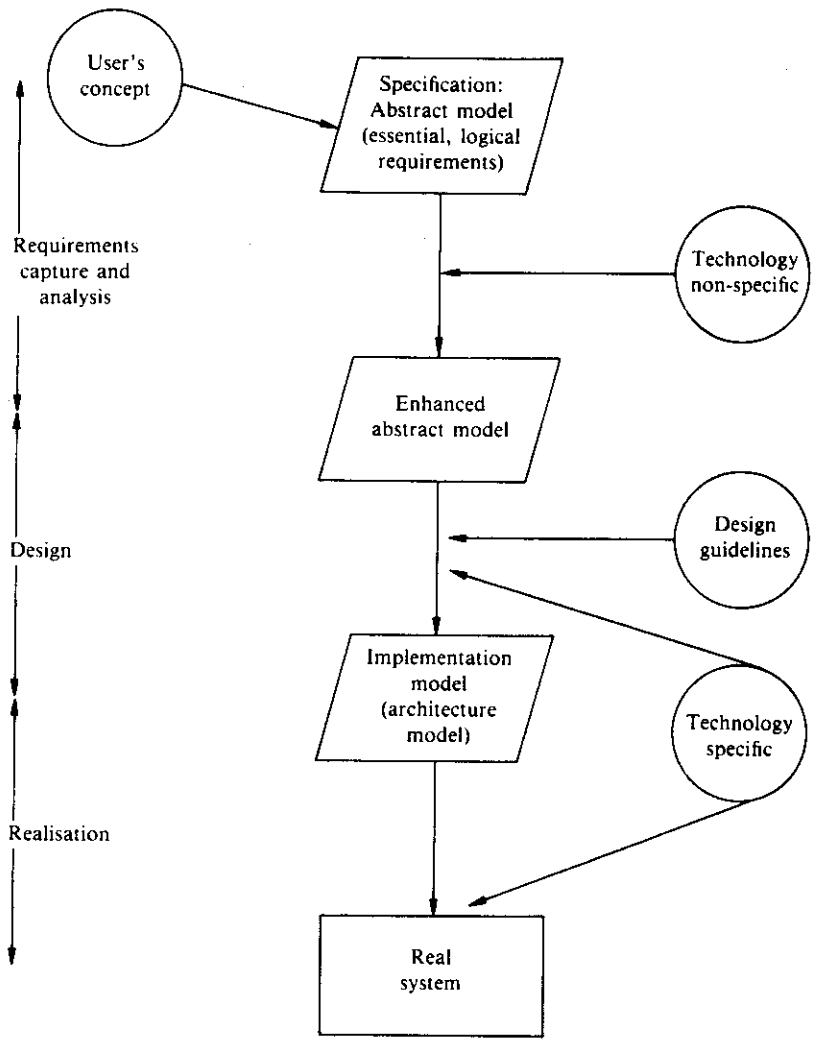

User's concept
Specification: Abstract model (essential, logical requirements)
Technology non-specific
Requirements capture and analysis
Enhanced abstract model
Design guidelines
Design
Implementation model (architecture model)
Technology specific
Realisation
Real system

Figure 8.1 Software modelling.

The Jackson System Development methodology has been widely used in non-real-time systems, as was the Jackson Structured Programming methodology, but has only recently been applied to real-time systems (Cooling, 1991, p. 358). It is a data-driven method and is now supported by a number of CASE tools (Program Design Facility, Speedbuilder and Network Builder) which are supplied by the Michael Jackson Company.

The abstract model in JSD is a network of processes. There are three types of process:

1. *Input*: these detect actions taking place in the environment (events) and pass these events to the internal system.
2. *Output*: these pass the system response back to the environment.
3. *Internal*: these deal with events reported by the input processes and actions resulting from other internal processes; they pass events and actions to output and other internal processes.

The processes are connected in two ways to form the network:

1. by buffered asynchronous data streams;
2. by state vector connections.

By using the state vector one process can inspect the internal state of another process.

In common with other methodologies originally developed for non-real-time applications there is no formal method of incorporating timing constraints. Also, as we will see when dealing with the Yourdon methodologies, JSD provides a notation for representing a specification and a design. It does not perform the design. JSD is a data-flow method and the basic design technique is to use functional decomposition and to preserve *natural* data flows.

Once the design has been produced and an implementation model obtained, then the realisation of this model proceeds in a systematic manner. With the appropriate CASE tools much of the code for JSD designs can be automatically generated, particularly if the implementation language is Ada or occam 2.

HOOD is a new addition to the real-time methodologies and is based on an object-oriented approach. It is targeted at implementations based on the use of Ada but can be adapted for use with other programming languages. Like MASCOT it is meant to be a design method which takes a requirements specification obtained by other means as its starting point. Also like MASCOT the diagrams used for the design have a direct textual equivalent which formalises the design and which can be used in the implementation stage of the development.

In recent years there has been extensive consideration of formal (mathematical) techniques for the specification of systems. The most widely used formal method is VDM and the language Z is also gaining support. These techniques are aimed at producing a *formal specification* of a system. The benefit of a formal specification is that it is possible to prove that it is consistent (it is not possible to prove that it is complete – there is still the 'we forgot to mention' factor – but formal analysis of consistency may well reveal incompleteness). The second advantage of formal techniques is the possibility (theoretically) of transforming a specification into a realised implementation and proving that each step in the transformation is correct. In practice the proofs are very difficult. A major limitation of the formal specification language approach is that at present none of the languages make any provision for the incorporation of timing constraints. In the next chapter we describe one methodology (PAISLey) that has attempted to combine formality and timing constraints.

8.2 YOURDON METHODOLOGY

The Yourdon methodology has been developed over many years. It is a structured methodology based on using *data-flow modelling* techniques and *functional decomposition*. It supports development from the initial analysis stage through to implementation. Both Ward and Mellor (1986) and Hatley and Pirbhai (1988) have introduced extensions to support the use of the Yourdon approach for the development of real-time systems and the key ideas of their methodologies are:

- subdivision of system into activities;
- hierarchical structure;
- separation of data and control flows;
- no early commitment to a particular technology; and
- traceability between specification, design and implementation.

Although the original method was developed as a pencil and paper technique most benefit can be obtained if there is CASE tool support and many software engineering CASE tools now support both the Ward and Mellor and the Hatley and Pirbhai versions of Yourdon. Examples are Software Through Pictures and EasyCase Plus.

There are many similarities between Ward and Mellor and Hatley and Pirbhai but to avoid confusion we will deal with them separately. As an example we shall use the system which is described in the next section.

8.3 REQUIREMENTS DEFINITION FOR DRYING OVEN

1. Introduction

1.1 Components are dried by being passed through an oven. The components are placed on a conveyor belt which conveys them slowly through the drying oven. The oven is heated by three gas-fired burners placed at intervals along the oven. The temperature in each of the areas heated by the burners is monitored and controlled. An operator console unit enables the operator to monitor and control the operation of the unit. The system is presently controlled by a hardwired control system. The requirement is to replace this hardwired control system with a computer-based system. The new computer-based system is also to provide links with the management computer over a communication link.

1.2 The general arrangement of the system is shown in figure 1.1 [Figure 8.2].

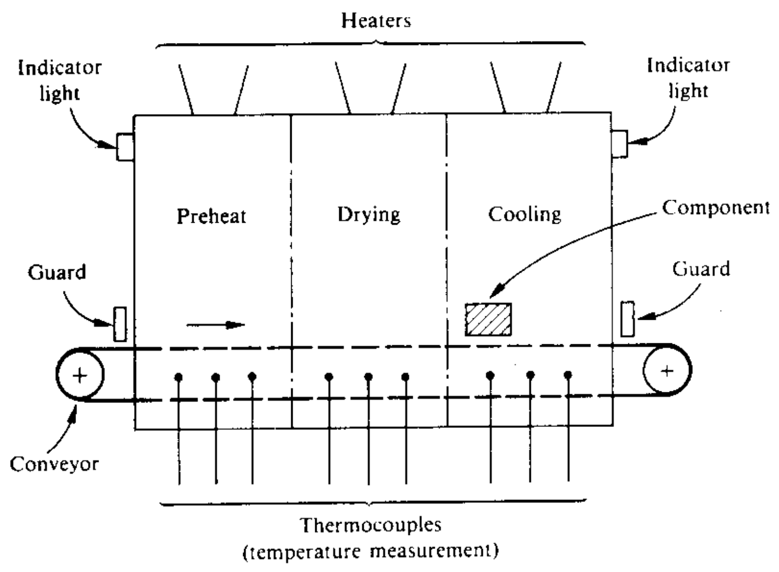


Figure 8.2 General arrangement of drying oven.

2. **Input/Output**

2.1 The inputs come from a plant interface cubicle and from the operator. There will need to be inputs obtained from the communication interface.

2.2 Plant Inputs

2.2.1 A thermocouple is provided in each heater area – the heater areas are *pre-heat*, *drying*, and *cooling*. The inputs are available as voltages in the range 0 to 10 volts at pins 1 to 9 on socket j22 in the interface cubicle.

2.2.2 The conveyor speed is measured by a pulse counting system and is available on pin 3 at socket j23 in the interface cubicle. It is referred to as *con-speed*.

2.2.3 There are three interlocked safety guards on the conveyor system and these are *in-guard*, *out-guard*, and *drop-guard*. Signals from these guards are provided on pins 4, 5, 6 of socket j23. These signals are set at logic HIGH to indicate that the guards are locked in place.

2.2.4 A *conveyor-halted* signal is provided on pin 1 of socket j23. This signal is logic HIGH when the conveyor is running.

2.3 Plant Outputs

2.3.1 Heater Control: each of the three heaters has a control unit. The input to the control unit is a voltage in the range 0 to 10 volts which corresponds to no heat output to maximum heat output.

2.3.2 Conveyor Start-up: a signal *convey-start* is output to the conveyor motor control unit. A second signal *convey-stop* is also output to the motor control unit. The connections are to pins 1, 2, 3 on j10 for convey-start, and to pin 5 on j10 for convey-stop.

2.3.3 Guard Locks: asserting the *guard-lock* line, pin 8 on j10, causes the guards to be locked in position and turns on the red indicator lights on the outside of the unit.

2.4 Operator Inputs

2.4.1 The operator sends the following command inputs: *Start*, *Stop*, *Reset*, *Re-start*, and *Pause*. The operator can also adjust the desired set points for each area of the dryer.

2.5 Operator Outputs

2.5.1 The operator VDU displays the temperature in each area, the conveyor belt speed, and the alarm status. It should also display the current date and time and the last operator command issued.

2.6 Communication Inputs

2.6.1 These have yet to be defined.

3. Functional Specification

3.1 Start-up: starting from cold the operator checks that all the guards are closed and issues the Start command. The guards are locked and if locking is correctly achieved the heaters are switched to on. Under this condition maximum heat is supplied. The temperature is monitored and when each area reaches a temperature within 20% of its set point control of the heaters is switched to normal.

3.2 Conveyor Start-up: when all areas have switched to normal control the conveyor start-up sequence is initiated. The conveyor is stepped through the start-up procedure. Motor position 1 is selected and held until the speed reaches 0.5 ft/min; then position 2 is selected. This is held until the speed reaches 1.5 ft/min; at this point the normal running position 3 is selected. If the conveyor fails to reach the desired speed within 30 seconds the conveyor is stopped and the *conveyor-fault* signal is asserted. The normal conveyor speed is 8 to 10 ft/min. If at any time the speed drops below this for more than 30 seconds the *conveyor-alarm* signal should be asserted.

3.3 Temperature Monitoring: the temperature measurement for each area is read at 2 second intervals. If the temperature for an area varies by more than 5% from the set point for that area then an alarm should be asserted.

3.4 Each area is controlled using a PID control algorithm.

3.5 Conveyor Failure: if the conveyor fails to start the operator can issue a Reset command which closes the whole system down. When it has been closed down and the system checked for obstructions the operator can issue the Start command again. If the conveyor stops or slows during normal running the operator can issue either a Re-start command which causes the conveyor first to be stopped and then to enter the full conveyor start-up cycle, or a reset signal that causes the whole system to be closed down.

3.6 Conveyor Pause: during normal running the operator may issue a Conveyor

Pause command. This halts the conveyor. It may for example be used to permit clearance of a blockage. The conveyor can be re-started by the operator issuing the Re-start command.

3.7 At any time during normal running the operator may issue the Stop command. The response is to turn off the heaters and the conveyor. When the conveyor stopped signal is asserted the guards are unlocked and the display lights are turned to green.

8.4 WARD AND MELLOR METHOD

The outline of the Ward and Mellor method is shown in Figure 8.3. The starting point is to build, from the analysis of the requirements, a software model representing the requirements in terms of the abstract entities. This model is called the *essential model*. It is in two parts: an *environmental* model which describes the relationship of the system being modelled with its environment; and the *behavioural*

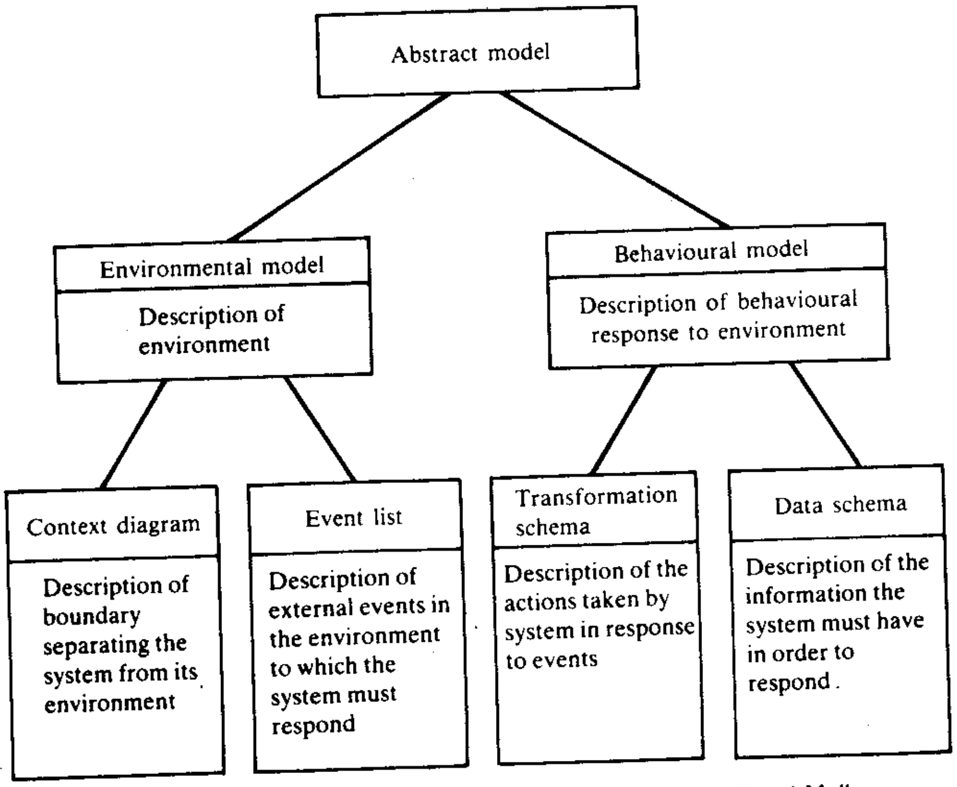

Abstract model
Environmental model
Description of environment
Behavioural model
Description of behavioural response to environment
Context diagram
Description of boundary separating the system from its environment
Event list
Description of external events in the environment to which the system must respond
Transformation schema
Description of the actions taken by system in response to events
Data schema
Description of the information the system must have in order to respond.

Figure 8.3 Outline of abstract modelling approach of Ward and Mellor.

model which describes the internal structure of the system. The second stage – the design stage – is to derive from the essential model an *implementation model* which defines how the system is implemented on a particular technology and shows the allocation of parts of the system to processors, the subdivision of activities allocated to each processor into tasks, and the structure of the code for each task.

The essential model represents *what* the system is required to do; the implementation model shows *how* the system will do what has to be done. The implementation model provides the design from which the implementors of the physical system can work. Correct use of the method results in documentation that provides *traceability* from the physical system to the abstract specification model. The type of documentation produced is shown in Figure 8.4.

8.4.1 Building the Essential Model – the Environmental Model

For most real-time systems the environmental model will comprise a *context* diagram and an *event* list and the entity relationship diagram will not be used. Figure 8.5 shows a context diagram for the drying oven. The rectangular boxes represent *terminator blocks* which are entities that exist in the environment. At this stage we are interested only in the logical function of the signals that connect these units to the system and not in the details of the units or the physical connections between the units and the system.

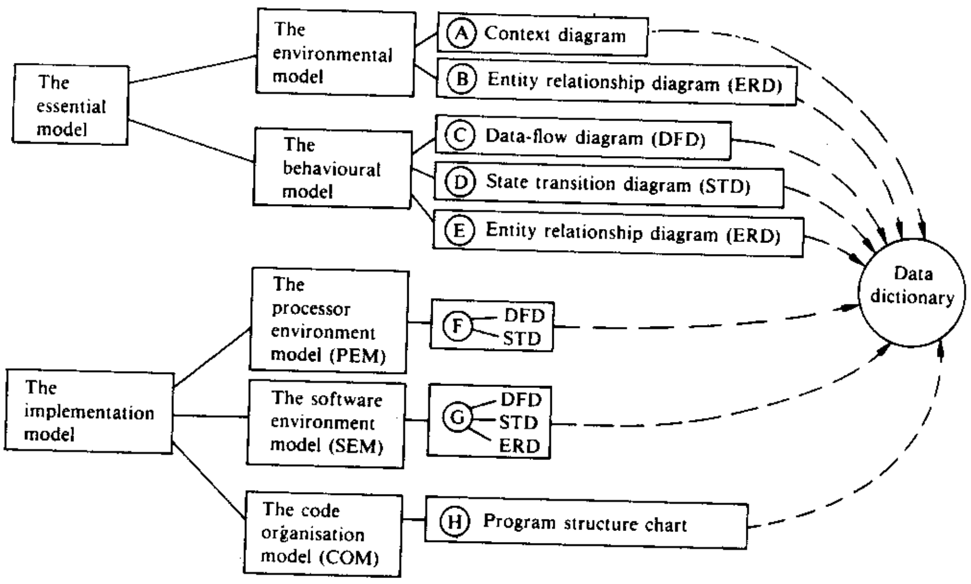

The essential model
The environmental model
The behavioural model
A Context diagram
B Entity relationship diagram (ERD)
C Data-flow diagram (DFD)
D State transition diagram (STD)
E Entity relationship diagram (ERD)
Data dictionary
The implementation model
The processor environment model (PEM)
The software environment model (SEM)
The code organisation model (COM)
F DFD STD
G DFD STD ERD
H Program structure chart

Figure 8.4 Relationship between models and diagrams.

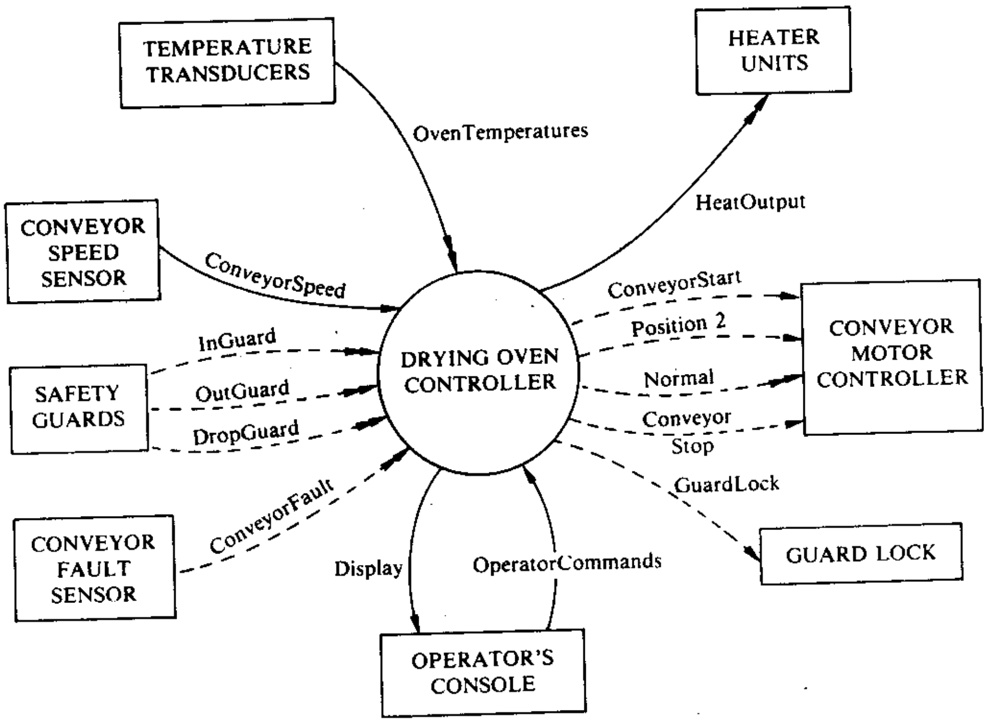


Figure 8.5 Drying oven – context diagram.

The system that we are building is represented as a single *bubble*. The directed lines represent *data* and *control* information that is passing between the system and the environment. The flows are classified as shown in Table 8.2.

Continuous flows are used to represent data or control signals for which there is always a value available to the system. *Discrete* flow is used for data or control which is generated as separate items or transactions and once the system has received or transmitted the item it is no longer available. A loose analogy is that discrete flows are the equivalent of a connection based on a channel and continuous flows with a connection based on a pool. As the context diagram is drawn, all flows should be named and each should be entered into a *data dictionary*. A data dictionary (also

Table 8.2 Flow notations – Ward and Mellor

Data	Continuous	Solid line, double arrow head
	Discrete	Solid line, single arrow head
Control	Continuous	Dotted line, double arrow head
	Discrete	Dotted line, single arrow head

known as a *requirements dictionary*) contains entries describing, in a rigorous manner, all the data elements of the system such that the user and system analyst will have a common understanding of them. Data elements in this sense include control flows as well as data flows and it is also common practice to include in the dictionary descriptions of the activities (see below) that the system performs (hence the use of the name requirements dictionary).

A data dictionary always forms part of a CASE tool. The exact structure can take many forms but must include the following information:

- *Name* – the primary name of the data or control item, or of an external entity, or of a process (activity).
- *Alias* – other names used instead of the primary name.
- *Usage* – where and how it is used, that is a listing of the processes/activities that use it and whether it is an input or output to a process or a store or external entity.
- *Content description* – a description using a standard notation of the content.
- *Other information* – data types, preset values, range of values, and other restrictions.

When using a CASE tool once an item has been entered into the dictionary consistency of naming can be enforced. For example, if a name already exists in the dictionary any attempt to use the name for another flow will be detected and the user will be warned.

An example of a data dictionary is given in Table 8.3. The symbols CD, DC are used to indicate continuous data and discrete control flows respectively and T is used to indicate that the entity is a terminator block. We would need to use symbols to indicate discrete data (DD) and continuous control (CC) as well as data and control stores (DS and CS) and processes (P). The first entry in Table 8.3 shows

Table 8.3 Example of a data dictionary

Name		*Description*
OvenTemperatures	CD	PreHeatTemp + DryingTemp + CoolingTemp *output from TEMPERATURE TRANSDUCERS, input to DRYING OVEN CONTROLLER*
PreHeatTemp	CD	range 0–100°C * measurement of temperature in PreHeat area of oven *
ConveyorStart	DC	range [on/off] * output from DRYING OVEN CONTROLLER, input to CONVEYOR MOTOR CONTROLLER *
GUARD LOCK	T	* external unit controls operation of guard locks *

a compound or group data flow made up of three elements representing the temperature measurement flows from each area of the oven. Group flows are widely used since they provide a concise means of describing the information that is being handled. For example, we could have used a group flow **GuardStatus** to describe the three control flows **InGuard, OutGuard** and **DropGuard** shown in the context diagram (Figure 8.5).

Associated with the context diagram is an event list. This is a table which lists all the events that can cause a change in the system and result in a change in an output. An event list for the drying oven is shown in Table 8.4. Note that the event list shown in Table 8.4 also shows the time response required for the response to the various events including the cycle time. This is not part of the Ward and Mellor requirement but is important and should be added. If possible the type of time constraint – hard or soft – and any tolerances should also be indicated at this stage.

8.4.2 Building the Essential Model – the Behavioural Model

The behavioural model shows how the system should respond to events taking place in the environment. A hierarchical approach to building the model is used. The system is divided into the various functions or activities that it has to perform. These functions are referred to as *transformations* in the Ward and Mellor method and are shown on a *transformation diagram*. Figure 8.6 shows the first-level transformation diagram for the **Drying Oven**.

The bubbles drawn with solid lines represent *data transformations* and those drawn with dotted lines represent *control transformations*. The transformation **ControlAreaTemp** has a double line round part of the bubble which indicates that

Table 8.4 Event list for drying oven

Event	*Action*	*Response*	*Time*
Start	Lock guards	Guardlock	< 0.5 s
InGuard OutGuard DropGuard	Start heat up cycle when heat normal	Set maximum heat output ConveyorStart	< 0.5 s ?
Pause	Stop conveyor	ConveyorStop	< 0.5 s
ConveyorFault	Raise alarm	ConveyorAlarm	< 0.1 s
Stop	Close system down	ConveyorStop Heaters off	< 0.5 s
OvenTemperature	Do control	HeatOutput	Cyclic 1.0 s
ConveyorSpeed	Check for normal	ConveyorAlarm	Cyclic 5.0 s

there are multiple instances of this transformation. It is shown in this way as the temperature control has to be replicated for each of the three areas of the oven. The dotted lines labelled **ENABLE** and **DISABLE** that enter **ControlAreaTemp** are known as *prompts* and indicate whether a particular transformation is *active* (that is, running) or not. Transformations without a prompt attached are assumed to be running all the time the system is running. The two entities placed between parallel

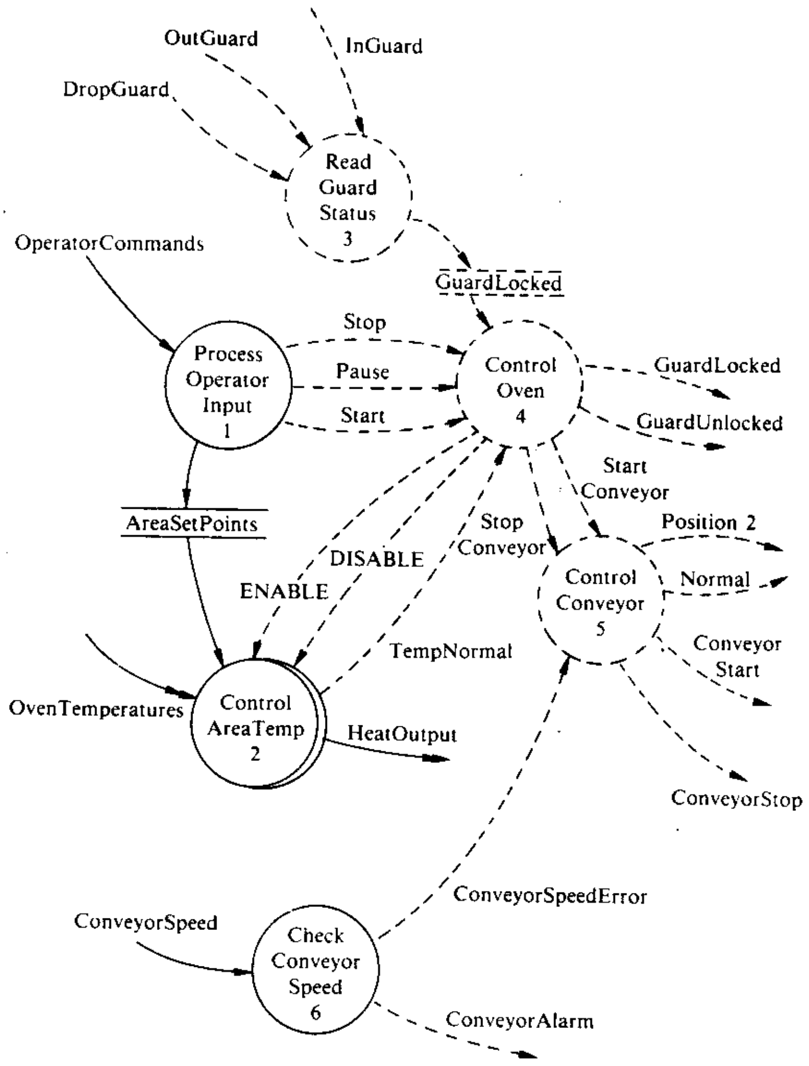


Figure 8.6 Level 1 transformation diagram.

lines and labelled **AreaSetPoints** and **GuardLocked** are respectively a *data store* and a *control store*. Flows entering and leaving stores are not named as they are assumed to take the name of the store. Flows that enter and leave the diagram must appear on the context diagram. Each transformation is given a number, for example **ControlConveyor** is numbered as 5 in the diagram. The single transformation in the context diagram is assigned the number 0 but this is usually not shown. All level 1 transformations have single-digit numbers (this indicates that they are level 1).

Building the model proceeds by taking each transformation in the level 1 diagram and breaking it down into smaller units. For example, Figure 8.7 shows the expansion of **ControlAreaTemp**. This diagram contains one new name, **TRIGGER**. This is a prompt which is used to indicate that the transformation is run once each time the **TRIGGER** becomes true. It is thus a way of indicating that a transformation runs in response to either a periodic signal or an event. You should

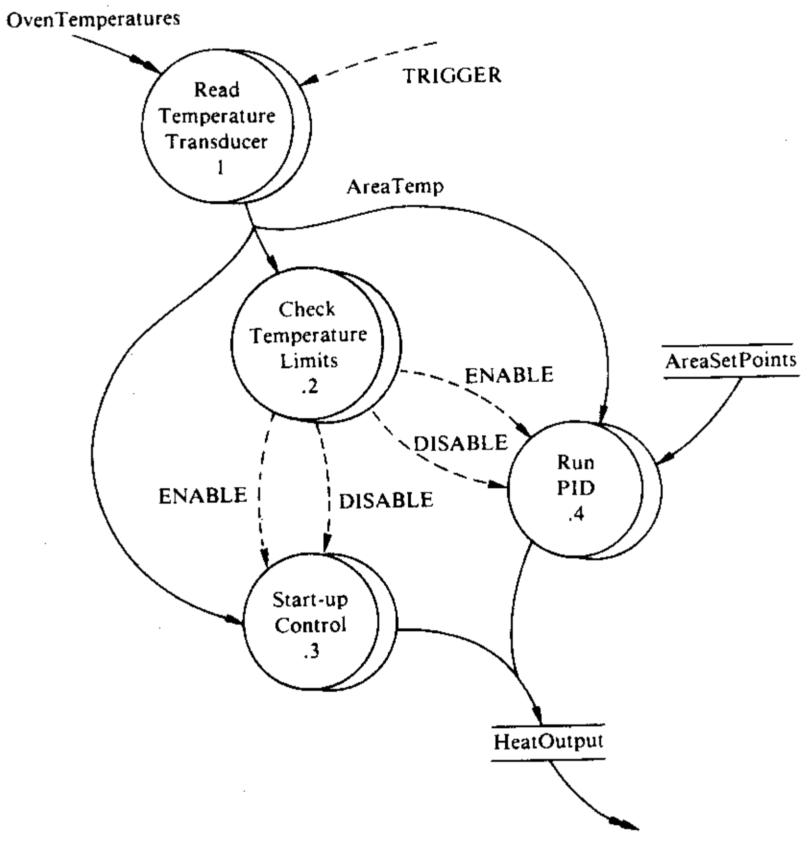


Figure 8.7 Level 2 transformation diagram.

notice that the input to **ReadTemperatureTransducer** is a continuous data flow and the output is a discrete data flow; this is a consequence of the transformation being triggered. Using the data store **HeatOutput** provides the reverse process of converting a discrete data flow into a continuous data flow.

Two other conventions are illustrated in the diagram: a common data flow passing to several transformations – **AreaTemp** – and a data flow being supplied from either of two transformations – **HeatOutput**. (The flow notation is summarised in Figure 8.8.) The transformations on this level 2 diagram (Figure 8.7) are numbered with a full stop in front of the number; this indicates that for full identification the number of the transformation diagram should be added. Thus the transformation **CheckTemperatureLimits** is 2.2 (the presence of two digits indicates that it is a level 2 transformation).

The process of subdivision continues until the analyst decides that no useful purpose is served by splitting up a transformation into smaller units. At this point a transformation specification is drawn up. For data transformations a *process specification* (PSPEC) is produced and for control transformations a *control specification* (CSPEC) is produced. These are described in sections 8.4.4 and 8.4.5 below.

8.4.3 Behavioural Model – Rules and Conventions

There are a number of rules and conventions associated with transformation

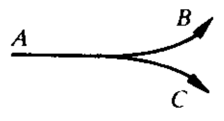
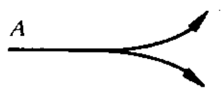
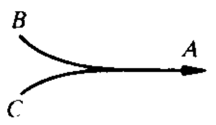
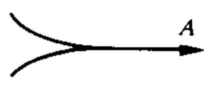
Notation	*Meaning*
	*B* and *C* are two subsets of *A* which pass on to two different transformations
	All of *A* is passed on to two different transformations
	*B* and *C* come from two different transformations and are combined to form a single data flow
	All of *A* may be provided by *either* of two different transformations

Figure 8.8 Summary of data-flow notation.

diagrams and in order to construct and interpret the diagrams these rules must be understood.

1. *Data transformations* with discrete input flows are assumed to be data triggered and hence there can be only one input from another transformation (inputs from data stores and prompts do not count). The reason

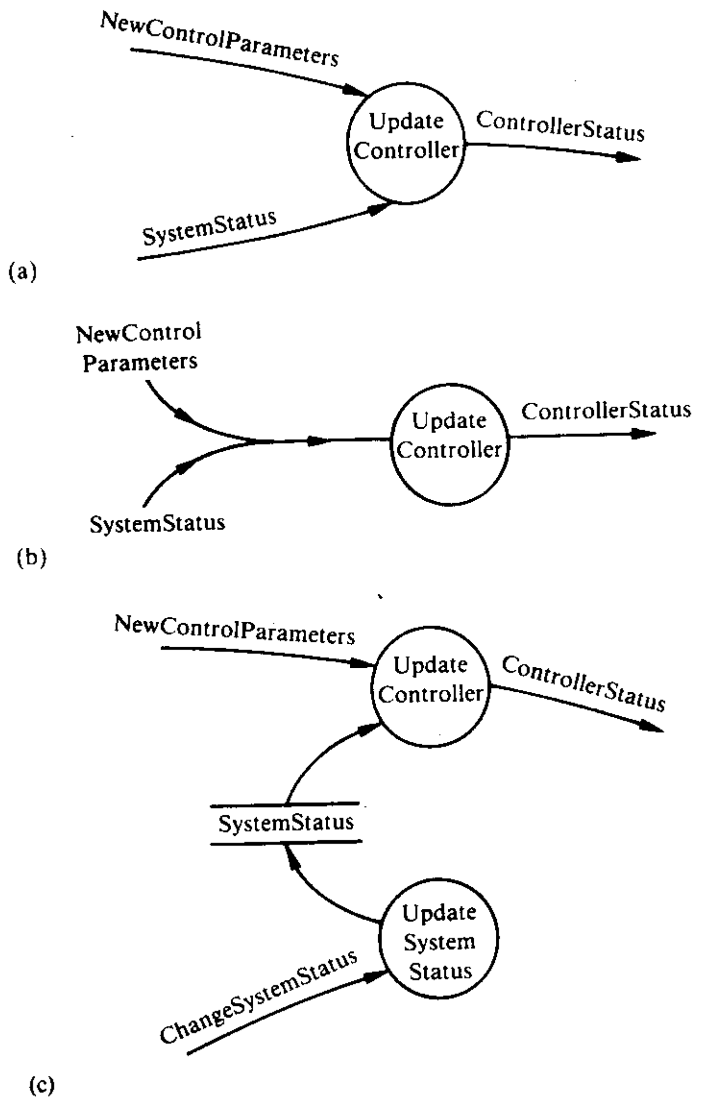


Figure 8.9 Synchronous data flows: (a) ambiguous diagram; (b) merged flows; (c) use of data store.

for this restriction is to avoid ambiguity. Consider the transformation shown in Figure 8.9a: should this be interpreted to mean that the transformation executes only if both inputs are present simultaneously? Or is it to be interpreted that one input will be stored until the other arrives and the transformation executed?

To avoid this ambiguity the transformation must be represented as shown in either Figure 8.9b or 8.9c. Figure 8.9b shows the input as a composite data flow and the convention is that the transformation can only be triggered when all elements of a composite data flow are present. In Figure 8.9c a data store is used to hold one of the data flows. The interpretation of this diagram is that **UpdateSystemStatus** runs when a transaction **ChangeSystemStatus** appears and **SystemStatus** is held in the data store. **UpdateController** runs when a transaction **NewControlParameters** appears and it obtains the current **SystemStatus** from the data store. This implies that the data store will have to have the characteristics of a *pool* since it may be read many times.

There is no need to restrict the number of continuous data flows entering a transformation since by definition a data value is always available; however, it is normal to show only one such input from another transformation and not to mix discrete and continuous data flows to the same transformation.

2. *Control transformations* – there is no restriction on the number of inputs to a control transformation.
3. *Input and output types*: the permitted mixture of input and output flows from the transformations is summarised in Table 8.5.
4. *Balancing* – since transformation diagrams at a given level represent the same information as the diagram at the next higher level (they simply provide more detail) the inputs and outputs must match the inputs and outputs of the higher-level diagram. The process of checking that the inputs and outputs correspond is referred to as balancing. Referring to Figures 8.6 and 8.7 above, the input to TFD 2 is **OvenTemperatures** and the output is **HeatOutput**; thus they balance with the input and output to **ControlAreaTemp** in TFD 0.

Table 8.5 Summary of inputs and outputs for transformations

Transform	*Inputs*	*Outputs*
Data	Data flow Prompt	Data flow Control flow
Control	Control flow Prompt	Control flow Prompt

8.4.4 Process Specifications

The process specification (PSPEC) is a description of the actions that the data transformation has to carry out. The description can be given in any form that the user wishes. Ward and Mellor (1986) discuss a number of methods of specifying data transformations including the standard procedural techniques of program design languages, pseudo-code and structured English. They also discuss and give examples of a non-procedural method based on the use of precondition–postcondition statements (Heniger, 1980). If the goal of producing an essential model devoid of implementation constraints is to be achieved it would seem important that specifications at this stage are expressed non-procedurally. The choice of methodology is open: any of the rapidly developing formal techniques can be used.

At some point in the design the transformation specification will have to be expressed procedurally and often a transformation will be specified using a procedural notation. The most common method is to use some form of *pseudo-code*. Pseudo-code can be thought of as an informal programming language. For example, the PSPEC for **ReadTemperatureTransducer** in Figure 8.7 could be written as follows:

```
PSPEC 2.1 ReadTemperatureTransducer

INPUTS: OvenTemperatures
OUTPUTS: AreaTemp

Every 1.0 seconds DO
    read OvenTemperatures
    convert to internal data representation
    output converted value as AreaTemp
END.
```

Most CASE tools provide a means by which PSPECs can be stored in text files which can be manipulated using a text editor. Many automatically insert in the file the names of the inputs and outputs (obtained from the transformation diagram) of the transformation for which the PSPEC is being created. As development proceeds the PSPEC can gradually be elaborated to form a code segment in the appropriate programming language.

8.4.5 Control Specifications

Control transformations are described using CSPECs. The most usual form for a CSPEC is a *state transition diagram* (STD) and/or a *state transition matrix* (STM). The general form of the STD is shown in Figure 8.10 and of the associated STM in Figure 8.11. The action resulting from an event may be the generation of an event signal (control flow) or the generation of a prompt, either an ENABLE/DISABLE

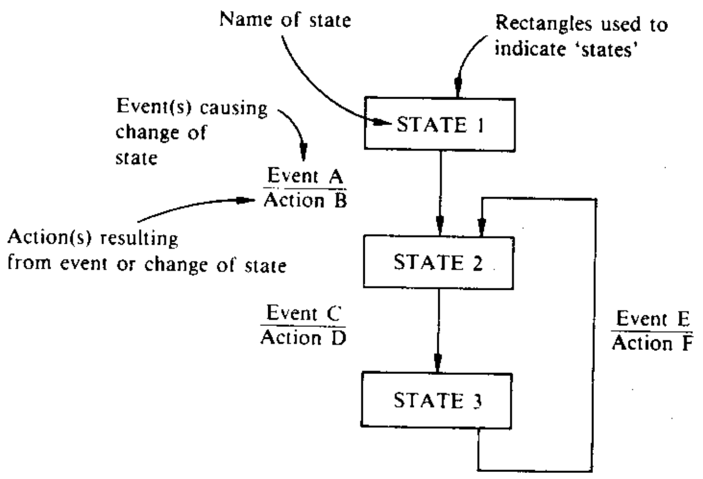

Name of state
Rectangles used to indicate 'states'
Event(s) causing change of state
STATE 1
Event A
Action B
Action(s) resulting from event or change of state
STATE 2
Event C
Action D
STATE 3
Event E
Action F

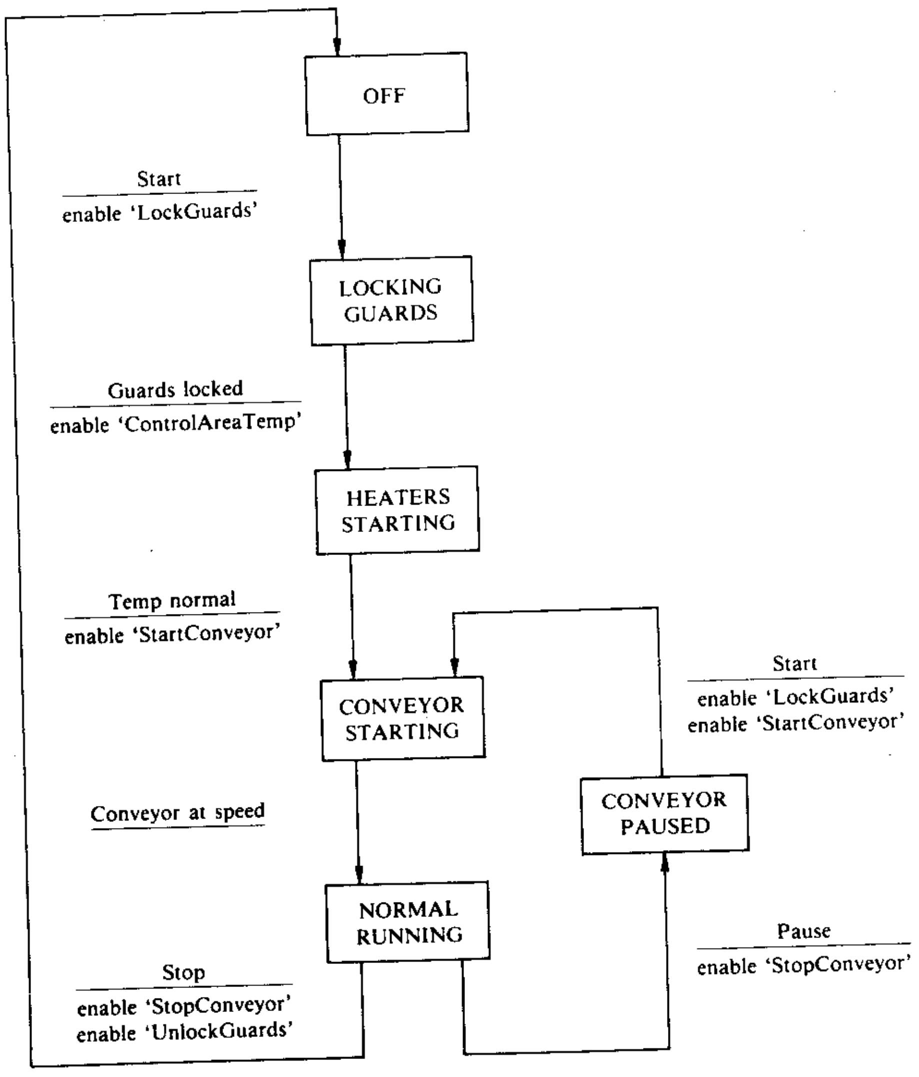
Figure 8.10 State transition diagram.

Present state	Event A	Event C	Event E	
STATE 1	STATE 2 / Action B			Next State / Action
STATE 2		STATE 3 / Action D		Next State / Action
STATE 3			STATE 2 / Action F	Next State / Action

Figure 8.11 State transition table.

or a TRIGGER. An example of a CSPEC is given in Figure 8.12. The associated STM is shown in Figure 8.13.

The blank entries in the STM represent non-operational states or undefined states. The value of using an STD to represent the CSPEC is that it leads to a concentration on the normal behaviour of the system. However, the associated STM should always be drawn up as the STM reveals the undefined states which frequently



CSPEC 4

Figure 8.12 Example of a CSPEC in state transition diagram form.

represent exceptions which the designer must take into account. For example, referring to Figure 8.13 we find that there is only one entry in the column for the event **Stop**. It should be clear that we need to know what to do if the **Stop** occurs when the system is in all other states. What action should be taken if the event **TempNormal** occurs when in state **LockingGuards**? It should be clear that this should not occur and hence it must represent a fault in the system. We will examine

	Start	Stop	Pause	Guards locked	Temp normal	Conveyor at speed
OFF	LOCKING GUARDS – – – lock guards					
LOCKING GUARDS				HEATERS STARTING – – – control area temp		
HEATERS STARTING					CONVEYOR STARTING – – – start conveyor	
CONVEYOR STARTING						NORMAL RUNNING
NORMAL RUNNING		OFF – – – stop conveyor unlock guards	CONVEYOR PAUSED – – – stop conveyor			
CONVEYOR PAUSED	CONVEYOR STARTING – – – lock guards start conveyor					

Figure 8.13 Example of state transition matrix.

an alternative approach to the use of state transition diagrams and state transition matrices in Chapter 10.

8.4.6 Checking the Essential Model

Ward and Mellor recommend checking the transformation schema of the behavioural model in two ways. The first is to use the rules for data flow to check for consistency. This is the equivalent of checking the syntax of a program and can be done by hand or, given the advances in graphics processing capabilities in recent years, it is now feasible to construct a graphics compiler to perform the necessary checks. The second level of checking is to determine whether the model can be executed – can it in some sense generate outputs from a given set of inputs? The approach suggested by Ward and Mellor is based on ideas derived from work on Petri nets. We shall deal with this in Chapter 10.

At this stage the abstract modelling is complete and we are ready to move to the implementation stage.

8.4.7 Building the Implementation Model

The construction of the implementation model divides into four phases:

- enhancing (or elaborating) the environmental model;
- allocation of processors;
- allocation of activities (transformations) to tasks for each processor; and
- definition of the structure of each task.

The latter three can be considered as being concerned with the allocation of resources and will be dealt with together.

8.4.8 Enhancing the Model

Enhancing the model is concerned with:

- clarifying the boundaries between the system and the environment and determining what activities the system – as defined by the behavioural model – will carry out and what will be done as part of the environment;
- elaborating data descriptions; and
- adding timing and process activation information.

Design begins at this stage and we have to begin to take into account the technology involved in the system.

In the context diagram we treated the terminal units as virtual devices which we assumed provided a clean input signal to the behavioural model. We now have to examine each detail to find out what sort of signal they provide and how much processing of that signal is required before it can be passed to the behavioural model. For example, consider the terminal unit TEMPERATURE TRANSDUCERS and take just one element for the **PreHeat** area of the oven. From the data dictionary we find that **PreHeat** is assumed to be provided as a temperature measured in degrees Centigrade in the range 0 to 100 and that it is a continuous data value. From the requirements document we find that the transducer is a thermocouple, the signal of which is amplified and is available as a voltage in the range 0 to 10 volts. We thus have to sample and digitise the thermocouple output, using an analog-to-digital converter, and then convert it to degrees Centigrade. The actions are shown in Figure 8.14, which represents part of the terminator block TEMPERATURE TRANSDUCERS; the rest of the block will consist of the conversion units for the other temperature measurements. In Figure 8.14 we have assumed that a precision of 12 bits will be adequate for the ADC. The data dictionary entry for **PreHeatTemp** will be updated to include information on the resolution of the temperature measurement (1 in 4096).

If we examine Figure 8.7 we find that the TRIGGER prompt does not have any timing information attached to it; as part of the enhancement we would add the comment (* 1 second, cyclic *) to it to indicate that the **ReadTemperatureTransducer** transformation has to be executed every 1 second.

Enhancing the model is a process of adding detail and beginning to take into account the possible technologies that might be used in implementing the system. Ward and Mellor regard the process as largely being concerned with providing an interface shell round the internal software system as shown in Figure 8.15. It separates the functional operations of the interfaces from their electrical and physical manifestations and also serves to hide many of the details of how the

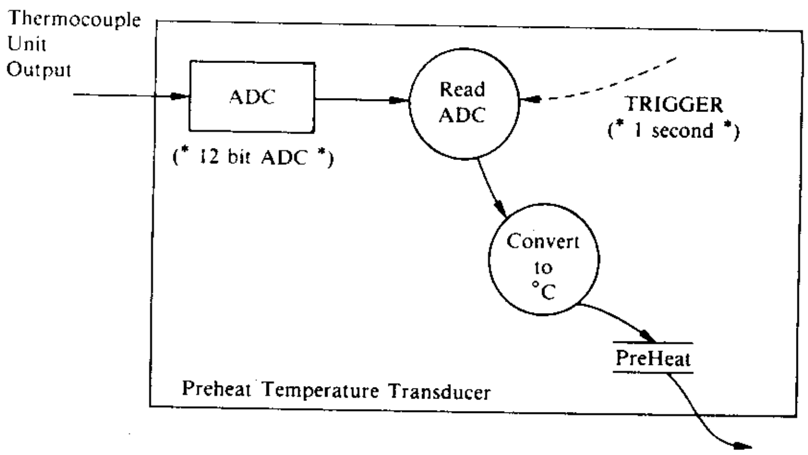


Figure 8.14 Example of a virtual transducer.

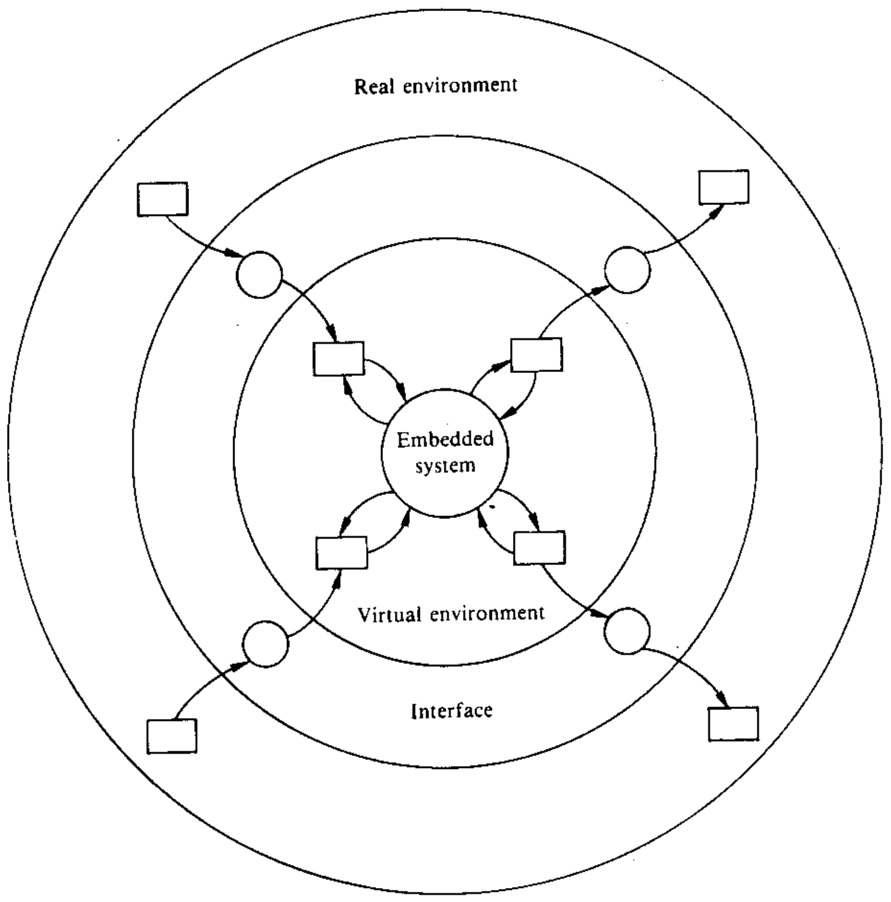


Figure 8.15 Relationship of real environment and virtual environment.

functions are implemented. The implementation may involve both hardware and software. The internal system is seen as communicating with virtual devices; the details of the actual devices are hidden within the interface modules. This approach encourages good design in that non-essential details are hidden and also technology-dependent operations are confined to specific parts of the system and not distributed throughout the software.

8.4.9 Allocation of Resources

The first stage of resource allocation is to decide on how many and what type of processing units are required and how the various functions to be performed are to be allocated to each. Processing units may be digital computers, lógic circuits, analog devices, mechanical devices or human beings. Typically the transformations specified in the behavioural model will be carried out using digital computer elements as the processing units, although some functions may be allocated to human beings. Use of analog devices and hardwired logic systems will usually occur in relation to interfacing to the environment. For example, in order to obtain the temperature measurements for the various oven areas we need to use an ADC and we may need to precede this with an analog filter.

The next stage is to decide on the task structure and the allocation of the tasks to the individual processors. In carrying out this process we need to keep in mind both the standard software engineering design heuristics of:

1. information hiding,
2. coupling and cohesion, and
3. interface minimisation,

and also some additional rules of guidance needed for real-time systems:

1. Separate actions (transformations) into groups according to whether the action is:
   (a) time dependent;
   (b) synchronised;
   (c) independent;
   and try to minimise the size and number of modules containing time-dependent actions.
2. Divide the time-dependent actions into:
   (a) hard time constraint
   (b) soft time constraint
   and try to minimise the size and number of modules with a hard time constraint.
3. Separate actions concerned with the environment from other actions.

The recommended design strategy can be expressed simply as: minimise the part of the system which falls into the category of having a *hard time constraint*.

The simplest processor allocation is to allocate one processor for the whole system. The choice of processor is then based on its ability to perform all the activities, and factors such as processor power, memory size, ability to handle interface devices, and reliability predominate. The more usual case is when it is necessary to distribute units of the essential model across a set of processors. The choice of appropriate processor may be dependent on, for example, the need for an extended instruction set, special memory requirements, or the ability to interface to special devices. The number and type of processors may also depend on the environment, the need or appropriateness of distributed processing, the need for low power consumption, etc. Also at this stage units of the essential model that might be best performed by special purpose hardware or by a human operator will be identified.

The second stage of the design process is to group units of the essential model

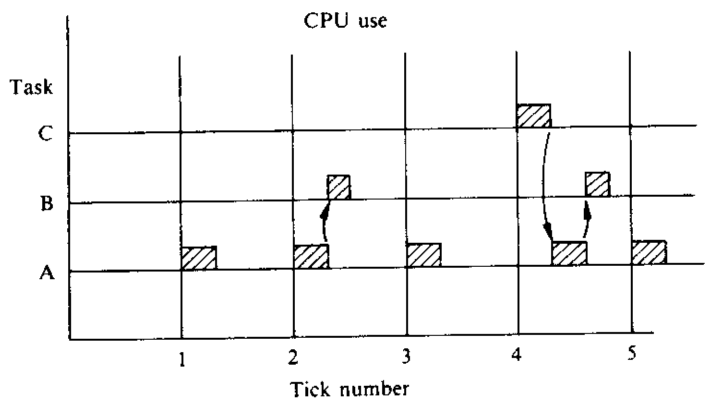


Figure 8.16 Example of a task activation diagram.

that have been allocated to a specific processor into *tasks*. Tasks are concurrent activities (even if run on a single processor); however, units of the essential model when grouped into a single task lose their potential concurrency and hence the grouping of essential model units to form a task introduces a distortion between the essential and implementation models. Also the essential model supports continuous data flows and data transforms; allocation to a task introduces *sampling* and the implementation constraint that a task takes a finite computation time.

Task allocation thus requires analysis of the system to estimate its performance in relation to some of the time constraints given in the specification. A simple way of analysing the timing requirements is to use a task activation diagram. An example of such a diagram is shown in Figure 8.16. The activation diagram shows the use of the CPU by each task that has to run at a fixed cycle time during each clock cycle (tick) of the real-time clock. Using the diagram the effects of task priority and pre-emption strategies are clearly seen and assessed.

8.5 HATLEY AND PIRBHAI METHOD

As might be expected the general approach of the Hatley and Pirbhai methodology is very close to that of Ward and Mellor. There are some differences in terminology which are summarised in Table 8.6.

8.5.1 Requirements Model

The basic structure of the requirements model is shown in Figure 8.17. The major

Table 8.6 Differences between the Ward and Mellor and the Hatley and Pirbhai methodologies

Ward and Mellor	*Hatley and Pirbhai*
Essential model	Requirements model
Implementation model	Architecture model
Transformation schema	Data-flow diagram
	Control flow diagram
Data transformations	Process model
Control transformation	Control model
Data dictionary	Requirements dictionary
	Architecture dictionary

differences between this and the essential model of Ward and Mellor are:

- separate diagrams are used for data and control;
- only one CSPEC can appear at any given CFD level; and
- all data flows and control flows are shown with single arrow heads; the distinction between continuous and discrete flows is determined by the way in which a process is activated. The normal assumption is that a flow is

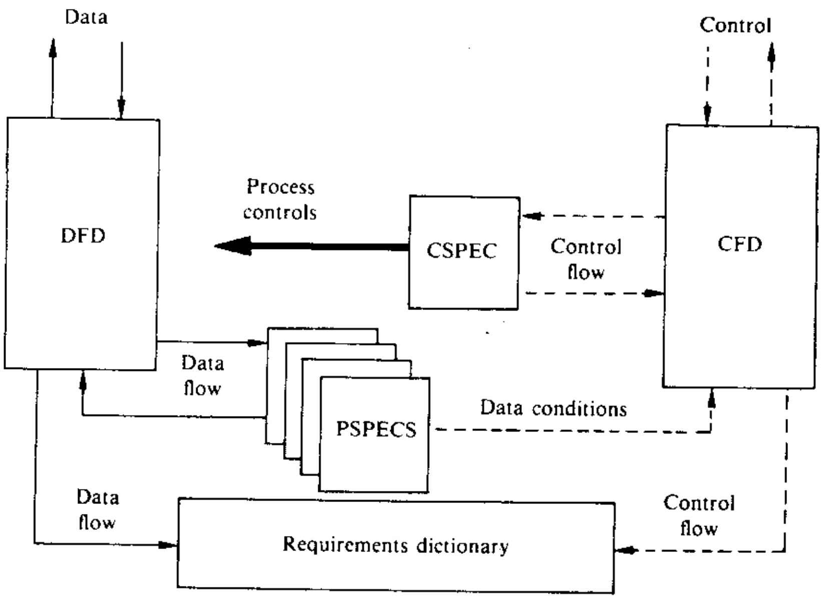


Figure 8.17 The structure of the requirements model. (Redrawn from Hatley and Pirbhai, *Strategies for Real-time System Specification*, Dorset House (1988).)

continuous (it is implicitly assumed that if the activity is implemented on a digital processor it will be carried out frequently enough to appear continuous).

Figure 8.18 shows the Drying Oven Controller in the Hatley and Pirbhai notation. There are several points to note:

1. The process bubbles (transformations) appear on both the DFD and the CFD. This is because, as the CFD shows, a process can produce a control

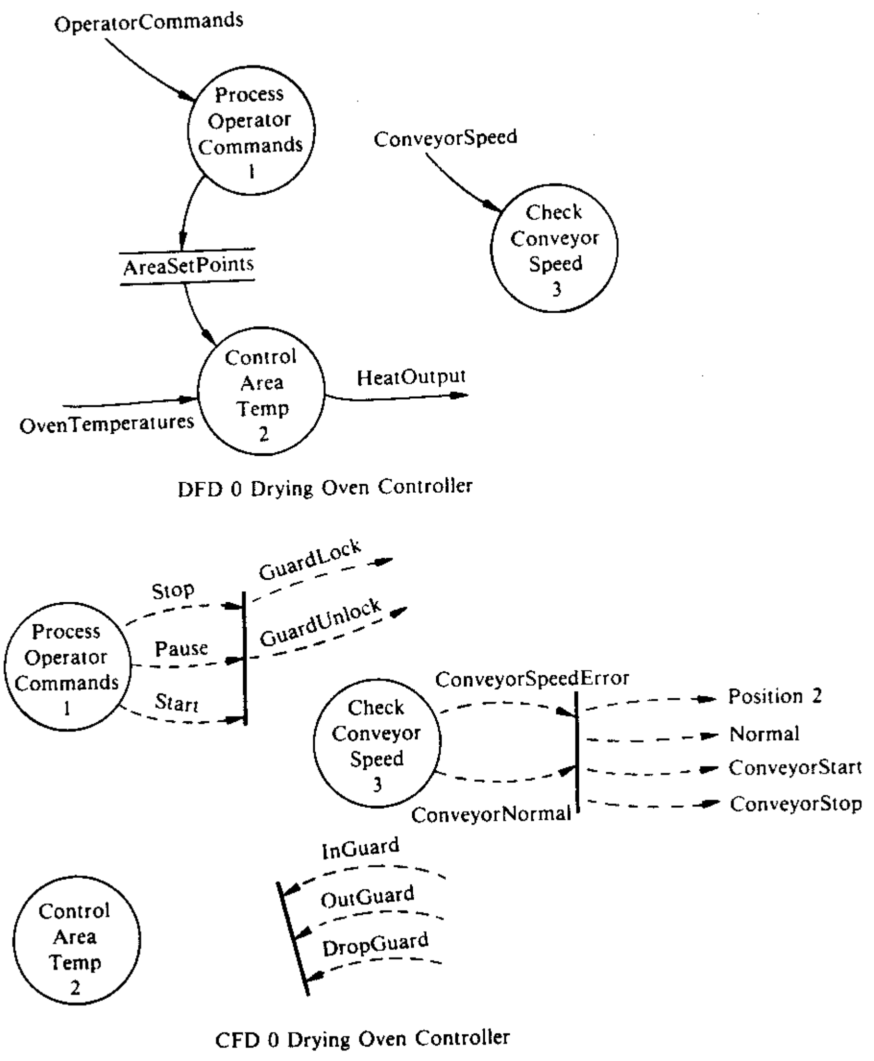


Figure 8.18 Hatley and Pirbhai notation.

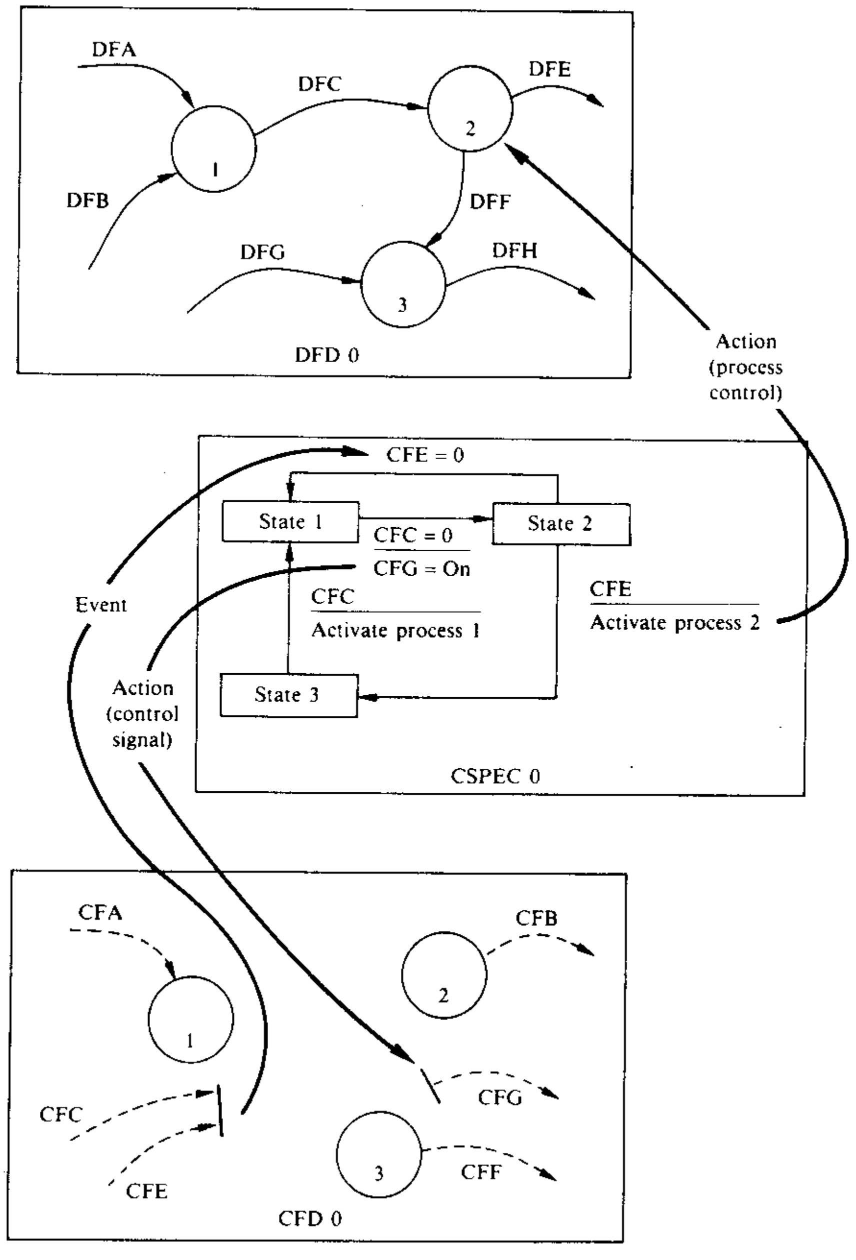


Figure 8.19 Sequential CSPEC, with its DFD and CFD. (Redrawn from Hatley and Pirbhai, *Strategies for Real-time System Specification*, Dorset House (1988).)

flow as an output. It usually arises as a result of some form of comparison which generates an event.

2. The CSPEC is represented by a bar. Although three bars are shown they form one CSPEC and, because there is only one, it does not need to be named on the diagram – it takes the number and name of the diagram. In this case it is numbered CSPEC 0.
3. There is no process activation information shown in the diagram. The process activation information is held in the CSPEC. The relationship between CSPECs and the DFD and CFD diagrams is shown in Figure 8.19.

8.5.2 Architecture Model

The general structure of the architecture model is shown in Figure 8.20 and as with the requirements model it is a hierarchical layered structure. In developing the architecture model a procedure based on using an *architecture template* is suggested. Figure 8.21 shows the form of the template. It is akin to the Ward and Mellor method for enhancing the model by developing virtual terminators but in this case it is suggested that the template be applied at each level in the requirements model hierarchy. The architecture model also includes diagrams showing the

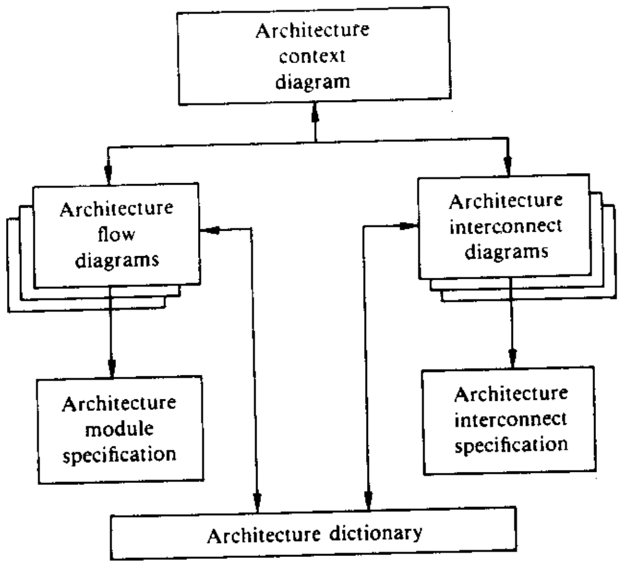

Architecture context diagram
Architecture flow diagrams
Architecture interconnect diagrams
Architecture module specification
Architecture interconnect specification
Architecture dictionary

Figure 8.20 Architecture model components. (Redrawn from Hatley and Pirbhai, *Strategies for Real-time System Specification*, Dorset House (1988).)

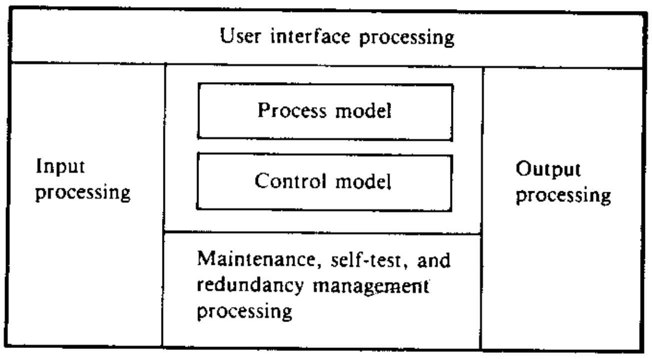

User interface processing
Input processing
Process model
Control model
Maintenance, self-test, and redundancy management processing
Output processing

Figure 8.21 Architecture template. (Redrawn from Hatley and Pirbhai, *Strategies for Real-time System Specification*, Dorset House (1988).)

interconnection technology between various elements of the system. Figure 8.22 shows in more detail the style of the architecture model elements.

8.6 COMMENTS ON THE YOURDON METHODOLOGIES

Both methodologies – Ward and Mellor and Hatley and Pirbhai – are simple to learn and have been widely used. They are founded on the well-established structured methods developed by the Yourdon organisation and hence over the years a lot of experience in using the techniques has been gained. For serious use on large-scale systems they both require the support of CASE tools. The labour involved in checking the models by hand is such that short cuts are likely to be taken and mistakes are bound to occur.

It can be argued that the methods are really only a set of procedures for documenting a specification and a design and to some extent this is true. The analysis procedures are minimal and adequate checking for consistency can be performed only with the support of a CASE tool. However, the methodologies are still useful in that the procedures they recommend provide a sensible way of preparing both a specification and a design in that they encourage the development of hierarchical, modular structures.

Of the two, the Hatley and Pirbhai method is the more structured and formalised in its approach. Its diagrams are less cluttered than those of the Ward and Mellor method and, once the separation is understood, are easier to follow. Many CASE tools provide alternative displays which allow a choice of either separate diagrams or a combined diagram with switching between the two forms.

The weakness of both methods lies in the allocation of processors and tasks. The suggestion that one allocates activities to processors and then subdivides the

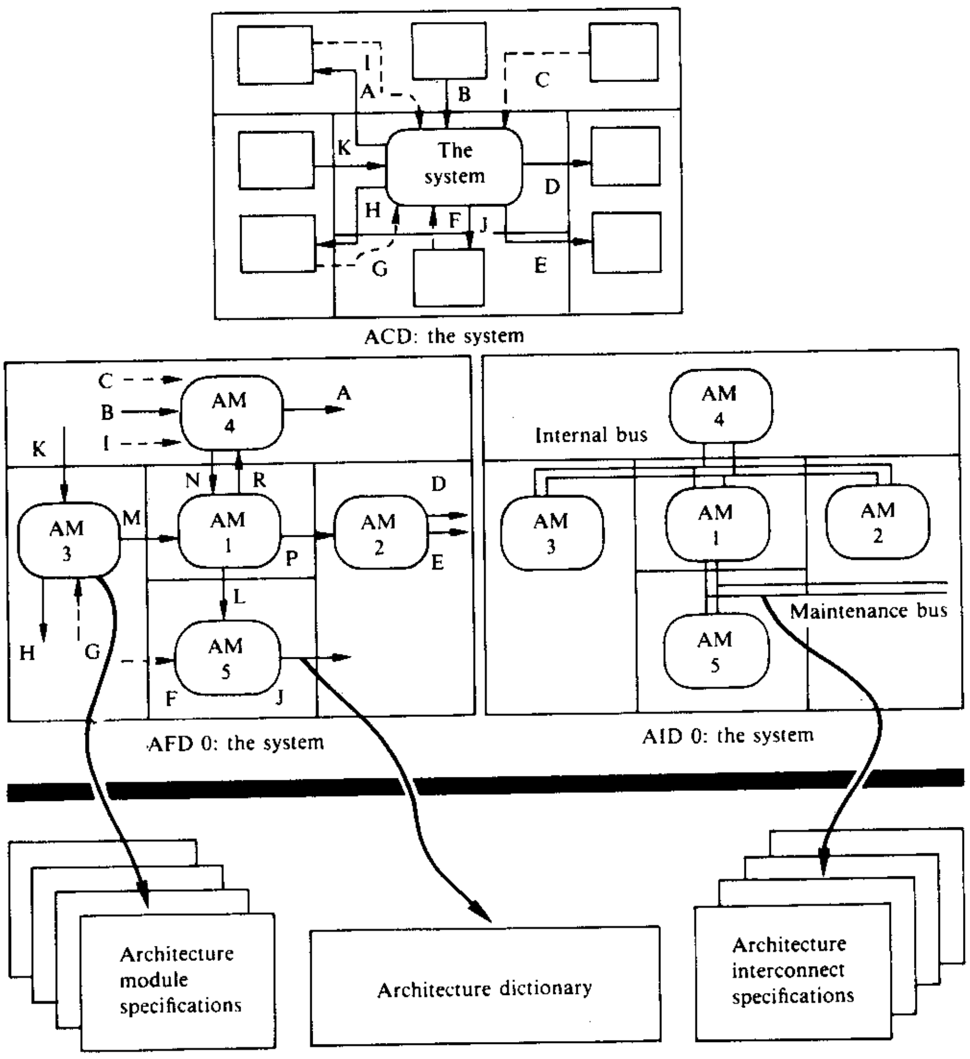


Figure 8.22 The structure of the architectural model. (Redrawn from Hatley and Pirbhai, *Strategies for Real-time System Specification*, Dorset House (1988).)

activities into tasks allocated to each processor appears at first sight a sensible way to proceed. However, when it is tried one soon realises that the information required to do this is not available. How can one determine the processor requirements until at least some detailed coding has been done? How can tasks be structured until some estimate of the feasibility of finding a suitable task schedule has been carried out? Both Ward and Mellor and Hatley and Pirbhai remain silent about these issues.

8.7 SUMMARY

The structured methodologies on which both Ward and Mellor and Hatley and Pirbhai are based are a widely used method of producing a requirements model. For serious use, however, the support of a CASE tool is essential. It is only through the use of a CASE tool that consistency, correctness and full adherence to the standards can be maintained.

The extension of the method to attempt to support design through the development of an implementation or architecture model is less successful. Of the two, the Hatley and Pirbhai approach is most well developed and useful. The lack of analysis tools is a weakness but this will eventually be remedied through the development of such tools within CASE environments. For a full understanding of the methods the books by Ward and Mellor (1986) and by Hatley and Pirbhai (1988) must be consulted.