9

Real-time System Development Methodologies – 2

In this chapter we will consider two further methodologies – MASCOT and PAISLey. MASCOT is one of the older methodologies but it has recently been extensively revised. It is interesting in that it assumes the system is being designed for a particular virtual machine and that the implementation of this virtual machine on a specific computer or set of computers is a separate problem. Hence at the design stage there is no need for consideration of specific technologies.

PAISLey is primarily a specification technique which is based on formal mathematics. Its most interesting features are: a means of specifying the timing constraints of the system; and execution of the specification.

9.1 MASCOT

MASCOT was the first formal real-time software development methodology. The first version of MASCOT was developed by Jackson and Simpson during the period 1971–5 (Jackson and Simpson, 1975). The official definition of MASCOT 1 was published in 1978 and a revised version – MASCOT 2 – was issued in 1983. Between 1983 and 1987 extensive changes to the technique were made and the official standard for MASCOT 3 was published in 1987.

The official handbook states that MASCOT is a Modular Approach to Software Construction Operation and Test which incorporates:

- a means of design representation;
- a method of deriving the design;
- a way of constructing software so that it is consistent with the design;
- a means of executing the constructed software so that the design structure remains visible at run-time; and
- facilities for testing the software in terms of the design structure.

9.2 BASIC FEATURES OF MASCOT

The application software is designed for a specific virtual machine and the problem of mapping the MASCOT machine onto a real computer is treated as a separate problem. In MASCOT software is represented as:

- a set of concurrent operations; and
- the flow of data between such operations.

The operations are referred to as *components*. The system consists of a set of interconnected but independent *components* that make no direct reference to each other. Each *component* has specific, user-defined, characteristics that determine how it can be connected to other *components*. *Components* are created from *templates*, that is patterns used to define the structure of the *component*. Two *classes* of *templates* are fundamental to MASCOT: (a) *activity* and (b) *intercommunication data area – IDA*.

An *activity template* is used to create one or more *activity components* each of which is a single sequential program thread that can be independently scheduled. It is assumed that at the implementation stage each *activity* will be mapped onto a software task. Such a task may run on its own processor or be scheduled by a run-time system (usually referred to as the MASCOT kernel) to run on a processor shared with other *activities*. The *activities* communicate through *IDAs*. The *IDA* provides the necessary synchronisation and mutual exclusion facilities.

An *IDA* is a passive element with the sole purpose of servicing the data communication needs of *activity components*. It can contain its own private data areas. It provides procedures which *activities* use for the transfer of data. Within an *IDA*, and only within an *IDA*, the designer has access to low-level synchronisation procedures and thus is not limited to using high-level operations such as monitors, message passing, or rendezvous, provided by the implementation language, but is able to use any technique appropriate to the problem. A structure containing *activity components* connected by means of one or more *IDAs* is referred to as a *network*.

MASCOT supports three forms of *IDA*: a generalised *IDA*; a *channel*; and a *pool* – their behaviour is defined as follows:

channel: supports communication between producers and consumers. It can contain one or more items of information. Writing to a *channel* adds an item without changing items already in it. The read operation is destructive – it removes an item from the *channel*. A *channel* can become empty and also, because its capacity is finite, it can become full.

pool: is typically used to represent a table or dictionary which *activities* periodically consult or update. The write operation on a *pool* is destructive and the read operation is non-destructive.

9.2.1 Simple Example

MASCOT can be used at a simple level to provide a virtual machine supporting *activities*, *pools* and *channels*. A design is constructed in the form of an *activity, pool* and *channel network* – an ACP diagram – as is shown in Figure 9.1.

The diagram represents part of a system for the control of a plant. The activity **Heater1Input** gets data from a plant interface. The data is held in a pool **Heater1In** from where it is read by activity **Heater1Alarm** and **Heater1Con**. The required output to the plant and the alarm status are held in a pool **Heater1Status**. An activity **Heater1Report** gets data from the pool holding status information and sends it via a channel **Heater1Ch** to some other activity (not shown). Also not shown are the activities required to pass the data to the plant control. This ACP differs from a MASCOT 2 ACP since the components now contain *ports* and *windows* (shown as solid circles and rectangles) the significance of which is explained below.

Once the ACP diagram has been produced, design of the templates for the

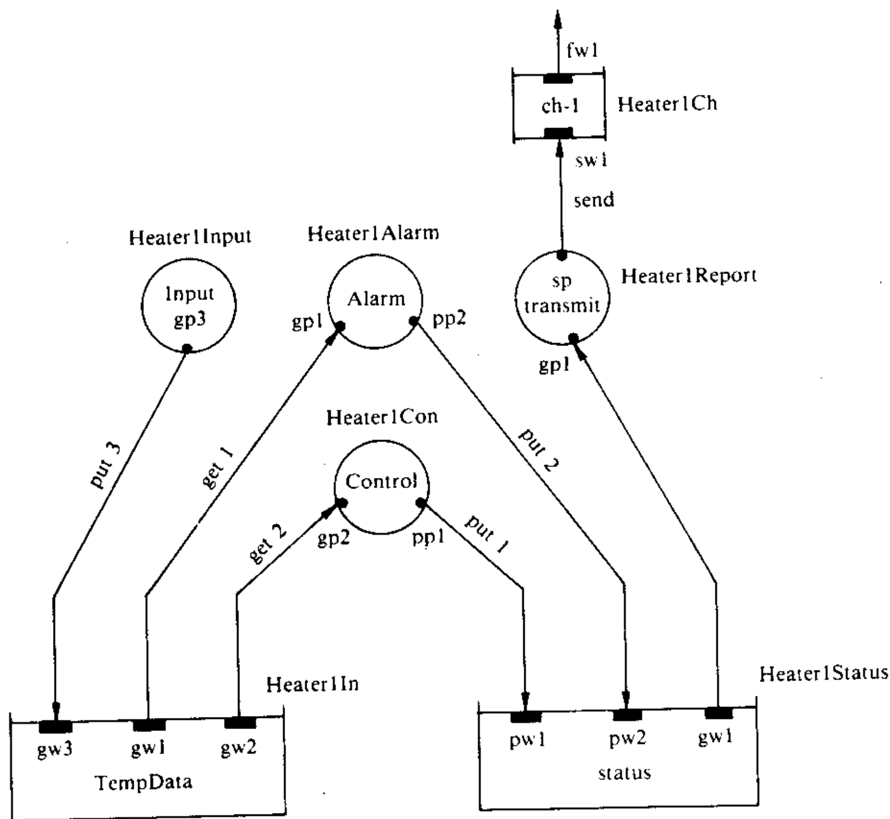

Figure 9.1 An example of a MASCOT ACP diagram.

individual components can proceed. Many component templates will be reusable and hence only application-specific ones will need to be designed. Instances of the component are created when the network is constructed by translating the ACP diagram to textual form and entering it into the MASCOT database.

At this level a design in MASCOT may be represented in either graphical (ACP diagram) or textual form. Both forms are equivalent and may be derived from each other. The textual form stored in the database can be progressively updated as the design proceeds.

9.2.2 Communication Methods

Entities in MASCOT communicate by means of *paths*. A path connection is made between a port and a window. A port is represented by a solid circle and a window by a solid rectangle. Thus in Figure 9.1 activity **Input** has a port **gp3** and is connected by a path labelled **put 3** to a window **gw3** in an IDA **TempData**. In path **put 3** the port is the source of the data and the window is the sink. However, a port can act as a sink and a window as a source as is the case with path **get 1**.

Windows are *passive* devices which provide a set of operations for use by an active device for the transmission of data. Ports are *active* devices which specify a set of operations required to transmit data. Windows are normally found in communication components whereas ports are found in both activities and communication components. (The use of ports in communication components enables IDAs to be connected together.)

9.3 GENERAL DESIGN APPROACH

The general approach to the design of the system is hierarchical:

1. Define system and external devices.
2. Decompose system into a network of concurrent subsystems, IDAs and hardware interface units – *servers*.
3. Continue decomposition of subsystems until further decomposition is not desirable. At this point – the component level – a particular subsystem will be composed of activities and IDAs.

Figure 9.2 shows the top-level network diagram for the Drying Oven which was used as an example in the previous chapter. Two subsystems – **ControlAreaTemp** and **GeneralOvenControl** – are used and they connect to the external devices through servers. The connection to the **Temperature transducer** is through the server **TT** which uses a path **fetch** to connect from window **TT.W** in the server to port **P1** in the subsystem.

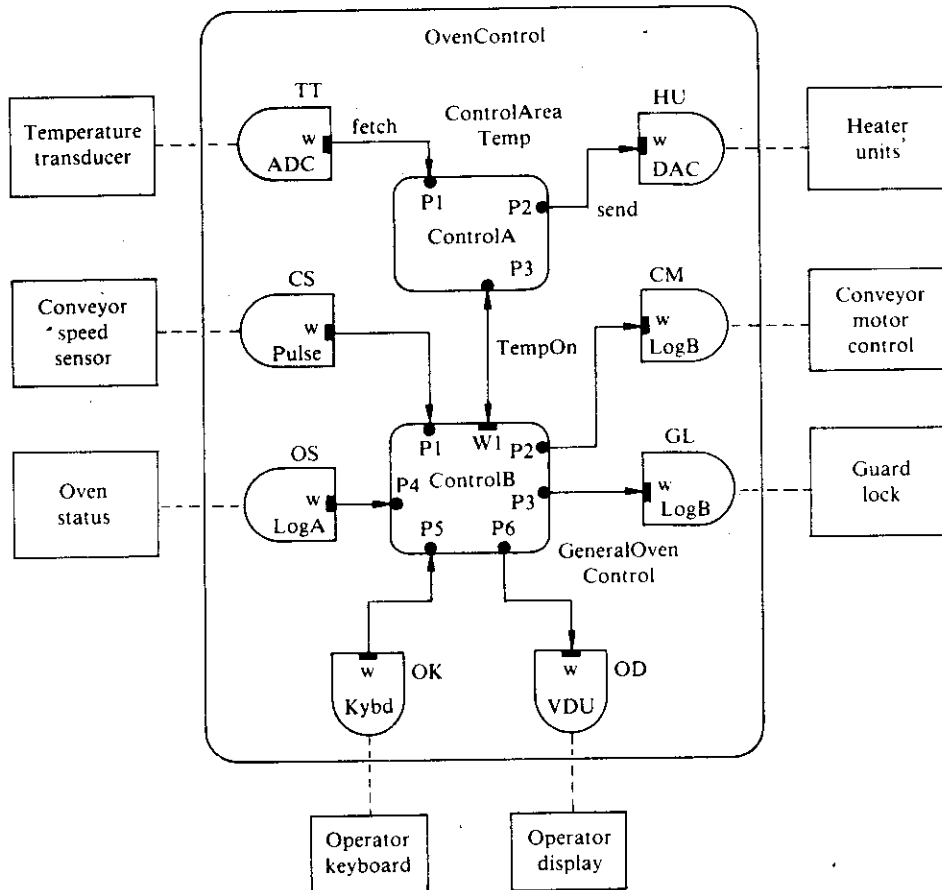

Figure 9.2 Drying oven – MASCOT network level.

Figure 9.3 shows subsystem **ControlAreaTemp** decomposed to the component level. There are two activities **Act1** and **Act2** which are named **ReadTemp** and **Control** respectively. The names inside the circles are the template names and those outside are the component names. A component is an instantiation of a template. **ReadTemp** is assumed to read the value **ControlOn** by using the access procedure **get** and if it is true it reads **Temp** and uses **send** to store the value in a pool **AreaTemp**. **Control** gets the value of the temperature from **AreaTemp** and calculates the heat output.

An activity represents a task and cannot be subdivided into smaller, separately schedulable units (they would themselves be activities) but it can be divided into smaller modules as shown in Figure 9.4. One module within the activity is defined as the *root* component, that is the main module, and it provides coding for the initial

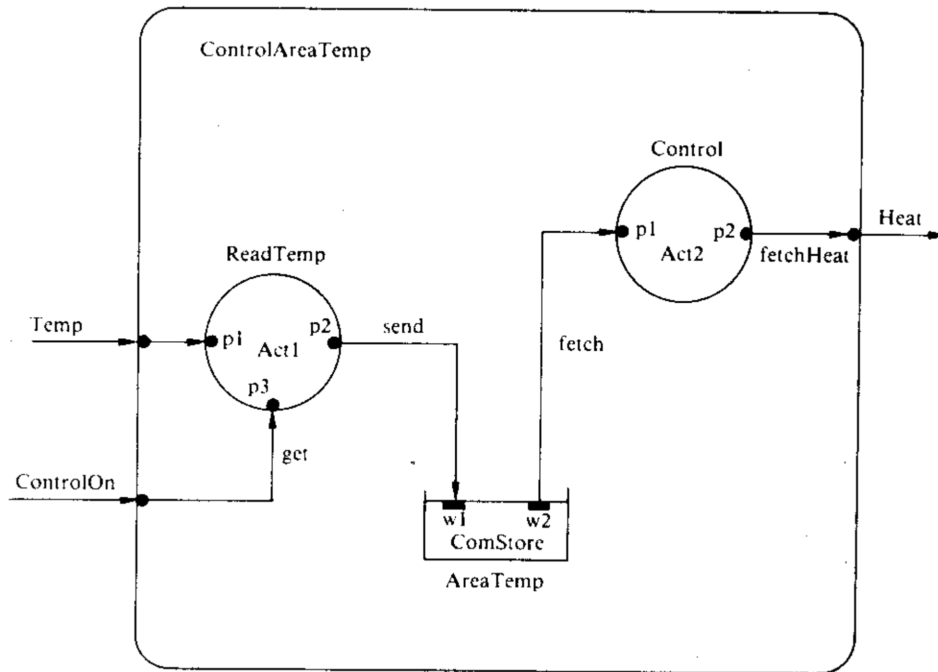

Figure 9.3 Subsystem – **ControlAreaTemp**.

entry and calls on the services provided by the other units which are known as *subroots*.

Subroots usually comprise a collection of procedures and can be designed using the standard structured methods. Thus in the example shown in Figure 9.4 the main root **M** calls **CheckTemp** which gets the **AreaTemp** from the pool (see Figure 9.3) and checks to see if it is within the normal range. If it is not within the normal range **MaxControl** (subroot **SR3**) is called, otherwise **PIDControl** (subroot **SR2**) is called. Finally subroot **SR4** which outputs the heat demand is called.

There are two general comments to make about this example. One is that the design procedures in MASCOT assume that a general set of templates will be created and used; thus all entities are created from generic types by a process of instantiation, and hence even if only one instance is required a template has to be created and then instantiated. This is in fact less cumbersome than it seems. The advantages are obvious even in this small system: subsystem **ControlAreaTemp** is a template and three instances can be created to deal with the **PreHeat, Drying** and **Cooling** areas of the oven. One of the purposes of this approach is to encourage the reuse of software.

The second comment is that although the subsystem **ControlAreaTemp** contains two activities that are run at periodic intervals this is not shown on the design

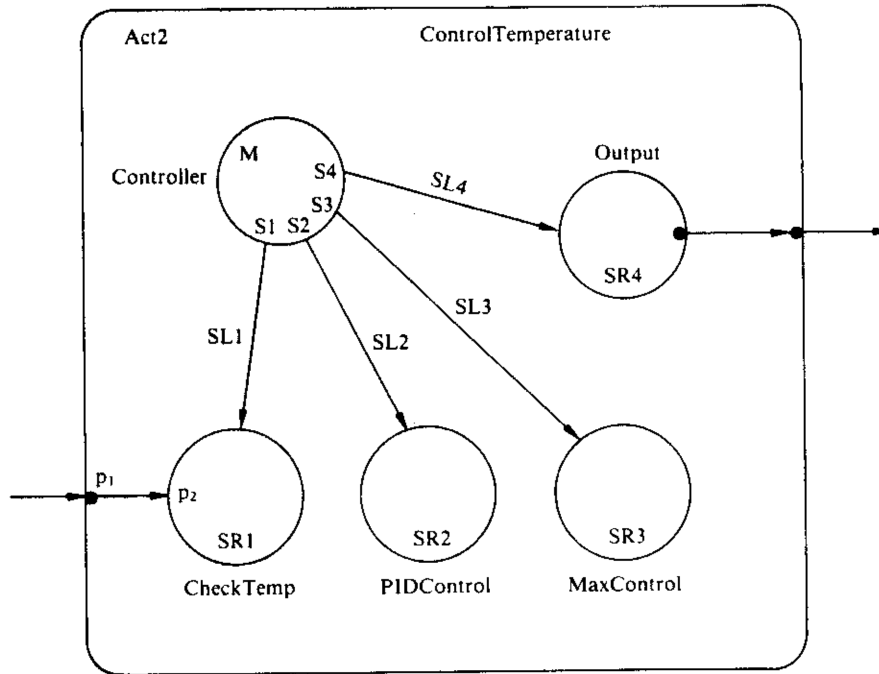

Figure 9.4 Diagram showing activities in **ControlTemperature**.

diagrams. The scheduling support is provided by the MASCOT kernel and the information is entered when the activities are made known to the kernel.

9.4 TEXTUAL REPRESENTATIONS OF MASCOT DESIGNS

In MASCOT there is a direct correspondence between the diagrams and their textual representation; in fact the text is the formal description of the system, not the diagrams. The description of the system shown in Figure 9.2 takes the form

```
SYSTEM OvenControl;
    {specification part goes here}
USES    ControlA, ControlB, ADC, DAC, Pulse,
        LogA, LogB, Kybd, VDU;
    SUBSYSTEM ControlAreaTemp:ControlA(p1=TT.w,
        p2=HU.w, p3=ControlB.w1);
    SUBSYSTEM GeneralControl:ControlB(p1=CS.w,
        p2=CM.w,p3=GL.w, p4=OS.w, p5=OK.w, p6=OD.w);
    END;
END.
```

The words in upper case are MASCOT key words. Each textual unit or module follows a similar pattern: it has associated with it an explicit class (in this case SYSTEM); a name part (OvenControl); a specification part (in this case empty); and, for a template, an implementation part (in this case the statements between USES and END). The USES statement lists the template names used to form components and is immediately followed by a list of the components, each preceded by its generic form (in this case SUBSYSTEM) and followed by its template name and, in parentheses, a list of connectors.

The lines connecting subsystems to the servers are paths. They represent data flow between a port of one component and the window of another. The procedures read and write would be available at windows **TT.w** and **HU.w** respectively. The coding for the two procedures would be included in the module for the servers **ADC** and **DAC** respectively. The outline for server **ADC TT** takes the form:

```
SERVER TT;
     PROVIDES w:  fetch;

     ACCESS PROCEDURE read (VAR item:PlantData);
     {body of procedure }
     END;
END.
```

A server is the only MASCOT design entity which is permitted to communicate with a device. It has all the features of an IDA but also is allowed to contain one or more handlers which can be invoked by hardware interrupts. It thus provides the means for low-level direct communication with the system hardware.

The identifier used to label a path indicates its type which is defined in a module called an *access interface*. Consider the subsystem **ControlAreaTemp** shown in Figure 9.3: the path connecting port **p1** in activity **Control** to the window **w2** in IDA **AreaTemp** is of type **fetch** and the path connecting port **ReadTemp.p2** to window **AreaTemp.w1** is of type **send**. The modules defining them take the form

```
ACCESS INTERFACE send;
     WITH PlantData;
     PROCEDURE write(item: PlantData);
END.

ACCESS INTERFACE fetch;
     WITH PlantData;
     PROCEDURE read(VAR item:PlantData);
END.
```

The WITH clause used in the specification indicates that the type definition PlantData is held in a common source accessible to all modules. The declaration

is made in a specification statement, for example

```
DEFINITION PlantData;
    TYPE
        PlantData = RECORD
            {put definition here}
        END {RECORD};
END.
```

The textual description of the composite activity shown in Figure 9.3 takes the form

```
ACTIVITY ControlTemp;
    {specification part}
    REQUIRES    p1:fetch;
                p2:send;
    USES M, SR1, SR2, SR3, SR4;
    ROOT Controller: M
    SUBROOT CheckTemp: SR1(p1=p1);
    SUBROOT PIDControl: SR2;
    SUBROOT MaxControl: SR3;
    SUBROOT Output: SR4(p1=p2);
END.
```

The templates for the root and subroots are defined as follows:

```
ROOT main;
    NEEDS  s1:sl1;
           s2:sl2;
           s3:sl3;
           s4:sl4;
    {code goes here}
END.
```

The `NEEDS` section specifies the links connecting the root module to the other components. As shown below the corresponding subroot will name the links in a `GIVES` statement. For example the subroot `SR1` takes the form

```
SUBROOT SR1;
    REQUIRES p1:fetch;
    GIVES sl1;
    {coding goes here}
END.
```

9.5 OTHER FEATURES OF MASCOT

9.5.1 Constants

The usefulness of the template method of creating components is enhanced by the facility to create from the same template components which differ in minor ways. In the specification of a template dummy constants are declared; their actual value is supplied when a component is created. The dummy constants are known as template constants. They can be considered to be the equivalent of dummy arguments in a macro declaration. For example, they permit servers with different device addresses and different interrupt levels to be created from a single template; or they can enable components with different buffer sizes or different iteration counts to be created.

9.5.2 Direct Data Visibility

In most real-time applications certain functions cannot be satisfactorily performed if the designer is restricted to using the data hiding approach provided by the IDA construct of MASCOT. A typical example is a module providing direct feedback control subjected to a hard time constraint. If a module of this type requires access to external data, for example in order to update controller parameters, it must have guaranteed access at all times and must not be kept waiting because another module is accessing the data. There are a variety of solutions to this problem; one is to allow the module to access the data directly without using the standard access procedures. MASCOT provides the designer with a means of providing direct access through a construct called an access interface.

9.5.3 Qualifiers

Software design techniques and implementation languages impose certain general constraints on what can and cannot be done. However, in most software systems there are areas in which the designer would wish either to relax the constraints or to impose locally more stringent constraints. For these purposes MASCOT provides a set of qualifiers which can be used to modify the behaviour of parts of the system.

1. *Connectivity constraints*: in the default mode windows are open to one or more ports; qualifiers can be used to restrict access to a single port, or to allow a window to exist without a port connection (used normally for test purposes).
2. *Data access constraints*: can be used to limit variables made directly accessible via an access interface.
3. *Data flow*: permits the direction of data flow to be shown in the textual form of the design.

4. *Context qualifiers*: provide a means of restricting certain functions provided by the support environment to particular types of template, for example certain low-level functions may be restricted to servers.
5. *Code generation constraints*: allow the designer to force the compiler to generate in-line code for an access procedure to a data area and hence avoid the overheads in making a procedural call.

9.6 DEVELOPMENT FACILITIES

As has already been mentioned the MASCOT design is captured by entering data relating to the design in textual form into a database. The method of constructing textual modules and the database have been designed so as to enable a design to be built up incrementally. As information is added, either in the form of a new module, or as an addition to an existing module, checks are made on the validity of the information. For example, the first stage is to register a module and for this process to be successful the name part must be defined and legal, and no module with the same name must have been previously registered. The various stages are listed in Table 9.1.

Table 9.1 MASCOT status conditions (reproduced from *The Official Handbook of MASCOT*)

Operation	*Status to be achieved*	*Module class*	*Preconditions*
Register	Registered	All	Name part defined and legal No other module with same name
Introduce	Partially introduced	All	Registered preconditions satisfied Specification dependencies registered Specification part defined and legal
	Fully introduced	All	Partially introduced preconditions satisfied Specification dependencies fully introduced
Enrol	Partially enrolled	Composite templates	Partially introduced preconditions satisfied Implementation dependencies introduced Implementation part defined and legal
	Fully enrolled	Simple template	Fully introduced preconditions satisfied Implementation dependencies fully introduced Implementation part defined and legal
		Composite templates	Partially enrolled preconditions satisfied Implementation dependencies fully enrolled

The progress status of each module in the design can be listed. The database thus provides management and designers with a support environment for project development.

9.7 THE MASCOT KERNEL

The MASCOT design procedures are based on the assumption that the design will be implemented using the features provided by a piece of software known as a MASCOT kernel. The implementation of this software on a particular computer using a particular operating system is considered to be a separate issue outside the application development. It is useful for an understanding of the MASCOT methodology to be aware of the main features of the kernel, which represents a *virtual* machine on which the MASCOT application will run.

Scheduling: the kernel must allocate processor time to the parallel activities that constitute a MASCOT system. It must provide a real-time clock and must support primitive synchronisation procedures. The synchronisation procedures are precisely defined and are similar to semaphores and signals.

Interrupt handling: the kernel must support the handling of hardware interrupts.

Subsystem control: a MASCOT design may include groups of activities (tasks) which form a subsystem. The kernel must provide a means of adding and removing such subsystems from the attention of the scheduler.

Monitoring: the kernel must provide a comprehensive set of facilities to aid testing and optimisation of the implementation.

In MASCOT 2 the exact structure of the support required was mandatory. The specific schemes for synchronisation, device handling, interrupts, process scheduling and priorities are analysed and a comparison with alternatives is given in Sears and Middleditch (1985). If the language being used in the implementation supports concurrency then the designer should consider mapping *activities* onto the appropriate language feature. Budgen (1985) describes a Modula-2 implementation of the MASCOT kernel. MASCOT 2 imposes one restriction: *activities* should not be created dynamically; the *system network* (*activities*, *IDAs* and *servers*) must remain invariant at run-time. The designer must document how the language features have been used to support the MASCOT virtual machine.

In MASCOT 3 the specific form of the kernel is not mandatory; it is only a recommendation and the implementer can provide the support in any appropriate way. MASCOT 3, however, requires the virtual machine to support additional features. For example, an activity providing direct feedback control that has a hard time constraint may require access to external data in order to update controller parameters. To meet its time constraint it must have guaranteed access at all times

and must not be kept waiting because another activity is accessing the data. A simple solution to this problem is to allow the module to access the data directly without using the standard access procedures. MASCOT provides for this through a construct called an *access interface* that must be supported by the kernel.

9.8 SUMMARY OF MASCOT

Although MASCOT is not widely used – its use has largely been for military applications within the UK – it has been dealt with at some length because it demonstrates some valuable ideas and features. Some of the more important are considered below.

Templates: the template construct encourages the reuse of software components, which contributes to increased reliability. Templates are generic entities and the hierarchical structure of systems, subsystems and activities lends itself to the development of knowledge-based support tools based on using the frame paradigm for the representation of knowledge (Bennett, 1992).

Encapsulation: entities in MASCOT have many of the features of objects in that the method forces the designer to encapsulate procedures for accessing data within communication units (IDAs) and to adopt message passing as a major means of communication between activities. So although MASCOT does not claim to be an object-oriented design method it contains many of the features of such methods.

Virtual machine: MASCOT systems are designed for a specific virtual machine, the MASCOT kernel. This has several advantages: the designer becomes familiar with the characteristics of the machine on which the system is to run; portability of designs is enhanced as the designer cannot utilise the peculiarities of one specific operating system or type of hardware; there is a clear separation of application design from system implementation. The disadvantage of the approach is the possibility of a less efficient implementation.

MASCOT 3 provides an excellent design methodology. It is sufficiently rich in concepts to provide design flexibility but has sufficient constraints for creating safe and reliable software. The introduction of hierarchical structures and the support for the generation of networks has overcome the limitations of MASCOT 2. However, the diagrams used in MASCOT 3 are much more complex to draw than those of the previous version and it is therefore more difficult to use the method quickly to sketch out ideas by hand. CASE tool support for both manipulating the diagrams and handling the textual representation is essential.

9.9 FORMAL METHODS

One of the features of MASCOT (and also of HOOD) is the way in which diagrams used as the basic design element are formalised by the use of textual equivalents. The textual form of the diagram is then successively elaborated to form the code for the system the diagram represents. Through the use of CASE tools conformity of the diagrams and textual representation to the rules can be checked and consistency can be enforced, as can the transformations between various representations. The weakness lies in showing that the design conforms to the specification and this issue is not addressed in MASCOT.

The aim of formal software engineering methods is to be able to transform a formal specification into implementation code and to be able to prove that each transformation step is correct. A first requirement for doing this is that the specification should be expressed in a formal (mathematical) language. There are a growing number of formal specification languages. Cooling (1991, p. 203) has listed the main ones as:

Model based	*Axiom based*
VDM (Vienna Development Method)	LARCH
Z	ACT-ONE
INA-JO	OBJ
me too	CLEAR

The most widely used of the model-based methods are VDM and Z and attempts have been made to use these for the specification of real-time systems. However, as yet they contain no facilities for specifying timing or concurrency. Cooling (1991, Chapter 7) gives a simple and brief introduction to the basic ideas of formal specification languages. There is one formal specification method that was developed specifically for real-time systems, namely PAISLey.

9.10 THE *PAISLEY* SYSTEM FOR REAL-TIME SOFTWARE DEVELOPMENT METHOD

PAISLey (Process-oriented, Applicative and Interpretable (executable) Specification Language) has been developed by Pamela Zave at the University of Maryland and the Bell Laboratories of AT & T. The specification is developed in terms of an explicit model of the environment interacting with an explicit model of the proposed system. Both the environment and the proposed system are modelled as a set of 'asynchronous interacting digital processes'. Non-digital objects in both the environment and the proposed system are modelled as discrete simulations of the object.

The behaviour of the asynchronous processes (which will be referred to as `processes` or `process` in the following sections) is described in a formal language. The language statements are executable and an interpreter is provided as part of the system. It is the intention (not yet achieved) that implementation code should be generated by applying formal transformation rules to the specification. The language is based on two well-established models of computing, asynchronous processes and functional programming, which have been merged. In doing so most of the benefits of each model have been preserved.

A system is specified in PAISLey as a set of processes that continually cycle through a set of state changes. The interval between the state changes is referred to as the *process step*; computations to be performed at each step are specified using a functional notation and each can be considered as a mapping of an input set of values to an output set of values. In order to restrict side-effects mappings should not use variables or assignment statements. Processes communicate by means of precisely defined, interprocess communication protocols.

9.10.1 A Simple System

Figure 9.5 shows a simple system module which contains two *processes* – machine and monitor. Information passes between processes by means of *channels*. A process is considered to be a finite state machine and its behaviour can be specified by defining:

1. a state space – that is, declaring a set of all possible states of the process (note that this is not the 'state space' of linear control but represents the set of discrete states of the system); and
2. a 'successor' function that defines the transition from the `current_state` to the `next_state`.

Thus if x is a member of the set X defined as $X = \{x_i \ldots x_n\}$ where n is finite, then $x(t+s) = \text{succ}[x(t)]$, where s is the finite interval of time required to compute the function 'succ'. The time s is referred to as the process step and is related to one basic cycle of the proposed activity (process) within the system.

Stage 1 of building the specification is to define the system structure:

```
(machine-cycle [initial-machine-state], monitor-
cycle[initial-machine-image]);
```

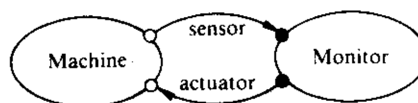

Figure 9.5 The process structure for a trivial process control system.

This states that the system is composed of two processes and that their successor mappings are `machine-cycle` and `monitor-cycle`, and their initial states are the values of `initial-machine-state` and `initial-machine-image` respectively. The set of values that can be taken by the initial states are defined by the statements

```
initial-machine-state: --> MACHINE-STATE;
    "MACHINE-STATE should be defined as the set of all
    possible states of this process."
initial-machine-image: --> MACHINE-IMAGE;
    "MACHINE-IMAGE should be defined as the set of all
    possible states of this process, capable of retaining
    whatever historical information is required."
```

The inverted commas are used to indicate a comment.

The next stage is to define each of the processes:

```
machine-cycle: MACHINE-STATE --> MACHINE-STATE;
machine-cycle: 1 --> = 0.1 s;
```

These two statements indicate that at each `machine-cycle` there is a mapping from one value in the set `MACHINE-STATE` to another, and that each `machine-cycle` takes exactly 0.1 s. The actual function evaluated to get the new value is given by

```
machine-cycle[state]
proj[(1,(simulate-,machine[(state,accept-feedback-if-
any[Null])],
offer-sensor-data[sense[state]]
))];
```

These statements indicate that each evaluation of `machine-cycle` includes the parallel evaluation of two expressions, one to compute another step of the discrete simulation algorithm and one to offer the most recent sensor data to the monitor. The value of the first expression becomes the next state of the process, while the value of the second is thrown away.

The system described in Figure 9.5 exchanges data between the processes by *channels* named `sensor` and `actuator`. Channels are specified by `Exchange Functions` which are described below. The statements describing this part of the specification for the machine process are

```
accept-feedback-if-any: FILLER --> FILLER U ACTUATOR-
SIGNAL;
accept-feedback-if-any[null] = xr-actuator[null];
simulate-machine:
MACHINE-STATE × (ACTUATOR-SIGNAL U FILLER) --> MACHINE-
STATE;
sense: MACHINE-STATE --> SENSOR-DATA;
offer-sensor-data: SENSOR-DATA --> SENSOR-DATA U FILLER;
```

The full PAISLey specification of this simple system with comments is given in Figure 9.6.

```
*------------------------------------------------------*
* SYSTEM STRUCTURE          *
*------------------------------------------------------*
(machine-cycle [initial-machine-state], monitor-cycle[initial-
machine-image1]);
"The system is composed of two processes. Their successor mappings
are machine-cycle and monitor-cycle, and their initial states are
the values of initial-machine-state and initial-machine-image
respectively."
*------------------------------------------------------*
* MACHINE PROCESS           *
*------------------------------------------------------*
initial-machine-state: --> MACHINE-STATE;
"MACHINE-STATE should be defined as the set of all possible states
of this process."
machine-cycle: MACHINE-STATE --> MACHINE-STATE;
machine-cycle: 1 --> = 0.1 s;
"Each evaluation of machine-cycle takes exactly 0.1 second."
machine-cycle[state]
proj[(1,(simulate-,machine[(state,accept-feedback-if-
any[Null])],
offer-sensor-data[sense[state]]
))];
"Each evaluation of machine-cycle includes the parallel
evaluation of two expressions, one to compute another step of the
discrete simulation algorithm and one to offer the most recent
sensor data to the monitor. The value of the first expression
becomes the next state of the process, while the value of the
second is thrown away."
accept-feedback-if-any: FILLER --> FILLER U ACTUATOR-SIGNAL;
accept-feedback-if-any[null] = xr-actuator[null];
"accept-feedback-if-any is defined as a nonwaiting interaction on
the actuator channel. If no interaction takes place the value
Null will be returned."
simulate-machine:
MACHINE-STATE x (ACTUATOR-SIGNAL U FILLER) --> MACHINE-STATE;
"This mapping should be defined as one step of the discrete
simulation algorithm."
sense: MACHINE-STATE --> SENSOR-DATA;
"This mapping should be defined to simulate the physical sensor
attached to the machine."
offer-sensor-data: SENSOR-DATA --> SENSOR-DATA U FILLER;
offer-sensor-data[data] = xr-sensor[data];
"offer-sensor-data is defined as a nonwaiting interaction on the
sensor channel."
```

Figure 9.6 The PAISLey specification listing for the trivial process control system *(continued overleaf)*.

```
*---------------------------------------------------.*
* MONITOR PROCESS          *
*---------------------------------------------------*
initial-machine-image: --> MACHINE-IMAGE;
"MACHINE-IMAGE should be defined as the set of all possible states
of this process, capable of retaining whatever historical
information is required."
monitor-cycle: MACHINE-IMAGE --> MACHINE-IMAGE;
monitor-cycle: 1 --> <= 2.0 s;
"Each evaluation of monitor-cycle must take less than or equal to
two seconds."
monitor-cycle[image] = process-sensor-data[(image,get-sensor-
data[Null])];
"Each evaluation of monitor-cycle consists of getting the most
recent sensor data and then processing it."
get-sensor-data: FILLER --> SENSOR-DATA;
get-sensor-data[null] = x-sensor[null];
"get-sensor-data is defined as a waiting interaction on the sensor
channel. Its value is the most recent sensor data."
process-sensor-data: MACHINE-IMAGE x SENSOR-DATA --> MACHINE-
IMAGE;
process-sensor-data[(image,data)] =
proj[(1,(maintain-machine-image[(image,data)],
give-feedback-if-needed
[check-machine-condition[(image,data)]]
))];
"Each evaluation of process-sensor data includes the parallel
evaluation of two expressions, one to incorporate the most recent
sensor data into the historical information being saved in the
process, and one to provide feedback to the machine if it is
needed. The value of the first expression becomes the next state of
the process, while the value of the second is thrown away."
maintain-machine-image:
MACHINE-IMAGE X SENSOR-DATA --> MACHINE-IMAGE;
"This mapping should be defined to save the most recent sensor
data."
check-machine-condition:
MACHINE-IMAGE X SENSOR-DATA --> { No-Problem } U ACTUATOR-SIGNAL;
"This mapping should be defined to decide whether feedback is
needed and if so what the actuator signal should be."
give-feedback-if-needed: { No-Problem } U ACTUATOR-SIGNAL -->
FILLER;
give-feedback-if-needed[signal] =
/equal[(signal,No-Problem)]: Null,
True                : give-feedback[signal]
/;
give-feedback: ACTUATOR-SIGNAL --> FILLER;
give-feedback[signal] = x-actuator[signal];
"give-feedback is defined as a waiting interaction on the actuator
channel."
```

Figure 9.6 *continued*

9.10.2 Exchange Functions

Processes interact by sending and receiving data through channels. The behaviour of a channel is defined by an exchange function. Three primitive exchange functions referred to as `x`, `xm` and `xr` are defined. The syntax of the exchange function is

```
<function type> - <channelname> [argument]
```

for example `x-msg[y]` defines a channel of type `x` with the name `msg` and argument `y`. The argument provides an item to be sent and returns an item received. The function types are:

- `x`: matches (synchronises) with a pending exchange function on its channel. If no exchange function is pending then it waits. If several requests are pending they are satisfied on a non-deterministic basis.
- `xm`: behaves like an `x`-type exchange function except that two `xm` functions on the same channel cannot match with each other.
- `xr`: behaves like an `x`-type except that it will not wait. If an `xr` exchange function cannot find an immediate match it will terminate and return its own argument value.

9.10.3 Timing Constraints

PAISLey supports the insertion of timing constraints into the specification. These are inserted as formal statements in the language; for example, the statement

```
machine-cycle ; ! =0.1s;
```

specifies a cyclic operation with a repetition time of exactly 0.1 s. The statement

```
monitor-cycle: ! < = 2.0s
```

specifies that `monitor_cycle` must compute its successor function within a time period of less than or equal to 2 seconds.

The language supports a wide range of timing constraints which include the ability to specify upper and lower bounds as well as precise hard constraints. During the execution of the specification these constraints are used to check for timing inconsistencies and conflicts and any such problems are reported.

EXAMPLE 9.1

Part of Automobile Management System

Figure 9.7 shows the state transition diagram for the main control section of an

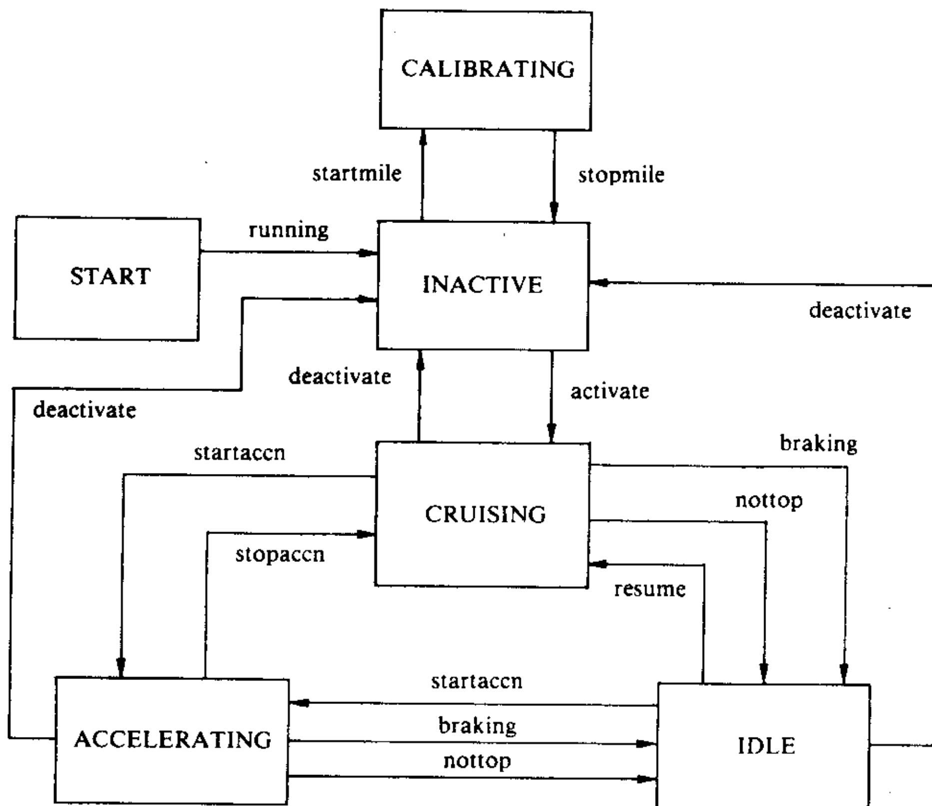

Figure 9.7 State diagram for the main states of the automobile management system.

automobile management system (this is part of a widely used case study – see Hatley and Pirbhai, 1988). The specification for this system written in PAISLey is shown in Figure 9.8.

9.11 *PAISLEY* SUMMARY

The major features of the language are:

- Support for both synchronous and asynchronous communication free from the problems of mutual exclusion.
- All computations are encapsulated; the mapping functions can be considered as black boxes.
- It is possible to execute incomplete specifications and hence rapid prototyping and incremental development are possible.

```
"Paisley file for Auto cruise state diagram"
"This file contains the Finite State Machine for the auto
cruise control. It consists of a number of states and a set
of rules for moving between them."
"A full range of driver command channels are taken by this
process and used for reading the drivers command. They are
read using the xr exchange function, so that we can pick up
one command from more than one channel. Very similar to a
multiplexor."
"The channels are;
running - Driver switches on the system
startmile - Start measured mile command
stopmile - Stop measured mile command
deactivate - deactivate the cruise control
activate - activate the cruise control
nottop - Not in top gear signal - inverted topgear signal
startaccn - start accelerating
stopaccn - stop accelerating
braking - Driver is braking
resume - Resume cruise control command
"
"The order in which these channels are read is important. If
the driver issues 'braking' and also 'startaccn' then
obviously 'braking' has to have priority."
state-machine: CRUISE-STATE --> CRUISE-STATE;
"Find out what state we're in and go off and see if there is
a change of state."
state-machine[state] =
/
equal[(state, 'START')]:          update-start,
equal[(state, 'INACTIVE')]:       update-inactive,
equal[(state, 'CALIBRATING')]:        update-calibrating,
equal[(state, 'CRUISING')]:       update-cruising,
equal[(state, 'ACCELERATING')]:  update-accelerating,
equal[(state, 'IDLE')]:           update-idle,
True:                    state    "Really an error"
/;
"Start state mapping"
update-start:    --> CRUISE-STATE;
update-start =
/
"Ordered by safety critical importance"
equal[(xr-running [Null],true)] : 'INACTIVE',
True                 : 'START'
/;
"Inactive state mapping"
update-inactive:    --> CRUISE-STATE;
update-inactive =
/
"Ordered by safety critical importance"
equal[(xr-activate [Null],True)]:
proj [(1, ('CRUISING', x-motionstate['ON'],
x-select['ON']))],
equal[(xr-startmile [Null],True)]:
proj [(1, ('CALIBRATING', x-measurestate['ON']))],
True
/;
"Calibrating state mapping"
update-calibrating: --> CRUISE-STATE;
update-calibrating =
/
```

Figure 9.8 PAISLey specification for system shown in Figure 9.7 (*continued overleaf*).

```
"Ordered by safety critical importance"
equal[(xr-stopmile [Null],True)]:
proj [(1, ('INACTIVE', x-measurestate['OFF']))],
True                          : 'CALIBRATING'
/;
"Cruising state mapping"
update-cruising:    --> CRUISE-STATE;
update-cruising =
/
"Ordered by safety critical importance"
equal[(xr-braking [Null],True)]: 'IDLE',
equal[(xr-nottop       [Null],True)]:'IDLE',
equal[(xr-deactivate [Null],True)]:
proj [(1, ('INACTIVE', x-motionstate['OFF']))],
equal[(xr-startaccn [Null], True)]: 'ACCELERATING',
True
proj [(1, ('CRUISING', x-maintainv['ON']))]
/;
"Accelerating state mapping"
update-accelerating:    --> CRUISE-STATE;
update-accelerating =
/
"Ordered by safety critical importance"
equal[(xr-braking [Null],True)]: 'IDLE',
equal[(xr-nottop       [Null],True)]: 'IDLE',
equal[(xr-stopaccn     [Null],True)]:
proj[(1, ('CRUISING',
x-select['ON']))],
equal[(xr-deactivate[Null],True)]: 'INACTIVE',
True
proj [(1, ('ACCELERATING', x-maintainv['ON']))]
/;
"Idle state mapping"
update-idle:    --> CRUISE-STATE;
update-idle =
/
"Ordered by safety critical importance"
equal[(xr-braking      [Null],True)]: 'IDLE',
equal[(xr-nottop       [Null],True)]: 'IDLE',
equal[(xr-deactivate   [Null],True)]: 'INACTIVE',
equal[(xr-startaccn    [Null],True)]: 'ACCELERATING',
equal[(xr-resume       [Null],True)]:
proj [(1, ('CRUISING',
x-select['ON'],
x-select['ON']))],
True                   : 'IDLE'
/;
```

Figure 9.8 *continued*

- Interprocess communication is precisely defined using exchange functions which hide problems of mutual exclusion and which can be used to simulate timing constraints on communication links.
- Both hard and soft timing constraints can be specified and these constraints are automatically checked for violation when the specification is executed.
- Bounded resource usage can be guaranteed.

The major weaknesses of PAISLey are largely those that are common to many approaches to formal specification, namely that for all but the simplest systems the specification becomes long and cumbersome, making it difficult to read and follow. To deal with this problem specifications can be broken down into segments which

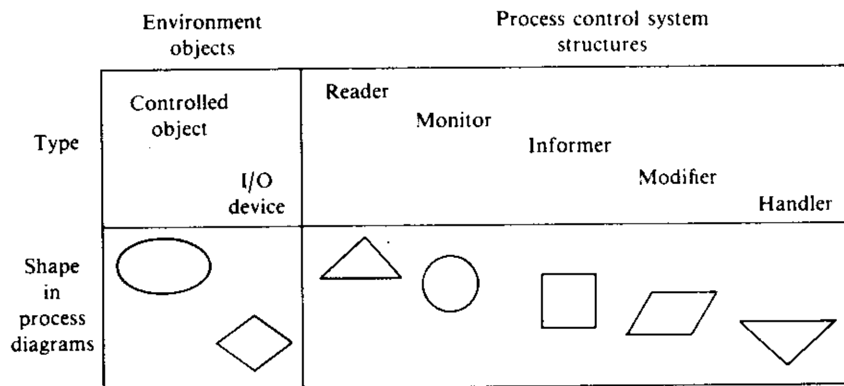

Figure 9.9 Graphical notation for PAISLey processes.

can be held in separate files. There has also been an attempt to add to PAISLey a graphical representation (see Figure 9.9) but unlike MASCOT there is no exact relationship between the graphical representations and the textual representation.

The other weakness of PAISLey is that the system is not available in a fully developed form in the sense that it lacks the good user interfaces that now characterise the majority of the CASE tools. A particular problem is that the output from the interpreter is in simple textual form which is difficult to analyse.

9.12 SUMMARY

In this chapter we have briefly examined two methods for dealing with real-time systems. MASCOT provides a well-established methodology which assumes that a limited set of basic elements are all that are required to implement the system. The additions that have been made to MASCOT 3 are such as to support an object-oriented approach to system design. A criticism of the method is that the timing requirements are not visible on the design documents and in fact the designer proceeds without direct reference to them. As MASCOT is not intended for specification and, as we have seen, it is difficult to make any use of the timing constraints until the realisation of the implementation, this is not a serious restriction on the use of MASCOT.

PAISLey is much less well developed and known than MASCOT. Its main interest lies in the attempt to support the specification of real-time systems using a formal language in which the timing constraints can be easily expressed. The other interesting feature is that the specification can be executed. It is a pity that the output from the execution is so difficult to interpret. What is required is full animation support.

10

Design Analysis

10.1 INTRODUCTION

The development methodologies considered in Chapters 8 and 9, with the exception of PAISLey, do not provide any means of analysing the design either to compare designs or to evaluate the implementation requirements. A weakness of both the Ward and Mellor and the Hatley and Pirbhai methods which we noted is that in allocating resources as part of the design process one really needs to be able to assess whether a feasible schedule, that is a schedule that meets the time constraints, exists for a particular design structure. Similarly both methods make extensive use of state transition diagrams in specifying and designing the control structure of the system. A method of analysing such diagrams to reveal unreachable states and/or undesirable states that are reachable would be useful. Ward and Mellor suggested that Petri nets might be used to analyse the essential model and hence in the next section we will look at Petri nets and their use.

10.2 PETRI NETS

Petri nets have been widely adopted as a method for *modelling* and *analysing* systems that can be described in terms of a set of states and a set of events. An event change results in a change from one state of the system to another state. Also, a change of state results in a change in the event set. The technique was originally proposed by Carl Adam Petri in 1962 as a basis for modelling computer systems with asynchronous communication between asynchronous components. It has since been used to model business systems, hardware systems and manufacturing systems. A full treatment of the technique is given in the book by Peterson (1981) which is recommended as a starting point for an in-depth study of the technique. An extensive bibliography on Petri nets can be found in the book by Reisig (1982).

10.2.1 Basic Ideas

A Petri net is used to model a system on the basis of two properties:

1. Condition: a Boolean description of the state of the system; a *condition* may be *true* or *false*.
2. Event: an action that depends on the state of the system.

The system model is represented by a set of conditions and a set of events. In the Petri net notation a condition is represented by a *place* and an event by a *transition*. In the graphical representation of a Petri net a place is drawn as a circle and a transition as a bar as is shown in Figure 10.1. Places and transitions are connected by arrows.

Referring to Figure 10.1, place p_1 is an *input place* for transition t_2 and an *output place* for transition t_1; whereas place p_2 is an input place for transition t_1 and an output place for transition t_2. Input places represent the necessary conditions for an event represented by a transition to occur and are referred to as *preconditions*.

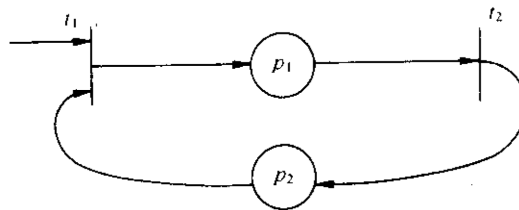

Figure 10.1 Graph representation of a Petri net.

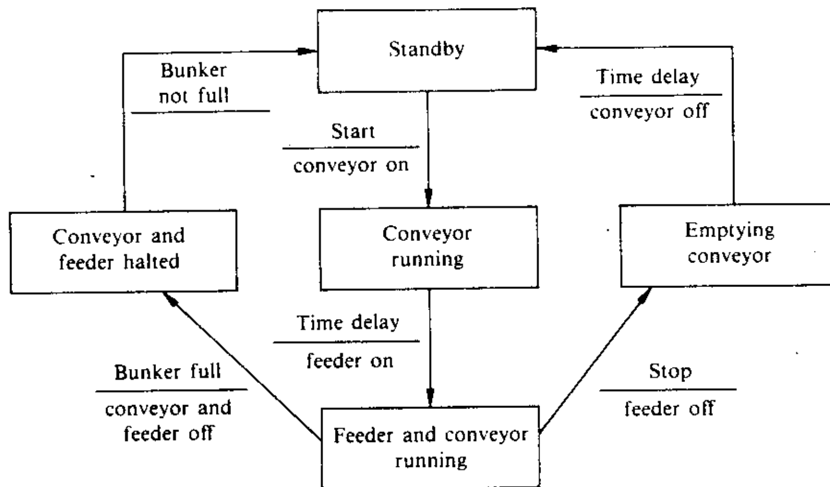

Figure 10.2 State transition diagram for a coal clearance system.

Output places represent the set of conditions that result from a transition and are called *postconditions*.

Given that Petri nets are used to model conditions and events there is obviously a similarity between state transition diagrams and Petri nets. For example, the STD shown in Figure 10.2 can be represented by the Petri net shown in Figure 10.3. A major difference arises from the idea of *executing* a Petri net. Dots are placed in

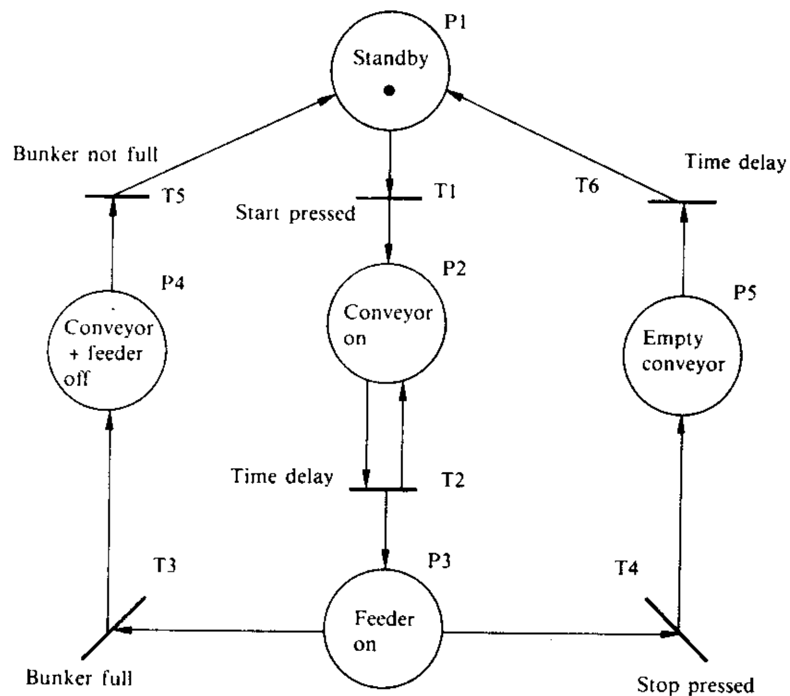

Figure 10.3 Coal clearance system equivalent Petri net.

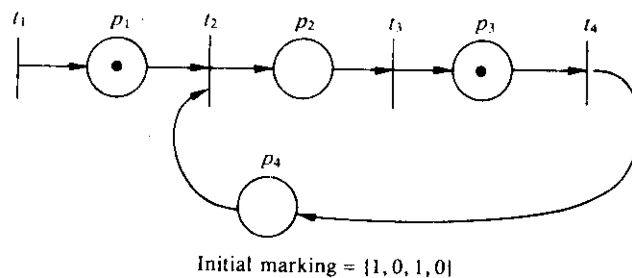

Figure 10.4 A marked Petri net.

the circles representing places for which the conditions of a place are known to be true. The dots are referred to as *tokens*. A distribution of tokens is known as a *marking* of the Petri net. A marked Petri net is shown in Figure 10.4. The marking can be represented as (1, 0, 1, 0) with each number representing the number of tokens present in places p_1 to p_4 respectively. A place can contain several tokens and hence can represent a queue – for example, a buffer holding several messages or a queue of parts awaiting processing. If the number of tokens at a place is large then instead of using dots a number is written in the place. Figure 10.5 shows a Petri net with multiple tokens; the marking is (10, 1, 20, 0).

Executing a marked Petri net causes the number and positions of the tokens to change. The rules for executing a Petri net are:

1. a transition is *enabled* if *all* its input places contain at least one token;
2. any enabled transition may *fire*;
3. firing of a transition results in one token being removed from each of its input places, and being deposited at each of its output places; and
4. execution halts when there are no enabled transitions.

Each time a transition fires the marking of the Petri net will change (usually). For example, consider the Petri net shown in Figure 10.4 where transition t_4 is enabled since it has one input place p_3 and this place contains a token. The effect of firing t_4 is to change the marking to (1, 0, 0, 1) as shown in Figure 10.6.

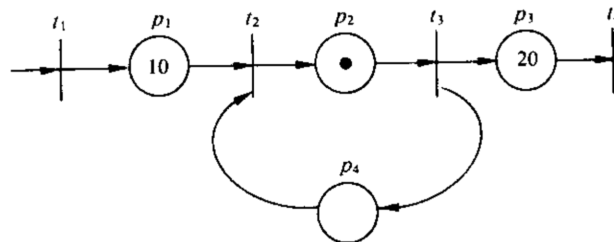

Figure 10.5 A marked Petri net with multiple tokens.

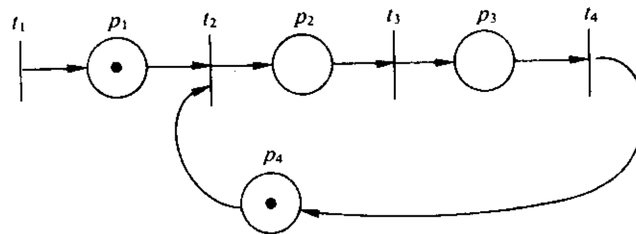

Figure 10.6 Executing a Petri net – stage 1.

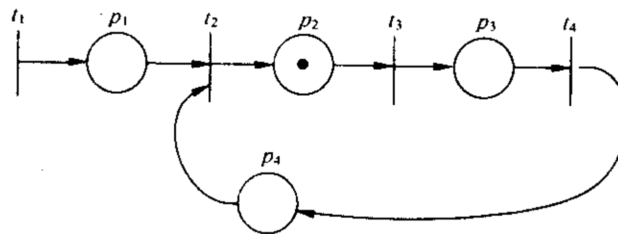

Figure 10.7 Executing a Petri net – stage 2.

Now transition t_2 can fire since places p_1 and p_4 contain tokens and as a result of t_2 firing the marking changes to (0, 1, 0, 0) and the marked net is now as shown in Figure 10.7.

Note that the firing of t_2 has changed the number of tokens in the net from two to one. This is because there are two input arrows to t_2 but only one output arrow.

In general the firing of a transition will result in a change in both the marking of a net and the number of tokens in the net. The number of tokens at an input place can never become negative since a transition can fire only if each of its input places contains at least one token. It is possible to model a system using a Petri net such that the total number of tokens in the net is kept constant. Such a model would be required if each token was being used to represent an object flowing through the system.

If several transitions are enabled then the order of firing is non-deterministic. When analysing a Petri net each possible order of firing must be considered. Use of this property enables us to use Petri nets to model systems which have concurrent events that do not have a unique ordering. You should note, however, that if the system being modelled does, in some way, guarantee precedence to certain events this must be explicitly modelled in the Petri net.

10.2.2 Modelling Mutual Exclusion

As an example of using Petri nets to model concurrent events let us consider the mutual exclusion problem. Consider two tasks **TA** and **TB** that share a resource **R**. The resource is protected by a semaphore **SR**. Assume that the tasks can be split into segments **TA1**, **TA2**, **TA3**, **TB1**, **TB2** and **TB3** where **TA2** and **TB2** represent the critical sections of each task respectively.

If we model each task segment as a place and the transition from one segment to the next as a transition (an event) we get the two independent Petri nets shown in Figure 10.8a. The presence of a token in a place will be used to indicate that a particular segment is active because there is an agreement between the task designers to insert code in each task that checks with the semaphore **SR** (for example, by executing a **SECURE (SR)** statement) before proceeding to segment **TA2** or **TB2**; the conditions for the event represented by **EA1** to occur are that **TA** should be at

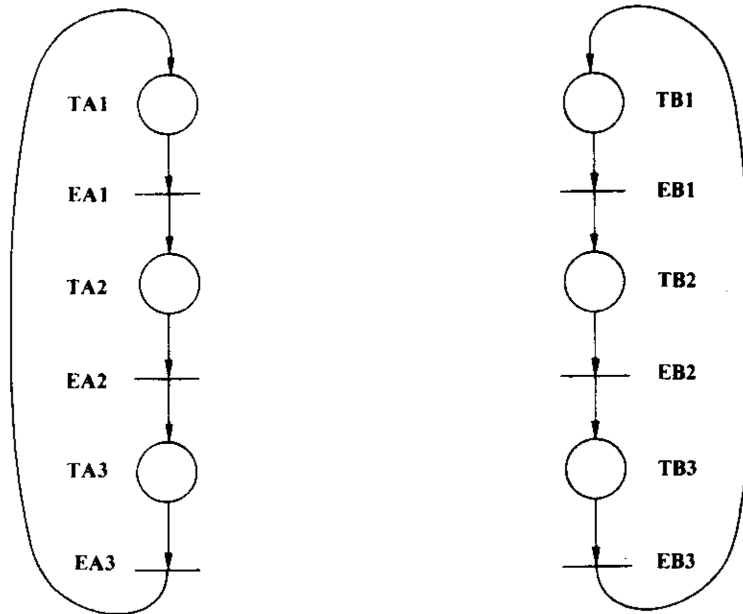

Figure 10.8a Mutual exclusion example – stage 1.

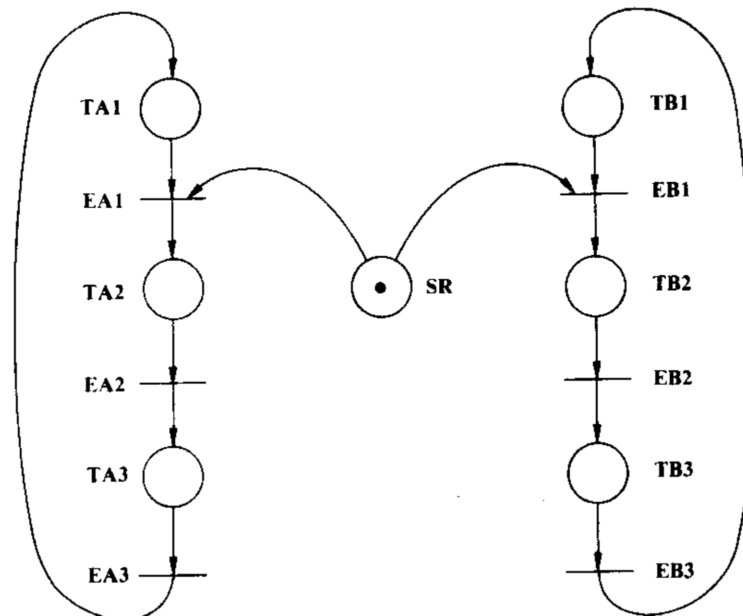

Figure 10.8b Mutual exclusion example – stage 2.

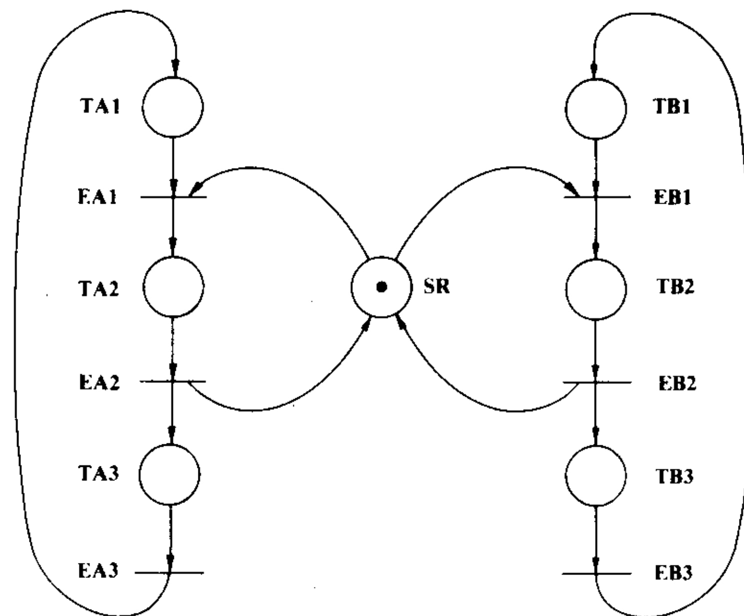

Figure 10.8c Mutual exclusion example – stage 3.

the end of segment **TA1** and that **SR** should indicate that the resource is free. We can represent the semaphore by a place and the condition that the resource is free by the presence of a token in the place. This is shown in Figure 10.8b. The presence of a token in **TA1** or **TB1** would now cause either **EA1** or **EB1** to fire, thus removing the token from **SR** and either **TA1** or **TB1** and putting a token in either **TA2** or **TB2**.

We complete the model by considering what happens when segment **TA2** or **TB2** completes. At the end of the critical section the task executes a **RELEASE (SR)** and thus we need to return a token to **SR** and the task then continues to execute the next segment. Hence we need to add arrows from **EA2** and **EB2** to **SR** as is shown in Figure 10.8c.

To execute the Petri net model we need to make an assumption about the initial marking. Let us assume that the initial marking is (1, 0, 0, 1, 0, 0, 1) as shown in Figure 10.9a. With this marking either transition t_1 or t_4 will fire – which fires is non-deterministic. Let us assume that t_1 fires; the net marking now becomes (0, 1, 0, 1, 0, 0, 0) as shown in Figure 10.9b, since the token is removed from p_1 and from p_7. Examining Figure 10.9b we can see that task **TA** has entered the critical section **TA2** and that task **TB** is prevented from entering segment **TB2** since the conditions for transition t_4 to fire no longer hold. However, t_2 can now fire since there is a token in p_2. When t_2 has fired the net marking becomes (0, 0, 1, 1, 0, 0, 1). The conditions now exist for t_4 and t_3 to fire.

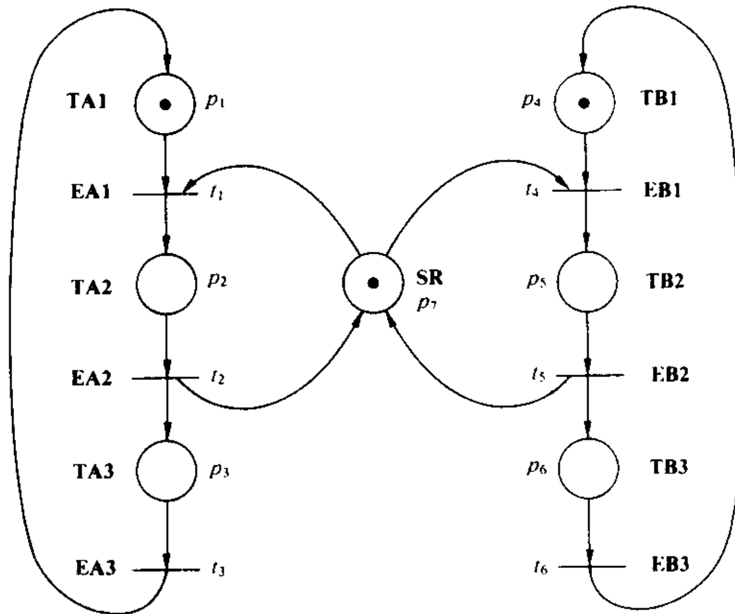

Figure 10.9a Mutual exclusion and execution – stage 1.

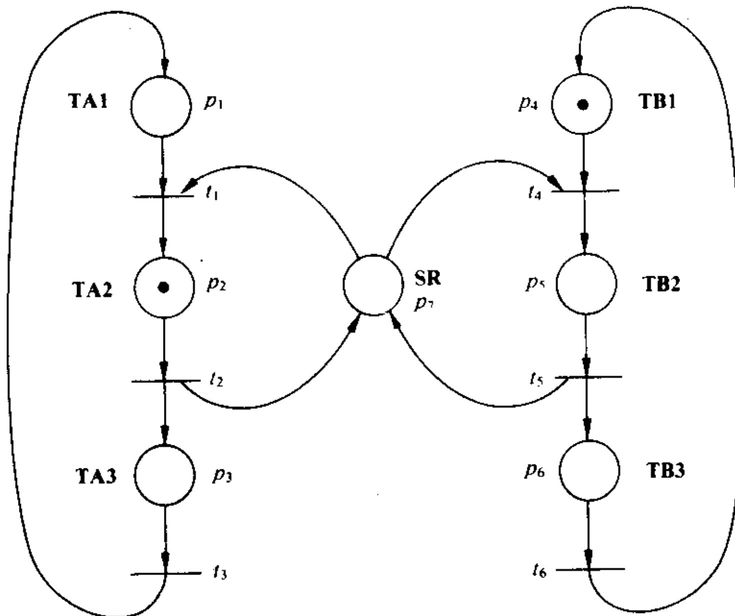

Figure 10.9b Mutual exclusion and execution – stage 2.

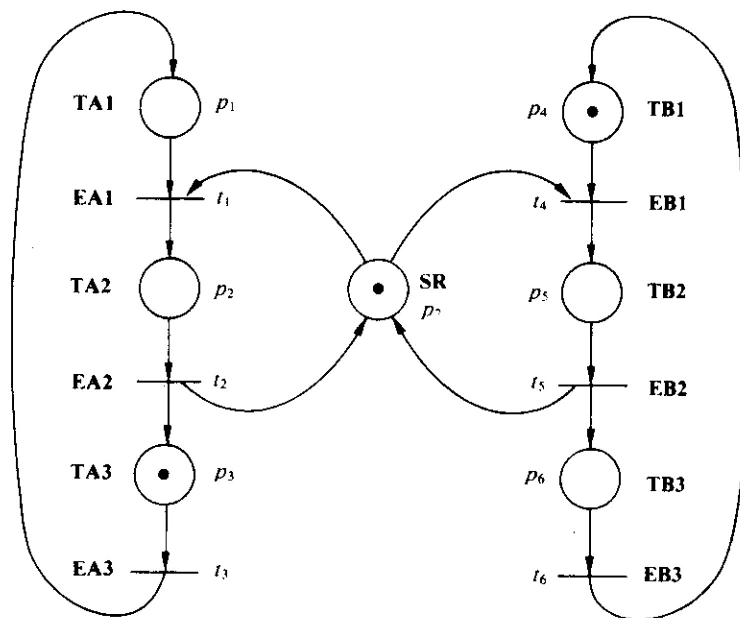

Figure 10.9c Mutual exclusion and execution – stage 3.

In following the execution stages in Figures 10.9a, 10.9b and 10.9c you should have noticed that once a token reaches p_2 the transition t_2 is enabled and can thus fire. The implication of this is that the Petri net does not model the timing of the two tasks since execution of the task segment **TA2** will take a finite length of time. The model we really require is one in which the timing of t_1 indicates the start of execution of segment **PA2**, and the presence of a token in p_2 indicates that **TA2** is being executed. The condition for the firing of t_2 is that **TA2** has finished executing. In order to produce such a model we need to use an extended form of the Petri net notation – the timed Petri net.

10.3 ANALYSING PETRI NETS

Although it is possible to obtain some information about the behaviour of a system modelled by a Petri net by executing the net either by hand or by using a computer simulation the number of possible sequences is such that the procedure is laborious and the information obtained uncertain for all except the most simple nets. Formal methods of analysis are required and these are based on set theory formulations.

A Petri net structure C can be represented as the four-tuple $C = (P, T, I, O)$ where $P = \{P_1, P_2, \ldots, P_n\}$, the set of places; $T = \{T_1, T_2, \ldots, T_n\}$, the set of transitions; I is an input function that maps each transition to its set of input places;

and O is an output function that maps each transition to its set of output places. Thus the Petri net shown in Figure 10.9a can be represented as:

$$C = (P, T, I, O)$$
$$P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$$
$$T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$$

$$I(t_1) = \{p_1, p_7\} \quad O(t_1) = \{p_2\}$$
$$I(t_2) = \{p_2\} \quad O(t_2) = \{p_3, p_7\}$$
$$I(t_3) = \{p_3\} \quad O(t_3) = \{p_1\}$$
$$I(t_4) = \{p_4, p_7\} \quad O(t_4) = \{p_5\}$$
$$I(t_5) = \{p_5\} \quad O(t_5) = \{p_6, p_7\}$$
$$I(t_6) = \{p_6\} \quad O(t_6) = \{p_4\}$$

In analysing a Petri net model of a concurrent system we are concerned with obtaining answers to questions concerning:

- safeness;
- boundedness;
- conservation;
- equivalence;
- reachability;
- coverability; and
- liveness.

Safeness: A Petri net is said to be *safe* if all the places in the net are safe. A place is said to be safe if the number of tokens in the place is either 0 or 1. For example, if we were using the net shown in Figure 10.9c to find out whether we could use a binary semaphore to implement the mutual exclusion condition we would want to know if place p_7 was 'safe'.

Boundedness: A Petri net is said to be bounded if all the places are bounded. A place is bounded if the number of tokens in it can never exceed some finite integer value N. If a place is bounded then the physical element that it models can be realised using a finite storage device.

Conservation: A Petri net is said to be conservative if the number of tokens in the net remains constant, that is tokens are neither created nor destroyed when a transition fires. Strict conservation implies that for each transition the number of input places must match the number of output places. The mutual exclusion model is not conservative since the firing of transition t_1 destroys a token and the firing of t_2 creates a token. Testing strictly for conservation is important in Petri nets where tokens are used to represent objects moving around a closed system; or where tokens represent resources available to the system.

Equivalence: For two Petri nets to be said to be equivalent all possible behaviours must be equivalent. Establishing equivalence is difficult since each net has to be analysed for reachability, coverability and firing sequence. The

equivalence property can be used to show that a given Petri net is a subset of another net. Its main use is in trying to optimise a system by removing redundant elements.

Reachability: Reachability is a basic property of a Petri net. It is concerned with answering the question: given an initial marking can a specified marking occur? The specified marking may be a desirable marking or it may be an undesirable marking (for example, a dangerous fault condition). In the mutual exclusion example (Figure 10.9a, b and c) we want an answer to the question: can any marking containing both p_2 and p_5 be reached from any initial marking?

Coverability: Coverability is the problem of determining if, given an initial state, there is a reachable marking that contains a particular marking subset. For example, in the mutual exclusion model we would like to know if it is possible to have simultaneously tokens in places p_2 and p_5, that is, are there any reachable markings that contain the subset $\{p_2 = 1,\ p_5 = 1\}$?

Liveness: A Petri net is said to be *live* if every transition can be enabled. Conversely a Petri net is *deadlocked* if one or more transitions cannot be enabled.

The two basic techniques for analysing Petri nets in order to seek answers to the above questions are:

- reachability trees; and
- matrix equations.

10.3.1 Reachability Tree

The basis of this method is: starting from an initial marking all reachable markings are found; then starting from each of these markings the reachable markings are found, etc. Figure 10.10 shows the reachability tree for the mutual exclusion net. The branches of the reachability tree are stopped *either* because with the marking of the net no further transition can be enabled *or* because the marking is equivalent to some other marking in the tree. For example, if we follow the transition firing path t_1, t_2, t_3 we reach a marking equivalent to the initial marking. Visual examination of the tree enables us to conclude that the net is safe – all the markings contain only the values 0 or 1 – and that there is no marking with tokens in both places p_2 and p_5.

The reachability tree also shows that the net possesses liveness since there is no leaf of the tree with a marking from which there is no transition. This in fact shows the net is free from deadlock: for a strict proof of liveness we have to show that the initial marking is reachable from all leaves of the tree. If a branch of the tree leads to an endless loop where one particular firing sequence repeats to the exclusion of all other firings then the system is *livelocked*.

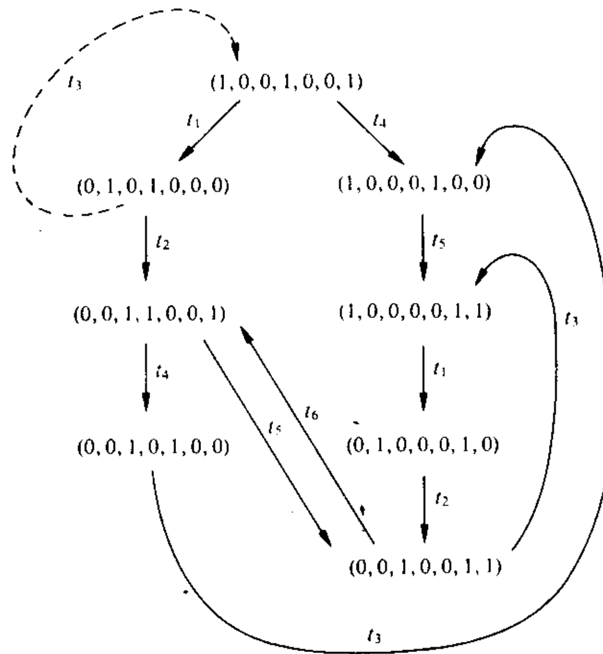

Figure 10.10 Reachability tree for mutual exclusion example.

10.4 SCHEDULING

The end point of most current real-time system development methodologies is an implementation model of the system that consists of a set of independently schedulable actions (asynchronous process) and a set of constraints (time, mutual exclusion and synchronisation). The methodologies then propose that realisation should proceed by first determining what processing elements are needed (analog circuits, general purpose digital processors, special digital processors, etc.) and by allocating groups of actions to processors. The second stage is to determine for groups of actions allocated to a general purpose digital computer which actions shall remain as independent schedulable processes and which shall be combined to form a single schedulable process.

The methodologies give no guidance on how these two stages should be carried out and on how decisions can be made on a rational basis.

Stated in general terms the problem is:

Given a set of processes
$P = \{p_1, p_2, \ldots, p_n\}$
how can a set of processors
$U = \{u_1, u_2, \ldots, u_m\}$

and process allocations
$V = \{(p_a, u_1), (p_b, u_2), \ldots, (p_x, u_m)\}$
be chosen so that the set of constraints
$C = \{c_1, c_2, \ldots, c_p\}$
on the system can be satisfied and that U and V are in some sense optimal?

An associated problem is:

Given P, U, V can we prove that constraints C are satisfied?

(Note that if the problem is extended to consider fault tolerance then U and V are not fixed. Also, in general the existence of a fault may result in a change to C.)
The constraints that have to be satisfied can involve:

- time;
- mutual exclusion; and
- precedence.

The most common form of time constraint is a *deadline*, that is the time by which the execution of a task must be completed, and it may be a hard or a soft constraint (see Chapter 1). Typical terms used when discussing task timing are as shown in Figure 10.11, which assumes that a task may be interrupted and hence executed in segments. The total execution time is the sum of the segment execution times plus any overheads involved in context switching. In general with a pre-emptive task scheduler we do not know how many segments a task will be split into and it will vary from task invocation to invocation. Some tasks may have a constraint on the *start time* as well as a deadline.

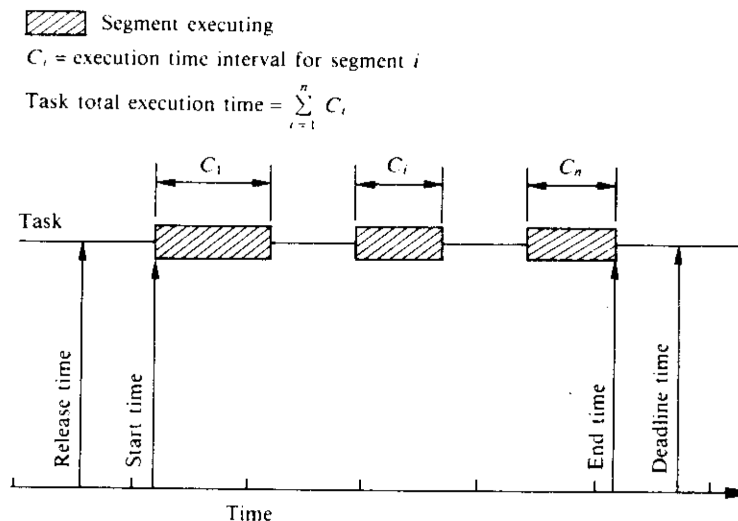

Figure 10.11 Task timing notation.

The mutual exclusion constraint arises when tasks share resources and hence task *A* (or a segment of task *A*) may not be able to run while task *B* (or a segment of task *B*) is using a particular resource. The precedence constraint arises because one task may need information generated by another task. For example, if task *C* (or a segment of *C*) requires a value produced by task *D* (or a segment of *D*) then there is no point in scheduling *D* to run before *C* (or the segment of *D* to run before the segment of *C*).

10.5 GENERAL APPROACHES TO THE SCHEDULING PROBLEM

There are two general approaches:

1. Run-time (on-line scheduling): this is the real-time scheduling (control) problem – can we design a scheduling algorithm (or algorithms) that will allocate resources (including time resources) such that the system meets its constraints?
2. Pre-run-time scheduling: this is the *design* problem – can we choose a set of resources such that a task execution schedule can be constructed that satisfies the constraints? Additional problems are: can we prove that the system does satisfy the constraints; can we choose a minimal set of resources?

The traditional approach to real-time systems has been to use an on-line scheduler. However, in recent years there has been an increased interest in the pre-run-time scheduling approach for systems in which all time constraints are hard. The difficulty with it is that any design change or, during run-time, the loss of a resource because of failure of part of the system means that a new schedule has to be found and implemented. The approach also requires systems to be designed on the basis of worst case conditions: upper bounds on execution time and communication delays, and maximum frequency of occurrence for events; hence there is overprovision of resources.

10.6 ON-LINE SCHEDULING – INDEPENDENT TASKS

The on-line scheduling approach divides into two sections:

1. assessment of schedulability; and
2. choice of scheduling algorithm.

The first of these is concerned with finding out whether there are sufficient resources allocated to the system for a feasible schedule to exist. The necessary and sufficient conditions for showing in general that a schedule that meets the constraints exists

are not known. Under certain restrictions and preconditions it is possible to check if a feasible schedule might exist.

Even if it is shown that a feasible schedule exists it does not mean that in practice all the time constraints will be met; this depends on how effective the scheduling algorithm being used by the operating system is in utilising the resources, or how well it is matched to the particular requirements.

10.6.1 Schedulability

The first requirement for carrying out a schedulability analysis is to determine or estimate the execution time for each task in the system. Given that execution times are not necessarily constant we need two estimates:

1. average execution time; and
2. worst case execution time.

We can get reasonable estimates for execution times only if certain restrictions have been applied when coding the system. For example, use of the following must be avoided:

- dynamic creation of tasks;
- dynamic allocation of memory; and
- recursion.

Also the following restrictions must be applied:

- all loops must have upper bounds (periodic tasks which are written as infinite loops are permitted);
- all intertask communications must have time outs; and
- all external communications must have time outs.

Note that these restrictions apply even if the execution times are being estimated by running the software, because without them the system will be non-deterministic.

Once a set of execution times for each task has been determined we can calculate

Table 10.1 Utilisation time for cyclic processes

Task	*Cycle time (s)*	*Execution time (s)*	*Utilisation (%)*
ReadInputs	0.1	0.02	20
CalculateControl	0.2	0.08	40
UpdateDisplay	5.0	0.30	6
SendToActuator	0.2	0.005	2.5
Total utilisation			68.5

the processor *utilisation* time. This is easily determined for a cyclic (periodic) task. For example, if the cycle period is 0.1 s, and the execution time is 0.02 s, the processor utilisation is (0.02/0.1) × 100%, that is 20%. Thus if we have a set of tasks which run on a single processor with cycle times and execution times as shown in Table 10.1 we can easily determine how much processor time they utilise.

Liu and Layland (1973) proved that for a single-processor system running a set of n independent periodic tasks with constraints consisting only of deadlines on time of the end of execution (that is, the task can be started at any time) a feasible schedule exists if processor utilisation satisfies the condition

$$\sum_{i=1}^{n} e_i/c_i \leqslant n(2^{(1/n)} - 1)$$

and a *rate monotonic scheduling* algorithm is used. This algorithm always chooses the highest-priority task. Task priority is ordered according to the cycle time of the tasks, the highest priority being given to the task with the smallest cycle time. For a large value of n this approaches 0.693; hence we can conclude that if the processor utilisation is less than 69% all sets of tasks are schedulable.

One must **not** conclude from this that sets of tasks with process utilisation greater than 69% cannot be scheduled. For example, task set 1 shown in Table 10.2 with 100% utilisation is schedulable (under the conditions given above) whereas the task set 2 given in Table 10.3 is not (Burns and Wellings, 1990, pp. 347–9). In order

Table 10.2 Task set 1 – utilisation 100%

Task	*Cycle time (ms)*	*Execution time (ms)*	*Utilisation (%)*
P_1	80	40	50
P_2	40	10	25
P_3	20	5	25
Total utilisation			100

Table 10.3 Task set 2 – utilisation 82%

Task	*Cycle time (ms)*	*Execution time (ms)*	*Utilisation (%)*
P_1	50	12	24
P_2	40	10	25
P_3	30	10	33
Total utilisation			82

to meet the deadlines, task set 1 must be run on a processor with a pre-emptive priority scheduler and the task priorities must be ordered from highest to lowest as follows: P_3, P_2, P_1. The task activation diagram for task set 1 is shown in Figure 10.12. From this diagram it can be clearly seen that without a priority scheduler P_1 and P_2 would not be able to meet their deadlines. It should also be clear that the start of P_3 is not accurately synchronised with real time; it will always run with a 15 ms offset, and similarly P_2 will always run with a 5 ms offset from its nominal start time.

Determining schedulability becomes more difficult when there are event-driven tasks in the system. Events are aperiodic and, because there is always a non-zero probability that an event will occur within a given time interval of a previous event regardless of how small that interval is, it is not possible to carry out a worst case analysis. However, by considering the particular application, we can usually set a minimum time interval between two occurrences of a given event. Events constrained in such a way are referred to as being *sporadic*. A worst case analysis can now be carried out by converting *sporadic* events into periodic events. We do this by taking the minimum time interval between successive occurrences of the event as the cycle time for the task that responds to the event. For example, consider the addition of a task **CheckAlarms**, which has to respond within 0.15 seconds of an event, to the set given in Table 10.1. We will assume that the response interval is the minimum interval between successive occurrences of the event and hence the cycle time for **CheckAlarms** is 0.15 seconds, and if we assume the execution time

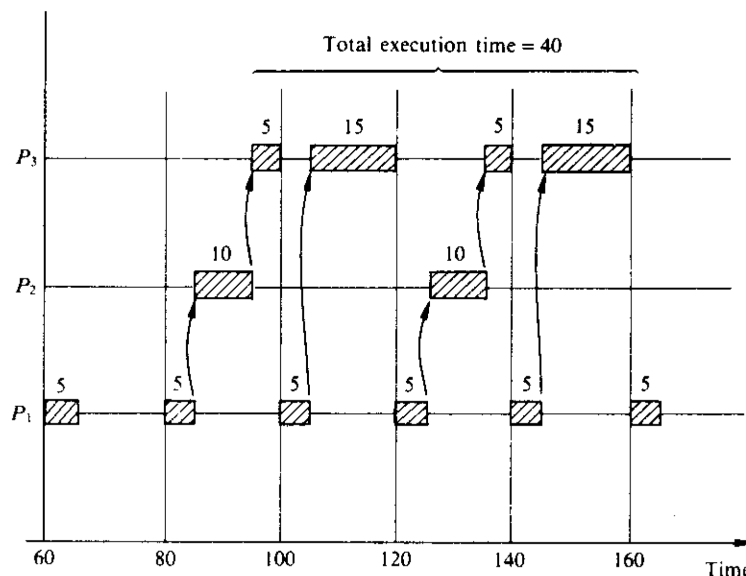

Figure 10.12 Task activation diagram.

is 0.01 seconds then the processor utilisation is 6.7%, taking the total utilisation to 75.2%.

Using the worst case analysis for sporadic tasks can lead to considerable overestimates of the processor utilisation and hence lead to low actual utilisation. The reason is easy to see – the average rate of event occurrences is likely to be much lower than the potential maximum rates.

Burns and Wellings (1990, p. 348) suggest as a guideline for assessing schedulability that the following conditions should be satisfied:

1. All tasks should be schedulable using average execution times.
2. All tasks with hard time constraints should be schedulable using worst case execution times.

A consequence of condition 1 is that there may be occasions when all tasks cannot meet their deadlines – this is referred to as *transient overload* – **but** if condition 2 is satisfied even under these conditions the tasks with hard time constraints will meet those constraints.

An analysis of processor utilisation can be useful when trying to determine the allocation of tasks in a system. In terms of simplicity of understanding, and of implementation, there are advantages in having only a small number of tasks. This may mean that some actions are performed more frequently than absolutely necessary, thus increasing total processor utilisation. However, if the set of tasks is still schedulable then such a task allocation can be adopted.

Examining Table 10.1 we can easily see that by combining tasks **CalculateControl** and **SendToActuator** we do not significantly change the processor utilisation since both tasks have the same cycle time (there will be a small saving in context switching time). But suppose we also try to combine them with **ReadInputs** and run all three at the cycle time of **ReadInputs**. The combined task has an execution time of 0.105 seconds (minus some allowance for reduced context switching) and a cycle time of 0.1 seconds and hence it is immediately clear that we cannot use this combination (the processor utilisation is 105%).

Let us consider the position when we add in the task **CheckAlarms** and let us assume that we decide to run it in combination with **ReadInputs**, that is at 0.1 second intervals rather than 0.15 second intervals. The utilisation calculation is shown in Table 10.4.

10.6.2 Scheduling Algorithms – Pre-emptive, Priority Based

We discussed some scheduling algorithms in Chapter 6 when we dealt with operating systems. In particular we assumed that for real-time applications the scheduler would use a priority-based pre-emptive algorithm. This is the simplest and most commonly used real-time scheduler. The scheduler always selects the task with the highest priority from the set of tasks that are ready to run.

Table 10.4 Effect of combining tasks on utilisation

Task	*Cycle time (s)*	*Execution time (s)*	*Utilisation (%)*
ReadInputs	0.1	0.03	30
CheckAlarms			
CalculateControl	0.2	0.085	42.5
SendToActuator			
UpdateDisplay	5.0	0.30	6
Total utilisation			78.5

By examining Figure 10.12 which was drawn on the assumption that a pre-emptive priority scheduler was being used, we can see that the algorithm guarantees that the highest-priority task is always run on time. This is the only assertion that we can make about this algorithm. The behaviour of a system which uses this scheduling technique is dependent on the particular choice of priority structure. Even then it is non-deterministic since the behaviour will also change according to the pattern of occurrence of events. A question then is: how should task priorities be assigned?

There are two basic approaches:

1. assign priorities according to the importance of the task; and
2. assign priorities according to the cycle time of the task with the task with the shortest cycle time being given the highest priority.

In practice designers use a mixture of the two approaches. Under normal operating conditions for a control system the tasks with the shortest cycle time will also be the ones that are the most important and hence in practice the two approaches coincide. However, under abnormal conditions the importance of some control loops may be downgraded compared to others. Hence a designer may choose a fixed order of priorities that also takes into account the requirements of abnormal running and hence departs from strict compliance with approach number 2 (alternatively the designer may choose to use dynamic priority reallocation and change to a different set of priorities during abnormal running conditions).

10.6.3 Scheduling Algorithms – Other Types

The two other, most commonly advocated scheduling algorithms are:

1. earliest deadline; and
2. least slack time.

To implement either of them the scheduler needs to know the deadline for each task.

The earliest deadline scheduler, as its name implies, simply chooses the task whose deadline is closest to the current time. Be aware that it chooses from the list of tasks that are ready to run, that is those tasks whose release time is earlier than the current time.

A least slack time scheduler needs to know, in addition to the deadline for each task, the amount of processor time that the task needs in order to complete its execution. Using these values the scheduler calculates which task has the least free or slack time before its deadline.

Little advantage over the priority-based scheduler is obtained from using these algorithms for systems which comprise mainly periodic tasks. However, they perform significantly better for systems with mainly aperiodic tasks as they do not involve the use of any prior assumptions on the rates of occurrence of the events.

The major weakness of the algorithms is that they operate only on the current state of the system and do not look ahead; hence under transient overload conditions decisions may be made in a non-optimum way.

10.7 PRE-RUN-TIME SCHEDULING

For control applications and in some other forms of hard real-time systems there is frequently a requirement for a task to run at exactly *T* second intervals. This is a constraint that applies to the control algorithms described in Chapter 4. The on-line schedulers described above seek a schedule that ensures that a periodic task runs once per time interval. For example, the priority-based scheduler can ensure that one task, that given the highest priority, will run at exactly *T* second intervals but no other. For small, hard real-time systems a method which is frequently used to provide precise scheduling for the critical tasks is to build into the scheduler a precalculated schedule.

One method of doing this is to construct a table with the number of columns equal to the number of tasks to be scheduled and the number of rows equal to the lowest common multiple of the task cycle intervals. (The size of the table can be reduced if the cycle intervals are expressed in terms of their greatest common factor.) Each time the scheduler runs it first checks if it is time to read the table, and if it is it reads the appropriate row. It selects those tasks for which the entry in that row is a '1' and runs them. The scheduler uses a counter to keep track of which row it is to read and the counter is reset when all the rows have been read and the sequence restarts. Depending on the ratio between the basic clock tick and the GCF of the cycle times for the task, the scheduler may check the table each time it is entered or only at some multiple of the basic tick.

For example, consider a system with four tasks *A*, *B*, *C* and *D* with cycle times of 4, 5, 10 and 15 ms respectively. The GCF is 1 and the LCM is 20; hence we need a table with 20 rows as shown in Table 10.5.

As will be seen from the table the approach does not guarantee exact cycle times at every task invocation since in row 16 we find that two tasks *A* and *B* are scheduled

Table 10.5 Task scheduling table

	A	*B*	*C*	*D*
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	1	0	0	0
5	0	0	0	0
6	0	1	0	0
7	0	0	0	0
8	1	0	0	0
9	0	0	0	0
10	0	0	0	0
11	0	1	0	0
12	1	0	1	0
13	0	0	0	0
14	0	0	0	0
15	0	0	0	0
16	1	1	0	0
17	0	0	0	0
18	0	0	0	0
19	0	0	0	0

to run at the same time and this occurrence will occur on every fifth invocation of task *A* and fourth invocation of task *B*. By using smaller scheduling time intervals and a much larger table it is possible to reduce the frequency at which tasks coincide but not to eliminate it entirely.

10.8 SCHEDULING – INCLUDING TASK SYNCHRONISATION

So far we have assumed that all the tasks are independent, that is there are no mutual exclusion or precedence relationships between the tasks. Of course in general this is not the case. Introducing these constraints increases the complexity of the problem greatly and methods for determining predictable schedules for a hard time constraint, real-time system with task synchronisation are the subject of much research. There are serious doubts about the use of on-line scheduling for hard time constraint systems and most work is being done on pre-run-time scheduling. The main techniques that are being used are:

1. Model-based techniques, for example Petri net models.
2. Temporal logic and extended states machines.

3. Algorithmic techniques (Xu and Parnas, 1990; Shepard, 1991).

One interesting approach that effectively avoids the scheduling problem is to divide each task into small segments for which the execution times are roughly equal (within an order of magnitude). Critical sections of code must not be split, all intertask communication must be by messages and all synchronisations between tasks must have time-out conditions attached. The segments are prioritised, usually in groups, and the scheduler simply executes the segments in order. That is, it looks at the highest-priority group and executes any segments that are waiting; if there are none it then looks at the next priority level and so on. No event-based tasks are permitted; they are turned into periodic tasks which are used to poll the source of the event. Because there is no pre-emption, each segment, once it starts, runs to completion. The behaviour of the system is predictable (on a worst case basis since the execution time of some tasks will depend on the value of inputs to those tasks).

10.9 SUMMARY

A major weakness of all the system development methodologies that we have examined is the lack of analysis tools. Without such tools the system designer has no means of evaluating design decisions. Resource allocation – which includes partitioning into tasks and scheduling decisions – is obviously a vital area for hard real-time systems. It is no use deciding on a particular task allocation if the resulting task set cannot be scheduled in a way that meets the time constraints.

In this chapter we have briefly examined some of the approaches to modelling and analysing systems for the purposes of evaluating design decisions. Some techniques – processor utilisation, for example – are simple and easily carried out but the information they offer is limited; other methods are more complex. Methods for carrying out the analysis of schedulability and evaluating the safeness of systems are being developed rapidly as are tools to support the methods. Information on techniques and tools, including simulation tools, can be found in Berryman and Sommerville (1991), Harel *et al.* (1990), Liu and Shyamasundar (1990), McCabe *et al.* (1985) and Pressman (1992). Such tools will gradually be added to CASE environments.

The problems become more difficult when distributed systems are used. Information on such systems can be found in Burns and Wellings (1990) and Levi and Agrawala (1990).

Some important points to remember are:

- Petri net models can be used to find out if a system can enter an unsafe state.
- Predictable performance is more important than efficiency in hard systems.
- Processor utilisation calculations give a simple check that can demonstrate immediately if a system cannot be scheduled. The check does not prove that it can be scheduled except for a particular limited set of conditions.

EXERCISES

10.1 Draw a Petri net diagram to represent the state transition diagram shown in Figure 9.6.

10.2 Plot a graph showing how the maximum processor usage for schedulability changes with the number of tasks (n) assuming the rate monotonic scheduling algorithm.

10.3 A system contains four tasks, A, B, C and D. A, C and D are cyclic tasks with periodic times of 0.2, 5 and 0.5 seconds respectively; and B is an aperiodic task with a response time of 0.3 seconds. The execution times for A, B, C and D are 0.08, 0.03, 0.9 and 0.03 seconds respectively. Find the free processor time. Can the tasks be scheduled? What would be the effect of (a) doubling the processor speed; and (b) halving the processor speed?