An Extended Data Flow Diagram Notation for Specification of Real-Time Systems

BY

Mohammad Nejad-Sattary

Department of Computer Science,
City University.

March 1990.

This thesis is submitted as part of the requirements for the degree of Doctor of Philosophy.

تقدیم به پدر و مادر عزیزم

To my parents whose love, support and guidance have been my greatest encouragement throughout my education.

Contents

List of Figures

Acknowledgements

There are a number of people I would like to thank for their help during my research. Thanks are due to Mr. Philip Winterbottom for his many useful suggestions during the implementation exercise. I would also like to thank Dr. Lee McCluskey for his help during the development of diagram syntax and semantics. Thanks are also due to Mr. Darren Whobrey for his kind review of the first draft, and to Mr. Paul Anderson for his many useful suggestions for typesetting this thesis. I am greatly indebted to Messrs David Bolton and David Till for their numerous suggestions and helpful remarks during the course of my research. My final and greatest thanks go to my supervisor, Professor Peter Osmon, without whose initial direction and continuous support, this research would not have been possible.

Declaration

ABSTRACT

The rapid demand for industrial automation has resulted in the development of very large systems. The development costs for such systems have highlighted the importance of a staged methodical approach to system development. One of the starting stages is the derivation and expression of *system specification.* Because it takes place very early in the development cycle, the techniques used to aid in deriving a specification should not only help system developers in recognising and resolving system requirements errors, they should also help in presenting those requirements clearly.

This thesis is concerned with the specification of a specific class of systems: *real-time systems.* After elaborating on what the terms "specification" and "real-time system" mean in the context of the thesis, it is proposed that the communication power of the notation used for specification plays a central role. General diagrammatic representation of engineering plans are then identified as one of the most desirable and communicable forms of such plans. A popular notation, used in the specification of data processing systems, is then briefly discussed, in order to identify its limitations for real-time system specification. Despite those limitations, its popularity is a strong incentive for extending the notation instead of inventing a new one. Two of the currently used extensions to this notation are then presented, and their main shortcomings are highlighted.

An alternative extension is then proposed, which attempts to overcome these shortcomings. It does so by separating the data and control interfaces of a system into complementary diagrams. Because real-time system behaviour is control dominated, the notation concentrates on this particular system feature by breaking it down into two categories: control over groups of system components, i.e. the conditions under which each group is enabled and disabled to perform its overall task, and control over individual system components, i.e. the condition under which each component is activated to carry out its (sub)task. The notation's constructs allow both types of control to be specified, without hindering the specifier, and in a fashion which highlights both low level concurrency (among individual components) and high level concurrency (among component groups). Special attention is also paid to the importance of synchronisation and temporal events by providing notational means for specifying both. These extensions are illustrated through a specification exercise before discussing issues related to the notation. Some comparisons are then made with four other approaches to system specification, before highlighting the more novel features of the notation and outlining possible future extensions to the work presented here.

Chapter 1

Introduction

1.1 Overview

The system life cycle from the conception of the system's purpose to its eventual implementation and maintenance has been the subject of much research. This is a direct result of the increasing demand for system capabilities by users and the consequent growth in system size and complexity. This rapid expansion has meant that the traditional informal interactions between users, analysts and implementors are no longer sufficient to ensure user satisfaction with the system on its delivery. The imbalance between maintenance and development costs [LL87] and the resulting user dissatisfaction, for many systems currently operating in industry, provides ample evidence for the case against such informal approaches, and has prompted system developers to rethink their approach to system development.

The system development community has, therefore, looked into the more established engineering disciplines to find better techniques. These disciplines have long standing planning mechanisms for deriving a physical design from customer requirements. This usually involves expressing customer requirements in some form, which is communicable between the engineers and customers, and which can be validated by the engineers before any actual physical implementation. The obvious advantage of this approach is the engineers' confidence in their design prior to actual construction. Following in those footsteps, system developers have developed a variety of development methods, which attempt to alleviate many of the problems associated with their existing development approaches, by providing notational aids for drawing plans of proposed systems in much the same way as traditional engineers do.

Moreover, the steps taken from the decision to create a new system to its actual implementation have been divided into a sequence of stages. This is aimed at further aiding the development process by separating different concerns into these stages. The resulting *life cycle* phases differ slightly amongst developers, but they approximately fit into the following model [RPTU84], often referred to as the *waterfall* model.

- *requirement statement*: the envisaged system's requirements are explored and stated.
- *specification of requirements*: a precise specification of system requirements is stated.
- *design*: the mechanisms through which the specified system behaviour is to be achieved, in a specific operational environment, are derived.
- *implementation*: the design is realised.
- *testing*: the implemented system is tested to ensure correct operation.
- *maintenance*: the system is modified as a result of the discovery of errors, omissions and consequently modified requirements.

Once a decision has been made to build a new system, its development begins by a statement of user requirements and expectations. This is normally in natural language and includes many user concerns including organisational issues, performance constraints, budget limitations, contractual detail, and possible restrictions on the implementation environment. The specification stage is concerned with stating these requirements more precisely by discovering and eliminating the errors in the requirements document, and arriving at an implementation independent statement of system requirements, to which implementation dependent constraints are appended. The resulting specification is passed onto the design stage, where restrictions imposed by a particular implementation environment are taken into account. The design is then passed to the implementation stage. Testing is intended to discover and remove any errors present in the implemented system. During the remainder of a system's useful life further discovery of errors and modified user requirements may prompt additional changes to it. These are included in the maintenance stage.

The outcome from the activities of each stage is documented and carried to the next, which in turn processes it and passes the resulting document to its next stage. Two additional activities which take place throughout the life cycle are *verification*, i.e. checking that the outcome of each stage is compatible with its input, and *validation*, i.e. checking that the contents of the document are compatible with user requirements.

The overall aim is to arrive at a system implementation which has the minimum number of specification, design, and operational errors, and behaves as expected by its users, thereby resulting in low maintenance costs. The path taken from the requirements specification to the final implementation should itself be resilient against errors, while providing a cost effective route.

Research literature suggests that errors detected in the earlier stages of the life cycle will result in substantial savings in manpower, time and their associated cost to the delivered system [RJ77]. It is, therefore, of paramount importance to detect errors as early as possible, in order to stop them from propagating to the later stages of development [DR79, WFP83]. Furthermore, since evolutionary changes in the user requirements are inevitable with all but the most trivial of systems, the outcome of each stage should also be *modifiable*. The structuring of specifications, designs and implementations is a major tool in minimising this aspect of the maintenance.

In order to provide quality products, which are resilient against changes in system requirements, and to minimise design flaws due to human error, more and more system designers are turning away from ad hoc in house methods, and are adopting specific proven methods suitable for each stage of the system life cycle. One such set of methods is aimed at the specification stage. By applying a chosen method to the specification of a system, analysts not only reduce the possibility of human introduced errors into the design, they also benefit from other advantages, such as proven techniques like hierarchical design, provided for them by the method. Hence the risk of the presence of errors in a system specification is reduced, resulting in easier and cheaper maintenance of the resulting system.

This realisation has lead the industrial and academic communities to explore various approaches for expressing system requirements. Many methods and notations have emerged. Each one has its roots in a particular discipline, ranging from mathematically based rigorous notations such as CSP [Hoa78, Hoa85] and CCS [Mil89], through object oriented approaches [Boo86, Mey88, SM88, PCW85, Bat87], finite state machine based approaches [HLN+88, Zav85a, Alf85], logic based approaches [PFAB86, Mai86. CFG+85, BEF+86], and functional decomposition approaches [DeM78, You89, WM86, HP88], to control modelling [Pet81]. Each of these approaches is aimed at a particular set of stages within the system life cycle. Some attempt to cover the the whole life cycle by combining a methodology with notational tools, while others provide a notation and leave its use to the discretion of system developers.

1.2 Scope of This Thesis

This thesis introduces a new notation for the *specification* of *real-time systems*. The terms "specification" and "real-time system" are both currently used in many fields of computer science. Usages have different interpretations according to the context. Before elaborating further on the scope of this thesis, the implied meanings of these terms are outlined.

1.2.1 What Is Specification?

The specification stage of the system life cycle is concerned with deriving the operational characteristics of a system from a requirements document. This is a non-trivial task because of the potential complexity of the system's operations. In addition, the stated requirements are often vague, and because they are stated in natural language, many ambiguities and inconsistences are hidden within the document. In some cases users may even be unsure of exactly what they require, which results in the additional problem of incompleteness in the requirements document. System development is partly concerned with discovering the problem as well as the solution [Som89].

The analysts' aim during the specification stage of the system life cycle is to detail the system behaviour independently of *real-world* concerns such as particular implementation factors. That is to say, they should only be concerned with *what* the system is to do, and not *how* its functions are achieved. Moreover, the analysts have to discover errors and ambiguities in the requirements document and resolve them while deriving a system specification. This description of system behaviour has been targeted by many authors as one of the most vital activities in system development. It has been referred to as *system specification*, *the logical system model* [DeM78], and *system essence* [MP84] in the literature.

To help analysts in achieving this goal, a *perfect* operating environment (world) is assumed. In such an (imaginary) environment the implementation technology enforces no bounds or restrictions on system operations. This allows the analysts to concentrate solely on *the problem*, without concern about the limitations which a particular *solution* to that problem may have to cater for. These matters are delegated to the later stages of system design and implementation. Non-functional system requirements, such as reliability and performance, are therefore stated along with the system specification, so that the designers and implementors can take them into account in the latter stages of the system development process.

1.2.2 What Is a Real-Time System?

The term "real-time system" has been defined by a number of authors in the literature [All81, AZ87, HP88, WM86, ONR87, Sta88]. Systems which satisfy these definitions fall into a number of categories, but they have a number distinctive features in common. Rather than giving a rigid definition for real-time systems, the distinguishing characteristics of the class of systems which behave in a real-time manner can be identified.

As implied by the term, a *real-time* system is expected to interact with its environment within certain timing constraints. The latter include response time constraints, i.e. given a set of inputs the system must produce a set of outputs within a time slot, as well as time scheduled operations such as regularly executed tasks, and activities carried out at specific points relative to a (conceptual) clock.

Moreover, real-time systems have the property that past and present events, both external and internal, change their behaviour pattern [HP88]. These changes are more fundamental than producing a different output value from a set of input values. They often require a change in the system behaviour which may include stopping and/or starting a subset of the system's operations. In other words, a real-time system reacts to events to affect the environment in which it is operating. In order to do so successfully, it must be responsive to changes in its environment.

Therefore, a real-time system is one whose behaviour is determined by the condition of its internal and external states[1], and whose responses to these conditions must occur within predetermined timing limits. This definition embraces a large number of systems with differing characteristics, which may be composed of both hardware and software components. It does not include *on-line* systems, which are also referred to as real-time systems in some of the current literature. These are interactive data processing systems that require fast response times. Our definition of the term "real-time" is targeted at systems, such as process control systems, which are usually part of large operational environments and whose temporal response to events is often a critical factor in the overall behaviour of the whole operational system. Such systems are often referred to as "embedded systems" in the system development literature [Zav82].

1.2.3 The Specification Approach

Having established the type of system targeted for specification, the approach taken in deriving such specifications is now discussed. Although the nature of a system is a major factor in determining the way in which its behaviour is described [Col84], a number of common criteria can be used when judging the suitability of a particular method. In particular, the vehicle (notation) used by the method to convey the information gathered is one of the most important factors when choosing a specification method.

The first and most obvious criterion is that the notation has the capability to describe all aspects of the subject matter. A method and its associated notation may be very well suited for specification of one class of system, but less usable for specification of others.

[1]In computer science the term *state* is often used to refer to the values stored within a program at any moment in time. It is also used in the literature to imply the whole operational status of an executing system at any point in time, including both the internal stored values and the current control conditions. The latter interpretation is used throughout this text.

Second, and perhaps the most important criterion of all, is the communication power of the notation used [Koo85, Was80]. Specification is only one stage in the system life cycle. The majority of systems, except the most trivial, usually engage different groups of people at different stages. These groups include system users, analysts, designers and implementors. Although these groups may not be disjoint, i.e. some people may belong to more than one group, it is necessary for the product of each stage in the life cycle to be an effective communication medium for the following stage. A notation's usefulness in this sense is what we mean by communication power.

A number of approaches have been applied to system specification ranging from unstructured natural language to rigorous mathematical notations such as CSP. Natural language is a highly communicable notation, but may contain or introduce undetected errors. Mathematically based notations, on the other hand, result in specifications which have proven properties such as consistency, but they cannot be used as effective communicable media between technical and non-technical people, since the latter may be unwilling to accept a notation which requires mathematical knowledge [BEF+86]. A specification notation is needed which is both precise and capable of providing an effective communication medium between the groups involved in system development.

Diagrammatic presentation of information is one of the most convenient and effective forms of communication between people [LS87, MM85], and between people and machines [Cha89]. Plans and maps have been used in the conventional engineering fields, such as civil and mechanical engineering, for a long time. The well established conventions for drawing such plans and their universal use by the engineering community shows that well defined diagrams can provide the best tools for conveying information between groups of people.

A further desirable property of a development strategy is its ability not only to guide analysts in deriving complete and unambiguous specification, but also to help in deriving it. It has been suggested that the human brain is only capable of concentrating on a small amount of information at any one time [Mil56]. Many advocates of system development strategies have therefore recommended a hierarchical approach [DeM78, Har87, EFRV86, Bat87]. This can help not only when deriving the specification, but it also provides analysts with a mechanism to present it in a comprehendable form to other people involved in the development.

Data Flow Diagrams (DFD's) have been in use in the data processing industry for a long time. They make up a substantial part of many methodologies including those used in information system's specification and design [DeM78, PJ88, WPSK86, BOT85], knowledge-based design [LH87], parallel processing [IOM+85, II85, II82], the design of a command interpreter [DS84], direct code generation [OWW85], object-oriented software design [Bai89, War89], and the development of real-time systems [WM86, HP88, You89, Fra85, Gom84, MJAS85]. The simplicity and easy use of DFD's has made them popular for showing system data interfaces, and many analysts are already familiar with them [Bai89]. This implies that there is already a substantial investment in using and understanding such diagrams by different groups of technical and non-technical people. The communication power of diagrams in general, and the popularity of Data Flow Diagrams in particular, imply that rather than inventing a new notation for specification of real-time systems, the DFD notation should be augmented to cater for such specifications.

1.3 Plan of The Thesis

The next chapter briefly describes Data Flow Diagrams and their use in describing a system's data interfaces. It goes on to outline why traditional DFD's are unsuitable for real-time system specification, implying that they must be extended to provide a notation suitable for specifying such systems. Two of the extensions, currently popular in industry, are then outlined before describing their shortcomings. The chapter concludes by showing the need for a new and better notation.

Chapter 3 concentrates on the new extended DFD notation. The symbols of the new notation are presented first, and their is use demonstrated through a worked example. The rules for forming these diagrams are then informally outlined.

Chapter 4 discusses the issues relevant to the notation. The chapter begins by outlining a set of objective criteria for selecting simple system processes. The next section discusses how system timing requirements are stated in a specification. The new notation is then claimed to be a language for programming in the large. The syntax and semantics of this language are then discussed. The usefulness of animating specifications is highlighted, and how it can be achieved for specifications in the new notation is shown. The next section discusses how specification quality may be judged. The process of transforming a specification to a design and implementing it in a particular environment is then illustrated through the experience gained from implementing an example specification. Some methodology guidelines are then given. The chapter ends by giving an overall conclusion.

Chapter 5 compares the notation with four currently popular notations, each of which has a different approach to system specification/design. Each section starts by briefly describing the particular notation and its use in system specification, before drawing some conclusions by comparing it with the new notation. The chapter ends by drawing some overall conclusions about features of the four notations discussed as compared with the new notation.

The concluding chapter outlines the importance of specification in the system life cycle. It goes on to describe some of the more novel features of the new notation. Extensions to the work presented in this thesis are then discussed, before drawing an overall conclusion from the research.

The development of the notation was guided by a number of example specifications. Those, other than the one given in Chapter 3, are shown in the first Appendix. Each exercise starts by giving the requirements for the example system, before presenting the hierarchy of diagrams in its specification.

Appendix B gives the complete specification and implementation code for an example system.

The final Appendix gives an abstract syntax for the notation. It also presents an alternative textual equivalent for the diagrams of the notation.

1.4 Glossary

The implied meanings (in this thesis) of the terms "real-time system" and "specification" were given above. There are a number of other terms, used in this thesis, which have been used in the literature for a variety of purposes. The short glossary below gives the implied meaning for each of those terms, when used in the following text.

- **System Development:** This term refers to all the stages involved in developing an operational system, starting from requirements capture through to implementation and testing.

- **User:** A person who has an interest in the final product of the system development process. This includes people who commission the system development, as well as those who will interact with it once it has been implemented.

- **Requirements document:** The requirements document is the product of the development stage, immediately preceding specification. It is a statement of what the users require the system to do.

- **Analyst, Designer:** These terms are used to refer to the people who carry system specification and design, respectively.

- **Host environment:** The host environment for a system identifies the physical entities that form part of the system implementation. This includes hardware, software, mechanical devices and other mechanisms such as manual tasks performed by people.

Chapter 2

Background

2.1 Overview

In this chapter the background to the original work described in this thesis is presented and discussed. DeMarco Data Flow Diagrams [DeM78] are briefly presented first. The following section outlines their unsuitability for specifying real-time systems. Two notations, currently used in industry, which are the direct predecessors of the notation outlined in this thesis, are then presented. These notations were put forward by Ward and Mellor [WM86] and Hatley and Pirbhai [HP88]. Finally their shortcomings are discussed to indicate the need for an improved notation for specification of real-time systems.

2.2 Data Flow Diagrams

Data Flow Diagrams [DeM78, GS79] are used, as part of many existing specification approaches, to show the data interfaces within a system and between a system and the environment within which it operates. DFD's take a functional viewpoint of systems by decomposing a system into a network of *processes*. Much of the successful use of DFD's is due to their ease of understanding and use in describing system data interfaces. This is shown in a hierarchy of system processes starting with the topmost view of a system, where it is viewed as a single process with data connections to its environment.

Pictorially, a process is represented by a circular named (and numbered) symbol and system environmental entities are shown by named rectangles. The latter are referred to as *sources*, *sinks*, or *terminators*. Each connection to the environment is called a *data flow*, which is represented on DFD's by a named directed arc connecting the system process to an environmental entity. The direction of data exchange is shown by an arrow head at the receiving end of the data flow.

The diagram showing a system's data interfaces to its environment is called a *Context Diagram*. Using an incremental specification strategy the system process on this diagram is expanded into a network of processes connected by data flows. These processes can also communicate via stored data, i.e. data that is occasionally updated but used many times by system processes. Data stores are an abstraction of the data a system remembers, whereas data flows are abstractions of direct (asynchronous) communications between system processes. In other words, data flows represent a temporary buffer for data, while data in stores represents the parts of system data which linger until overwritten or deleted. Stored data is shown by a pair of parallel lines on a DFD. The name of the data store is placed between the lines. The symbols of (DeMarco) Data Flow Diagrams are shown in figure 2.1. The construction of lower level DFD's is governed by *balancing rules*, which require the a DFD's inherited data flows and data stores should be those connected to its parent process and vice versa.

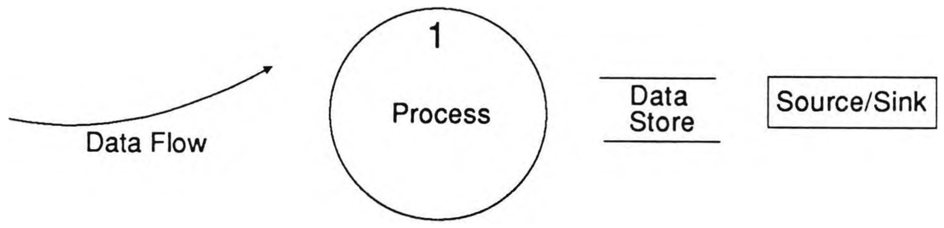


Figure 2.1: The Symbols of Data Flow Diagrams

The number of items (processes and stores) on each diagram is kept to a guideline of seven (plus or minus two) in order to control the complexity of each diagram. This will result in clearer diagrams that are easy to understand [Mil56]. The subdivision of (complex) processes is continued until all the (*leaf*) processes are small enough to be described by a small piece of text, typically no more than a page. Each such specification is called a *minispecification* or a *minispec* for short. The diagram hierarchy is accompanied by a *data dictionary*, which holds the definitions of the data flows and stores of the system. Each data flow entry shows the decomposition of the data carried by the data flow. Each data store entry contains a similar decomposition of a single record to be stored in the store as well as access key identifiers.

The diagram hierarchy together with the minispecs of the leaf processes and the data dictionary specify the complete flow of data through a system.

2.3 Extended Data Flow Diagrams

Data Flow Diagrams are quite adequate for specifying the behaviour of the class of systems where the flow of control through the system is largely determined by the arrival of data, i.e. where system process execution sequences are governed by the availability of data to operate on. This is typical of data processing systems, for example. However, notations based around Data Flow Diagrams lack the means to model systems that include control via events other than those implicitly associated with the arrival of data. In real-time systems, for example, many of the inputs to the system are signals that indicate the occurrence of some event, e.g. a critical condition has occurred. These inputs do not pass any data to the system to be processed. Frequently they indicate a change in system behaviour. That is to say, real-time systems are event driven [WL85] and their specification requires a clear definition of system stimuli [Bai89].

Distinguishing between data and event inputs to a system will help in understanding and describing the operational behaviour of a system. One way to achieve this distinction is by using specific naming conventions for data flows [TRH87]. This restricts the freedom of analysts when naming data flows, and gives a dual purpose to the data flow label. It no longer only identifies what the data flow carries; the type of information carried by the flow is made explicit by its name. This is clearly undesirable.

Furthermore, many systems are made up of subsystems. These can readily be identified as groups of closely related leaf processes, which share a common control structure, in the DFD process hierarchy. In other words, a subsystem is an abstraction of a group of processes which operate over iterations of a stream of data or event tokens. Any such group of processes may be active or inactive at a particular time during system operation [All81]. This is typical of real-time systems. The ability to enable and disable parts of a system, when certain events have occurred, is often a part of the operational requirements for that system. Early notations based on Data Flow Diagrams are unsuitable for modelling subsystem control. Specifying a control structure at subsystem level is at the very best cumbersome, and the very worst nearly impossible with early data flow diagram methodologies and notations.

It is, therefore, clear that traditional Data Flow Diagrams cannot easily be used for writing down elegant specifications for the class of systems whose control structure is not simply governed by the availability of data, and whose input events may cause some parts (subsystems) to be activated and deactivated.

2.4 The Transformation Schema

In order to make Data Flow Diagrams more suitable for the specification and design of real-time systems, Ward and Mellor [WM86, War86] introduced a number of notational extensions. They called the resulting notation *the transformation schema.*

2.4.1 Notation

The basic DFD symbols are retained in the transformation schema. Processes, renamed *data transformations*, keep their circular symbols, and data flows are represented by arcs joining processes and data stores, which are shown with parallel lines. The data flow notation is extended to enable the representation of joining, merging, splitting and copying of data flows, figure 2.2. The interpretation of a data flow is determined by its labelling. Figure 2.2(a) shows how two pieces of data can be joined together, i.e. Z = X + Y (where '+' is used to indicate combination of data tokens not the sum of X and Y; this follows the convention used in DeMarco style data dictionaries). Part (b) shows the merging of two data pieces: X can be provided by either of the incoming data items. In part (c), a data item is split into a number of parts, Z = X + Y. Part (d) shows how multiple copies of the same data item can be sent to a number of data transformations.

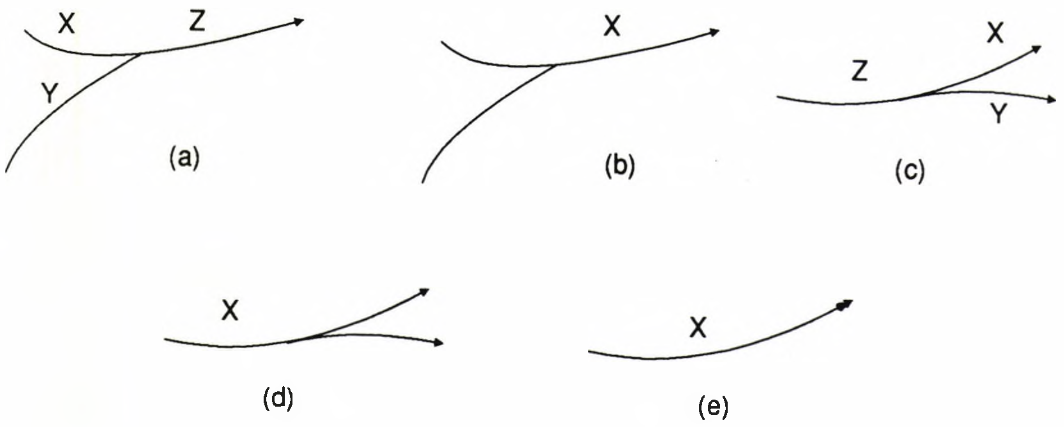


Figure 2.2: Data Flows of The Transformation Schema

The transformation schema also distinguishes between two types of data flows: time-continuous and time-discrete data flows. Time-continuous data flows represent inputs to systems that continually vary over time. Typical examples are readings from the system environment such as temperature and pressure. Such data are available to the system all the time, but their values are only of interest at certain points during system operation. Unlike time-continuous data flows, time-discrete data flows represent data that is available to a system at certain points in time. This is roughly equivalent to the notion of a transaction in the data processing terminology [WM86]. Time-continuous data flows are shown with a double arrow head in the transformation schema, figure 2.2(e).

As well as slightly altering the data flow notation, the transformation schema introduces a new set of symbols to represent the flow of control through a system. Ward and Mellor [WM86] note that most real-time systems contain some flows that have no content; they are simply signals that indicate something has happened. These are the events that are exchanged between system processes (transformations) and between the system and its environment. Events are shown in the transformation schema by dotted arcs. Ward and Mellor identify three types of events: *flow-direct* events are those associated with the arrival of data (discussed below), *flow-indirect* events are generated by the system transformations when a specific condition has been satisfied, and *temporal* events signal the passage of time. There is no notational distinction between the latter two; both are shown with the dotted line symbol. Flow-direct events are not explicitly shown on the transformation schema.

A transformation that accepts only event flows and time-continuous data flows as inputs and produces only event flows as outputs is called a *control transformation*, and is represented in the transformation schema by a dotted circle. There is also an analogue of a data store called an *event store*. It is used to remember the occurrence of event flows, and is represented by parallel dotted lines. The control symbols of the transformation schema are shown in figure 2.3.

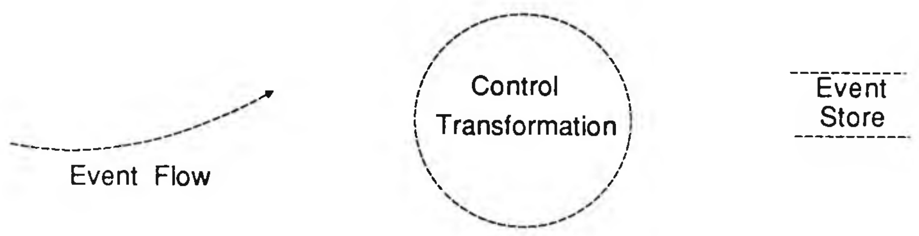


Figure 2.3: The Control Symbols of The Transformation Schema

2.4.2 Deriving a Specification

Ward and Mellor's system model has two parts: the first part defines the system interactions: the *environmental model*, and a second part which describes the required behaviour of the system: the *behavioural model* [WM86]. The environmental model can be divided into two parts: a description of the boundary between the system and its environment, showing the interfaces between the two parts, and a description of the events that occur in the environment to which the system must respond. The behavioural model also consists of two parts: the transformation schema and the data schema. The transformation schema denotes graphically the transformations that operate on flows that cross the system boundary and is the active portion of the system that responds to environmental events. The data schema denotes graphically the information that must be remembered by the system.

Specification of the system starts by deriving the environmental model. The environmental terminators of data and events, i.e. data and event sources and sinks, are first identified. These are the entities the system interacts with. The data and events exchanged between the system and each of these entities are then determined. The nature of each data flow exchanged between the system and its environment is examined to determine whether it is a time-continuous or a time-discrete data flow. A system's environmental interface is shown in a *context schema*. In a similar way to a DFD context diagram, a context schema represents environmental entities by rectangular boxes, and a black box view of the system shows it as a single process with data and event connections to its environment, figure 2.4 [WM86].

Once the system interface to its environment is defined, its internal behaviour is derived. The behavioural model specifies a system's required behaviour in a hierarchical set of *schema*. This is derived by using an incremental strategy to expand data transformations into sub-networks of data and control transformations, starting with the data transformation of the whole system on the context schema. An example schema is shown in figure 2.5 [WM86]. As with DeMarco DFD's, this expansion continues until the data transformations are small enough to be specified in text.

The two types of control in a system are specified as follows. Implicit control carried by discrete data flows is not explicitly modelled in the transformation schema. Instead, every data transformation at the leaves of the transformation hierarchy tree is restricted to one discrete input data flow. If a data transformation requires the data carried by more than one discrete data flow, the data must first be joined together in one data flow and then input to the data transformation.

Control over groups of transformations is specified by using control transformations. A control transformation can enable/disable or trigger a data transformation. While a data transformation is enabled, its children can respond to and transform data input to it. When disabled, inputs are ignored

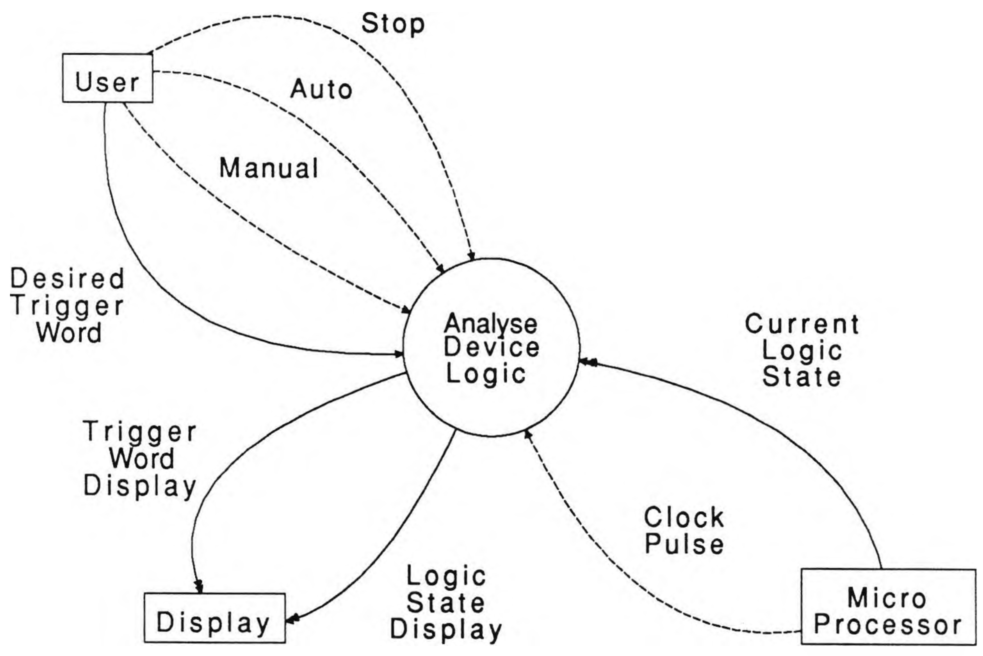


Figure 2.4: An Example Context Schema

until the data transformation is re-enabled.

A data transformation that does not have an active input, i.e. one with only data store and/or continuous data flow connections, must be explicitly triggered. Such a trigger is generated by the control transformation(s) at the same level as the data transformation. That is, control is localised to the control transformations within a schema: data transformations may exercise control external to the schema by producing output event flows, but only control transformations may prompt transformations internal to the schema.

Control transformations, unlike data transformations, cannot be expanded into a further sub-network, but like leaf data transformations, their operation must be specified. The operation of each control transformation is described by using a finite automaton model. Each state of such a finite state machine represents part of the system state. The input and output event flows of a control transformation are the *input events* and *output actions* of its finite state machine model. The occurrence of an event may cause either a change of state or the production of output or both. A change of state reflects the behaviour change by the system. This may imply enabling/disabling or triggering system processes, which is shown by corresponding output event flows from the control transformation.

Pictorially, a FSM can be represented by a *state transition diagram.* To

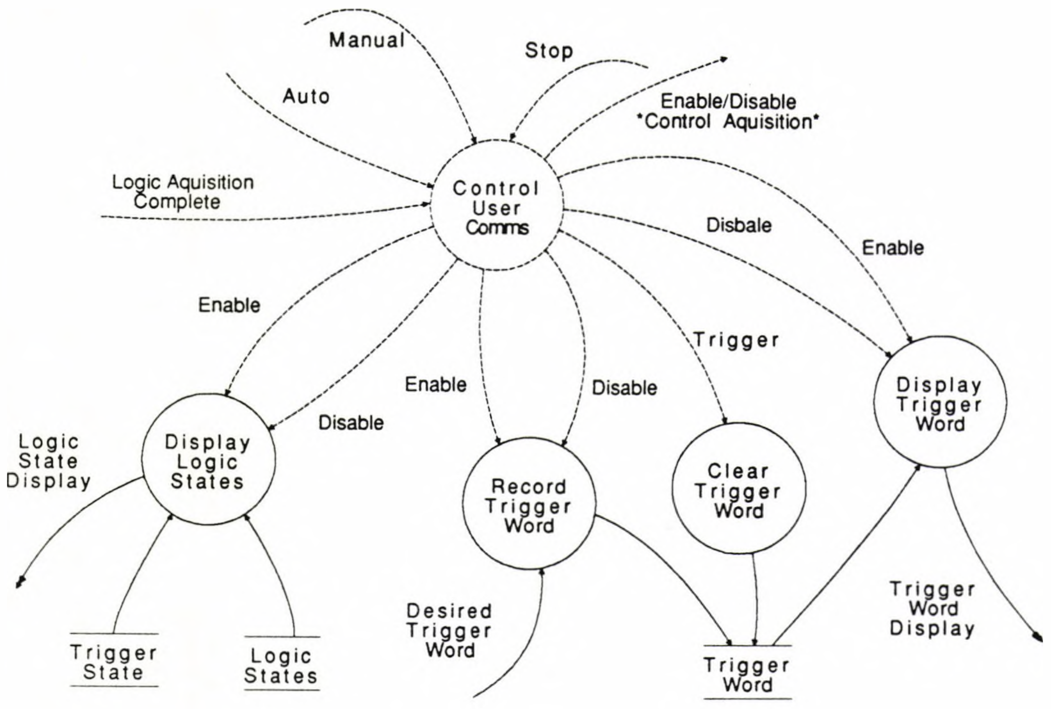


Figure 2.5: An Example Schema

avoid confusion between STD's and schema, state transition diagrams use rectangles and straight lines (instead of the traditional circles and arcs), respectively, for states and transitions. Transitions are labelled with the event that causes the state change and the transformation output. These are placed above and below a horizontal line in the transition label. Figure 2.6 [WM86] shows the state transition diagram for the control transformation of figure 2.5.

Operation of a control transformation can equivalently be specified by using a *state transition table* to show the state change and an *action table* to show the output of the transformation for each state change. This alternative representation is particularly useful for control transformations with large state transition diagrams.

Coordination of data transformations is the responsibility of the control transformation of a schema. This is achieved by the exchange of events between data and control transformations. Control transformations may exchange events for synchronisation as well as enablement and disablement. By using a levelled set of schemas with control transformations a hierarchy of control can be specified for a system.

The notation put forward by Ward and Mellor [WM86] has several additional features. The data schema, mentioned above, uses an entity relationship model to specify the layout of the stored information for a system.

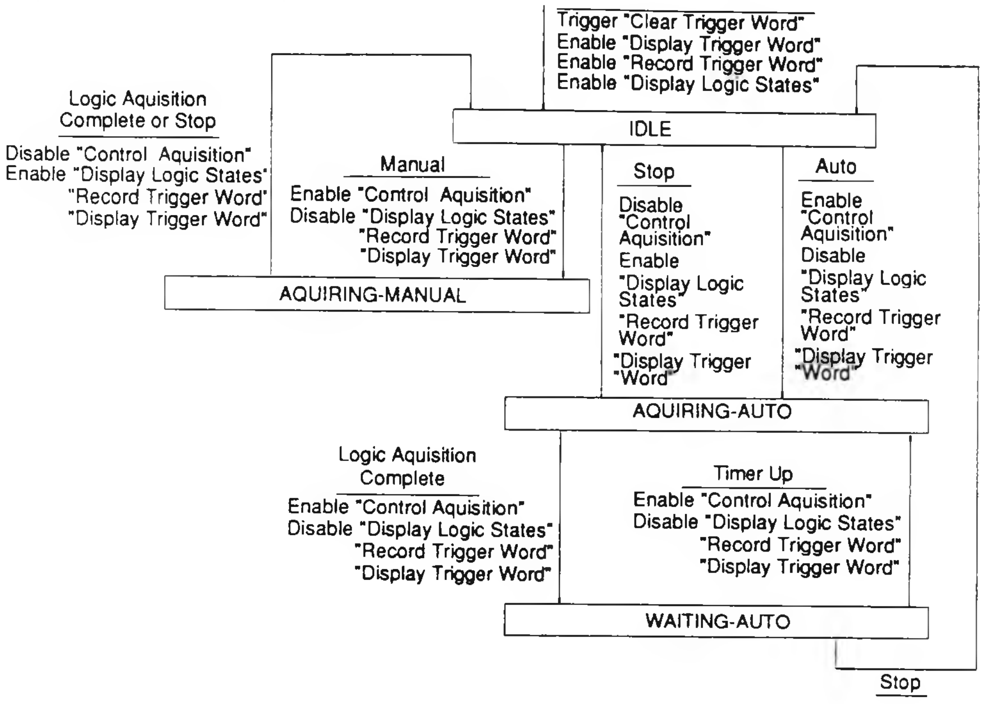

Trigger "Clear Trigger Word"
Enable "Display Trigger Word"
Enable "Record Trigger Word"
Enable "Display Logic States"
IDLE
Logic Aquisition Complete or Stop
Disable "Control Aquisition"
Enable "Display Logic States"
"Record Trigger Word"
"Display Trigger Word"
Manual
Enable "Control Aquisition"
Disable "Display Logic States"
"Record Trigger Word"
"Display Trigger Word"
AQUIRING-MANUAL
Stop
Disable "Control Aquisition"
Enable "Display Logic States" "Record Trigger Word" "Display Trigger Word"
Auto
Enable "Control Aquisition"
Disable "Display Logic States" "Record Trigger Word" "Display Trigger Word"
AQUIRING-AUTO
Logic Aquisition Complete
Enable "Control Aquisition"
Disable "Display Logic States"
"Record Trigger Word"
"Display Trigger Word"
Timer Up
Enable "Control Aquisition"
Disable "Display Logic States"
"Record Trigger Word"
"Display Trigger Word"
WAITING-AUTO
Stop

Figure 2.6: Example State Transition Diagram

Data flows and stores can be defined, in a manner similar to those in DeMarco DFD's, in a data dictionary. Once a complete specification has been derived, the schema can be executed using a technique based on the execution of petri nets: tokens are placed on data and event flows to represent the arrival of data and events and the progress of the system is observed by continuous execution of *ready* transformations. The methodology also offers a comprehensive set of guidelines to assist in going from a set of specification diagrams to a physical design.

2.5 Hatley and Pirbhai's Notation

Like Ward and Mellor, Hatley and Pirbhai [HP88] recognised the deficiencies of DeMarco Data Flow Diagrams when specifying real-time systems. Starting with Data Flow Diagrams, they introduced their own extensions to create a DFD style notation which is more suitable for this purpose.

2.5.1 Notation

Hatley and Pirbhai also retain the DFD symbols for processes, data flows and data stores. Their data flow notation has many more varieties than Ward and Mellor's, figure 2.7, but there are no semantic differences. The interpretation of each data flow is again determined by the way it is labelled.

Unlike Ward and Mellor, Hatley and Pirbhai's notation does not add the

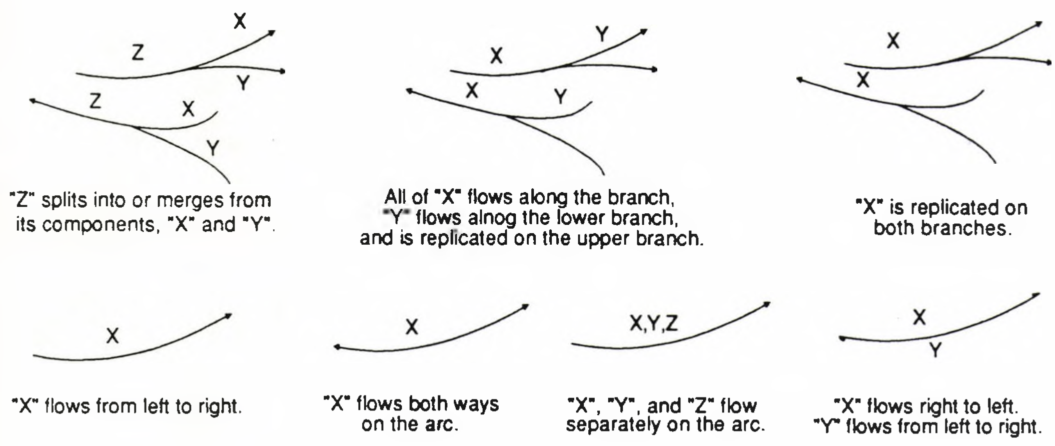

Z
X
Y
Z
X
Y
"Z" splits into or merges from its components, "X" and "Y".
X
Y
X
Y
All of "X" flows along the branch, "Y" flows alnog the lower branch, and is replicated on the upper branch.
X
X
"X" is replicated on both branches.
X
"X" flows from left to right.
X
"X" flows both ways on the arc.
X,Y,Z
"X", "Y", and "Z" flow separately on the arc.
X
Y
"X" flows right to left. "Y" flows from left to right.

Figure 2.7: Data Flows of Hatley and Pirbhai's Notation

control flow view to the Data Flow Diagrams. Instead, a diagram accompanying the DFD shows the flow of control. These diagrams are named *Control Flow Diagrams (CFD's)*. A CFD will contain a *shadow* of every process on its DFD counterpart. *Control flows* show the flow of control signals down through the system process hierarchy. These are shown by dashed arcs. Control flows, unlike event flows of the transformation schema, can carry composite values, which allow (composite) control flows to take any of the data flow types shown in figure 2.7. As a result the data dictionary, renamed the *requirements dictionary* by Hatley and Pirbhai, also contains control flow definitions.

Hatley and Pirbhai, like Ward and Mellor, use a finite automaton model to represent control at any level of granularity. Such automata are represented by a short bar on CFD's. The input *alphabet* (events) of an automaton are the control flows entering the bar symbol. Some of its outputs (*actions*) are shown as emerging control flows and others are shown in tables, see below. The control flow symbols of Hatley and Pirbhai's notation are shown in figure 2.8.

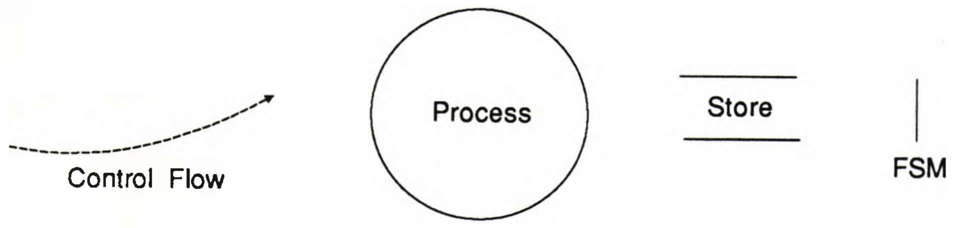


Figure 2.8: The Control Symbols of Hatley and Pirbhai's notation

Stores may also be placed on CFD's. These represent the recording of a control flow. A store may contain either data or control or both, so there is no special symbol for a control store. It is shown with a pair of parallel lines. Store definitions in the requirements dictionary contain a description of what each store holds. Unlike DeMarco DFD's, a store is only shown on a single DFD/CFD (on DeMarco DFD's and in the transformation schema a store is shown wherever it is referenced). Flows going from stores to lower level processes are labelled instead with the store name.

2.5.2 Deriving a Specification

Hatley and Pirbhai divide the design process into two parts: the *requirement modelling stage*, which derives from user requirements an implementation independent specification of the system; and the *architecture modelling stage*, which maps the specification onto a design restricted by real world constraints and implemented on specific hardware.

The system requirements are captured through an integrated model that views a system from two aspects: the information processing (functional) behaviour, and its control (state) behaviour [HP88]. These are called the *process* and *control* models and are shown on DFD's and CFD's, respectively.

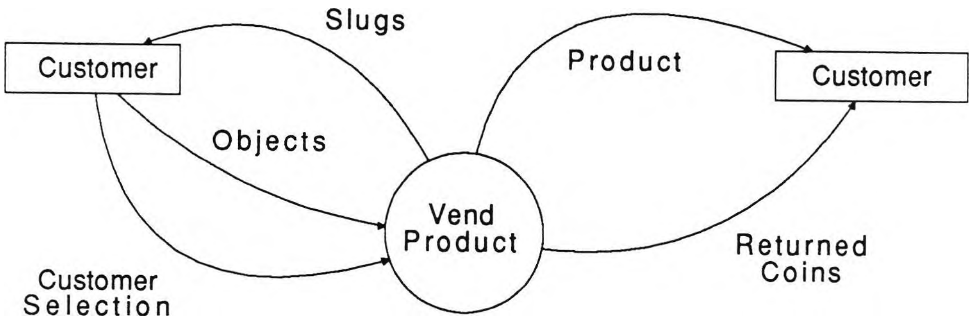


Figure 2.9: An Example Data Context Diagram

Hatley and Pirbhai start by deriving the system interface to its environment. This is shown on a pair of diagrams. The *Data Context Diagram* shows the data exchanged between the system and the entities in its environment, figure 2.9, and the *Control Context Diagram* shows the control interface, figure 2.10 [HP88].

Note that in order to reduce cluttering of diagrams, Hatley and Pirbhai allow multiple symbols to be drawn for some entities. All items with the same name represent the same entity, e.g. "Customer" in the DCD and CCD of figures 2.9 and 2.10. Other repeated symbols are stores, short bars and flows.

The requirement specification continues by incremental expansion of each process into sub-networks of processes until each process reaches the size of a *primitive* process, i.e. one whose operation can be described by a small piece

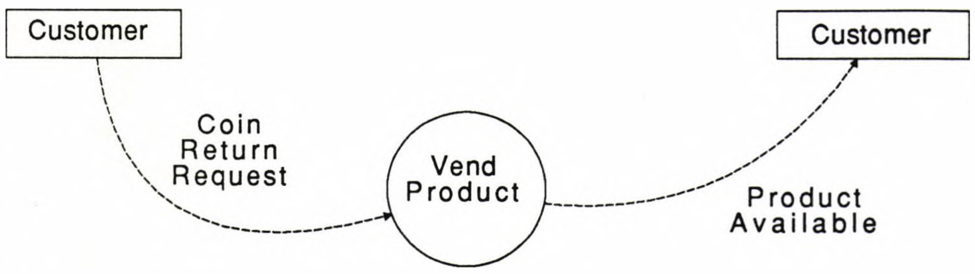


Figure 2.10: The Control Context Diagram for The DCD of figure 2.9

of text. This text is called the *process specification (PSPEC)*. At every level, the process model is first derived and shown on a DFD, figure 2.11 [HP88].

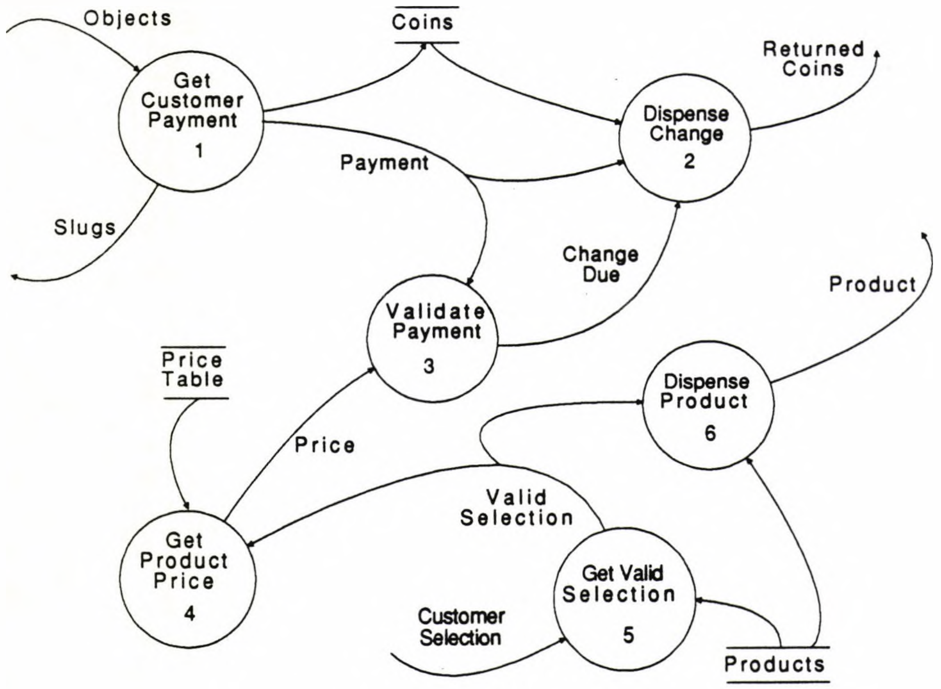


Figure 2.11: An Example DFD

The naming, numbering and balancing rules of DeMarco Data Flow Diagrams are followed and aid the easy comprehension of Hatley and Pirbhai DFD's. Once the process model has been specified, the control model is derived and shown on a CFD. A CFD is formed by first *shadowing* every process and store. These retain the same symbols as those on the DFD, figure 2.12 [HP88].

A process on the CFD does *not* represent processing of control flows entering it, nor is it activated or deactivated by those control flows. Control flow diagrams are only used to show the routing of control signals in the system (they share the naming and numbering of those on the corresponding DFD).

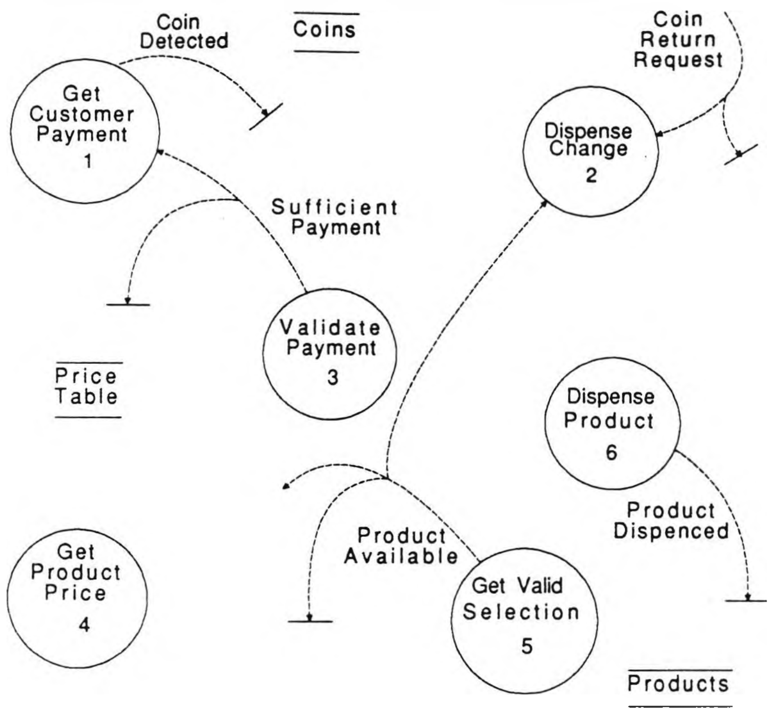


Figure 2.12: The CFD for the DFD of figure 2.11

Like Ward and Mellor, Hatley and Pirbhai do not explicitly show control carried by data, but unlike them, they do not insist on having only one discrete (active) data flow input to a process. The triggering effect of data on a process may be deduced from its PSPEC.

Control local to a level is described using a finite automaton. Pictorially this is shown by a short bar on a CFD. Although the diagram of figure 2.12 has several of these, they all represent the same finite state machine. The operation of such a finite state machine is given in an associated *control specification (CSPEC)*.

Hatley and Pirbhai divide finite state machines into two categories: *combinational machines*, where the machine output is dependent only on its inputs, and *sequential machines*, where the machine output depends not only on the current inputs to the machine, but also on the history of past inputs. The operations of these are specified by decision tables and state transitions diagrams, respectively. Decision tables list the machine output for each of its inputs. STD's are similar to those used in the transformation schema. When a FSM is too complex to be easily specified by a STD, it may be specified by a *state transition table* or a *state transition matrix*. The automaton in figure 2.12 is a sequential machine, whose STD is shown in figure 2.13.

The finite state (sequential) machines in Hatley and Pirbhai's notation differ in two ways from those in the transformation schema. First, the enabling events, from the automaton to processes, are not shown by using control flows (in the transformation schema, enabling signals are shown by con-

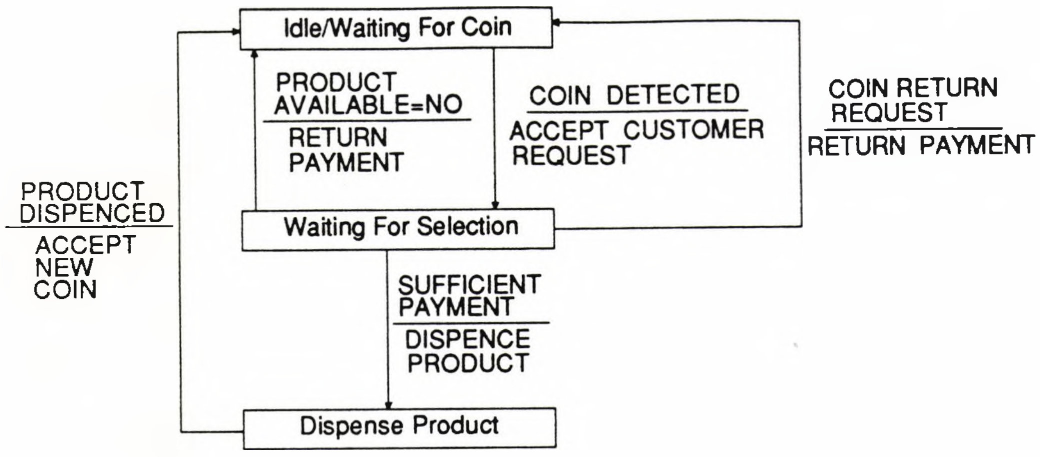


Figure 2.13: CSPEC of figure 2.12

necting an event arc from a control transformation to a process). They are shown, either on the STD in a similar fashion to those shown on transformation schema STD's, or in *activation tables*. Activation tables are used to reduce STD cluttering, e.g. figure 2.14 shows the activation table for the STD of figure 2.13. An activation table is part of the CSPEC for a FSM.

Process Activated / Control Action	Dispense Change	Dispense Product	Get Valid Selection
Accept Customer Request	0	0	1
Return Payment	1	0	0
Accept New Coin	0	0	0
Dispense Product	1	1	0

Figure 2.14: The Activation Table for the STD of figure 2.13

Second, and more important, in the transformation schema a transformation stays active until either a further (output) event from its controlling transformation deactivates it or its parent is deactivated, whereas activated processes in Hatley and Pirbhai's diagrams stay active only until the next transition. That is, if a process is to be active over two consecutive states, it must be activated prior to entering both states and, hence, processes do not need explicit deactivations. Once activated a process, i.e. its primitive children, can respond to data items until the process is deactivated by the next transition.

Hatley and Pirbhai encourage the use of combinational machines to reduce specification complexity. They also point out that in cases where a sequential machine must be used to specify a controlling mechanism, using combinational machines to generate input for the sequential machine from the input control flows and to generate output to processes and the environment from the sequential machine's output, will help in reducing the complexity of sequential machines.

Hatley and Pirbhai's notation also includes a number of other techniques. Critical system timing is defined by them to be the timing observable from outside the system, i.e. the timing between an input set and getting the corresponding outputs (internal timing is considered a design issue). This is specified in *timing specification tables.* Time (both relative and absolute) is also available to PSPEC's and CSPEC's. The architecture model provides a guide for going from the specification to a particular implementation.

2.6 Why Introduce Another Notation?

The purpose in deriving a specification for a system is to be able to write down the complete and unambiguous operational requirements for that system. The derived product must be useful for communication between the groups of people involved in the development of a system, i.e. the users, analysts, designers and implementors. In order to be useful, the specification notation must not only convey the system's operation concisely, but it must also result in diagrams that can be easily followed so that the required behaviour of the system can be understood. In order to achieve this, the notation must present clearly the data and control interfaces of the system processes, with each other and with the system environment.

The data interfaces of the system have already been the subject of much research and the successful use of Data Flow Diagrams in the data processing industry is evidence for the fact that, to present the data interfaces within a system, the notation must show the data exchanged by the system processes and the data stored by the system. These are abstracted in data flows and data stores in Data Flow Diagrams. Both of the above notations follow the established conventions of data interfaces in Data Flow Diagrams.

The motivation for creating a new notation stems from the fact that data flow diagrams are unsuitable for modelling the control structure within a system. To present a concise and easy to follow specification of the control structure of a system, it is necessary to be able to specify two kinds of control: control of groups of processes and control of individual (*leaf*) processes. The notation must, therefore, be able to show clearly the subdivision of a system into process groups (subsystems), how each such subsystem is enabled and disabled, and how each individual process is *triggered* for operation.

Two notations based on DFD's have been outlined above, but neither quite reaches the goals outlined here. Although there are notational differences between them [WK87, BJKW88], they have a common approach to specifying control over groups of processes: they both use a finite automaton model for this purpose. A finite automaton can only be in one state at any time, i.e. it is inherently a sequential machine (the term "finite automaton" is used here to refer to sequential machines such as those based on the Mealy and Moore models [TB73, Gil62, HU79]).

There are several disadvantages to this kind of notation. First, a sequential model of control may force some unnecessary sequential behaviour into a system specification. Concurrent computations are not expressed naturally by an FSM [CDK85, Har87, HLN+88]. The sequentiality inherent in an automaton model means that events can only be treated one at a time. A number of concurrent events may, as a result, have to be serialised in order to model them in an FSM. Such a serialised response may not be a part of the required system behaviour, and may only be included because the specification technique is incapable of modelling the concurrent events.

Second, finite automata notations are susceptible to combinatorial explosion in the number of states [Har87, Mir89]. For example, the behaviour of some (sub)systems may require a change of the system state after the occurrence of a number of events. If these events can occur in an arbitrary order, the number of states between the initial and final (system) states of a finite state machine representation will quickly increase so as to require at best an unreadable, and at worst an unmanageable, diagram.

Third, the only way to specify concurrent behaviour in any part of the system is to have multiple processes enabled in some states. These are specified on transition action labels of STD's and process activation tables in the above notations. Neither the number of processes enabled, nor the behaviour change caused by an event at a hierarchy level, can be read directly from Ward and Mellor Schema or Hatley and Pirbhai Control Flow Diagrams. In the transformation schema this information is given in the state transition diagram or table, but the number of enabled processes in a state cannot be realised just by looking at the current state of the system. It is necessary to look at previous states to see which processes were already enabled prior to entering the new state. The subsystem control picture is even hazier in Hatley and Pirbhai's notation. Such information is divided between the CFD, which separates the data driven processes from the others, and the CSPEC, which gives the enabling information for the latter. A process activation table gives the list of active processes in each state (processes not controlled by the CSPEC are permanently enabled). Therefore, it is not easy to deduce how each group of processes is enabled and disabled in either notation.

Although a variety of extensions to finite state machines, which alleviate some of the above problems, have been proposed for use in specifying telephone switching systems [CDK85, Zav85a, CCI84, RS82, McF82], the design of reactive systems [Har87, HLN+88, Har88, HPSS87], the design of weapon systems [Alf77, Alf85], general software specification and design [Wil77, Den77, Hol87, Sal76, Tay80], and specification of communication protocols [BZ83], these extensions cannot easily be used in conjunction with Data Flow Diagrams.

Furthermore, it is not immediately apparent from the diagrams, produced using either of the notations above, when a leaf process fires, i.e. when it is activated to perform its task. This can only be deduced after looking at a number of diagrams and text descriptions. This information can be found more easily in the transformation schema because it restricts each data driven process to a single active data input. Once enabled, such a process is driven by its single active input. Processes with no active inputs are triggered by control transformations. The triggering agent for each leaf process has to be extracted from its PSPEC, or from activations tables in Hatley and Pirbhai's specifications. A diagram, such as a schema or a CFD, which is intended to show the flow of control through the system, should show clearly when, and by what agent, each leaf process fires, without any need to consult other diagrams or text.

There are other less important drawbacks to these notations. Since all the control at any particular schema must go through the control transformations of that schema, the resulting diagrams can get very cluttered with event exchanges between control and data transformations. The decision to remove redundancy in diagrams has lead Hatley and Pirbhai to abandon the convention of showing a data store at every level it is referenced. A data flow inherited from a store is instead labelled with the data store name. Since data flows from stores cannot provide active input, i.e. they cannot trigger the receiving process, this convention may introduce a little (unintentional) ambiguity into the diagrams. An inherited store flow will look like an active flow, especially several levels below where the store is placed, since it is likely the diagram reader has forgotten where this flow is from. This problem may be worsened since triggering information is not included on any of the diagrams produced according to Hatley and Pirbhai's notation.

Therefore, it is clear that a specification notation suitable for use in deriving specifications for the class of systems which include concurrency, and whose control structure is dependent on events as well as data, should not be based on finite automata. A notation should also enable the reader to grasp the system control structure by means of a simple walk through the diagram hierarchy. The reader needs to be able to see how each subsystem is enabled/disabled and how each leaf process is fired for execution. To achieve this, a notation must show both the data and control interfaces of a system concisely and in a way that is easy to follow. A new notation which satisfies these requirements, and attempts to overcome the shortcomings of the two notations outlined in this chapter, is introduced in the next chapter.

Chapter 3

The New Notation

3.1 Introduction

The previous chapter outlined the need for an extended Data Flow Diagram based notation which overcomes the deficiencies of two of the best known notations currently used in industry. The intention of the work presented here is to provide a notation which clearly shows [NSO89, NSO90]:

- all stored data,
- all data interfaces to processes,
- the division into subsystems which may be enabled and disabled separately,
- the conditions for enablement and disablement of each subsystem,
- all processes down to the level of leaf (or *atomic*) processes, and
- the order in which processes are required to fire.

To these ends the flows of data and events through the system are separated into two diagrams. At each hierarchy level, a Data Flow Diagram shows the data interfaces of the processes at that level. The corresponding *Event Flow Diagram* (EFD) shows not only the event interfaces of processes, but specially the firing agent for each leaf process. A third special diagram, named *Subsystem Control Diagram* (SCD), is used to show enabling and disabling of processes (subsystems) at levels where such high level control is part of the operational requirements of the specified system. Minispecs describe the operations of processes at the leaves of the system process tree, and an event dictionary is included, to hold information on the events in the specification, along with the data dictionary.

In the following sections each of these diagrams is considered in more detail. The first sections give a brief description of the symbols used in each of the three diagrams. The sections that follow give a worked example in order to demonstrate the use of these symbols. The final section outlines an informal set of rules for drawing the diagrams in the new notation.

3.2 Symbols Of The Notation

The symbols for DFD's, EFD's and SCD's are described in the following sections.

3.2.1 Data Flow Diagrams

The DFD's in the proposed notation follow the conventions of traditional Data Flow Diagrams with some minor extensions. The first of these is an extension to data flows similar to the data flow extensions of Ward and Mellor [WM86]. The new data flow constructs show data divisions, merges, and copies. These are shown in figure 3.1.

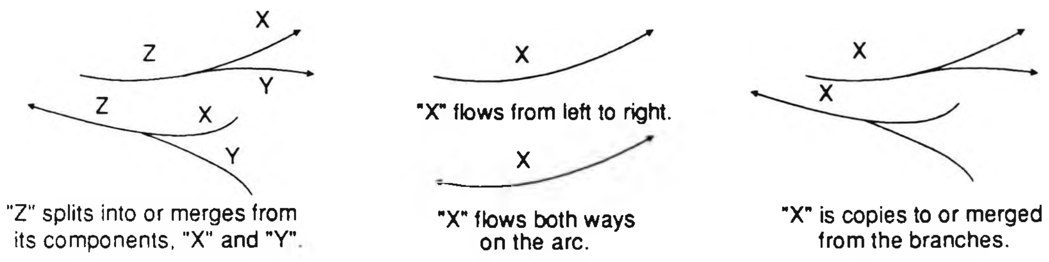

X
Z
Y
Z
X
Y
"Z" splits into or merges from its components, "X" and "Y".
X
"X" flows from left to right.
X
"X" flows both ways on the arc.
X
X
"X" is copies to or merged from the branches.

Figure 3.1: Data Flows

The interpretation of each data flow is determined by its labelling, as indicated by the annotations on the diagram.

A more significant extension is the distinction between *atomic* and higher level process symbols. Experience has shown that it is quite cumbersome to identify atomic processes if they have the same pictorial representation as other processes. This is particularly the case for systems with large specifications. In such specifications, it is difficult to identify atomic processes, i.e. those whose operations are not described by a (lower level) network of processes and stores, without looking further down the hierarchy of diagrams.

In order to make the diagram hierarchy instantly comprehensible, atomic processes are shown with a double circle symbol. These are the processes that have an associated minispec. The symbols for processes are shown along with the remaining symbols of our DFD's in figure 3.2. DeMarco style DFD symbols for environmental entities the system interacts with, called sources and sinks, and symbols for data repositories, called data stores, are retained.

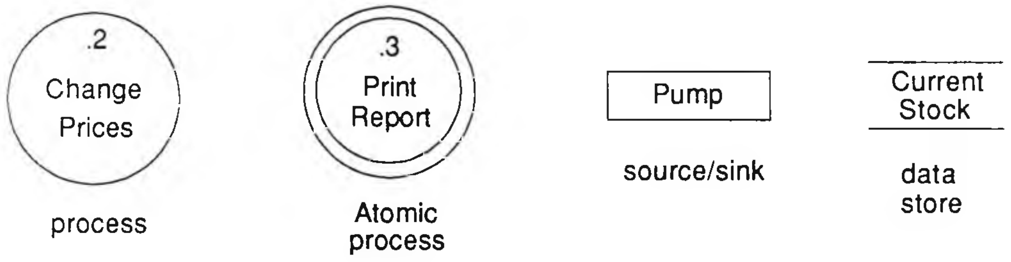


Figure 3.2: Other DFD Symbols

Data Flow Diagrams in this notation show the data interfaces of system processes. These include the data exchanged between processes, the data exchanged between processes and their environment, and stored data interfaces.

3.2.2 Event Flow Diagrams

There are two reasons for showing data and event flow on separate diagrams. The first is clarity. Including event flow information on DFD's can result in cluttered diagrams, which are difficult to read and comprehend. One of the major incentives for using diagrammatic specification techniques is the effective communication of the specifications to others. Adding event flow information to DFD's degrades their clarity, and hence defeats one of their original purposes.

Second, the ability to look at the two (data and event) types of process interface is an invaluable analysis tool. By separating the two, they can be studied in isolation or side by side.

For these reasons, process event interfaces are shown on Event Flow Diagrams. The symbols used on EFD's are shown in figure 3.3.

Dashed arcs are used to represent the flow of events into or out of processes. These are abstractions of inputs to processes that, unlike inputs carried on data flows, have no content. An event usually indicates the occurrence of some happening within or outside the system. These include events resulting from external and internal data conditions.

Processes, on EFD's, are represented by dashed circles. They show the event interfaces of system processes. Event stores can also appear on EFD's. These contain control state information. Research has shown that the occurrence of some events may need to be recorded in order for a system's processes to be able to react to them at a later stage of system operation. These are recorded in event stores, in the form of boolean flags or integer counts.

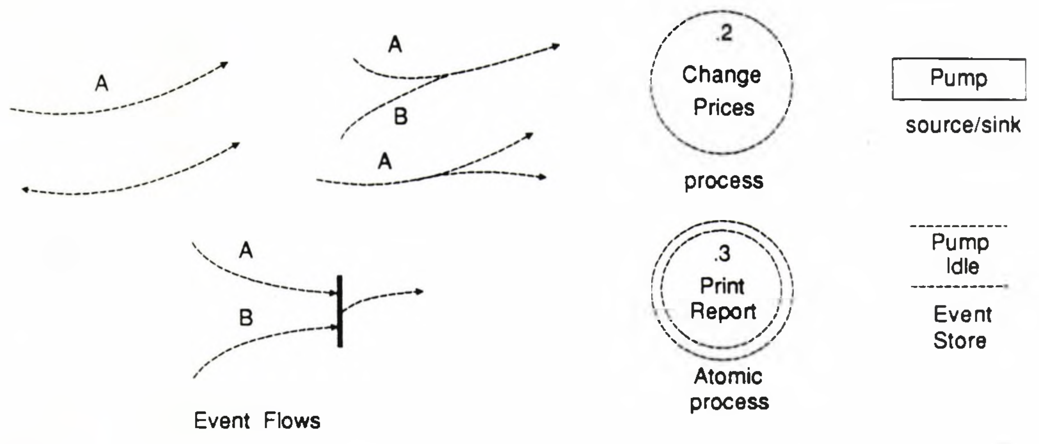


Figure 3.3: Event Flow Diagram Symbols

Like data flows, event flows can also be merged or copied, but the interpretation of such flows is different. Since an event is a singular entity and cannot be split into parts; there is no equivalent for a *split* data flow in the event flow notation. A merged event indicates an *or* of the events merged, and the branches of a copied event flow can be renamed (relabelled) to suit the purpose of the event.

The remaining event flow construct of figure 3.3, the vertical bar, is used to represent synchronisation of a number of events. Synchronisation of independent activities is an important part of real-time systems operation, which is why synchronisation is given a distinct symbol in this notation: a straight solid line.

Event Flow Diagrams show the event interfaces of system processes. This includes stored events, the events exchanged between the system processes and between those processes and the system environment. In particular, they show the firing event for each atomic process on the diagram.

3.2.3 Subsystem Control Diagrams

Two types of control were identified in the previous chapter: control over groups of processes and control over individual atomic processes. The EFD shows only the control or sequencing of atomic processes. The other type of control, which is concerned with streams of data and events (i.e. the enabling/disabling of process groups (subsystems)) is shown on *Subsystem Control Diagrams* (SCD). A DFD/EFD process is shown by a roundangle (a rectangle with rounded corners) on the corresponding SCD. Events are again shown with dashed symbols but, to distinguish between an EFD and SCD, only straight lines are used. Enabling *transitions* enter the subsystem symbol from the left and those disabling it emerge from its right hand side. There are three types of enablement/disablement: Enable/Halt, Enable/Finish, and Resume/Suspend (These are discussed in more detail below). The end of each transition connected to a subsystem roundangle is annotated by a letter in a circle to indicate the type of enablement/disablement imposed on the subsystem. SCD symbols are shown on figure 3.4.

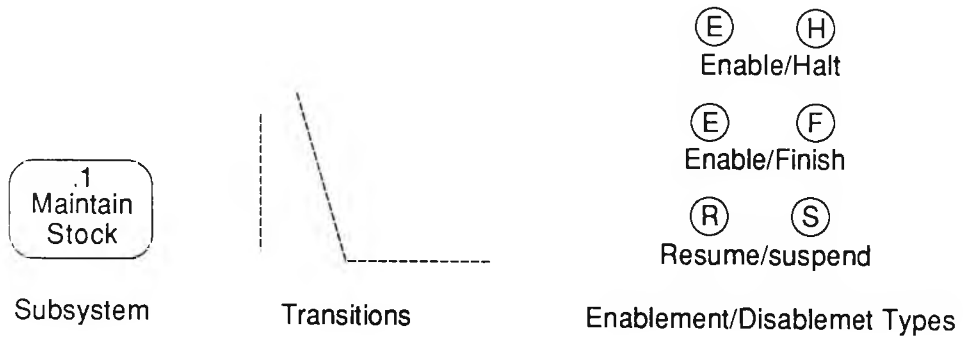


Figure 3.4: Subsystem Control Diagram Symbols

Subsystem Control Diagrams are used to show control over groups of system processes. They show how each process group is enabled and disabled.

3.3 A Worked Example

In order to show how the above symbols are used to form each of the diagrams and what role the diagrams play in a hierarchical specification of a system, a worked example is given in the sections below. Although the example is small in size, it includes most features of the notation, which is why it was chosen from the set of example specifications given in Appendix A.

3.3.1 The Petrol Station: System Requirements

The following describes the day to day operation of a petrol station. The petrol station is equipped with a number of pumps. Each pump, once enabled, is able to deliver petrol at several grades. Each grade of petrol is stored in a separate tank on site. An attendant is responsible for looking after the smooth operation of the station. He has a console in front of him which displays information about the pumps and tanks. Each pump is also equipped with a display which, during delivery, shows the selected grade, the price of that grade per litre, the amount of petrol delivered so far, and the cost of the delivered petrol.

To get petrol, a customer drives up to a pump and presses a grade selection button. A bell sounds on the attendant's console and a light corresponding to the pump is lit. The attendant enables the pump by pressing the button for the pump. Delivery of petrol is then delegated to the pump. It will commence when the customer presses the delivery lever. The pump display is constantly updated during delivery.

When the customer has acquired sufficient fuel, (s)he replaces the delivery nozzle. The pump will again warn the attendant with the bell and light. It also sends the details of the transaction to the system. Once the customer has paid for the petrol, the attendant presses the pump button again. If a second customer is waiting for service, the pump is also enabled by that button press. In order to keep paper costs to a minimum, it is company policy to issue receipts to customers only on request.

The console displays the completed transactions for every pump. If the stock level for any grade falls below a threshold value, the attendant is warned by a light on his console.

Deliveries are made both on a regular basis and on demand. When a delivery tanker arrives, the attendant presses a delivery button on the console. The ongoing deliveries are completed, but no further pump enablement is allowed until the delivery is complete. Once complete, the details of the delivery are entered via the console.

When the price of any petrol grade changes a supervisor will visit the station to alter that grade's price. For security reasons, the supervisor must first enter a preset code before (s)he is allowed to make the changes. Once the code is validated, the supervisor is instructed to commence entering the price changes. The details of price changes are forwarded to the pumps which change grade prices accordingly.

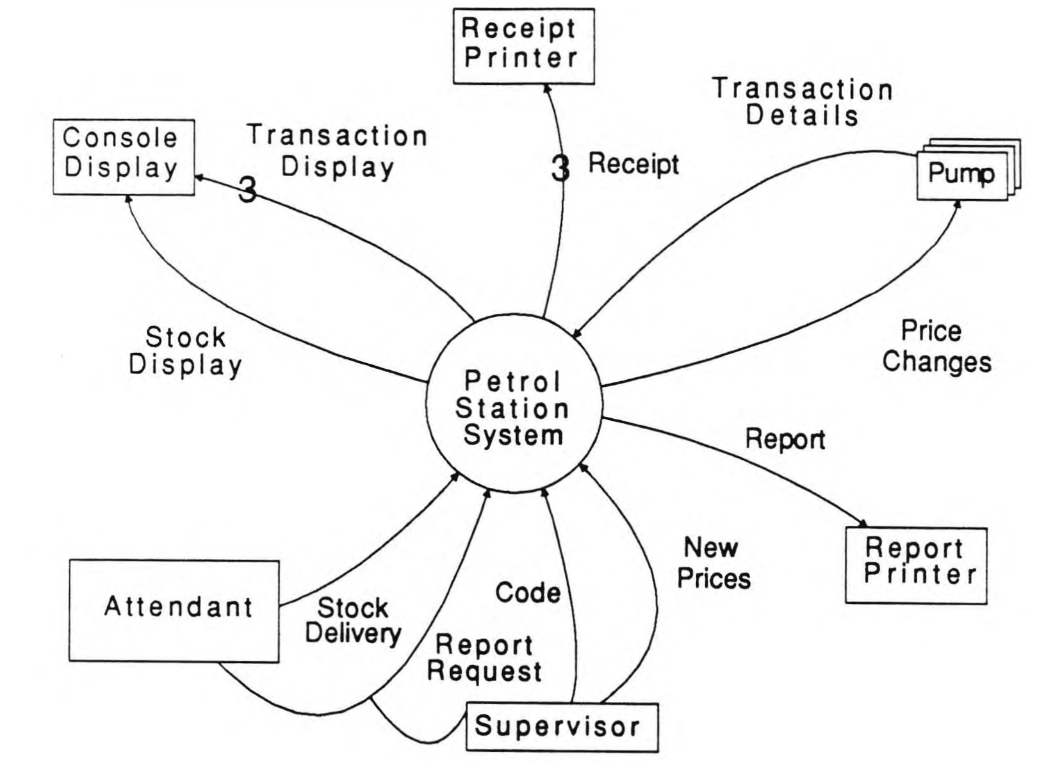
Finally, sales and stock reports are produced on demand by either the attendant or a company supervisor. A computer system is to be installed in the petrol station to help with its day to day operation.

3.3.2 The Petrol Station: Specification

The System Environment

The presentation of a derived specification in the notation described in this thesis follows in the footsteps of its predecessors by using an incremental approach which employs a hierarchical set of diagrams to reveal increasing levels of processing detail for a system. It starts by showing the system interface with its environment. Like other levels of the diagram hierarchy, this is divided into two views: the data interface and the event interface. These are shown on the *context diagram*, figure 3.5.

The context diagram is the only place where system data and event exchanges with the environment are shown directly. For this reason, rectangular symbols representing environmental entities appear only on this diagram. The data and events exchanged between the system and these



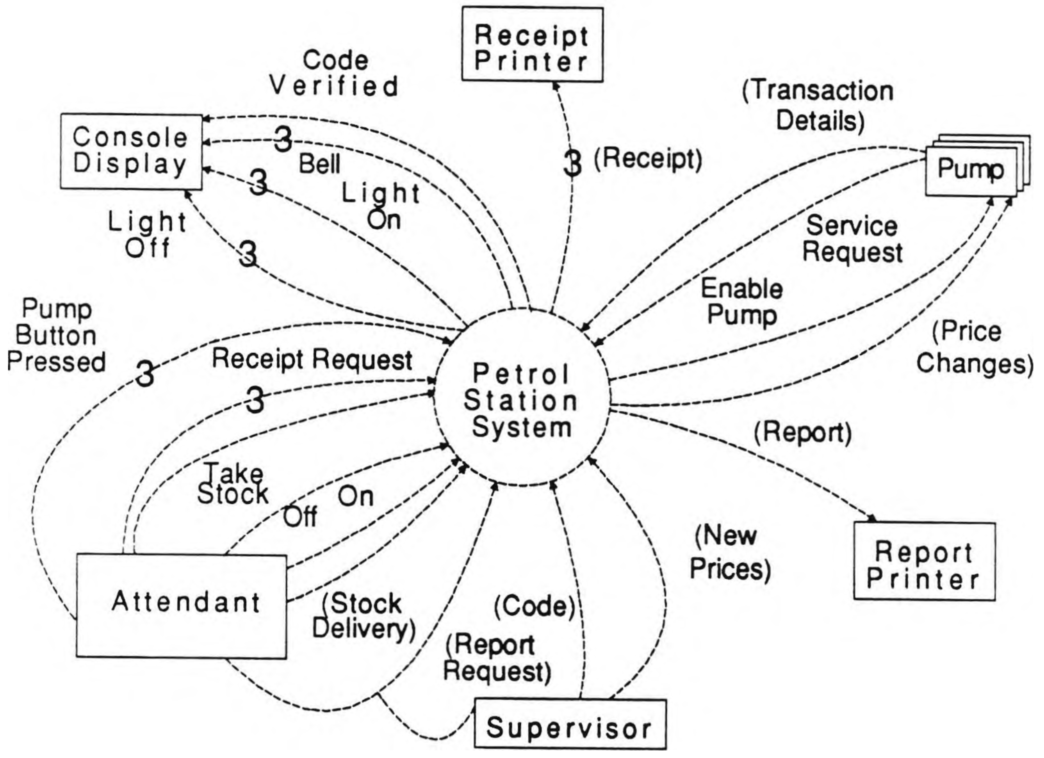


Figure 3.5: Petrol Station System: Context Digram

entities are abstracted by data and event flows on the context diagram.

To avoid cluttering the diagram, instances of identical environmental entities are overlaid on top of each other, e.g. there are three pumps in this specification. The data and event flows connecting the system bubble to the rectangles representing such environmental entities are, by inference, also duplicated. Where such duplication cannot be inferred from the diagram, a number may be placed on the flow to indicate its multiplicity. This will further help in keeping the diagram less cluttered with unnecessary detail. For example, the data flow "Transaction Display" and the event flow "Bell" have three instances each.

Only one other feature of figure 3.5 remains to be explained: the event flows labelled with parenthesised names. Ward and Mellor [WM86] identified two types of data input to a system: *time-continuous* and *time-discrete* data. The former are input data whose value varies over time, e.g. inputs from temperature or pressure transducers; the latter are data that are available at discrete points in time. The two types of data are also distinguished here. These are called *latched* data, i.e. input data whose value is updated from time to time and can be inspected when required-e.g. temperature or pressure - and *active* data, which is accompanied by an implied event, and is processed on arrival. This distinction is quite important when determining the firing agent, explained below, for an atomic process.

Output data flows may also be active, i.e. carry implicit events, indicating that the output data should be processed by the receiving agent when it occurs. Output data flows which do not carry implicit events are latched for the recipient. Since the mechanisms through which this latching is achieved depend on the capabilities of the devices used in the final implementation, they are left to the system design stage. In figure 3.5, the "Receipt" is processed by the "Receipt Printer" as soon as it is output, whereas "Transaction Display" is to be latched by the console display unit.

The implicit event carried by a piece of active data is abstracted by an event flow labelled with the parenthesised name of the corresponding data flow.

The First Division

On the next diagram level, the first division of the system into processes is shown. These are the major subsystems that make up the overall operation of the whole system. For the example system considered here, these are subsystems for monitoring the operation of each pump, effecting price changes, maintaining station stock, and printing reports, as shown in figure 3.6.

Note that the overlaying technique for instances of identical entities is again used here to avoid diagram cluttering; there are three instances of the process "Monitor Pump Operation", one for each pump. The same convention is followed for repetition of stores. Figure 3.7, which is part of the Bottling

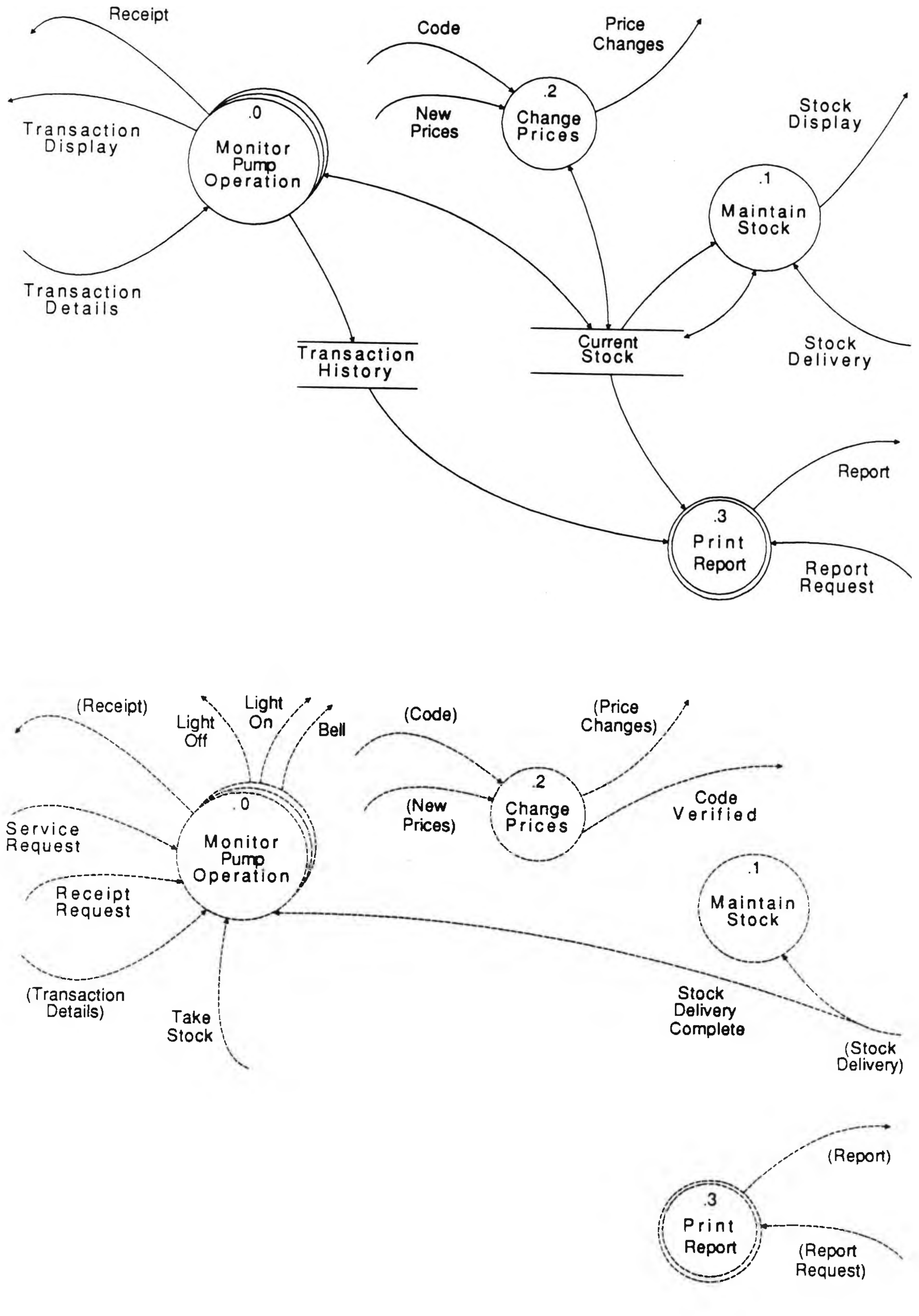


Figure 3.6: Petrol Station System (DFD/EFD)

System exercise in Appendix A, illustrates this. Again, the flows entering and emerging from repeated entities are, by inference, duplicated. Note that, when processing elements, both those inside and outside the system, are connected together in this way, there must be a one to one correspondence between the connected nodes, i.e. there must be the same number connected to either end(s) of the flow(s). In the petrol station exercise, for example, there are three pumps and three "Monitor Pump Operation" processes. In contrast, this one to one relationship does not apply to repeated stores. A number of processes may write to the same store, e.g. the pump monitoring processes all write to "Transaction History"; and a single process may read from any number of (repeated or otherwise) stores. In the Bottling System, for example, "Monitor Area" reads from all the stores shown on figure 3.7, see Appendix A.

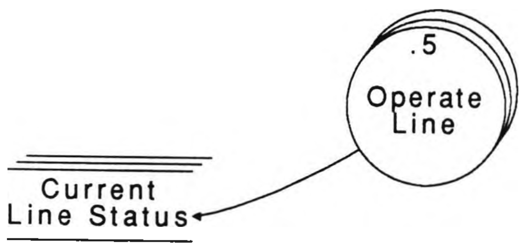


Figure 3.7: Duplication of Identical Store Instances

The data interfaces of the processes are shown by the DFD part of the diagram of figure 3.6. It shows the data exchanged between those processes, the data stored by the system which is shared by those processes, as well as the data exchanged by those processes and their environment. The DFD follows the DeMarco balancing rules, i.e. all the data flows coming into and going out of the diagram must also appear on its parent, the context diagram in this case. A similar style of numbering processes is also used to make it easier for specification readers to follow larger specifications, and to enable the analyst to identify a process by a unique digit string. This string can be used when labeling the diagrammatic expansion of that process or in its minispec, e.g. see figure 3.12.

The EFD part of the diagram shows the event interfaces of the processes which appear on the DFD. There are several points to note. First, every process on the DFD is *shadowed* on the corresponding EFD. These EFD processes do not represent new processes; they show a different aspect of the same processes as those on the DFD. They share the names and numbers of the shadowed DFD processes. The dashed circular symbol re-emphasises the purpose of EFD's: to show the event interfaces of processes.

Second, any atomic process on the diagram, must have a minispec specifying the algorithm for transforming its inputs to its outputs. The latter include both data and event outputs. For example, "Print Report" outputs "Report" in response to "Report Request".

Third, note that an atomic process may output either data or events or both. Examples appear in the diagrams below. This implies that a process may be used for generating events which result from internal system conditions such as those caused by data comparisons. In notations that use FSM's for control specification [WM86, HP88], this role is delegated to a collaboration between finite state machines and processes. In those notations, a process signals the FSM of the occurrence of an event. The FSM may generate a corresponding event upon changing state as a result of the first event. In the notation presented here, an atomic process may pass an event directly to another instead of going through a third party.

Fourth, as pointed out above, implicit events carried by input data flows are shown with an event flow labelled with the parenthesised name of the data flow. "Report Request" and "Stock Delivery" are examples of such events. Note that the lower arm of the latter event has been relabelled. Relabelling event flows can be useful in portraying the purpose of the event carried by the event flow. In this example, since entering the data for a stock delivery indicates the end of stock delivery, the corresponding event flow has been relabelled to reflect this fact.

Fifth, note that the only atomic process on figure 3.6, "Print Report", has a single event flow entering it. This is a syntactic rule for EFD atomic processes. The discussion in the previous chapter identifies the indication of when an atomic process is activated as a useful feature of a specification notation. This is why the notation presented here enforces this syntactic rule. Once a process has been identified as an atomic process with an associated minispec, the point at which it starts must be identified. This is abstracted by the single event flow input to the process: the process starts execution when the event occurs.

If a process requires a number of events to occur, e.g. it needs several pieces of data before it can start, then these events must be synchronised to form the firing event for the process, e.g. the subprocess "Update Transaction Display" of the process "Monitor Pump Operation" is fired when the two events "Transaction Complete" and "Delivery Complete" have both occurred, see figure 3.12. When any one of a number of events can individually fire an atomic process, they are merged together. For example, in figure 3.8, which is part of the Autoteller System exercise in Appendix A, "Request Service Selection" can be fired by any of the four merged events which make up its event flow input.

Sixth, an atomic process may output an event on completion. Such an event may be used subsequently to fire other system processes, enable/disable pro-

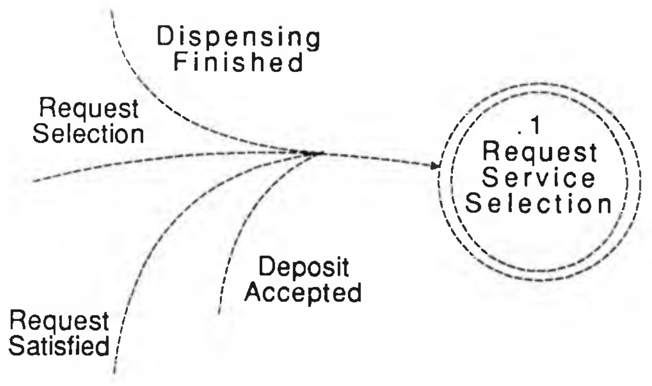


Figure 3.8: Merged Events

cesses, and/or signal to environmental entities. It may be copied to several destinations. When more than one event flow emerges from an atomic process, a choice is implied between those output events, i.e. the process may output *one* of these events upon completion of its task. Instances of these cases appear in the example specifications given in Appendix A, e.g. the process "Monitor Ph Limit" in the Bottling System exercise is shown in figure 3.9: on termination it may output either "Ph Out Of Range" or "Restart".

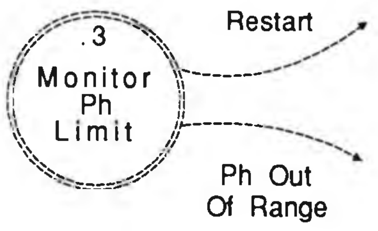


Figure 3.9: A Choice of Output Events

The balancing rules are slightly modified for EFD's. Although all events entering and emerging from the parent bubble must appear on its lower network expansion (EFD or SCD), some events output by processes on this network may not appear on the parent diagram. These are the events used on the Subsystem Control Diagram to enable and/or disable process groups, e.g. the two events "Restart" and "Ph Out Of Range" in the Bottling System exercise are used to enable and disable subsystems on the first Subsystem Control Diagram for the system (see Appendix A). They are not passed onto the parent diagram. Note that an event may be connected to the processes on the EFD, the subsystems on the SCD, or both. In the example given here, both "Stock Delivery Complete" and "Take Stock" are

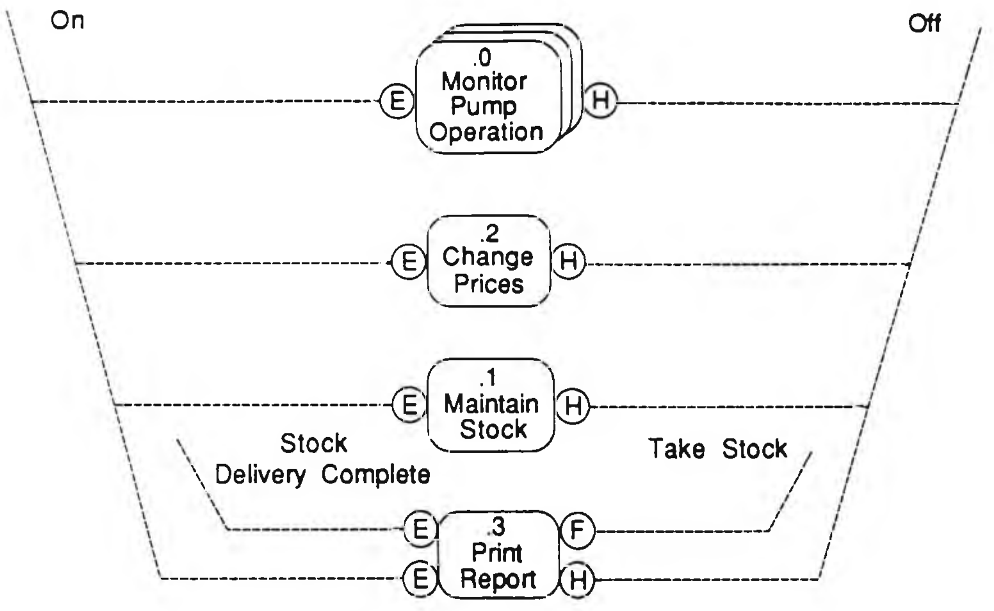


Figure 3.10: Petrol Station System (SCD)

connected to both the EFD and SCD, discussed next. All other events are connected to EFD processes.

A Subsystem Control Diagram is used to specify control over groups of processes at any level of granularity in the process specification. This implies a rule of aggregation, with nesting of subsystems to specify elaborate system control structures. The example here includes one level of nesting, figures 3.10 and 3.13. The SCD for this level of the example system is shown on figure 3.10.

Arcs entering and emerging from nodes on DFD's and EFD's indicate input to and output from those nodes. The events entering and emerging from SCD subsystems are **not** inputs to or outputs from those processes. These events are like the transitions on state transition diagrams of FSM's. The difference is that in a SCD a number of processes may be enabled *concurrently*, whereas in a STD only one state may be occupied at a time. Transitions entering a subsystem from the left enable it. Those emerging from its right hand side disable it. While enabled the child processes of the subsystem respond to events. When disabled they ignore those events. In other words, nested subsystems of a subsystem are disabled while their parent is disabled and atomic process below a disabled subsystem ignore firing events.

The annotated circles at the end of transitions on a SCD indicate the type of enablement/disablement imposed on its subsystems. Three such types have been identified. These are grouped into three pairs of enabling/disabling events.

The first type is identified by the letters E and H in the corresponding circles. The letter E indicates that the subsystem starts in its initial state, i.e. all its atomic processes are enabled to react to firing events and its (nested) subsystems can react to enabling/disabling events. The disabling event, marked with the letter H, indicates that the subsystem is disabled upon the occurrence of the event, and any ongoing work is immediately *halted*. For example, all the first level subsystems of the petrol station are enabled when the system is turned *on*. Turning the system *off* will result in halting all system activity.

The letters E and F identify the second enablement/disablement type. The letter E has the same interpretation as in a E/H pair. The difference between this and the first type is in the way the subsystem is disabled. The letter F indicates that the subsystem is to *finish* any ongoing work before stopping. In other words, all unprocessed (data and event) tokens in the subsystem are dealt with before the subsystem halts, awaiting further enablement. If a stock delivery commences in the middle of a report in the petrol station system, for example, the report request is completely satisfied before the report printer is halted. New events are, however, ignored by a subsystem's processes while it is completing unfinished work during this transition period. Further report requests are, for example, ignored while the current report is completed by the report printer before halting.

The third type of enablement/disablement is indicated by using the letters R and S. These are used when a subsystem is to *resume* from its *suspended* state. The letter R annotates the enabling event to indicate that the subsystem is to restart from its suspended state, and the letter S annotates the disabling event to indicate the requirement to save a subsystem's state, so that it can return to that state when re-enabled. On system startup a resumption event acts in the same way as an E-type enablement. Unlike the other two types of enablement, where an enabling/disabling transition may be connected to a subsystem without its corresponding disabling/enabling transition (e.g. see Appendix A examples), a subsystem required to resume after a particular event must have a corresponding suspension condition indicated by a S type disablement. The Bottling System exercise of Appendix B includes examples of this type of enablement/disablement, figure 3.11.

A Subsystem Control Diagram is only shown at those levels of the system's process hierarchy where stream control occurs, i.e. while enabled a subsystem processes a succession of data and event tokens input to it to produce a series of data and event outputs. At levels where no SCD is shown, processes can be thought of as being permanently enabled while their parents are. It is plausible to have a Subsystem Control Diagram at every level of the system process hierarchy, but most of these will be unnecessary as they will only show every process on them enabled all the time. Systems specifications without any SCD's indicate that the specified system's (atomic) actions are controlled entirely by data/event inputs, and that there is no enablement/disablement control imposed over any of its process groups (subsystems).

In addition, the processes on a SCD may not be one to one with those on the DFD/EFD. Only processes that have enablement/disablement requirements are shown on the SCD. Processes not shown on the SCD for any hierarchy level can again be assumed to be permanently enabled, e.g. see the "Monitor Pump Operation" SCD on figure 3.13. For nested subsystems, this (permanent) enablement applies only when the parent subsystems are also enabled. While a subsystem is disabled, so are all its children, including any subsystems. When a subsystem is (re)enabled, its child subsystems can react to enabling and disabling events. Those which do not have any control imposed on them, are enabled and disabled with their parent.

Furthermore, a subsystem may be initially enabled with its parent, but disabled by subsequent events (while its parent is enabled). Such requirements are indicated by following a convention similar to that used on state transition diagrams to show the starting state: an unlabelled vertical transition enters the top of the subsystem symbol. Note that unlike STD's, where only a single state may be indicated as the starting state, any number of subsystems may be start subsystems, i.e. enabled with their parent. For instance, one of the Bottling System's SCD's indicates this requirement for the "Maintain Vat" subsystem, figure 3.11.

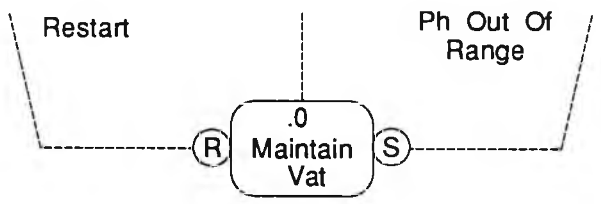


Figure 3.11: A Start Subsystem

The example SCD of figure 3.10 indicates that the whole system is enabled by the "On" event and disabled by the "Off" event. In addition, the report printer is disabled during stock deliveries. The latter starts by the "Take Stock" event and ends with the "Stock Delivery Complete" event.

Note that a process group may be composed of a single process. For example, "Print Report" makes up a degenerate subsystem which contains a single process. This convention allows for enabling and disabling atomic processes as well as groups of atomic processes.

The Remainder Of The Diagram Hierarchy

The levels below the first show the further subdivision of its composite processes into networks of processes and stores. The most complex of the processes in the example system are the pump monitoring processes. The intricate operation of each pump monitor is detailed in figure 3.12.

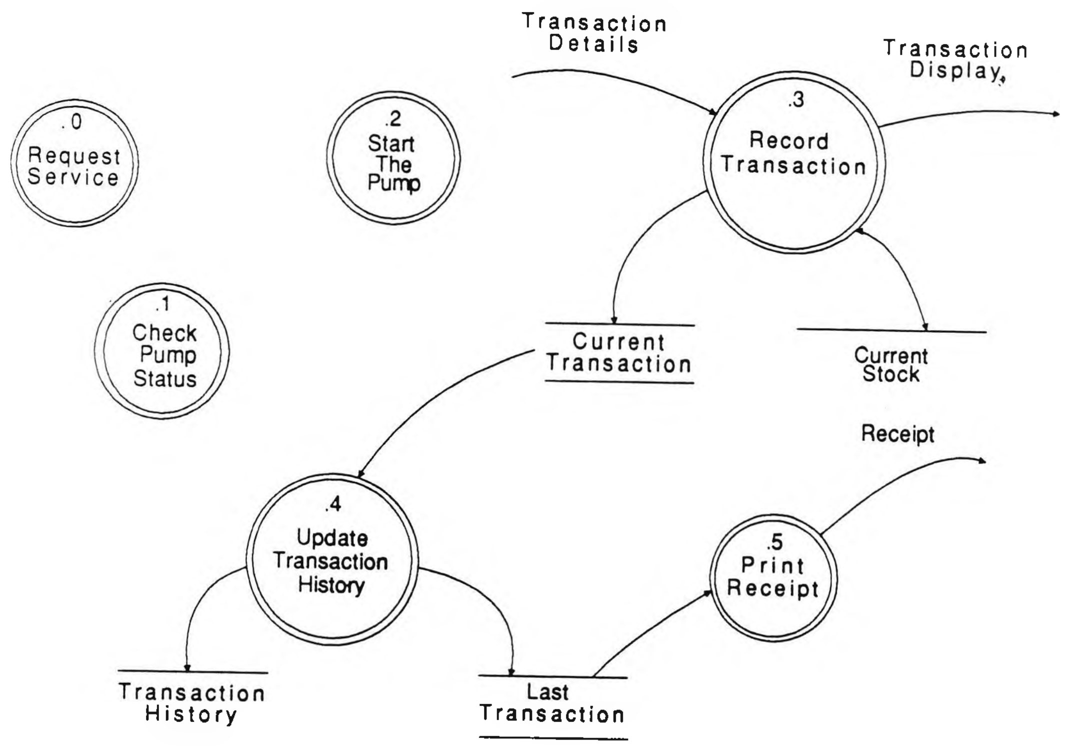
The most notable feature of this diagram is the clear indication of the control intensive nature of this part of the system. Three of the processes on the DFD have no data input or output. This indicates their role as pure event processors, i.e. those that generate events from events. Also note, unlike Hatley and Pirbhai [HP88] but like DeMarco style DFD's, the notation here shows data stores at every level they are used. Here, inherited stores are shown using a single line, rather than the parallel pair, to make it easier to recognise them.

The EFD shows some new features of the notation. Event stores are used to remember the occurrence of events for later use by processes. For example, "Pending Request" stores outstanding service requests, so that they can be serviced when the pump button is pressed. Synchronisation is used to generate the appropriate firing events for "Update transaction History" and "Print Receipt", and output events are copied and relabelled to show their purpose.

The output events of "Check Pump Status" perhaps deserve clarification. The multiplicity of output events indicates that the process selects an event from a choice of three. These indicate that the pump should be started-"Start Pump", the customer has paid-"Transaction Complete", or both-"End Transact.". According to the requirements specification, the last event should occur when a subsequent customer requests service before a previous customer has paid for his/her transaction. Since an atomic process is restricted to a single output event (at the conclusion of its task), the branches of the middle event flow are merged with the (destinations) of the other two event flows to indicate that both events may be output by the process, resulting in the peculiar output event flow constructs of "Check Pump Status".

Another feature of this part of the petrol station system, which may not be immediately apparent from its EFD, is the link between the sequences of events that are generated by the system and its environment. The EFD is well suited for showing a sequential string of executions of a number of (atomic) processes in the cases when the firing events (except perhaps the starting and ending events) are generated within the system. It can be noted that an event is generated as a result of another when the first is the input and the second is the output event of an atomic process. If the output event is used to subsequently fire another process, the output event of that process can be linked to the original input event.

When part of a sequence of actions lies outside the system, however, the correspondence between the events involved is not immediately obvious from the EFD. In figure 3.12, for example, "Service Request" causes the console bell and light, which will eventually result in a button press from the sta-



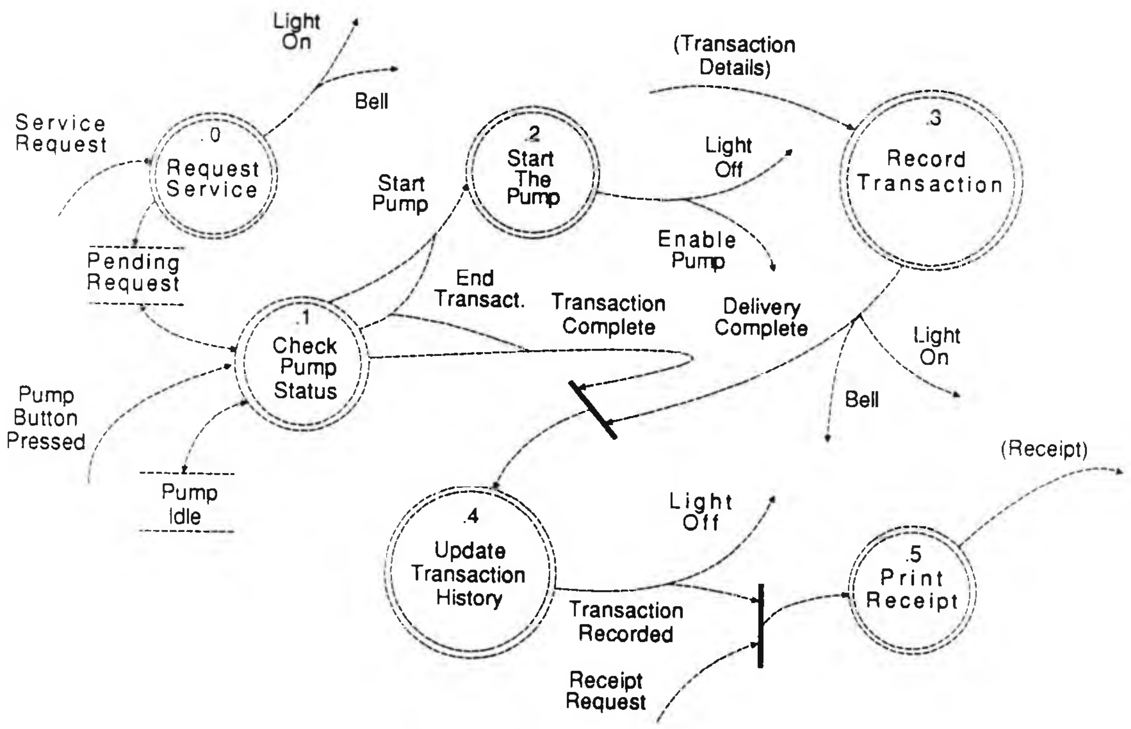


Figure 3.12: .0 Monitor Pump Operation (DFD/EFD)

tion attendant. That may, in turn, result in a pump enablement, which will prompt the pump into action. The pump will subsequently forward "Transaction Details", which will cause a further bell and light on the console. The light is eventually turned off when the attendant presses the pump button a second time, and the sequence restarts. This sequence indicates a *dialogue* between the system and its environment.

The identification of dialogues is an essential part of the analysis activities during system specification. They can provide useful guides for grouping of processes under subsystems and for design and implementation strategies (see the notes under Methodology and on design and implementation in the next chapter). This implies that a useful extension to the notation would allow dialogues to be identifiable on EFD's. EFD's already have several distinct symbols and convey a large amount of information. Adding extra symbols or annotations to indicate dialogues may clutter them beyond a comfortably understandable form. The event dictionary is a more suitable place to indicate event dependencies of all types, including conversation sequences. We suggest the Event Dictionary for the petrol station which appears as part of its specification in Appendix B.

All the processes of figure 3.12 are atomic and so require no further expansion into another diagram. The only process on this diagram that requires subsystem control is the pump enabling process, "Start The Pump". Like the report printer, it must be disabled during stock deliveries. It also has the same enablement/disablement type as the report printer: once "Start The Pump" has been fired, it will not halt until it has enabled the pump, even if it should become disabled. This is shown on the SCD for "Monitor Pump Operation", figure 3.13.

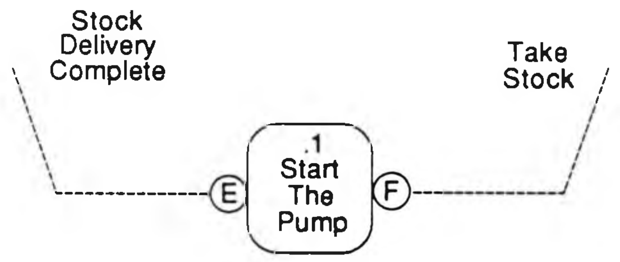


Figure 3.13: .0 Monitor Pump Operation (SCD)

Note that since "Start The Pump" is the only process requiring subsystem control, it is the only process that appears on the corresponding SCD.

The detailed specifications of the other two processes of figure 3.6 are shown on figure 3.14.

Note the use of the timing process "Clock" used by "Monitor Stock". As pointed out above every atomic process must have a single firing event. Since "Monitor Stock" does not have any active data inputs (its only data

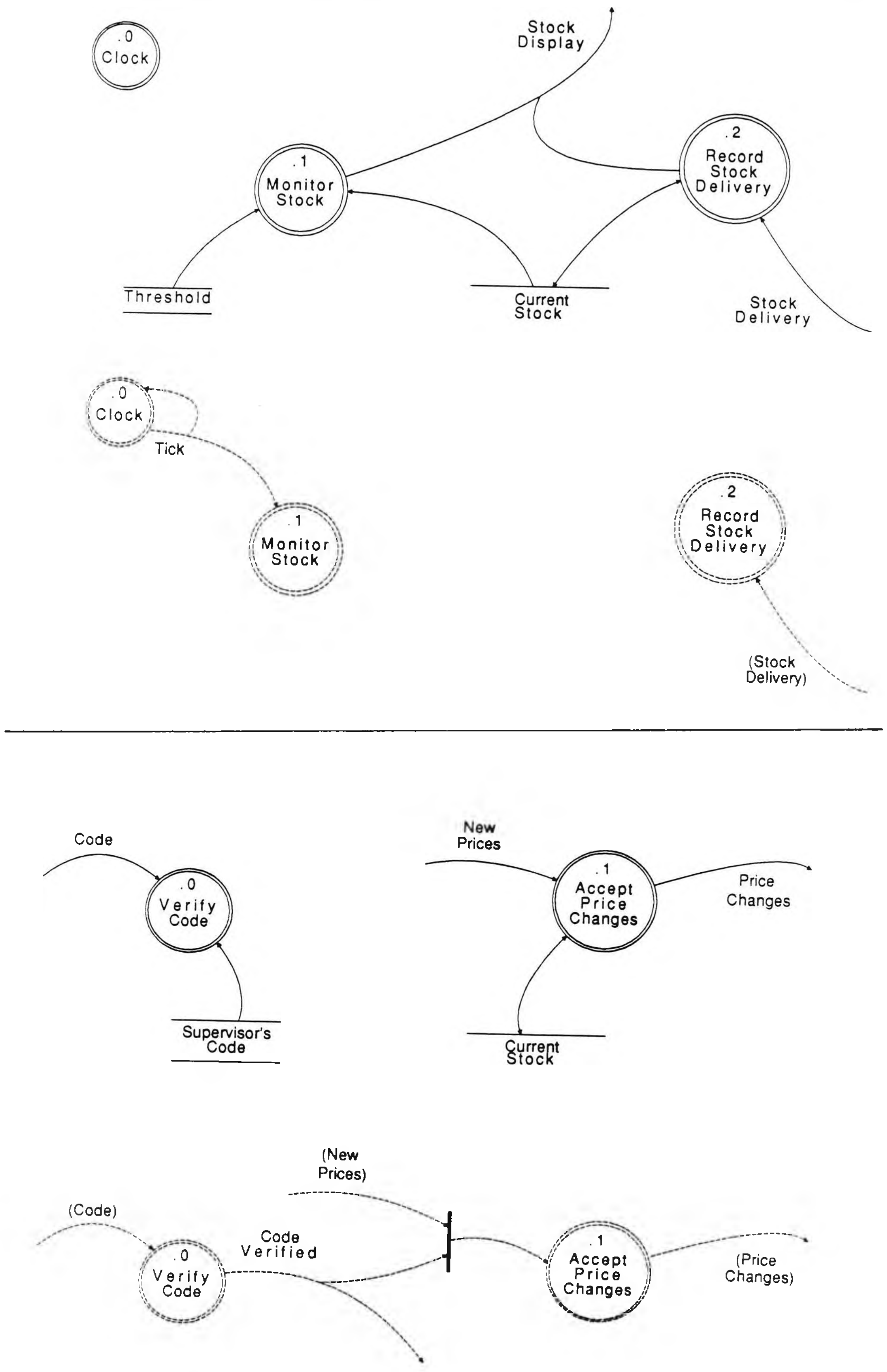


Figure 3.14: .1 Maintain Stock (Above) and .2 Change Prices (Below)

inputs come from stores), a firing event must be provided by either the process's environment or a system process. In this case the process runs on a regular basis to check stock levels. A timing process is, therefore, provided to fire the process. Note that this approach is different to that taken by many other notations, where time is either available to all system processes or it is an environment derived quantity. The approach taken here makes more explicit the temporal events to which a particular process has to respond by considering time a system derived quantity, resulting in clearer specifications which are easy to follow (see also the section on timing requirements in the next chapter).

Other example specifications are given in Appendix A.

3.4 Diagram Rules

The example specification above illustrates how the symbols of the notation are used to form DFD's, EFD's, and SCD's. Many of the rules for drawing each diagram were given mixed in the discussion. This section gives the full set of rules for drawing these diagrams. A more formal discussion of these rules is given in the following chapter.

A specification consists of a Context Diagram and a hierarchical set of diagram levels. A Context Diagram consists of the Context Data Flow Diagram and the Context Event Flow Diagram. It is labelled with "Context Diagram" and the system process name.

A Context DFD consists of a (non-empty) set of system data sources and sinks (terminators), which are connected to the system process by data flows. A context data flow cannot connect either two terminators or a node to itself. Conversely, every context data flow connects the system process to a data terminator, and every data terminator is connected to the system process. All the data flows of the Context DFD appear on its child diagram. Every symbol on the Context DFD is named and all names are unique.

A Context EFD consists of a (non-empty) set of system event sources and sinks (terminators), which are connected to the system process by event flows. A context event flow cannot connect either two terminators or a node to itself. Conversely, every context event flow connects the system process to an event terminator, and every event terminator is connected to the system process. All the event flows of the Context EFD appear on its child. Every symbol on the Context EFD is named and all names are unique.

Each diagram level consists of a Data Flow Diagram, an Event Flow Diagram and an optional Subsystem Control Diagram. It is labelled with the process name of its parent. There is one and only one diagram in the set of diagram levels labelled with the system process name.

A DFD consists of a (non-empty) set of processes and a set of data stores connected together by a set of data flows. A data flow cannot directly connect two data stores. Nor can it connect a node to itself. Every inherited data store and its connections to the processes of a DFD are connected to the DFD's parent process. Similarly, every inherited data flow on a DFD is connected to its parent process (which can be the Context Diagram). All the DFD symbols are uniquely named.

An EFD consists of a (non-empty) set of processes and a set of event stores connected together by a set of event flows. An event flow cannot directly connect two event stores. Nor can it connect an event store to itself, but it can connect an atomic process to itself (for self perpetuating timing processes). Every atomic process on an EFD has *one and only one* active input event flow. Every inherited event store and its connections to the processes of a EFD are connected to the EFD's parent process. Every inherited output event flow on an EFD is connected either to its parent process (which can be the Context Diagram), or to a subsystem on the corresponding SCD. Every inherited input event flow on an EFD is connected to its parent process (which can be the Context Diagram). Every synchronisation symbol on an EFD must have only one output event and more than one input event. All the EFD symbols are uniquely named.

The process sets of the DFD and EFD for a level are identical, and every process must have at least one (data or event) input and one (data or event) output flow. Every expandable process has an expansion in the set of diagram levels, and every store and flow that is connected to that process appears on its expansion.

A SCD consists of a set of subsystems (which are a subset of the processes on the corresponding DFD/EFD). Each subsystem has a set of enabling and disabling transitions connected to it. Each transition is labelled with an event and an enablement/disablement type. A subsystem cannot be enabled and disabled by the same event. Every transition on a SCD is either connected to the SCD's parent process, or is output from one of the processes on the corresponding EFD.

3.5 Summary

This chapter introduced the symbols of the new notation and how each one is used to form the three diagrams of the notation. The rules governing the formation of these diagrams were also given. The following chapter discusses a number of issues related to the notation.

Chapter 4

Notational Issues

4.1 Overview

The previous chapter introduced the symbols of the new notation and how they are used to form the diagrams of the new notation. This chapter deals with the many issues related to the new notation. Each section below discusses one of these issues. An overall conclusion ends the chapter.

4.2 Atomic Processes

The term *atomic* has been used frequently in this text to refer to a process which is a leaf of the system process hierarchy tree, i.e. a process whose operation is not so complicated as to require further breaking up into a subnetwork of processes and stores, but is described by a minispec. So far no guidelines have been given which aid the analyst in identifying such a process. This section elaborates further on our meaning of the term *atomic*.

Neither traditional DeMarco style DFD's nor the direct predecessors of the notation described here give any *objective* guidelines to help in deciding when to stop expanding processes. The most widely accepted *subjective* guide given by the designers of these notations and their accompanying methodologies is that the specification of a leaf process should not exceed half a page of A4 paper [LL87, DeM78].

This simple subjective guideline may result in the misuse of a notation and hence in low quality specifications. Therefore, a set of objective guidelines must be provided to help the analysts identify *atomic* processes. The first of these is, we envisage, that each atomic process, once *fired* by its event input, will run to complete its task. A further event is required to re-fire the process for a subsequent execution. This implies that an atomic process must receive all its active data at the start of its execution cycle: it cannot receive data, other than what it reads from stores, during its execution.

This restriction will help in clearly identifying what data is required before a process can start, which in turn indicates when a process can be started. Moreover, all outputs, except those to stores, from an atomic process occur at the end of its execution. In other words, the (active) outputs of the process are not available until the conclusion of its task. This makes clear the sequencing that is implied by the exchange of data (or events) by atomic processes. A similar approach is taken in [DT86], where traditional DFD's are executed.

Furthermore, an atomic process may output an event at the conclusion of its task. The corresponding event flow may be copied to several destinations, and relabelled to indicate its purposes. An atomic process may not, however, output multiple (distinct) events during or at the completion of its operation. The requirement to do so is a direct indication that the process is too complex to be atomic and must be further divided into a subnetwork of processes and stores. In other words, a simple definition of an atomic process may be given as one whose operation starts upon the occurrence of a single event and concludes by outputting a single event. It has no idea what event started it and what will be fired or enabled by its output events. This definition is similar to that of an *atomic action* used in the fault tolerance literature [AL81], where the absence of interactions in taken as a criterion for atomicity.

Second, atomic processes are *functions*. In other words, each execution of an atomic process carries no internal data from past executions, i.e. atomic processes are stateless. If an atomic process requires information to be carried from one execution to the next, it must store such data in a data store outside itself. This will result in clearer specification, because the operation of each atomic process can be studied in isolation and without having to discover its internal state.

The above criteria can be used as useful guidelines by an analyst when trying to distinguish between atomic and composite processes. Clearer specifications will result because the analyst is now able to identify atomic processes using objective rather than subjective or rule of thumb criteria from past experience.

4.3 Timing Requirements

Any notation aimed at specifying reactive systems must cater for the specification of the system timing requirements. The latter fall into two categories. The first encompasses timings that are taken relative to a clock. Examples include system activities that must take place at regular time *intervals* and those that must be performed at particular *points* in time. The second type of timing requirement includes system response times, i.e. the time taken by the system to produce an output set from a given input set.

The first type of timing occurs in a variety of systems and has been dealt with in a variety of fashions by specification notations. The existence of a conceptual clock and the availability of its current value to all system processes has, for example, been used by Hatley and Pirbhai [HP88]. One of the aims in producing a new specification notation was to produce one which results in clear specifications. In achieving this aim, timing requirements of this kind can best be specified by using special self perpetuating timing processes such as the "Clock" process in stock maintenance subsystem in the Petrol Station System. These processes provide firing events for those atomic processes which must execute at regular or particular time slots. When a number of processes run with the same regularity, the same timing process can drive them, e.g. see the Bottle Filling example in Appendix A. In contrast, if timing requirements of a group of processes differ, separate timing processes can be used, e.g. see the Patient Monitoring System in Appendix A.

The second type of timing is concerned with system response. Timing constraints of this kind vary from those desirable for efficient system performance to those critical for correct system operation. Since specification is concerned with detailing what a system must do, not how it is to achieve it, performance constraints can only be stated along with a specification. The means of achieving them depend entirely on the system's implementation environment, and are the responsibility of system designers during system design and implementation. To achieve them, system designers must be aware of such requirements. They can be stated in a tabular format similar to that used by Hatley and Pirbhai [HP88], and accompany the diagram hierarchy with other non-functional requirements such as fault tolerance.

4.4 A Graphical Language

Derivation of a system specification in the new notation can be viewed as a programming activity. The control constructs of EFD's are very similar to those used in (imperative) programming languages, e.g. sequence, selection, and iteration. The grains of the program are, however, more coarse than the statements of high level programming languages. The atomic process is the smallest grain of the program in the notation. In other words, specification is *programming in the large.* The syntax and semantics of the new notation, which is the language of this activity, must therefore, be more formally described.

4.4.1 Language Syntax

The rules for forming syntactically correct diagrams were given informally at the end of the previous chapter. This section briefly outlines a formal approach to describing the language syntax.

The syntactic rules for the graphical language can be described by using sets of tuples. This is a common approach for describing graphical notations. Petri nets are, for example, often described in this way. This approach gives an abstract syntax for the language which can form the basis for checking the syntactic validity of a set of diagrams.

The sets alone are not sufficient to describe the language completely. Unlike petri nets (described later), giving the set of components for a particular diagram is not sufficient. There are additional rules that govern how diagrams can be formed. These relate to naming, numbering and balancing the diagrams. For example, nodes on each diagram must be uniquely named, processes must be uniquely numbered, and all inherited data flows on a DFD must be connected to their parent process. These additional rules can be formally stated by a set of logic predicates.

The set description and the additional rules are given in Appendix C. This appendix also describes an alternative textual language for describing a set of diagrams in the new notation. This language can be used as an alternative (textual) interface for storing and analysing a specification. The syntactic rules for forming the sentences of this language and an example specification are also given in Appendix C.

4.4.2 Language Semantics

Since real-time systems are event driven, the interpretation of the control parts of a specification are of the most interest to system developers. The operational semantics of a given specification should, therefore, be formally described.

The dynamic behaviour of a system is described by its state at any given moment in time. This is the collective state of all its subsystems and atomic processes at that moment. One way of describing this state is by using a finite state machine based model, where each state of the machine is a compound state composed of the collective state of the system components. As pointed out in the Chapter 2, because of their inherent sequentiality, concurrent behaviour is not naturally expressed by FSM models. An alternative model is required that lends itself more naturally to modelling concurrent system behaviour.

Petri Nets, EFD's and SCD's

Petri nets have been used extensively for describing the concurrent behaviour of a variety of systems. They are described in detail in the following chapter. This section describes the investigation which was undertaken to evaluate the mapping of Event Flow Diagrams onto petri nets so that the established body of petri-net formal theory can be used as a semantics for the control parts of specifications.

An atomic process can be represented by a *transition* in a petri net. Its input and output events are represented by the input and output *places* of this transition. A system event source is represented by a place that is not the output place for any of the net's transitions. Conversely, an event sink is represented by a place that is not an input place for any of the net's transitions. The occurrence of an event is indicated by placing a token in the place that represents it. Multiple connections between places and transitions can be used when modelling the various event flows of EFD's. Figure 4.1 illustrates these mappings.

A transition in an *ordinary* petri net fires when there are sufficient tokens in its input places. Upon firing it removes the enabling tokens and deposits a token per connection into *each* of its output places. The only way to represent a choice in such a network is to introduce a *conflict* between the receiving transitions, figure 4.2. This does not reflect a true choice since the execution path followed depends on which of the receiving transitions is chosen for firing. Extended notations exist which allow the representation of choice on petri net graphs [Pet81, Bae73]. In figure 4.1 choice is shown by a $\oplus$ symbol. Output disjuncts, representing the placement of takens in a subset of a transition's output places, is also shown using this symbol.

The transition representation of a process fires when its input place has a token in it, i.e. when the firing event for the process occurs, but a process can only respond to firing events if it is enabled. In other words, the control picture contained in Subsystem Control Diagrams must be incorporated into the petri net model. Dummy transitions and places are used to remove firing tokens so that a transition cannot fire when the corresponding process is disabled. For example, the petri net for two processes, P1 and P2, which are fired by unique events, El and E2, but enabled and disabled by common events, E3 and E4, is shown in figure 4.3. Tokens representing new firing events for a disabled atomic process must be *absorbed*, until it is re-enabled.

Subsystems can also have higher level control imposed on them. The enabling tokens for the processes in such subsystems must again be prevented from firing the enabling transition until the subsystem is itself enabled. The hierarchical control structure specified by a set of EFD's/SCD's can be derived by introducing dummy transitions and places for each SCD in the hierarchy. For example, if the (higher level) process containing the processes, P1 and P2, of figure 4.3 is enabled and disabled by two events, E5 and E6, the subnet of figure 4.4 is added to the front of the petri net of figure 4.3.

The initial marking of places in a derived net determines which parts of the net are initially enabled to respond to external events. This marking can be derived from Subsystem Control Diagrams (initially enabled subsystems

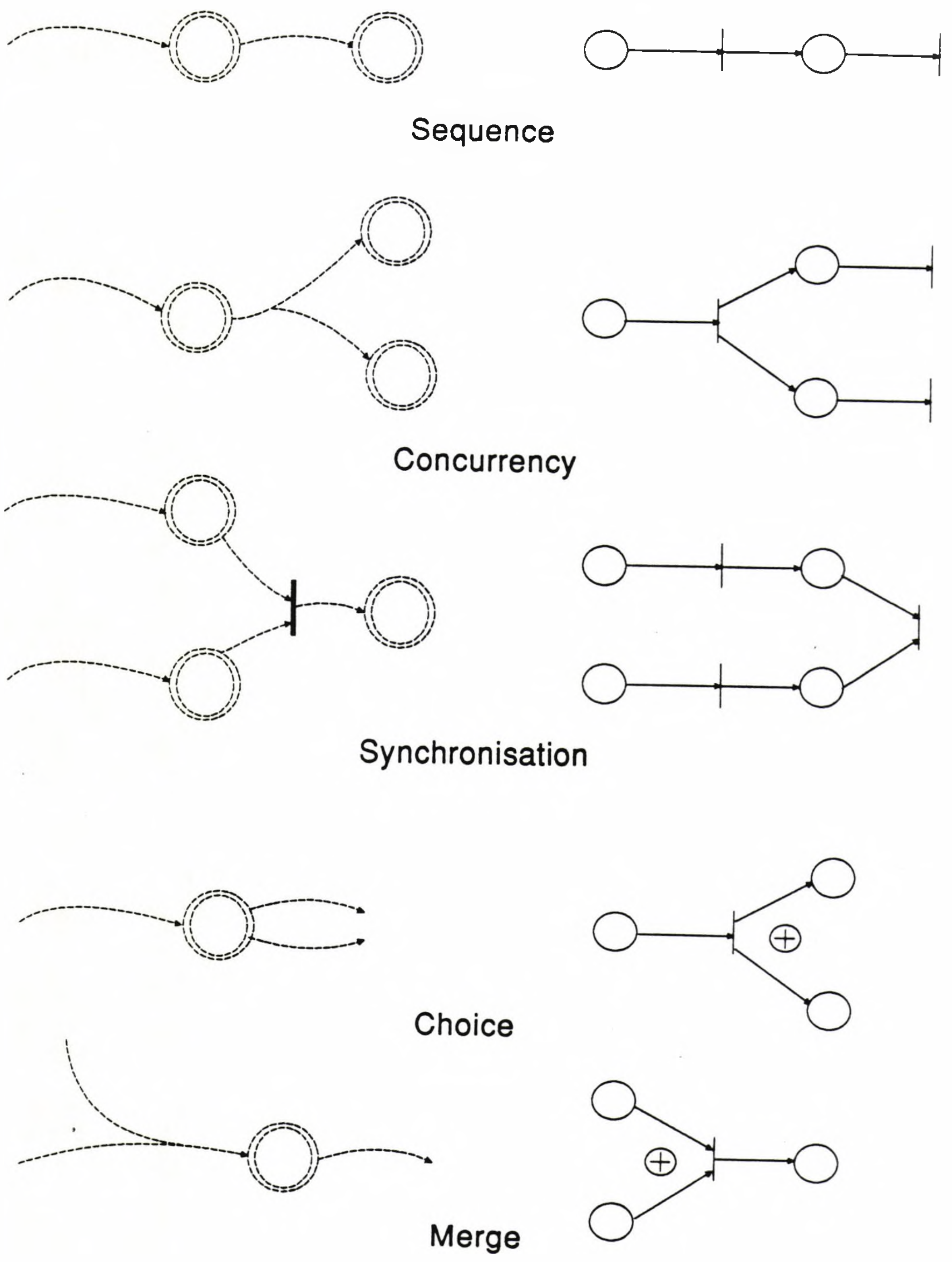


Figure 4.1: Mapping The Event Flows of EFD's to Petri (Sub)nets

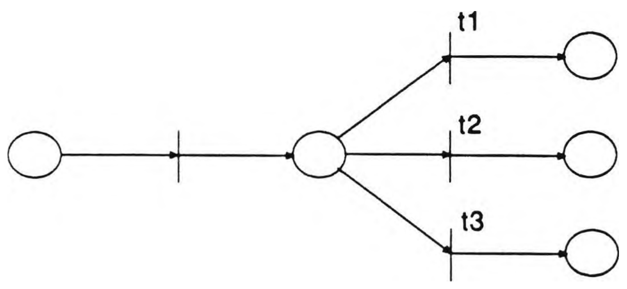


Figure 4.2: Conflict Between t_1, t_2 and t_3

include those enabled with their parent, i.e. those processes shown on the DFD/EFD pair but not on the SCD, and those on the SCD which have vertical unlabelled directed lines entering them).

Using the above mappings in an extended petri net model (which allows input and output disjuncts for transitions) a petri net equivalent can be derived for a set of specification diagrams. EFD's are therefore a form of petri net, but this flat petri net equivalent has a number of drawbacks.

It can be observed from the above two simple mappings that the resulting petri net, for anything but the most trivial of systems, will be very large. It may be possible to arrive at a more compact net for a system by using a petri net variant which attempts to reduce the size of the net, e.g. coloured petri nets [Jen81]. In addition, to lessen the sudden expansion of the net in cases where nested control is prominent, an alternative approach can be taken. Rather than using dummy transitions and places to construct a net equivalent for a leveled set of EFD/SCD's, a simpler net equivalent of atomic processes can be constructed. This net can then be allowed to grow and shrink as subsystems are enabled and disabled. However, the petri net for very large systems may still result in a net which is a jumble of places, transitions and arcs. Because of topological restrictions, the network of circles and bars for such a net cannot be understood easily. The resulting network can hence only be useful in its mathematical representation.

The worst drawback of a petri net model is the loss of information. Since petri nets can only reflect the control view for a system effectively, the data processing part of a system can not easily be represented by a petri net model. Although a process can be represented by a transition, and its firing is modelled by the firing of that transition, the actual operations on the input which result in the output from that process cannot effectively be modelled by a petri net. Since the data values moving around a system affect its operational behaviour, they must play a part in the control flow view of that system. This is reflected in the decision to show the implicit control carried by data flows on EFD's. The inability to show data processing

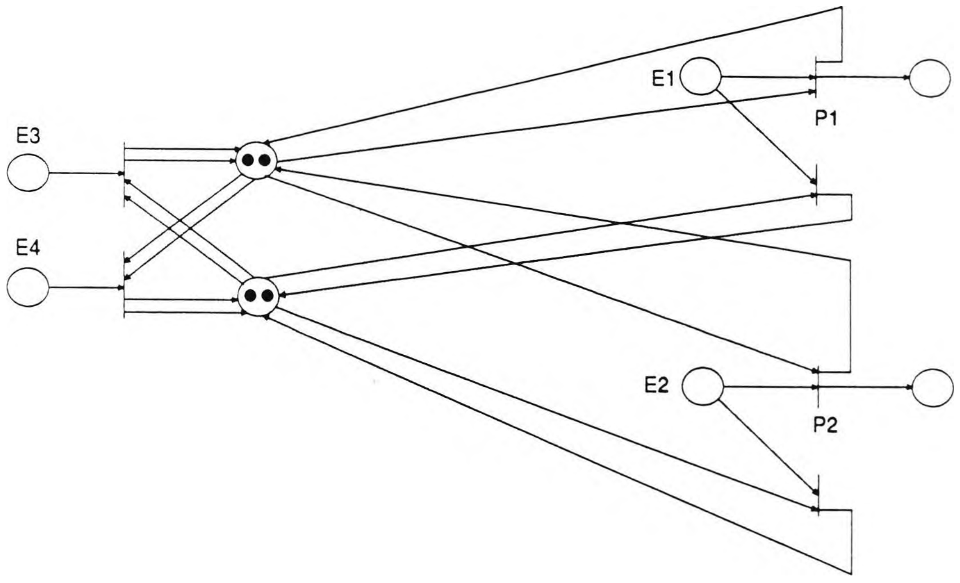


Figure 4.3: An Example Petri Net Mapping for Two Processes

aspects of systems results in a loss of information when using a petri net model for the control flow through that system. Significantly, many choices as to which execution strand to follow through a system are made based on the value of some piece of data. Since this type of choice cannot easily be represented in the petri net model, the mechanisms by which the choice is made are lost when going from a DFD/EFD/SCD view to a petri net model.

Hence, the petri-net model itself is not as usable as EFD's, but it is comforting to be able to rely on petri net semantics.

4.5 Animation and Prototyping

User participation throughout system development can contribute enormously to the successful completion of that project. User involvement in validating a specification is not only desirable in achieving a project's objectives, it is also of great value to project managers in recognising the milestones reached during specification, as well as determining the contractual obligations fulfilled by reaching those milestones.

The ability to analyse various aspects of a partially or totally derived specification is, therefore, a great asset to analysts while developing a specification. One way to carry out such analysis is by *simulating* or *animating* the behaviour of a particular part of the system, before its full implementation. This has been referred to by many authors as rapid pro-

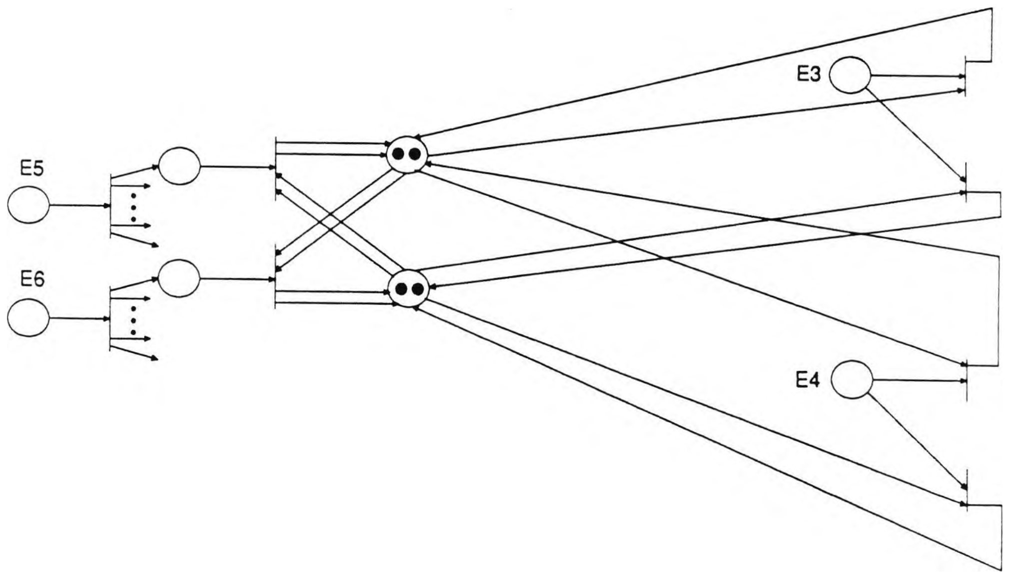


Figure 4.4: Petri Subnet for Subsystem Enablement/Disablement

totyping [BM85, KN88, FLL86]. Such animations can be used as part of the demonstration to system users, in order to guarantee that the derived specification conforms to their requirements. The alterations prompted at this stage are, of course, much cheaper to organise than those to the fully implemented system.

Since a system's control flow is completely captured by the Event Flow and Subsystem Control Diagrams in its specification, simulating the behaviour of any system part is only a matter of isolating that part and analysing it through animation. As shown above, a set of EFD/SCD's can be mapped onto a petri net equivalent. This net can then be used for a token like execution of (any part of) the specification. This can be achieved by first deriving the initial marking of the net from Subsystem Control Diagrams and then observing system behaviour while it responds to a series of interactive or batched input events.

The drawback of using a petri net model is the loss of information. Since petri nets can only reflect the control view for a system effectively, the actual processing within each atomic process can not be easily represented by a petri net model. Nevertheless, this equivalent model can be used to study the dynamic behaviour of the system by providing stubs for atomic processes and selecting execution strands by either prompting the user for a choice (interactively), or picking one from a preselected list (in batched mode).

During such animations transition firings are only useful if they are translated back to subsystem enablement/disablement and process firings for presentation to the user. This can ideally be done as part of an automated CASE tool (see also the section on CASE tool support in Chapter 6).

4.6 Specification Quality

It is desirable to have some criteria for judging the quality of specification diagrams in the new notation. Since the representation of system control aspects is the major extension to the DFD notation, EFD's should be subject to such criteria. Similar criteria already exist for imperative programs, and it was decided to attempt to derive similar ones for EFD's. The following sections describe the result of the investigation.

4.6.1 Background

This section gives a brief history of the theorem which defines a structured program.

D-structures

A *D-structure* is a one-in one-out structure that can be recursively defined as follows: a D-structure is either a basic action or it is constructed from simpler D-structures each of which may be a sequence of D-structures, an alternation structure, or an iteration structure. A basic action is one that has one entry point and one exit point, where the steps between the entry and exit points cannot cause a transfer of control. In an alternation structure control is transferred by taking one of a number of routes available from the entry point to the exit point. The route taken depends on the value of a condition and each route is a D-structure. In an iterative structure a single route is repeatedly followed until a certain condition is satisfied. Again the route is a D-structure.

D-structures can be represented diagrammatically as shown in figure 4.5.

Boehm and Jacopini's Theorem

The theorem attributed to Boehm and Jacopini states that a solution to any programming problem can be constructed by using D-structures [BJ66, Coo67, BS72, LM81, Mil75, Har80]. In other words, for every programming problem, there exists a solution which is entirely made up of D-structures. Furthermore, for every program whose control structure is not made of D-structures [Tse87a, Tse87b, Wil83], there exists an equivalent program made up of D-structures. Such equivalent programs can be derived by applying mechanical restructuring techniques [LM81]. A number of informal

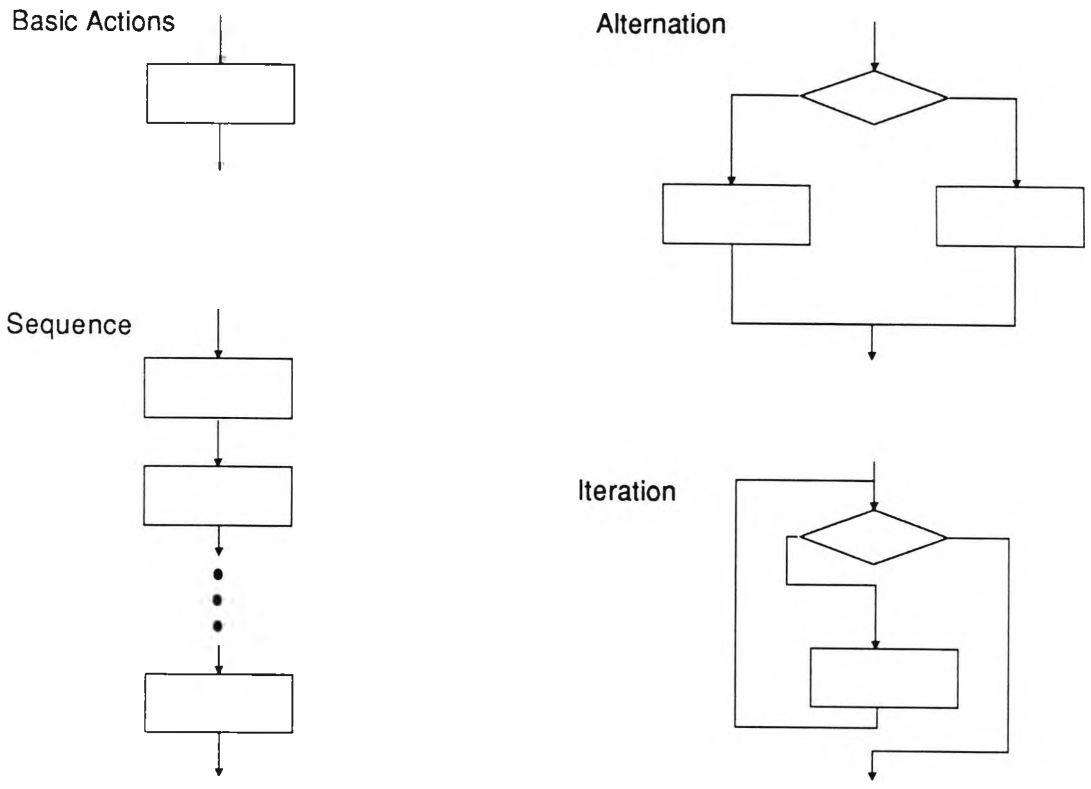


Figure 4.5: D-structure definition

and formal proofs for this theorem have been derived and presented in the literature [LM81].

4.6.2 Well Formed Diagrams

According to the above theorem, structured programs can be constructed from four constructs: basic actions, sequences of actions, alternative structures and iterative structures. A similar set of constructs may be identified for EFD's. The basic action in EFD's is the "atomic process". A sequence of atomic processes is equivalent to a sequence of basic actions in D-structures. Alternation and iteration structures can also be constructed from atomic processes in EFD's.

D-structures were developed to describe structured *sequential* programs. There is no provision for concurrency. The basic set of constructs must, therefore, be augmented in order to make it applicable to EFD's by adding a concurrency construct to the basic set. Note that since an atomic process does not change the sequence of control by means other than using its output event, each atomic process can be considered a one-in one-out structure in a similar way to basic actions in D-structures.

Figure 4.6 shows the diagrammatic representation of the set of constructs for structured EFD's.

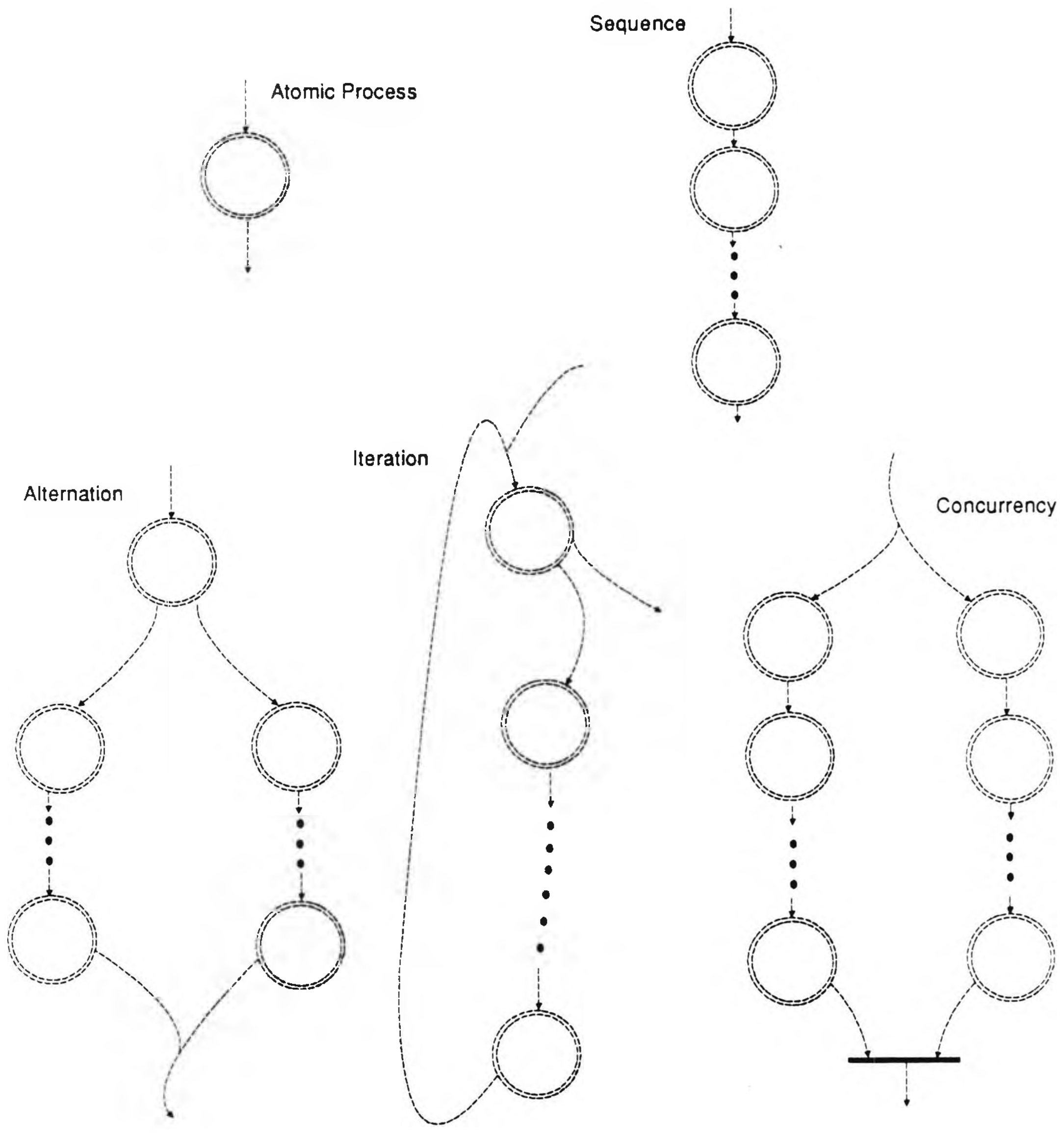


Figure 4.6: Basic Structured EFD constructs

In a similar way to the definition of structured programs (using only D-structures), the new set of constructs can be used for definition of well-formed diagrams. Processes on such diagrams will be restricted to having a single input event flow and a single output event flow (multiple output flows indicate a choice of execution threads). The elegance of Boehm and Jacopini's theorem results from it simplicity. This simplicity is due to the recursive nature of the definition for a structured program. Imposing a similar recursive definition on EFD's produces several associated problems.

First, although atomic processes normally output an event, there are many simple processes, such as those that clear a data store at the start of a processing cycle, that do not output an event at the completion of their task. The firing event for these processes is simply absorbed by the process and plays no further part in the operation of the system. In structured programs every strand of execution eventually emerges from its block: lines are joined from the termination points of alternatives within the block. For EFD's to follow a similar convention, the EFD notation must be extended to include *earthed* events emerging from such atomic processes.

Second, and most important, D-structures are defined recursively, which implies that when checking a program to see if it is a D-structure or converting one to be a D-structure, a top down stepwise strategy can be used. The definition of well-formed diagrams cannot inherit this recursive nature easily. This is because non-atomic processes usually have a number of input and output event flows. To place a one-in one-out structure on intermediate EFD processes will mean that only processes whose execution starts with a single event and ends with the generation of a single event can be grouped together. As a result of this, many processes which are grouped together for other reasons, e.g. because they share common data, have to be placed in unique groups. This will result in processes being brought up the hierarchy levels closer to the Context Diagram.

On the other hand, a common feature of real-time systems is the domination of control. In such systems the operations over a set of inputs starts with a single event and ends with the resulting output(s). For systems that exhibit this type of control domination, it may be reasonable to group processes according to control (see also the discussion under Methodology).

Third, many threads of system execution will involve interactions between the system and its environment. This implies that some events input to the system from its operating environment may be as a result of an event or data previously output to the environment. Since the diagram hierarchy only details the interactions between system processes, such event relationships cannot be derived from the diagrams. An analysis of a diagram hierarchy to ensure it conforms to structuring rules will fail some diagrams if such interactions are not included in the analysis. The basic need is to analyse and decompose the environment to the same level as the system itself. This forms the basis of the CORE viewpoint analysis approach [Mul84].

However, expanding the system boundary to encompass the details of the environmental interactions may result in large specification, parts of which are not part of the internal workings of the system being specified.

Fourth, whereas any given program can either satisfy or fail the conditions that prove it is a D-structure, a diagram that fails to satisfy the above formation rules may still be a valid diagram. For example, if two events that require synchronisation originate from distinct sources in the system's environment, the resulting EFD will fail to satisfy the above requirements for a well-formed diagram: it fails the concurrency structure. Such a synchronisation may, however, be part of the system's operational requirements. Further examination of the event sources may reveal that the events originate from the same ultimate source. The specification boundary may also have to be extended to include environmental processes to ensure a set of specification diagrams conform to structuring rules.

4.6.3 Concluding Remarks

The above discussion indicates that there are difficulties in the application of structuring theorems to event flow diagrams. The ability to judge a derived specification and provide some measure of quality is, nonetheless, an invaluable analysis (and management) asset.

The simplicity of Boehm and Jacopini's theorem is a direct result of its recursive definition. This recursive definition is possible because their constructs do not include a program's interactions with its environment, i.e. data exchange with the programs environment is not included as part of the control flow model. Such exchanges are explicitly indicated in EFD's by showing the (implicit) events carried by the data exchanges between system processes and between those processes and the system environment. As pointed out above, a similar recursive definition for EFD's may result in difficulties when deriving and presenting system specifications.

Furthermore, a system execution thread may include some exchanges with the environment. This means that a pair of events exchanged between system and its environment may be part of the same *dialogue sequence.* In order to establish such relationships between events, the system model should include a more detailed description of the environment. In other words, quality analysis must examine a *closed world* model of the system, where a series of correlated events must be easily identifiable.

Therefore, it is possible to apply a structuring theorem to EFD's, but since it must deal with constructs that portray much more information than D-structures, the rules should be relaxed. An automated aid can then incorporate these rules, and check derived specifications against them, so as to give some measure of quality.

4.7 Design And Implementation

As pointed out by many practitioners of systems analysis and design, a system specification should be devoid of any implementation bias [HS87]. In addition, the analyst should not have to be concerned with complications such as errors due to a system's operating environment. For these reasons, a perfect operational environment is assumed for a system while deriving its specification. This will ensure that the resulting specification does not have to deal with the complexities of the system's environment, and will give a pure description of that system's operational behaviour.

Once the specification has been derived, it must pass through a *design* stage, which will impose most of the restrictions due to the system's environment. In addition, it must deal with many other issues such as the system's interactions with the agents in its environment, e.g. people, and software organisation for the target hardware, i.e. the specification is *enhanced* by adding the processing required to deal with implementation specific concerns. The outcome of the design stage will pave the way for the implementation of the proposed system. In other words, specification states *what* the system operations are and design states *how* they can be achieved in a particular host environment.

Although the nature of many of the tasks that make up the design stage depend largely on the particular system under design, the characteristics of its operating environment, and the specifics of the proposed host hardware, there are some general points that can be applied while going from a system specification to its design. These result in enhancements to the derived specification, which cater for the real-world issues the system must deal with.

To investigate the applicability of these guidelines, an implementation of one of the specification exercises was undertaken. The Petrol Station example not only covers many of the new notation's features, it also includes operational concerns such as endangering data integrity by allowing multiple access to stored data. Hence this example was selected for the implementation exercise. The host environment chosen was the Sun workstation family and the external entities such as printers and pumps were simulated by (UNIX) processes.

Before commencing with the design and implementation, the specification was completed by adding the minispecs and the dictionaries to it. The complete specification is given in Appendix B. This Appendix also includes the final implementation code. The implementation exercise highlights a number of useful general guidelines for deriving a design by considering the system specification and the environment within which it must operate. These are outlined below.

4.7.1 System Interactions

The introduction identified the interactions of the system with its environment as one of the issues that the design stage should address. The specification stage does not distinguish between the agents the system communicates with, but since the system must interface with its human users in a radically different way from the way it interacts with other agents in its operating environment, the user interface is usually of particular interest during design. The interactions of the system with its environment will, therefore, require specific software to be designed in addition to the software already specified. In the Petrol Station exercise, for example, an entire module is dedicated to the user interface. The user interface can be designed in an implementation free manner, so that it can be used along with the specification for implementation on a particular target environment.

Mechanisms for system interactions with entities in the system environment other than its users are, of course, entirely dependent on the host system and the exact specifics of the operation of those entities. In the exercise presented here, for example, the operating system facility of pipes has been used to effect communication between (pump and monitoring) processes.

4.7.2 Process Groupings

The division of the software into modules may be based on a number of criteria. The subsystem divisions in the specification may have already provided a natural grouping of atomic processes into software modules. The atomic processes grouped under a common subsystem can be the routines collected together in a software module. This is true of the Petrol Station example.

In an environment which allows highly concurrent implementations, minimum interfaces between modules allows the inherent concurrency in a system to be usefully exploited, resulting in efficient systems. If the interface across concurrent tasks is small, the tasks can proceed independently and at their own speed, with minimum interaction with other tasks. In the implementation of the Petrol Station, for example, the pseudo concurrency of (UNIX) processes is used to carry the concurrency in the specification through to the implementation. The interfaces among the groups are implemented through pipes, whose asynchronous nature allows more concurrency than would be possible if synchronous communication is used.

As well as grouping processes together, process groups may be broken up to give a better implementation. Different reasons may compel designers to make such a decision. Maximising the concurrency in the implementation is again an important factor here. As well as the high level concurrency present amongst subsystems, there is often local concurrency between atomic processes within a group. In some cases it may be desirable to separate these processes in the implementation. The stock maintenance subsystem, for example, has been divided into two parts. The first part is incorporated into one the main system (UNIX) process modules, while the monitoring part is placed in a process of its own.

4.7.3 Error Handling

Error checking and handling are part of the additions to the software during the design stage. In an imperfect world, system users are fallible. Possible errors must be trapped and the system must be able to recover from them. This is a major part of the system enhancements during the design stage.

Errors may result from a number of sources. Communication between the system and the agents in the environment can seldom be guaranteed to be error free. It is necessary to ensure the validity of system inputs for correct system operation. If such validations are of critical importance, then this may make up a substantial part of the final design.

Like other strategies, error detection and recovery mechanisms depend largely on the system environment. This is particularly true of errors in inputs from peripheral devices connected to the system, but errors present in the input from its users may be predicted in many instances. The example in Appendix B includes error traps and recoveries for all user inputs in its design.

4.7.4 Host Services

The assumption of a perfect system operating environment implies that data integrity is guaranteed by system processes. In reality, software designers must be able to provide such a guarantee through correct system implementation. The services offered by the host environment may provide the ideal solution to many of the problems due to the imperfect system environment. File locking mechanisms have, for example, been used in the example here to ensure exclusive store access to processes, which guarantees the integrity of the data in those stores.

Designers may be able to satisfy many other system requirements by using services offered by the host environment. The use of pipes for communication amongst processes, for example, provides the ideal mechanism for the exchanges between the system processes in the petrol station implementation. Another example of a host service used in the implementation of the Petrol Station is the use of the operating system timing facility "sleep". The call to this routine performs the task of the "Clock" process, eliminating the need to write a special purpose timing routine. Using host services may alleviate the task of designing and implementing many of the functions required by the system.

4.7.5 Process Firing and Enablement/Disablement of Subsystems

The execution sequence of atomic processes, which may be implemented as target language routines, is clearly indicated by the firing agents for those processes. By noting the firing event for an atomic process, a call may be placed to the corresponding routine when the conditions satisfying the event are met.

The enabling and disabling conditions for each subsystem are clearly specified on its Subsystem Control Diagram. The mechanisms through which these are implemented depend on the complexity of the system and its control structure. It was found for the example system that many such enablements and disablements can be effected through the use of semaphore like flags which act as switches to allow/disallow operation of subsystems.

4.7.6 A General Design Hueristic

The dynamic behaviour of a system specification can be viewed in a number of ways. The first considers each part of the system to operate over one set of inputs: once a set of inputs enters that part of the system, no other inputs are admitted until the corresponding set of outputs have been produced. A second view may consider system parts as *pipelines*: once a set of inputs has passed through one processing stage another input set can be admitted into the pipeline, i.e. processing of inputs is overlapped in time. The Petrol Station System exhibits both types behaviour. Only one set of prices are changed at a time, whereas a pump monitor may overlap serving two customers (a new customer can receive petrol while the previous transaction is paid for).

Yet another view may consider inputs to cause instances of the program to be made available; a second set of inputs uses a second set of copies, i.e. each set of inputs initiates a process in the same way as operating systems creating copies of programs to operate on each set of inputs. The choice of which view is taken is determined by the design methodology used.

4.7.7 Concluding Remark

There are many factors that may influence the design and implementation of a system. As pointed out in the introduction, these are mainly determined by the constraints of the environment within which the system must operate. The guidelines above are the result of a relatively small implementation exercise. To formulate a general set of guidelines a more comprehensive study of the applicable techniques must be carried out (see also Chapter 6). Apart from the factors pointed out above, the design and implementation decisions for a system may be affected by many other factors. These include reliability, safety, maintainability, testability, cost, available technology, performance, growth, and expansion capability. The degree of importance of these factors varies depending on the nature of the system under design.

4.8 Methodology

Many advocates of system specification and design notations have recognised the importance of accompanying a notation with a methodology. The latter would not only provide techniques for deriving a specification, it would also define techniques for validating that specification. It is not enough to provide the tools for specifying a system's operational behaviour; a strategy must be provided for deriving that specification. Without such a strategy a specification notation can be misused or incorrectly used, resulting in low quality or even incorrect specifications, in the same way as programming languages can be used to write bad programs if structured programming techniques such as stepwise refinement are not followed.

However, the nature of every system is unique, and different systems lend themselves to different specification derivation approaches [WHF82]. As pointed out by Levy [Lev86], "while the general principles and objectives of a software development method may be the same for most projects, the embodyment of the method will probably vary with the characteristics of the specific product being developed". This implies that a rigid methodology will be a hindrance rather than an aid to the analyst. The methodology must be flexible enough to allow the analyst to select the appropriate derivation strategy for the particular system being specified.

Although a specification, derived using structured analysis, is presented in a top down manner, it is rarely derived in that way [CCW89]. The incremental presentation of detail will greatly ease the task of reading specifications, but it is not necessarily the best way to approach their derivation. Bottom up strategies can, on the other hand, overwhelm the analyst' when specifying large systems. A middle out strategy can, therefore, provide the best results. The point at which analysis commences is largely dependent on the system to be specified, but some guidelines may be provided for the analyst.

Even when a top down strategy is not employed in deriving a specification, the most useful point to start analysis is the system context, i.e. its interactions with its environment. Many methodologies aimed at the early stages of the system life cycle identify this activity as one the most important. The viewpoint analysis of CORE [Mul84, Som89] and the entity/action step of JSD [Sut88, Jac83] are, for example, specifically aimed at deriving a model for the system environment. Interactions with the environment include both data and events exchanged between the system and the agents in its environment. Once the data have been identified, they can be further subdivided into two groups: active and latched. Every piece of data exchanged between the system and its environment falls into one of these categories. Identifying the type of all system data exchanges will help the analyst in later stages of the analysis, especially when determining the firing agents for atomic processes (as only active data can provide the firing event for such processes).

Once all events, including implicit events carried by the data, have been identified, the system responses to each event may be examined. For example, in the Petrol Station exercise when the event "Take Stock" occurs, the system must respond by disabling the printing of any further reports and the commencement of any further transactions. Ward and Mellor [WM86] place great emphasis on identifying events and the system responses to each event. Identification of these responses will again aid in the later stages of the analysis (when determining subsystem and atomic process behaviour).

Event-response analysis plays a more dominant part in deriving the specification of real-time systems when compared with specification derivation for other types of systems. This is directly due to the control intensive nature of real-time systems. Consideration of responses to events can also guide the analyst in identifying subsystems. A subsystem can be viewed as a processing unit that, while enabled, continuously processes a stream of events. The latter includes internal and external events, some of which may be implicit in the arrival of data.

Once the analysis moves within the system internals, there is already a body of well established guidelines [DeM78, WM86, HP88, MP84] to provide help in deciding how to divide a process's task into a subnetwork. The most important of these are keeping process interfaces to a minimum and grouping closely related data in data stores. The former will result in diagrams that are not only clear and easy to follow, but will also help in the later stages of design and implementation. The less the communication among system processes, the more the concurrency in the system can be exploited. Since the data used and stored by many real-time systems is of a simple nature, the second guideline is perhaps less applicable to these systems than to data processing systems.

Other factors can be used to guide the derivation of system specifications. An overall examination of system requirements often leads to clear identification of many system subtasks. These are parts of the system whose operation should clearly be separately described. A typical example of this is instances of identical subsystems. For example, in the Petrol Station example, the parts of the system dealing with each pump's operation are delegated to a subsystem. Further examples of systems with identical subsystems can be seen in the Patient Monitoring System and the Bottling System of Appendix A. Another case of separation of system activities into distinct subsystems occurs when one subsystem controls other parts of the system by reacting to the system environment. For example, in the Bottling System of Appendix A, the pH monitor suspends and restarts other parts of the system according to the current pH of a liquid.

In many systems, the first subdivision of the system can be derived by the clear identification of subsystems in the requirements document. In such cases as the Autoteller System example of Appendix A, there are clear subtasks for the system in achieving its overall goal. This feature can be identified more easily for systems whose overall activity is sequential, although many simple tasks may be performed in parallel. For instance, the Autoteller system "validates the customer's card", "requests a service selection", and "performs the selected service" in strict sequence. Although such a feature may not be immediately identifiable for a system exhibiting highly concurrent behaviour, subsystems can still be identified by a clear subdivision of the system activities into subtasks. Using criteria such as grouping processes which take part in a dialogue sequence between the system and its environment, the system subtasks can be identified.

Furthermore, guidelines given here for atomic process identifications will help the analyst in deciding when to stop expanding a process, at which point the firing agent for that process must be identified. This may be found in the events inherited by the process's network from its parent, or an event generated by one its neighbouring processes.

The above guidelines coupled with the experience gained through repeated application of the notation can provide an invaluable aid to the analyst during system specification. Many of the techniques outlined here are based on a functional decomposition strategy. There is currently a strong tendency in the system development industry towards this approach. In particular, notations based on Data Flow Diagrams have almost always been used to arrive at functionally based system specifications. The section on specification quality above demonstrates that this may not always be the appropriate strategy to follow for the class of systems we are considering here. A control decomposition method may be more appropriate. Once all input events have been been identified, they can be followed inside the system to identify the internal system actions they may cause.

The above strategies are the result of the experiences gained through applying the new notation to a number of case studies. The outcome of this exercise is given in the sample specifications of Appendix A. A more detailed study of the techniques appropriate for the derivation of real-time system specification is required to arrive at a solid methodology that results in specifications whose quality can be judged.

4.9 Conclusions

A new extension to Data Flow Diagrams was presented which aims to remove the limitations and disadvantages of current DFD extensions used in real-time system specification. The new notation defines more precisely the semantics of processes used on the diagrams. DeMarco DFD processes do not have a precise definition: each process *somehow* transforms its inputs to its outputs. However, processes at the leaves of the system process hierarchy in this notation are identified by using objective criteria and the condition for starting each one is clearly shown by the single event flow entering the process. Each atomic process starts when its firing event occurs, and produces its outputs at the conclusion of its task.

Moreover, high level concurrency can be described by the Subsystem Control Diagram in a fashion that is not restrictive. Any concurrency identified within the system operational behaviour can be specified easily. The enablement/disablement conditions for each subsystem are then indicated on the SCD. This capability, plus the precise definition of the operational characteristics of each atomic process, allows a system specification to be *animated* in a token style execution model. Such an animation will be an invaluable aid to analysts in determining whether the derived specification conforms to user expectations.

Furthermore, the separation of the data and control views of a specification allow separate analysis of these system aspects. A single view of any system level can be composed by superimposing its EFD on its DFD.

Hence the new notation builds on the well established practices in the system development industry by using experience of and familiarity with existing notational conventions to allow easier derivation of system specifications, which can be comfortably understood by the parties involved in the development process, and whose operational characteristics can be studied before proceeding to system design and implementation.

The following chapter presents four other approaches to system specification/design and compares them to the new notation.

Chapter 5

Related Work

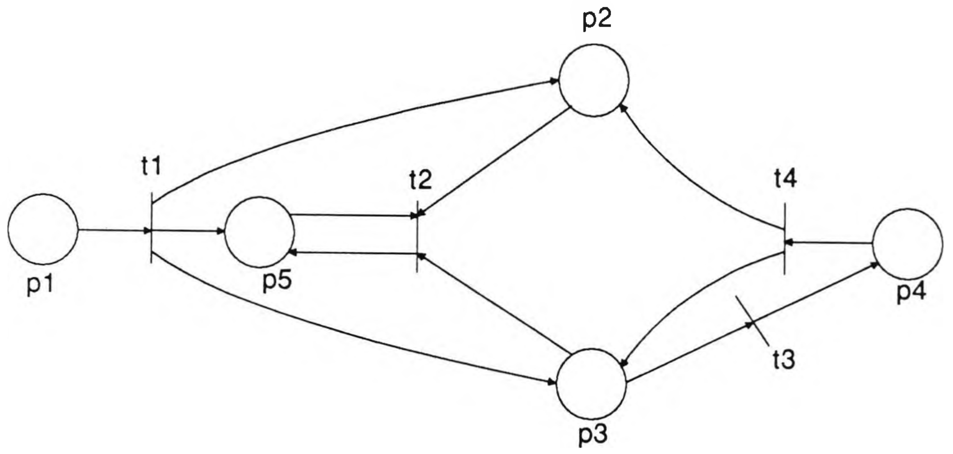
5.1 Introduction

In this chapter some of the relevant software specification/design notations are discussed. There are a wide variety of notations currently in practice. These include, amongst others, object oriented approaches [Boo86, Mey88, SM88, PCW85, PC86], formal mathematically based approaches [Hol88, Bjo87, Hoa85, Mil89], specification languages [Zav85b, Zav82, ZS86], FOREST [CFG+85, JKM86, FP86, BEF+86, Mai86, PFAB86], SARA [WE82, Est78, EFRV86], STATECHARTS [Har87, HLN+88, Har88, HPSS87], and SREM [Alf77, Alf85]. Since it is not practical to cover them all, and because many share the same underlying principles, a subset of these notations has been chosen. These are currently popular in the specification and design of concurrent and real-time systems both in industry and in academic institutions. The selection includes a control flow analysis notation (petri nets), a functional hierarchical notation which incorporates information hiding (MASCOT), an object oriented entity/action based notation (Jackson System Development), and a formal algebraic language (Communicating Sequential Processes). The first four sections below present an overview of each of these notations. The final section discusses the suitability of each notation for real-time system specification by comparing them through a set of criteria.

5.2 Petri Nets

5.2.1 Basic Concepts

A *petri net* is a bipartite directed multigraph consisting of two types of nodes: places and transitions. Algebraically, a petri net can be described by a four-tuple C=(P,T,I,O): P is a set of places, T is a set of transitions, I is the input function and O is the output function. In a graphical notation, places are represented by circles and transitions are represented by vertical bars. Directed arcs connect places and transitions. An arc from a place to a transition defines the place to be an input of the transition. An output place is indicated by an arc from the transition to the place. Multiple arcs may connect a place and a transition. An example petri net, adapted from [Pet81], is shown in figure 5.1.



$$P = \{p_1,p_2,p_3,p_4,p_5\}$$
$$T = \{t_1,t_2,t_3,t_4\}$$

$I(t_1) = \{p_1\}$	$O(t_1) = \{p_2,p_3,p_5\}$
$I(t_2) = \{p_2,p_3,p_5\}$	$O(t_2) = \{p_5\}$
$I(t_3) = \{p_3\}$	$O(t_3) = \{p_4\}$
$I(t_4) = \{p_4\}$	$O(t_1) = \{p_2,p_3\}$

Figure 5.1: An example petri net.

5.2.2 Analysing a Petri Net

Petri nets are primarily used to study the dynamic behaviour of a modelled system. In order to do this, petri net markings are introduced into the model. To mark a petri net *tokens* are assigned to places in the net. Tokens can be thought of as residing in places. Figure 5.2 shows a marking of the petri net in figure 5.1.

A transition can *fire* when it is enabled. A transition is enabled if each of its input places has at least as many tokens as there are arcs from that place to the transition (multiple arcs may connect a transition and a place). When fired a transition removes its enabling tokens and places a token, per arc, in its output places. In figure 5.2, t_1 is the only enabled transition. Figure 5.3 shows the new marking after it has fired.

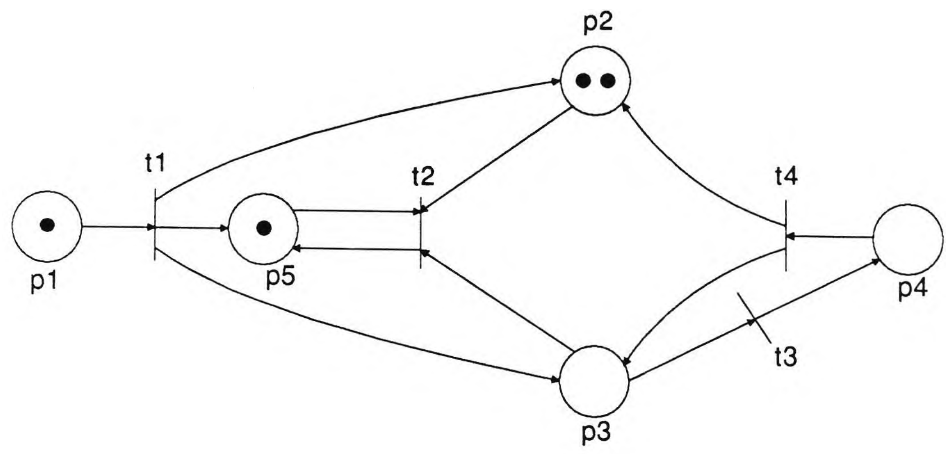


Figure 5.2: A marked petri net.

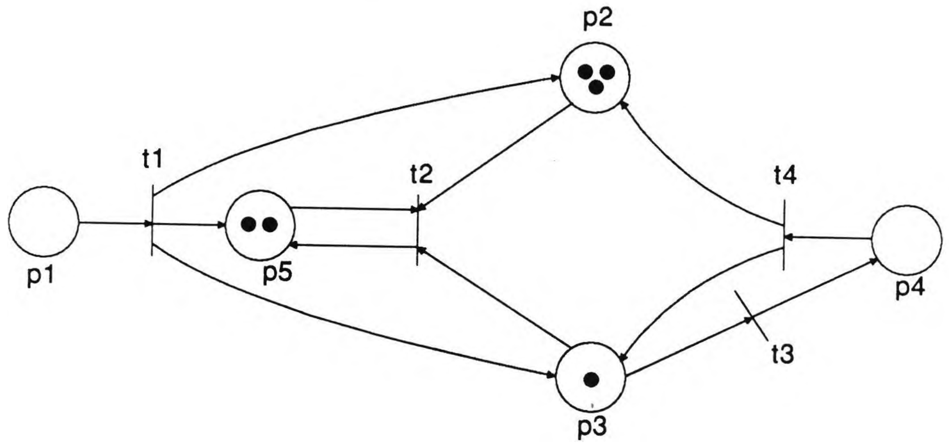


Figure 5.3: The marking resulting from firing t_1.

The state of a petri net (and that of the system it models) is defined by its marking. Firing a transition changes the marking and hence the state of the petri net. The dynamic behaviour of a system can be investigated by assigning an initial marking to its petri net model and observing its state changes by continuous transition firings. Execution of a petri net continues as long as there is at least one enabled transition. When there are no enabled transitions, the execution halts. A petri net model can also be statically analysed to discover system faults by reachability analysis, for example [Pet81, Pet77, Rei82, Age79, Jen81, GH81].

5.2.3 Modelling with Petri Nets

To derive a model the petri net view of a system concentrates on two primitive concepts: events and conditions. Events are actions that take place in the system. The occurrence of an event is controlled by the system state which can be described as a set of conditions. For an event to occur a number of (pre)conditions must hold. Its occurrence may cause other (post)conditions to hold. On petri net graphs, pre and post conditions are shown as markings and the occurrence of an event is represented by firing the appropriate transition. A system model is derived by identifying the events, i.e. system actions, and the pre and post conditions for each event [BO85].

Petri nets were designed specifically for modelling systems with concurrent interacting parts. Because of their general form petri nets have been used to model a variety of systems [Pet81], including computer hardware and software, interactions of subatomic particles, queuing theory, brain modelling and many others.

The original form of the petri net model has proved to be too simple and limited to model real systems [Pet81]. The first of these limitation results from the flat nature of petri nets. Because of the potentially large number of events in large systems, there is an enormous amount of detail to be considered at once. The proven principle of incremental topdown design cannot be applied easily when using such a model.

Furthermore, the flatness of petri nets will result in unmanageably large nets whose graphs are a spaghetti of arcs between places and transitions. Studying such a model so as to understand the behavioural aspects of a system is a difficult task and can be nearly impossible for a non-trivial system. A modelling technique that lends itself to a hierarchical design and presentation strategy will make it easier for a reader to grasp a system's behaviour by giving him/her adequate detail at decreasing levels of granularity.

In addition, the firing rules for a transition dictate a conjunction of its preconditions: all its preconditions must hold before a transition fires, i.e. they are *anded* together. There are many systems whose operational requirements may require an action to take place when just one of many conditions holds, i.e. an *or* of the preconditions. Similarly, when a transition has completed its task, its requirements may dictate that it places tokens in a subset of its output places. These cannot be modelled easily with petri nets [Bae73].

Extensions have been proposed for petri nets to alleviate some of these problems [Pet81, BM85]. For example, a transition can be expanded to a further petri net, which describes in more detail the steps involved in the transition's action. But the subnet must be started by the preconditions of its parent transition and must end by generating its parent's postconditions. As a direct consequence the place/transition groupings are restricted, which will result in restrictions during a topdown hierarchical design process. Many other extensions such as inhibitor arcs [Pet81], disjunctive input/output transitions with switches and token absorbers [Bae73], associating actions with arcs connecting places and transitions [BM85], attaching an execution time table to each node in the network representing a process in a system [CR83], and placing predicates on transition firing, have also been proposed.

Another limitation of petri nets is that they are essentially aimed at modelling the flow of control through a system. Processing and the flow of data through the system and between the system and its environment cannot be derived easily from a petri net model. A petri net model is, therefore, an auxiliary tool which should be used within a more all-embracing technique to completely specify a system's operational characteristics [Pet81, BO85].

Tse and Pong have proposed Formal Data Flow Diagrams (FDFD's) to create a notation that is both familiar and can be subjected to formal verification analysis [TP89] . This notation is based on traditional DeMarco type Data Flow Diagrams. In order to apply formal analysis techniques to FDFD's, the relationship among input/output data flows of a task (process) must be explicitly defined. This is done by placing *and* and *or* operators between data flows on diagrams. An FDFD can be mapped directly onto an extended petri net making it possible to apply many of the petri net formal analysis techniques to FDFD's. For the same reasons that DeMarco type Data Flow Diagrams are inadequate for specifying the class of systems we are considering (i.e. the inability to model events and the two levels of control), FDFD's are also not adequate.

5.3 MASCOT

5.3.1 Basic Concepts

MASCOT [Sim82, Bat87, Jac84, Dib82, Irv84] is an acronym for Modular Approach to Software Construction, Operation and Test. It is a machine and language independent approach to software design and implementation which has at its heart a particular form of software structure based on independent parallel processes, known as *Activities*, whose sole means of communication is through *Intercommunication Data Areas* (IDA's). It aims to represent, directly, the system's concurrent functions and the data flows between them. Its origins lie in work at the Royal Radar and Signals Establishment during the late 60's and early 70's. As a result MASCOT is primarily aimed at real-time embedded areas, where the software is complex and highly interactive [Bat87].

On MASCOT 3 ACP (Activity, Channel, Pool) diagrams, activities are represented by circles. Two special types of IDA were identified early in MASCOT. A *channel* is used to represents producer/consumer type communication in a similar fashion to data flows on Data Flow Diagrams. Data is written to the channel by the producer. The consumer removes the data from the channel by reading from it, i.e. reads from a channel are destructive. The producer and consumer are connected to opposite sides of the symbol to emphasise the nature of their communication. A *pool* represents a repository for data. Its role is similar to data stores of Data Flow Diagrams. Static data, which may be updated occasionally, is stored in a pool. Activities connected to a pool can read data from/write data to it, i.e. in pools writes are destructive. IDA's that do not fit either of the two specially identified classes of IDA (channels and pools) are shown on ACP diagrams with a rectangular symbol. Activities are connected to channels and pools by directed arcs, called *paths*, which show the direction of data flow. An environmental entity, known as a *device* or *server* in MASCOT, is represented on an ACP by a hatched rectangle. Figure 5.4 shows the basic components used in a MASCOT 3 ACP diagram.

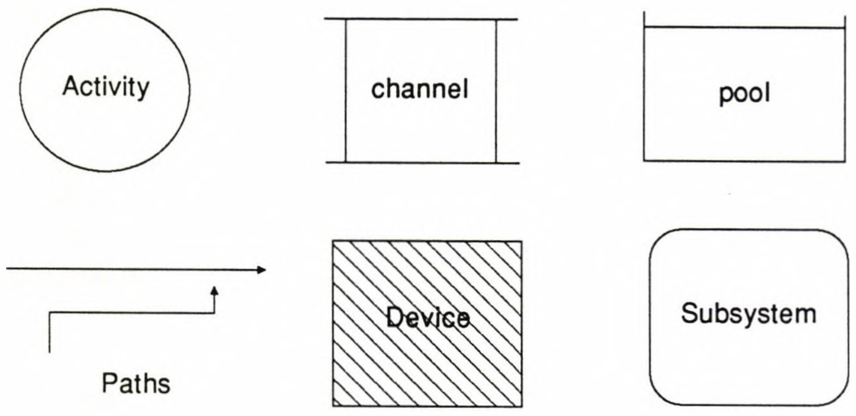


Figure 5.4: MASCOT ACP Diagram Symbols

The clear separation of active and non-active components in MASCOT is a direct result of the MASCOT designers' intention to provide a design method which caters for large scale concurrency. Activities and *MASCOT subsystems*, explained below, on an ACP diagram can be thought of as running in parallel. In order that these asynchronously executing concurrent processes exchange information in a secure manner, MASCOT provides mechanisms to effect mutual exclusion and cross-stimulation for use at the points where data is transferred to or from common storage areas. These mechanism are implemented through IDA *Access Procedures.* The data stored in an IDA is private to it. An activity can access this data only through calls to the IDA's access procedures. In MASCOT 3, an IDA offers a subset of its access procedures through each of its *windows.* An activity must connect to an IDA window through a *port.* The window and the port must be of the same type (see below). On ACP diagrams, windows and ports are shown with a small filled in rectangles and a small filled in circles, respectively. By using this window/port connection protocol restrictions can be put on the type of access permitted to each activity connected to an

IDA.

5.3.2 Deriving a MASCOT Design

The unit of construction during an MASCOT design is the *subsystem*. This is merely a collection of activities which have been connected to their IDA's at the same time [Bat87]. In MASCOT 2 a design was conceived as a flat data flow network with the subsystem as the basic construction unit. Because of the potentially large networks that may result from such a scheme, and to incorporate the advantages of incremental design, a design can be described in a hierarchical manner in MASCOT 3 [Bat87, Sim84]. The subsystem is still the basic unit of construction, but a subsystem may contain lower level subsystems. A subsystem is represented by a roundangle on an ACP diagram.

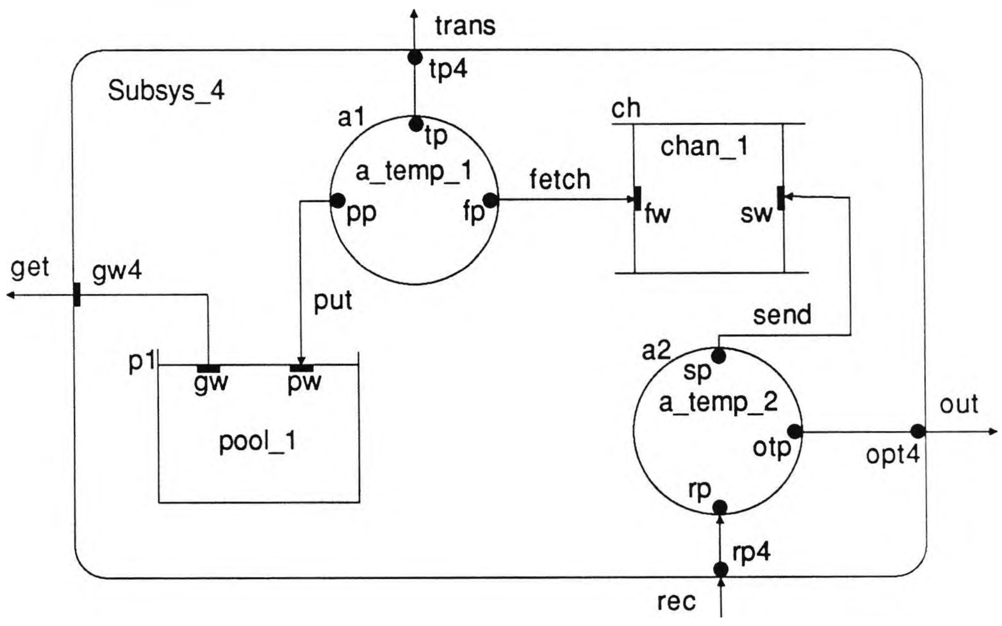


Figure 5.5: Example of a MASCOT 3 subsystem ACP

Software design using MASCOT can be divided into three stages: network design, component design, and integration/testing [Bat87]. The first stage consists of deriving a hierarchy of ACP diagrams. All the subsystems, activities and IDA's are identified and connected together on ACP diagrams. Figure 5.5 shows an example ACP for a subsystem. A MASCOT 3 ACP diagram contains more detail than just the connection of the components present on it. The name of the component being designed is placed inside the roundangle, "subsys_4" in figure 5.5. Ports and windows are named, e.g. "pp" and "gw". The type associated with a port/window pair annotates the path connecting them, e.g. "put" and "send". Each subsystem, activity and pool is not only labelled with a unique name, it also has the name of the template, explained below, describing its behaviour placed inside it. For example, the operation of activity "a1" is described by template "a_temp_1".

A special case of a subsystem, called a *system*, shows the external devices connected to the system as well the initial division of the system into subsystems, activities and IDA's. This subsystem does not offer any port or window connections, i.e. it is closed. Connections to the environment are represented by device servers (described below). These reside within the system (and nested subsystems) symbols. The subsystems in the system are then expanded into their own ACP networks. After subdividing the software system into its constituent activities and IDA's, those components whose implementation may be too complex can be further subdivided. A complex activity may be broken down into smaller components. Its ACP diagram shows the execution of its components which communicate via a procedural interface. In figure 5.6 the activity "a1" has been broken down into four procedures: "main", "sub1", "sub2" and "sub3". Their relationships are represented by lines bearing hollow arrow marks, known as *links*. Links represent procedure calls, e.g. "main" calls the other three procedures.

An IDA may also be composite, i.e. an IDA can be broken down into a connected network of IDA's. In figure 5.7 "cida" has been broken down into three component IDA's. They are connected together by window/port pairs. Using composite IDA's data can be completely hidden away from outside the IDA; only internal IDA access is allowed to such data. In this example, the data in the IDA "ex" is private to "cida". Complex activities and IDA's are shown with thick borders on ACP diagrams.

During component design, each component on every ACP diagram is designed in more detail. A *template* is used to describe the *module* for each component subsystem, activity and IDA. Other textual support, such as data type definitions, is provided to enable the complete description of a MASCOT design in text. The design of each component is carried out in minute detail. For example, a subsystem template will include the types of all the windows and ports it offers to its environment, the template types for the channels, pools and activities that are contained within it, and the connections amongst them; and the types of all paths are identified to determine the type of access allowed through a port/window pair. All the design data for a *MASCOT machine* is maintained in a database, so that identical software components can use the same template definition. Using this information many checks, such as checking compatibility of a port/window pair, can be performed on the designed software.

The final stage, integration and testing, consists of creating executable software for a specific hardware configuration from the designed components in the database. The definitions of the software components in the database

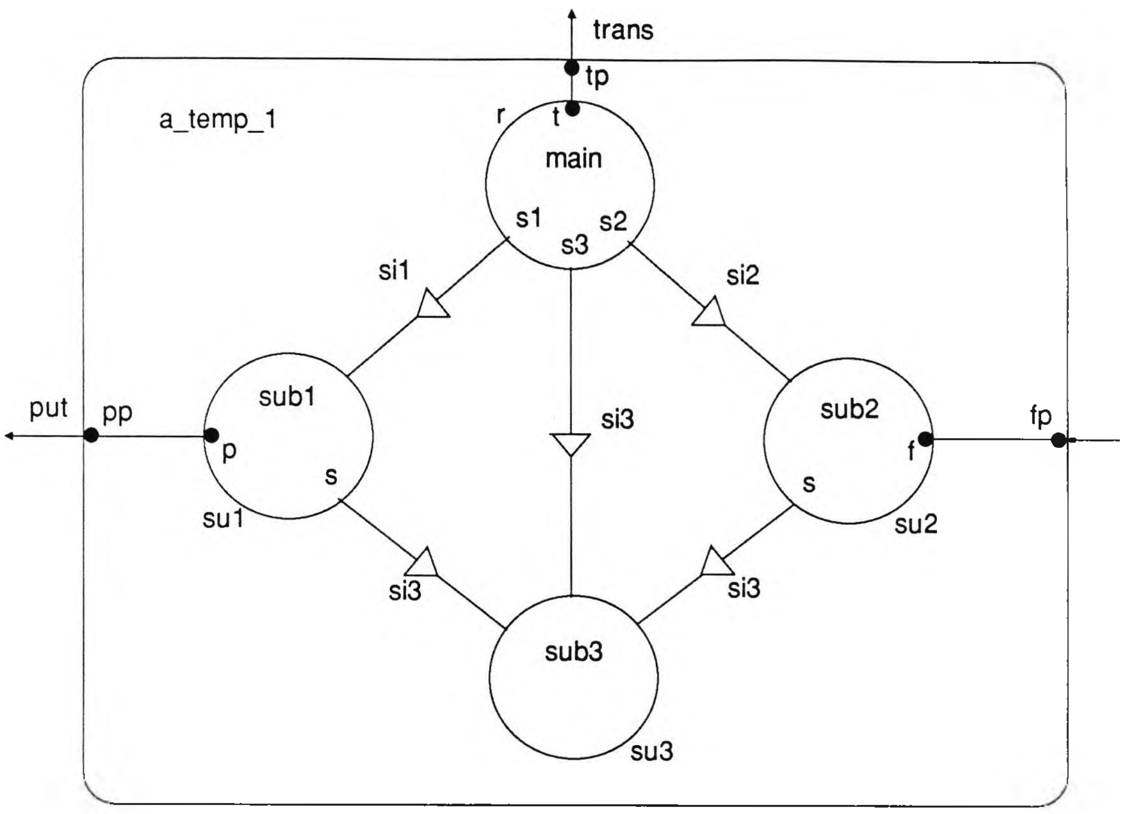


Figure 5.6: ACP Diagram of a Complex Activity

are mapped onto the implementation language constructs to be integrated with supporting software to run on the implementation hardware. This support is provided by the *Context Software* which provides a run time environment for the application to run in. This includes primitives for synchronisation of activities, device handling, scheduling of activities, allocating activities to processors in a multi-processor environment, monitoring software preformance and error handling. Context software primitives may be applied to activities and subsystems to start, stop, suspend or resume them. The software can then be tested to ensure that it conforms to its requirements. By using facilities provided by the context software performance can also be measured.

5.3.3 Concluding Remarks

MASCOT is aimed at deriving a design for a software system to run on a specific hardware configuration. Although the network design stage can be language independent, the component design stage may tend to be biased towards a specific target language (CORAL 66 was mainly used for MASCOT 2 designs [Fou84], but because of its parallel programming capa-

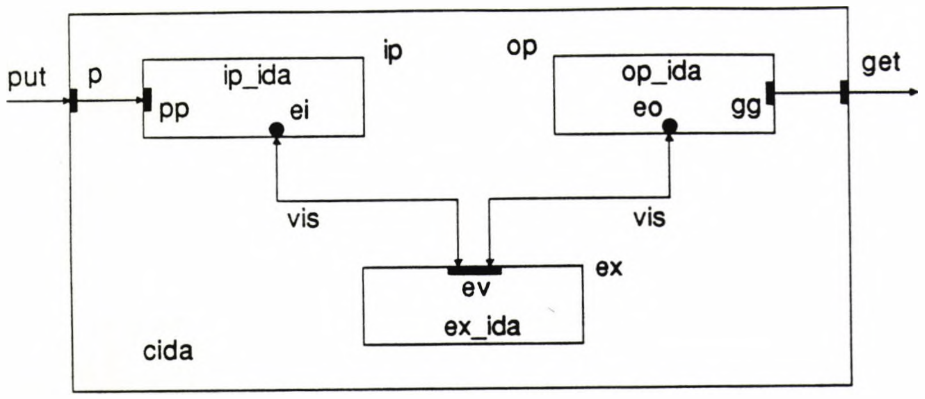


Figure 5.7: ACP Diagram of a Composite IDA

bilities, ADA is the recommended implementation language for MASCOT 3). Although many languages have been tried in MASCOT designs, the concentration on the main two languages, CORAL and ADA, may introduce a bias in the way tasks are divided between activities and how IDA's are chosen to store data. However, it should be noted that MASCOT's aim is to derive a design, and as such its outcome is expected to be more implementation biased than a specification approach.

The flow of control through a MASCOT machine is distributed among several parts of the software. Explicit synchronisation of activities is achieved by primitives that operate on special objects, called *control queues*. Since synchronisation takes place only in respect of access to IDA's and servers, described below, each control queue is conceptually part of the structure of an IDA or a server [Bat87]. Using the MASCOT *JOIN*, *LEAVE*, *STIM*, *WAIT*, and *WAITFOR* primitives, activities can join control queues to wait for service and leave them to be served [SJ79]. For example, when an activity requires access to the data in an IDA, it joins the queue for that IDA. An activity at the head of the queue is said to own the queue, and can use the LEAVE primitive to leave the queue for service. Cross-stimulation is effected by using the STIM, WAIT and WAITFOR primitives.

Other parts of the control flow in a MASCOT design are handled by the context software. Initialising, suspending, resuming and terminating of activities is delegated to the run time support environment in MASCOT. When a MASCOT system has been built, each of its components is said to be *unestablished*. Before any constituent activity can be executed it must first be *established*, i.e. all relevant initialisation code must be executed first. Once an activity has been established it can be started, suspended, resumed and terminated. These operations are part of the context software that deal with scheduling and execution control.

MASCOT is based on data flow. An activity is designed to execute when data is ready at its input channel. In order to achieve real-time behaviour an activity may receive an event. Events are also exchanged via channels. MASCOT, therefore, does not distinguish between the two types of exchange. The ordering of activity executions is, hence, achieved by the exchange of data and events via channels. Such orderings cannot be derived from any part of the MASCOT network diagrams. In many systems, particularly those with a real-time nature, the order of activity executions and the conditions under which subsystems are enabled and disabled form a vital part of the requirements document. It is, therefore, important to be able to derive these easily from the specification.

MASCOT, therefore, differs mainly from our approach in its design specific characteristics. Whereas our aim is to derive a *specification* of a system, MASCOT is aimed at deriving the final design for it. The provision of special processes, called *servers*, to handle the environmental interactions of MASCOT designs is a clear indication of MASCOT bias towards hardware specific design. Requirements specification techniques such as CORE [Mul84, KNPW88, KN88] have been recommended for use in the early stages of a MASCOT design [Bat87, SJF88] so that the MASCOT user can derive the system requirements before proceeding with a design.

The role of diagram hierarchy in our notation is to allow a reader to comfortably derive the flow of data and control through the specified system. Such information cannot be derived easily from a set of MASCOT network diagrams. MASCOT uses the principle of information hiding by forcing all data accesses by activities to go through IDA access procedures. Although this will ensure data integrity as well as safe synchronisation among concurrent activities competing for access to the same data, the diagram readability is somewhat impared. Some processing may also be hidden. In composite IDA's, for example, an access procedure may cause data private to the IDA to be updated through a call to the internal access procedures of that IDA, e.g. the data in "ex" in figure 5.7 may be updated by either "ip" or "op".

Although MASCOT provides a rigorous technique for deriving a design for a software system, which will aid designers in deriving safe designs for highly concurrent systems, it does not meet the objectives of our notation (to show clearly the flow of data and control through a specified system).

5.4 Jackson System Development

5.4.1 Basic Concepts and Design Derivation

Jackson System Development (JSD) [Jac83, San89, Sut88] is a system design approach which has its roots in a program design methodology, Jackson Structured Programming (JSP) [Jac75, Cam82, Cam83]. Unlike other methodologies, JSD is not based on a hierarchical functional decomposition strategy. Instead, emphasis is placed on first modelling the system's environment; and only then going on to consider the full details of the tasks which the system is to perform.

The JSD design method is subdivided into three stages: *modelling stage*, *network stage* and *implementation stage*. JSD starts to define the subject matter by describing the real world of the system in terms of *entities* and the *actions* they perform or suffer. In other words, a conceptual boundary is drawn around the aspects of the real world which are closely linked with the operation of the system. An *entity* is an object of interest in the system which will undergo or cause change during the system's activity [Sut88]. An *action* is an event which happens to an entity. Jackson [Jac83] suggests that entities and actions can be derived by listing the nouns and the verbs, respectively, of the requirement document for a system. Strategies are then introduced to shorten these lists, e.g. aliases for entities are eliminated.

Once all entities and the actions for every entity have been identified, each entity is considered in turn and its actions are arranged in time ordering. A *Process Structure Diagram (PSD)* is used to show the relationship between each entity and the time ordering of the actions it performs or suffers. PSD's allow the expression of the three classical constructs of structured programming: sequence, selection and iteration. Figure 5.8, adapted from [Jac83], shows an example of a PSD.

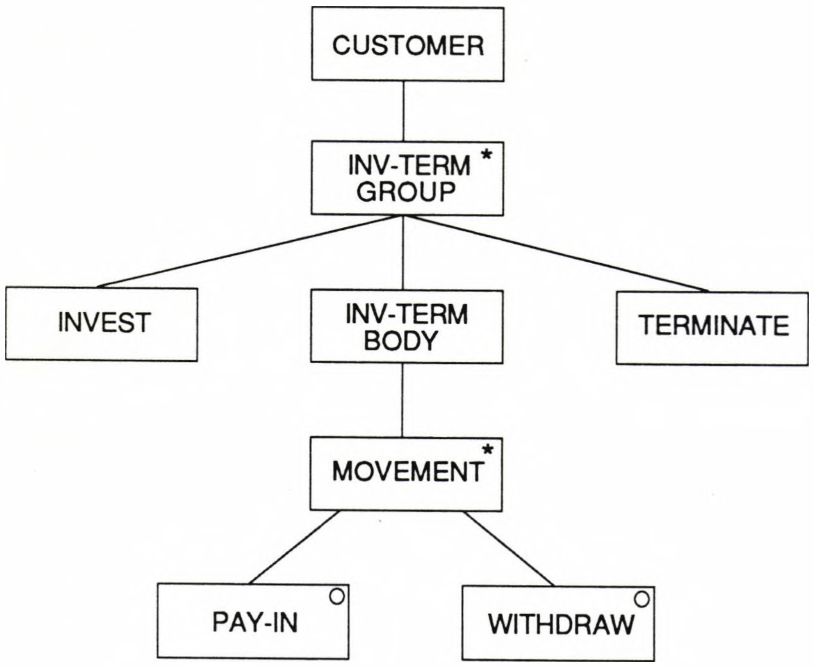


Figure 5.8: Example Process Structure Diagram

Process structure diagrams are tree shaped. Each node is a named rectangular box, which may be an entity, an action, or a name for a collection of actions. Sequence is indicated by left to right ordering, e.g. the customer *INVESTs* before he *TERMINATEs* his account according to figure 5.8. Selection and iteration are shown by a circle and an asterisk, respectively, placed in the top right corner of a rectangle. For example, "INV-TERM BODY" is composed of zero or more "MOVEMENTs", each of which is is either a "PAY IN" or a "WITHDRAW". The root of each tree is an entity; the actions that happen to it are at the leaves of the tree. PSD's can be converted to *structure text* which will, in addition, include elements such as executable operations and conditional tests that do not fit easily into a PSD. For example, the structure text for the structure diagram of figure 5.8 is:

```
CUSTOMER seq
   read C;
   INVEST; read C;
   INV-TERM BODY itr while(PAY-IN or WITHDRAW)
      MOVEMENT alt(PAY-IN)
         PAY-IN; read C;
      MOVEMENT alt(WITHDRAW)
         WITHDRAW; read C;
      MOVEMENT end
   INV-TERM BODY end
   TERMINATE;
CUSTOMER end
```

The capital letter words are the entity described or its actions and the bold typeface is used to show the constructs, e.g. sequence (**seq**), of the structure diagram. Other elements shown in the structure text are termination conditions for the loop and the points at which the process *read*s from its data stream C, see below.

In the *network stage*, the description of reality, in terms of entities and actions, is realised in a process model and connections between the model and the real world. A set of processes is specified which model the real world entities and their behaviour. The skeleton for the behaviour of each entity has already been derived in the previous step in the form of a process structure diagram (or structure text). The only remaining aspect to be specified for each entity is the way it communicates with other system entities and the system environment.

Processes can connect together in two ways. In *data stream* connection, one process writes a sequential data stream; the other process reads this stream. In *state vector* connection, one process directly inspects the state vector, i.e. the internal local variables, of the other process. *System Specification Diagrams* (SSD's) are used to show process connections, figure 5.9 [Sut88].

Data stream connection is shown by a circular symbol connected to the producer and consumer by lines, with an arrow head at the consumer

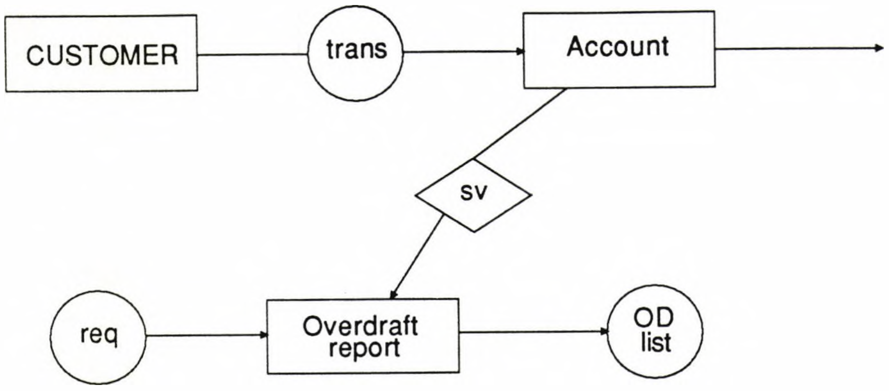


Figure 5.9: Example System Specification Diagram

end. Events (an event in JSD is the point in time when something happens [Sut88]) are passed from the producer to the consumer down the data stream. Actions respond to events. These events are communicated to the system as data messages and are referred to in JSD as *attributes* of the action. Entities also have attributes. These are internal data which record what stage the entity is at in its life history, i.e. the *state* of the entity. Read and write statements are added to the structure text for each process to indicate places at which (data stream) communication occurs.

State vector connections are shown by a diamond symbol. The direction of data transfer is again shown with an arrow head at the receiving process. Data stream connections represent producer/consumer type connections, whereas a state vector connection is used when data is read on demand. A special operation, *Get SV*, is used in structure text to indicate when a reader inspects a state vector.

Communication with the system environment is modelled by using both types of connection in JSD. Data stream connection resembles that of a discrete data flow; and state vector connection is like data reads from externally stored data or continuous data flows.

Once all the real world entities have been modelled, other processes (which are required to complete the operational requirements of the system) are added to the model. Jackson [Jac83] uses the term *function* for these processes to separate them from those that model real world entities. Entities represent groups of time ordered actions which describe a significantly long life history of something within the system, whereas functions are a set of actions which take place in a short space of time to accomplish a task [Sut88]. These functions fall into two categories: *embedded* and *imposed* functions [Sut88]. Elementary operations may be added to existing model processes. These are the embedded functions. Larger changes, such as functions to produce system outputs, impose new processes on the model.

A change in the system's operational requirements may also prompt the addition of new processes into the model. For example, the "Overdraft report" function in figure 5.9 is an output function which generates an overdraft report when the account goes overdrawn. Other function classes include those responsible for input validation and user interface.

The final stage in JSD is concerned with converting the derived specification into an executable program. During this stage, the developer considers what hardware and software should be provided for the system, and applies the techniques of transformation and scheduling along with techniques of database definition to allow the system to be efficiently and conveniently run.

5.4.2 Concluding Remarks

JSD views a system as a set of concurrently running processes which communicate with each other by messages [Sut88]. All the model processes can, in effect, be considered to run in parallel. Since each PSD is dedicated to modelling one process, this concurrency is not apparent from within any single PSD. It is the collection of PSD's that represent the concurrency in the system.

A process can execute as long as it has data available to operate on. When no data is available, a process is *blocked.* The execution of ready processes, i.e. those with available input data, is controlled by the scheduling mechanisms built into the system through system processes and the dedicated scheduler. JSD aims, therefore, to model only low level concurrency amongst model processes. Concurrency of groups of processes is not modelled explicitly.

Hence, there is no notion of a "subsystem" in JSD and control of a group of processes can only be effected through mechanisms distributed amongst model processes and the scheduler. No single diagram (or text) specification can be consulted for a description of subsystem control at any level of granularity. If part of a system is to be disabled because of an error condition, for example, the disablement must be reflected in the PSD for every process in the group of processes belonging to that part of the system. One of the primary aims of the notation we have introduced is to enable an analyst to specify high level (subsystem) concurrency among groups of processes (by using Subsystem Control Diagrams). The specification of subsystem control is encouraged by our notation, whereas it may be left to the later stages of JSD. JSD, however, places (early) importance on the order of atomic actions. This information is portrayed in process structure diagrams for every model process.

The inability of JSD to model high level concurrency explicitly may be due to its designers' decision to abandon a top down (incremental) design strategy. JSD advocates design by composition [Cam86] rather than design by decomposition. In the latter high level processes are *exploded* to reveal further detail, i.e. a specification is gradually decomposed from a black box view of the system to its atomic functions. In JSD, small increments are precisely defined before being combined together to make up the whole system.

In addition, since JSD modelling starts at a very low level, potentially huge amount of detail have to be derived in the entity action step from a requirements document for a large system. Sutcliffe [Sut88] notes that one of the dominant difficulties in introducing JSD is the problem of not knowing where to start looking for entities. Sommerville [Som89] also points out this difficulty. For a large system the number of possible entities, and the actions performed by each one, can easily overwhelm the analyst with detail. Even experienced JSD users, who may be able to derive the entities and actions for such systems, may be faced with large and unmanageable communication models for these systems. Such models may result in huge cluttered SSD's if many system processes exchange information with each other. Godwin et al [GGS89] note that JSD is problem size sensitive and that for a large application the network diagrams can become extremely difficult to handle. In practice, it may be necessary to subdivide such systems into a number of parts before applying JSD to each part of the design.

Static (stored) data is not explicitly modelled in JSD. Such data is held in the state vectors of processes. It is only during the implementation stage of JSD that storage considerations for state vectors are taken into account. Storing data in process state vectors may also mean that a collection of data items which logically belong together (like those represented by data stores in DFD's) may be spread among several processes. As a result a process that requires access to a number of data items will require several state vector connections. In addition, if a data item is in demand by several processes, the process storing that item will have many state vector connections. Both of these may result in cluttered SSD's. Furthermore, since a state vector connection is read-only, the only process capable of updating a particular data item is the one storing it. Other processes have to achieve updates through messages passed to the process holding the data item.

JSD provides a methodology which can lead a developer directly to a design. The last step of JSD provides good guidelines for the developer to convert a JSD specification derived in the earlier stages to a physical design. The approach taken by JSD does not, however, reach the goals that we have set for our notation. This is because it is not immediately clear from JSD diagrams what information flows between processes, or what the interfaces are to stored data items. Nor is it possible to see the flow of control through many parts the system. The flow of control at levels higher than the process level is a key feature of many systems, and its clear presentation is vital for the correct specification and implementation of such systems. This information is spread amongst several processes, including the scheduler, in JSD designs. Furthermore, hierarchical presentation of information is the computer scientist's main weapon for dealing with the problem of scale in complexity and detail. JSD does not follow a hierarchical strategy, which makes it a cumbersome methodology when specifying and designing large systems. However, It should be noted that JSD follows an object oriented approach. Such an approach cannot be expected to reach many of the goals set for a functional decomposition strategy.

5.5 Communicating Sequential Processes

5.5.1 Basic Concepts

Communicating Sequential Processes is an algebraic notation that can be used to specify, design and implement computer software systems [BHR84, Hoa85]. CSP recognises that input and output are primitives of programming and parallel composition of communicating sequential processes is a fundamental program structuring method [Hoa78]. In CSP, a process is an object whose behaviour can be described in terms of the limited set of events it can engage in. This set is named the *alphabet* of a process, and is denoted by the greek letter α. A prefix notation can be used to describe the behaviour of a process. For example, (x$\longrightarrow$P), pronounced x then P, defines a process which accepts the event x and then behaves exactly as described by P. A vending machine [Hoa85] that accepts coins and dispenses chocolates can be described as follows.

$$\alpha\text{VMS} = \{\text{coin, choc}\}$$
$$\text{VMS} = (\text{coin}\longrightarrow(\text{choc}\longrightarrow\text{VMS}))$$

This process has two events in its alphabet and is defined by a recursive formula which indicates that the process continuously accepts coins and delivers chocolates. However, the operation of many processes involves taking alternative routes of behaviour. Processes whose behaviour can be influenced by their environment are described by using the choice operator, |. For example, the process (x$\longrightarrow$P | y$\longrightarrow$Q) can initially engage in either of the distinct events x or y. After the first event has occurred, the process behaves like P or Q depending on whether the event was x or y, respectively. A process which accepts a coin and delivers either a chocolate or a toffee can be defined as follows.

$$\alpha\text{VMCT} = \{\text{coin, choc, toffee}\}$$
$$\text{VMCT} = (\text{coin}\longrightarrow(\text{choc}\longrightarrow\text{VMCT} \mid \text{toffee}\longrightarrow\text{VMCT}))$$

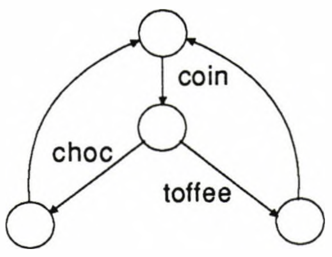


Figure 5.10: The Pictorial Representation of VMCT

The behaviour of a process can be represented pictorially as shown in figure 5.10.

These diagrams follow the traditional notation of state machines. Circles represent process states, and the arrows linking these states represent the transitions. Each transition is labelled with the event name that causes the state change. The root of the tree, usually drawn at the top of the diagram, is the starting state. Terminating states, if any, are usually drawn on bottom of the diagram.

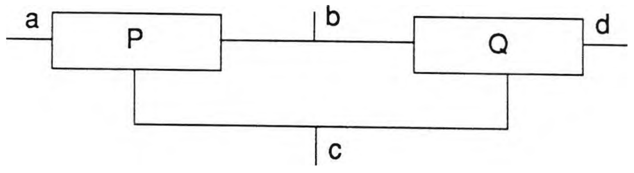
The behaviour of a process over a period of time can also be described by a *trace* of that process. This is a finite sequence of symbols recording the events in which the process has engaged up to some moment in time. Traces are shown using angeled brackets in CSP. For example, both of the following are traces of VMS.

$$<\text{coin, choc}> \qquad\qquad <\text{coin, choc, coin, choc}>$$

Complex processes can be constructed by combining simpler processes in parallel. When processes are combined in this way, they will often need to interact. Such interactions are regarded as events that require the participation of all combined processes. If the alphabets of the combined processes are the same, each event that actually occurs must be a possible event in the independent behaviour of each process separately, i.e. for an event to occur all processes are required to participate. In the case of the vending machine, a chocolate can only be extracted if the customer wants it and only when the vending machine is prepared to give it. Processes formed by the concurrent combination of smaller processes are defined by using the parallel operator, ||, e.g.

$$\begin{gathered}(\text{CUST} \parallel \text{VMS})\\ \alpha\text{CUST} = \{\text{coin, choc}\}\\ \text{CUST} = (\text{coin}\longrightarrow(\text{choc}\longrightarrow\text{CUST}))\end{gathered}$$

If the alphabets of the combined processes are different, the participation of all processes is only required for common events. Processes combined together by the parallel operator can be represented pictorially in a *connection* *diagram.* Each process is pictured by a named rectangular box from which emerge a number of lines each labelled with an event from its alphabet, figure 5.11 [Hoa85]. The lines for common events are joined together.



αP = {a, b, c} αQ = {b, c, d}

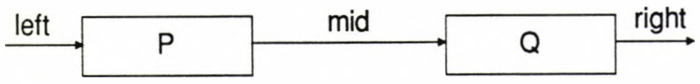
Figure 5.11: A Connection Diagram

Processes can be combined in other ways by using other CSP operators. The general choice operator, $\Box$, is used to define a process whose behaviour is determined by the first event that occurs. The process (c$\longrightarrow$P $\Box$ d$\longrightarrow$Q) will behave like process P if the first event c. If the first event is d, then the behaviour of the combined process is like that of Q after that event. If P and Q share a common first action, Hoare [Hoa85] indicates that the choice between which one is taken is nondetrministic. Processes can also be *interleaved* by using the interleaving operator |||. The interleaving operator is used when processes are joined together to operate concurrently but are not required to synchronise on common events. In this case each action of the combined process is the action of exactly one process. When a common event occurs, the choice between which process performs that action is nondeterministic. For example, a vending machine that will accept up to two coins before dispensing up to two chocolates is defined by (VMS ||| VMS). Processes can also be combined in sequence. This is indicated by using the sequence operator, ;. The process (P;Q) defines a process which first behaves like P and upon successful termination of P its behaviour is like that of Q.

In addition to using the above interactions, two processes can exchange information by using the CSP communication primitives. A communication is an event that is described by the pair c.v, where c is the channel on which the communication takes place and v is the value of the message which passes. The process (c!v$\longrightarrow$P) output the value v on channel c and then behaves like P. Similarly, the process (c?v$\longrightarrow$P) inputs any value v communicable on channel c and then behaves like P. For example, a process which reads a value from channel "in" and outputs that value to channel "out" is defined by

$$\text{COPY} = (\text{in?x}\longrightarrow(\text{out!x}\longrightarrow\text{COPY}))$$

The possible values communicated along a channel are denoted by $\alpha c(P)$, where c is the channel name and P is the process that engages in the communication event c.v. On a connection diagram for a process, the channels are drawn as arrows in the appropriate direction, and labelled with the name of the channel. When the processes P and Q in figure 5.12 are combined together, (P || Q), the value read on channel left is first doubled and then incremented by one before being output to right.



$$P = (\text{left?x} \longrightarrow (\text{mid!}(x \times 2) \longrightarrow P))$$
$$Q = (\text{mid?y} \longrightarrow (\text{right!}(y+1) \longrightarrow Q))$$

Figure 5.12: The Connection Diagram for (P || Q)

Two processes connected in this way are called a *pipe* in CSP, and are shown with a special symbol, P $\gg$ Q. Of course, a process may have more than one input and one output channel, but a channel may connect only two processes, and the communication is unidirectional. Communication along such channels is synchronous. That is, there is no buffering between the two processes: the communication event requires the participation of both processes.

Finally, in a similar way as conventional programming languages, CSP provides as part of its notational syntax statements to declare variables, assign to those variables and conditional statements branching on the value of variables. CSP also provides a wide range of operators. Operators, other than those described above, are provided in CSP for specifying special combinations of processes, e.g. when a process, P, is dedicated to serving the need's of another process, Q, the combination of these processes is shown by using the subordination operator, P // Q. Other classes of operators include those that operate on process traces and process alphabets. CSP also provides definitions for useful sets, relations and operators that can be used when studying the behaviour of processes. These can be used along with operators to form a *specification* for the behaviour of a process.

5.5.2 Concluding Remarks

CSP provides an extensive notation in a formal programming language which can be used to specify and design a wide variety of systems. The most obvious application of CSP is to the specification, design and implementation of computer systems which continuously act and interact with their environment [Hoa85]. Since CSP is a programming notation, it only provides the means of specifying systems without an accompanying method for doing so. System specification and design methodologies are usually inclined towards a particular strategy such as top down or object oriented design. The flexibility of CSP, on the other hand, allows its user to follow a method of his/her choice. CSP is very well suited to bottom up design: small processes can easily be defined by identifying system actors and their actions (alphabets), and then combined by using any of the many CSP operators to define more complex processes. It is also possible to follow a top down hierarchical design strategy using CSP: complex processes can be decomposed into a network of simpler processes by using the process combination operators.

Nondeterminism is one the underlying principles of CSP. A CSP system description may include processes whose behaviour is nondeterministic for a particular set of events. The general choice operator, for example, specifies the behaviour of a process as a choice determined by the first event. If the constituent parts of the complex process share a common first event, the choice is nondeterministic. Nonderterminism also exists in other parts of CSP specifications. Hoare [Hoa85] indicates that the choice of which execution branch is taken is left to the implementor, so that the most convenient implementation can be chosen. This implies that some decisions about the precise way a system reacts to an event may be deferred to a late stage of the design, which in turn may cause inconsistences in the specification to filter through to the final stages of system implementation.

Using CSP, the activating (firing) agent for a process can be explicitly defined. Every atomic process in our notation has an identified firing event. Such an event can play a similar role in a CSP process definition by making it the initial event of that process. The portrayal of higher level subsystem control is a little more complex. A subsystem can first be formed by grouping the processes within it. The starting conditions for the combined process and its constituent processes must then be formed to conform to the control requirements for that subsystem. In addition, it may be necessary to include extra actions in the definition of the constituent processes to specify deactivations of those processes when a particular event occurs. It may also be necessary to place restrictions on the behaviour of these processes by using CSP behaviour *specifications.*

Static data can be represented in two ways in CSP. It can either be represented as shared data or as a process. Hoare [Hoa85] discourages the use of shared variables because of the possible violation of data integrity by concurrently interacting processes. Stored data can also be represented as a process, where the data is private to a guardian process and other processes read and write to the data by using channel communication with that process. This representation model follows in the footsteps of data representation in object oriented languages and methodologies. In the case of a data item in demand, the resulting process that represents it may require several channel connections to other processes. A data flow type connection can be represented in CSP by using a single item buffer process which engages in synchronous communication with the producer and consumer in turn.

The formal nature of CSP allows the precise specification of a system with mathematical rigor. The resulting specification can be subjected to proofs to investigate many properties, such as the absence of deadlock and livelock, of the specified system. Although CSP can prove to be a very useful notation in this respect, it provides a poor communication medium between technical and non-technical people. The proofs derived using CSP cannot be easily used (in their mathematical form) to show users without a substantial mathematical background that a specification satisfies their requirements. Diagrammatic notations have proved to be more useful than algebraic ones for this purpose. CSP does provide limited pictorial representation of processes and their behaviour in state transition and connection diagrams, but these representations are only used as an aid to understanding; they are not intended to be used for practical transformations and manipulation of large-scale processes [Hoa78].

The above discussion shows that a specification in our notation can be converted to one in CSP, but this may involve some fine tuning of the resulting specification to ensure that it conforms to user requirements. There are two major points to note when doing so. First, communication in CSP is synchronous, i.e. both partners take part in the communication. The communication model in our diagrams is asynchronous: data and events are delivered when ready. This implies that a CSP equivalent for a data flow is a single buffer process which (synchronously) receives a piece of data from the producer and sends it to the consumer. Second, CSP processes are permitted to communicate during their operation. This is not allowed for atomic processes. The equivalent CSP processes must hence be restricted to those which do not exchange data in the middle of their operation.

Therefore, it may be possible to carry out checks on a derived specification by converting it to CSP and then using CSP proof techniques to prove the satisfaction of system requirements. Since CSP may not be suitable for communicating the specification to a user, the diagrams may have to be used. Any resulting modifications to the specification must then be reflected in the CSP specification in order to recheck the system properties.

5.6 Conclusions

A number of criteria can be identified for comparing notations to assess their suitability and expressive power for specification of real-time systems. A broad selection of these criteria are given in the following sections and the ability of each notation to satisfy each criterion is briefly discussed. It is not claimed here that the list given below is complete, but it provides a base for broad comparison of the notations outlined in this chapter.

The comparison criteria can be subdivided into two groups. The first assesses the capability of a notation for describing certain aspects of systems, and the second examines the overall suitability of notations for describing systems. There are areas where these categories overlap.

5.6.1 System Aspects

There are a number of system aspects that are of interest when specifying the operational behaviour of a system. Each of these is discussed in turn below.

Activities: The overall operation of a system is accomplished by the collaboration of a number of simple activities. These activities are often taken to form the base elements that must be considered [GGS89]. All the notations discussed in this chapter have a representation for a simple activity. In petri nets, an activity is represented by a transition. The firing of that transition represents the execution of the activity. In MASCOT each simple task is represented by a MASCOT activity, and the scheduling of that activity represents the execution of the task it represents. Activities are modelled by processes and functions in JSD. In CSP a process can be used to represent a simple task. In the new notation, atomic processes are the base elements within a specification, but the subjective criteria used in identifying an atomic process gives analysts additional help when deriving specifications.

Data Interfaces: There are two types of data interface between system activities: transient data, and static (stored) data. Transient data is exchanged by system activities during system operation. Any system specification notation must be able to represent such exchanges of data. Data exchange between system activities can be represented by marking input/output places of transitions in petri nets, but this representation cannot easily show what data is exchanged. MASCOT provides elaborate data exchange specification through the use of IDA's, where each IDA treats each collection of data as an object and provides access to it via procedures. JSD uses data streams for this purpose. In CSP such exchanges are represented as communication via named channels. Data flows represent transient data exchanges in the new notation. The name of a data flow identifies the type of the data exchange, while the composition of each exchange is given in a data dictionary entry.

Unlike transient data, static data is not represented well in all notations. Since petri nets are essentially a notation for modelling the control sequences in a system, static data cannot easily be represented in a petri net graph. MASCOT IDA's are used to model stored data. Again, the data is an object and access is provided via a procedural interface. In JSD, the only way of representing stored data is by using internal variables of processes, i.e. in state vectors. The decisions about what part of system data is to be stored is left to the later stages of JSD. As discussed in the section on CSP, static data can be represented by a process. This representation follows the principles of object oriented programming by providing access to data only through the process that owns the data. Stored data is represented explicitly by data store symbols on the Data Flow Diagrams of the new notation.

Control Interfaces: The control interfaces of a system fall into two categories: control of individual activities and control of a group of activities. The firing of a transition in petri nets represents the execution of an activity. Control of activities is, hence, effected by placement of tokens in input places of transitions. In MASCOT control sequences are planned using synchronisation via IDA's. In JSD, control sequences of individual activities are covered by the later (scheduling) stages of a design. The firing of an activity is modelled in CSP by having its activating event as the first event it can engage in. The flat nature of petri nets makes them unsuitable and difficult to use in modelling control over a group of activities. This inherent flatness means that only one level of concurrency, that amongst simple activities, can be modelled in a petri net. Although MASCOT incorporates subsystems in its design hierarchy, it does not provide specification tools for showing control over a subsystem explicitly. Such control is achieved by using the primitives that operate on control queues. Since JSD does not follow a hierarchical design strategy, there is no notion of activity groups and the control of subsystems is not easily representable in JSD. Subsystem control can be represented in CSP by using process groupings. Because of the control intensive nature of real-time systems, the ability to specify both categories of system control interface is an invaluable analysis aid. The firing event for each atomic process on the Event Flow Diagrams and the enabling/disabling transitions on the Subsystem Control Diagrams of the new notation cater for specifying both the control over individual activities and the control over activity groups.

System State: Once a system has been specified, its dynamic behaviour can be analysed by examining its state at various points in its operation. It is, therefore, important to be able to derive this state information easily. Petri nets are executed by placing tokens in places, i.e. marking the net, and firing enabled transitions. The state of the system at any given point in time is described by the marking of its petri net model. Since much of the MASCOT control flow is achieved by data/event exchange via IDA's and operations on control queues, there is no explicit feature of MASCOT activities and subsystems that can be examined to derive state information. The same is almost true of JSD, but the state of an entity can be derived by looking at the values of its internal variables, i.e. its state vector. The state of the whole system is the collection of its entities' states. In CSP system state is portrayed by the trace of a system at any given point in time. The approach taken for specifying subsystem control in the new notation means that, like petri nets and unlike finite state model based notations, the dynamic state of a system is not localised within a single state. It is distributed amongst its processes. Hence system state can only be derived by collecting subsystem and atomic process states.

5.6.2 Overall Capabilities

Godwin et al [GGS89] identify a number of significant factors that can be considered when assessing the overall capabilities of a specification approach. Among these are: analysis power, communication power, and size sensitivity.

Analysis Power: When using a specification notation, one of the purposes may be to produce an understanding of the system being described. In other words, the notation must be capable of pointing out ambiguities and errors in user requirements, and provide techniques to ensure the derived system conforms to those requirements. All the notations described in this chapter claim to guide the analyst in discovering such errors. The mathematical base of petri nets and CSP makes them more suitable for rigorous analysis of specifications (as the existing literature indicates). The clarity of specifications in the new notation and its features which help in discovering requirements errors (see also Chapter 6) point to the capabilities of the new notation for providing help during analysis.

Communication Power: The description of a system presented in any notation can be used for communication between the various groups, e.g. analysts, implementors and users, involved with the system development. Diagrammatic notations have proved to be the most useful for this purpose. Because of its strong mathematical base CSP is least useful of the above notations for communication between technical and non-technical groups. The flat natures of petri net graphs and JSD SSD's result in large diagrams that are difficult to digest at once. This may become a major factor in presenting information about large systems when one of these techniques is used. MASCOT uses both a hierarchical and a diagrammatic approach, so it is easier to use it as a communication vehicle between groups of people than the other notations, but the lack of diagrammatic control flow representation reduces the extent of the detail MASCOT network diagrams can portray. The clear representation of all system aspects on the diagrams of the new notation greatly enhance its communication power.

Size Sensitivity: A notation should be able to cope with the specification of systems of varying size. The size of the system becomes a very significant factor when designing large systems. If a notation provides a systematic approach to the division of the problem, it has a better chance of providing a satisfactory result when deriving a system specification. A large problem cannot be tackled at once, which implies that an incremental approach is required for such systems. Hierarchical design has proved most effective in breaking a problem down. The only notation that advocates a hierarchical design among the above notations is MASCOT (CSP does not encourage any strategy for deriving a specification, even though it can be done hierarchically). Specifications in petri nets and JSD are flat, and as pointed out in the earlier sections above, may cause difficulties when deriving a large specification. The new notation follows in the footsteps of its predecessors by adopting a hierarchical presentation strategy. It is, therefore, not as sensitive to the system size as those with a flat approach to specification.

5.6.3 Concluding Remark

In order to specify a real-time system, the flow of data and control must both be completely described in the specification. The aim of the notation we have introduced is to show clearly all stored data, all data interfaces to processes, the division into subsystems which may be enabled and disabled separately, the conditions for enablement and disablement of each subsystem, all processes down to the level of atomic processes, and the order in which processes are required to fire. It can be deduced from the above discussion that each of the above notations covers some of these aspects, but none of them covers them all.

Chapter 6

Conclusions

6.1 Overview

Specification has been identified as one of the most important stages of the system life cycle. It is the stepping stone into the process of system design and implementation. Hence, any errors propagated from a specification to the later stages of a system's life cycle will have disastrous results, ranging from minor alterations to modules and subsystems to complete redesigns of major parts of that system. It is, therefore, essential for analysts to have a toolkit of notational aids which enables them to present complete and unambiguous specifications which ideally have no errors.

This thesis has attempted to provide a notation for the specification of real-time systems, which aims to achieve these goals. The introduction chapter identified the characteristics of such systems and what the outcome of the specification should be. It also outlined the need for a notation that: is easy to use when deriving such specifications, can be used effectively as a communication medium between the various groups of people involved in system development, and can guide its users in deriving complete and unambiguous specifications.

Data Flow Diagrams are identified as an effective tool for the specification of data processing systems in the background chapter, which also outlined why they are unsuitable for the specification of real-time systems. Two extended DFD notations, designed for use in real-time system specification, were then presented in order to establish some of their shortcomings. The next chapter introduces the use of a new notation through a worked example, outlining the various symbols of the notation and their use in constructing the diagrams that describe the operational behaviour of a system. The following chapter discusses many of the issues relevant to the new notation. Finally, the chapter on related work drew some comparisons with other specification techniques through discussing four particular notations.

This chapter begins by discussing some of the more novel features of the new notation. Possible extensions to the notation and future directions for research are then discussed. The chapter concludes by drawing an overall conclusion from the research work described in this thesis.

6.2 Features Of The Notation

Data Flow Diagram based notations aimed at the specification of real-time systems, in use in industry at present, are deficient and imprecise as well as clumsy in various respects. Taking two of the best known of these notations as a starting point, a new notation has been devised which is both more precise and more comfortable to use. We believe the notation presented here has a number of advantages over its predecessors. The more important of these are discussed below.

6.2.1 Clarity

The new notation results in clearer specifications for a number of reasons. The first of these is the way control over process groups (subsystems) is specified. The approach taken here to describe the operational characteristics of subsystems is radically different from that taken by similar notations. The most popular notations, currently used in industry, use finite state modelling for specifying system behaviour in response to events. This inherently sequential model inhibits clearly showing the concurrency among parts (subsystems) of a system. This is a direct result of *centralising* the (sub)system state in a single state of the finite state model. In any such state many system parts may be responding to system inputs. Identifying these parts usually involves not only studying the FSM model, but it can also include looking at additional documentation such as activation tables. In short, when each part of the system is active, and which parts may be operating concurrently, cannot easily be deduced from a finite state model based notations.

Subsystem Control Diagrams, on the other hand, show both of these features *clearly*. Abandoning the centralised state feature of the finite state model, an SCD specifies the enabling and disabling conditions for each individual subsystem. Subsystems are no longer lumped together into particular groupings, each of which is enabled/disabled upon the occurrence of an event. Furthermore, this separation promotes a clear understanding of which subsystems may be acting concurrently. This is immediately apparent from an SCD by looking at the overall effect of each event appearing on the diagram. Nested subsystems provide the means of specifying more elaborate control requirements.

The concept of an atomic process (a leaf process of a diagram hierarchy) has been defined clearly by giving a number of objective guidelines for identifying such a process. These guidelines identify an atomic processes as a function whose operation starts with a single event, and whose outputs are not produced until the termination of its task. Objective identification of atomic processes should help analysts in deriving clearer specifications.

Moreover, the activating agent for every atomic process is uniquely identified in a specification presented in this notation. This is abstracted by the single event flow entering each atomic process. Predecessors of this notation do not have a specific indication of how each of the processes at the leaves of the system process hierarchy is activated to carry out its task. This information may be hidden away in either finite state models or process specifications, whereas each atomic process on an Event Flow Diagram has one and only one input event flow, which identifies its firing agent *clearly.*

The clear presentation of high level concurrency in SCD's has been emphasised as one of the features of the new notation. Low level concurrency among atomic level process is also shown clearly. This is a direct result of showing the firing agents for all atomic processes on their corresponding EFD. Local concurrency, such as that among a number of atomic processes started by the same event, can be identified readily by inspecting the appropriate EFD.

6.2.2 Ambiguity And Incompleteness

The process of deriving a specification from the users' requirements for a system is not a mechanical one. The requirements document is often incomplete and may include ambiguities [TI77]. It may be incomplete in the sense that the system's required behaviour under some circumstances may have been omitted from the requirements document, whereas ambiguity is the result of an opposite error: more than one behaviour may have been indicated under the same conditions or the required behaviour may not be precisely described. The derivation of a system's specification may, therefore, involve many interactions between its users and its analysts.

Before such interactions can take place, the analysts must first identify the ambiguities and incompleteness problems contained in the requirements document. A specification notation should not only guide the analysts in deriving complete and unambiguous specifications, it should also help them in discovering ambiguity and incompleteness problems contained in requirements documents.

The combination of the Event Flow Diagram and the Subsystem Control Diagram specifies completely and unambiguously the control structure of a system. Such control falls into two categories: high and low level control. High level control over groups of processes is specified using the SCD. As pointed out in the discussion above, the latter identifies these groups clearly and how each one is enabled and disabled by (internal or external) events. Low level control, i.e. control over individual atomic processes, is specified on the EFD by the single input event flow restriction of each atomic process. Again, the controlling conditions are shown clearly.

In addition, application of the firing rule may prompt the analysts to discover ambiguities and incompleteness problems in system requirements. Once a process has been identified as a leaf process in the system process hierarchy, the analysts must identify its firing event. This may be provided by one of several sources: an implicit event carried by (external or internal) data, an environmental event, an event generated by a system process, or an event resulting from the synchronisation of a number of (internal or external) events.

If none of the data flows input to an atomic process can be used to fire that process, i.e. they are all latched or stored data inputs, and it cannot be fired by any of the events present in the process's environment, the analysts will have to consult the users of the system to determine when that part of the system operates. For example, the analysts may discover that the process may need a temporal, i.e. time generated, event. In this way incompleteness can be discovered and resolved by the analysts.

Ambiguity can also be discovered as a result of applying the firing rule. If an atomic process can be fired by a number of events, the combining relationship between these events must be indicated on the EFD. Each event may be sufficient to fire the process, in which case a merged event flow is used to form the input event of the process. Conversely, the occurrence of all those events may be required before the process can start, e.g. a number of data pieces must arrive for the process to operate on them. In such cases the events are synchronised to form the process's input event. Other combinations are possible, and can be specified using the event flow constructs. Examples of these are shown in the specification examples of Appendix A. An ambiguity query may result if the relationship between the events that form an atomic process's firing event is not clear from the requirements document. The users of the system will have to be consulted again to resolve such an ambiguity.

The SCD can provide help in discovering ambiguity and incompleteness in a similar fashion. When high level control is associated with a hierarchy level, i.e. that level has an SCD as well as the DFD/EFD pair, enabling and disabling events must be identified for each subsystem. A subsystem may be enabled/disabled by one or more events, or it may be left out of the SCD indicating that it is enabled and disabled with its parent(s). In identifying the enabling and disabling conditions for each subsystem, incompleteness may show up as unspecified subsystem behaviour under some conditions and ambiguity may be the result of unclear indication of subsystem behaviour under certain circumstances.

The above discussion indicates that the Event Flow Diagram and the Subsystem Control Diagram can effectively help analysts in discovering incompleteness and ambiguity in user requirements both at the coarse grain subsystem level and the finer grain atomic process level.

6.2.3 Ease Of Specifying Concurrency

The discussion above points out that both high and low level concurrency can be shown clearly on Subsystem Control and Event Flow Diagrams. A further feature of the notation worth noting is the *ease* of specifying both types of concurrency.

Due to the restriction of having only one state occupied at any time, a finite state model is inherently sequential. Any concurrency present amongst subsystems can only be represented by multiple subsystem enablement in any state. Consider a situation in which a number of subsystems are enabled/disabled by a number of unique events. If these events can occur in an arbitrary order, the number of states in a finite state model increases at an alarming rate. This proliferation of states requires careful consideration by the analysts, when writing down the specification, in order to ensure that the correct number of subsystems are enabled/disabled in each state. In a case where the number of subsystems and events is more than a handful, the control specification can become unmanageable at best and unreadable at worst.

Furthermore, in a finite state model, the operation of a finite state machine can become more complex as it may also be responsible for triggering atomic processes. This further complicates the control operation of that machine, increasing the difficulties in deriving the specification, and hence increasing the possibility of erroneous specification. The separate control issue of firing atomic processes is not included in the coarse grain control structure given on the SCD: it is considered separately and presented on the EFD. As pointed out above, local concurrency is represented clearly and easily by the firing sequences of atomic processes.

Discarding the finite state model for specifying control at higher levels eliminates the sequentiality such models can impose on the analysts. In deriving an SCD, each subsystem is considered in isolation, and its enablement/disablement requirements in response to events are specified. This separation greatly aids the analysts in writing down specifications for subsystem levels. Moreover, the finer grain control in atomic process firings is not mixed in with the more coarse grain control of subsystems. This provides further help in deriving concurrent system behaviour. In fact, the ease of specifying such concurrency may encourage the analysts to look for and include concurrency in the system specification.

6.2.4 Other Features

Combinatorial explosion of states in a finite state machine can also occur in other situations. Consider, for example, a situation in which a number of events must be synchronised to form the firing event for an atomic process. If those events can occur in any order, a proliferation of states, similar to that outlined above, can take place. This proliferation can be controlled to a certain extent, if the analysts serialise the events, i.e. if they consider only a subset of the orderings in which the events can occur. The first case can result in yet another case of an unmanageable specification, yet the second enforces requirements on system behaviour which are not part of those outlined by system users.

The synchronisation symbol of the EFD enforces no particular ordering on the events synchronised. It synchronises those events to generate a compound event. In other words, no assumptions are made about what order the events may occur in and how their synchronisation is achieved. The mechanics of synchronisation depend largely on the host environment for a system. This implies that they are design issues, and are hence left to the later stages of system development.

A further feature of the new notation is its objective criteria for selecting atomic processes. Chapter 4 discusses these criteria and the advantages of applying them when deriving a specification. The criteria can further aid in deriving and presenting clear and unambiguous specifications.

6.3 Future Directions

The research work described in this thesis may be enhanced by further research to extend that work. Some of the major enhancements possible in future research are outlined in the sections below.

6.3.1 Specification→Design→Implementation

Once a specification has been derived for a system, it is passed on to the next stages of the system life cycle: *design* and *implementation*. The first of these tightens up some of the relaxations about a perfect system operating environment and adds enhancements such as the processing required for the system to communicate with its environment. Unlike the specification stage, the full implications of the host environment such as its limitations and the variety of errors that may occur during operation for the system are taken into account during design. The outcome of this stage paves the way for a full implementation of the system in a specific host environment.

Chapter 4 briefly touched on the subject of deriving the design and implementation of a system from its specification by discussing the experiences gained through implementing one of the example specifications. Further research is required in this area to derive some general guidelines, more concrete than those given in Chapter 4, which can be used when deriving a design and implementation from a specification given in the new notation.

The two direct predecessors of the notation both advocate using their notations to carry a specification into the later stages. (Specific notations, e.g. Hatley and Pirbhai's Architectural Model [HP88], are devised for this purpose. These use many of the symbols used in the preceding specification notation). The design of a system can, however, be affected radically by the choice of its host environment. As pointed out by Kalinsky and Ready [KR89], a system specification may require major reorganisation of system functions as a result of the host environment choice. This is a direct result of the different concerns during specification and design stages. While the analysts' task is to produce a complete and unambiguous description of the system's operational behaviour, the designers are concerned with fitting those requirements into the rigid and restricted host environment selected for the system. Therefore, the process of deriving a design from a specification is not just padding the latter with more processes. Because of real-world constraints, it can involve reorganisation and reconstruction of some system parts. Hence, a specification notation may not be sufficient or even appropriate for this task. Although the system processing division derived in the specification can be left unaltered in the implementation, further research is required to investigate the possible alterations and enhancements to a specification when deriving a design for implementation in a specific host environment.

6.3.2 Quality Of Specifications

Applying a programming language to a problem can yield several solutions. All such solutions may satisfy the problem's specification, i.e. all the programs may perform in a similar manner when they are executing. In order to judge one solution against another, some criteria must be provided. One method of comparing similar solutions to a problem is by assessing the quality of each solution. Such qualitative criteria already exist for programs. A program can be checked for *structuredness* by using structuring theorems [BJ66, BS72, Coo67]. Using these criteria, programs can be qualitatively categorised.

A specification notation can, in a fashion similar to a programming language, result in different specifications for the same system. For a number of reasons, e.g. user concern, analyst performance measurement, or as part of project management, it is desirable to be able to judge the quality of a specification, so that it can be assessed. Qualitative criteria are, therefore, required for specifications presented in the new notation. Chapter 4 briefly described one attempt to attach structuring criteria, similar to those used for programs, to the control parts of a specification. The problems with such an approach were also outlined.

It is not possible to evaluate the effectiveness of the new notation without any measurement for the quality of derived specifications. As part of future enhancements to the approach taken here for real-time system specification, quality criteria should be derived, so that specifications can be assessed.

6.3.3 Automated Tools

The activities during the specification stage of a system's life cycle can prompt numerous alterations to its specification document. Modifying parts of a graphical notation can prove more cumbersome than making changes to a textual description. Perhaps this has been one of the major obstacles to persuading systems analysts to use such notations. The decreasing cost of computer hardware, specially for graphic workstations, and their increasing processor power has lead to a large number of Computer Aided Software Engineering (CASE) tools, which support the various approaches to system design [Was87]. As well as providing editing facilities for the diagrams of a notation, a CASE tool can provide many invaluable aids, such as syntax checks, to analysts [TCL89].

A CASE tool based on the new notation can not only alleviate many of the mechanical tasks, such as interactive syntax checking of diagrams and deriving event flows for implicit events carried by data, it can also include facilities for analysing those diagrams. From an analysis point of view, it is useful to be able to examine separately different aspects of parts of the system. A CASE tool can provide the ideal means of doing so. DFD's, EFD's and SCD's can be viewed separately as well as together; an EFD can be superimposed on its corresponding DFD to show all aspects of the processes on the same diagram; regrouping of processes can be performed automatically to study different system configurations. Many more useful capabilities can be given to a such a CASE tool. Perhaps the most useful of these is the ability to observe a token like execution, similar to that of the transformation schema [War86, WM86], and based on the petri net equivalent (described in the Chapter 4) of a specification, in interactive or batch type modes. By providing stubs for atomic processes, for example, an animation [KN88] of a system's control structure may be studied. Such an animation will only be concerned with some behavioural aspects of the system, providing much more rapid prototyping than that provided by executable specifications [Zav82].

Despite their relatively recent invention, CASE tools have already proved their usefulness as an analysis/design tool in an industrial environment. The currently expanding market in CASE products bears evidence to this fact. Hence, a CASE tool based on the new notation will, no doubt, prove an invaluable aid to future users of the notation.

6.4 Concluding Remark

The importance of a comprehensive study of system requirements and their presentation in a form understandable by the parties involved with the system development process, before commencing a full system implementation, is being increasingly realised by system users and analysts. The increasing use of CASE tools and notations in real-time system development projects is a direct consequence of this realisation. The currently active research efforts investigating methodologies and their associated notations aimed at the various stages of the system life cycle point to the fact that such notations are still in their infancy. As they become more mature, their acceptance in industry will become more widespread. It is hoped that the work presented in this thesis makes a step towards achieving this result.

References

[Age79] T. Agerwala. Putting Petri Nets to Work. *Computer*, 12(12):85–94, Dec. 1979.

[AL81] T. Anderson and T.A. Lee. *Fault Tolerance: Principles and Practice.* Prentice-Hall International, 1981.

[Alf77] M.W. Alford. A Requirements Methodology for Real-Time Processing Requirements. *IEEE transactions on Software Engineering*, SE-3(1):60–9, 1977.

[Alf85] M. Alford. SREM at The Age of Eight: The Distributed Design System. *Computer*, 18(4):36–46, April 1985. SREM SREP R-Net RSL REVS SYSREM DDL Requirements-Specification.

[All81] S.T. Allworth. *Introduction to Real-Time Software Design.* McMillan, 1981.

[AZ87] S.T. Allworth and R.N. Zobel. *Introduction to Real-Time Software Design, 2nd Edition.* MacMillan, London, 1987.

[Bae73] J.L Baer. Modelling for Parallel Computation: A Case Study. In *Proceedings of The Sigamore Computer Conference on Parallel Processing*, pages 13–22, Syracuse University, August 1973.

[Bai89] S.C. Bailin. An Object-Oriented Requirements Specification Method. *Communications of The ACM*, 32(5):608–623, May 1989.

[Bat87] G. Bate. *The Official Handbook of MASCOT.* Defence research Information Centre, Glasgow, UK, June 1987.

[BEF+86] J.P. Booth, L.R.B. Elton, A.C.W. Finkelstein, S.M.D. Glenister, S.J. Goldsack, D. Jordan, R.D. Tavendale, and W.J. Quirk. Development of a Strategy for Technology Transfer In Relation to The FOREST Project. Alvey Initiative FOREST Report R11, Department of Computing, Imperial College, UK, 1986.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of The ACM*, 31(3):561–599, July 1984.

[BJ66] C. Boehm and G. Jacopini. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. *Communications of The ACM*, 9(5):366–371, May 1966.

[BJKW88] W. Bruyn, R. Jensen, D. Keskar, and P. Ward. ESML: An Extended System Modelling Language Based on The Data Flow Diagram. *ACM SIGSOFT Software Engineering Notes*, 13(1):58–67, Jan. 1988.

[Bjo87] D. Bjorner. On The Use of Formal Methods in Software Development. In *Proceedings of The 9th International Conference on Software Engineering*, pages 17–29, Monterey, Cal., March 1987. IEEE Computer Society Press.

[BM85] G. Bruno and M. Marchetto. Rapid Prototyping of Control Systems Using High Level Petri Nets. In *Proceedings of The 8th International Software Engineering Conference*, pages 230–235, London, Sept. 1985. IEEE.

[BO85] N.D. Birrell and M.A. Ould. *A Practical Handbook for Software Development*. Cambridge University Press, 1985.

[Boo86] G. Booch. Object-Oriented Development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221, Feb. 1986.

[BOT85] D. Bolton, P. Osmon, and P. Thompson. A Data Flow Methodology for System Development. In *Proceedings of The Third International Workshop on Software Specifications and Design*, pages 22–24, London, 26-27 August 1985. IEEE Computer Society Press.

[BS72] J. Bruno and K. Steiglitz. The Expression of Algorithms by Charts. *Journal of The ACM*, 19(3):517–525, July 1972.

[BZ83] D. Brand and P. Zafiropulo. On Communicating Finite State Machines. *Journal of The Association for Computing Machinery*, 30(2):323–42, April 1983.

[Cam82] J.R. Cameron. Two Pairs of Examples in The Jackson Approach to System Development. In *Proceedings of The 15th Hawaii International Conference on System Sciences*, Jan. 1982.

[Cam83] J.R. Cameron. *JSD and JSP: The Jackson Approach to Software Development.* IEEE Computer Society Press, Los Angeles, 1983.

[Cam86] J.R. Cameron. An Overview of JSD. *IEEE Transactions on Software Engineering*, SE-12(2):222–240, Feb. 1986.

[CCI84] CCITT. Recommendations Z100-104: Functional Specification and Description Language (ADL), 1984.

[CCW89] J.R. Cameron, A. Campbell, and P.T. Ward. Comparative Methods Work and The Future of CASE. In *Proceedings of The CASE Workshop at Imperial College*, July 1989.

[CDK85] M. Chandrasekharan, B. Dasarathy, and Z. Kishimoto. Requirements-Based Testing of Real-Time System: Modeling for Testability. *Computer*, 18(4):71–80, April 1985.

[CFG+85] R.J. Cunningham, A. Finkelstein, S. Goldsack, T. Maibaum, and C. Potts. Formal Requirements Specification-The FOREST Project. In *Proceedings of The Third International Workshop on Software Specification and Design*, London, UK, August 26-27, 1985. IEEE Computer Society Press.

[Cha89] S.K. Chang. *Principles of Pictorial Information Systems Design.* Prentice-Hall International, 1989.

[Col84] M.A. Colter. A Comparative Examination of System Analysis Techniques. *Management Information Systems*, 8(1):51–66, March 1984.

[Coo67] D.C. Cooper. Boehm and Jacopini's Reduction of Flow Charts. *Letter to the Editor, Communications of The ACM*, 10(8):463 and 473, Aug. 1967.

[CR83] J.F. Coolahan and N. Roussopoulos. Timing Requirements for Time Driven Systems Using Augmented Petri Nets. *IEEE Transactions on Software Engineering*, SE-9:603–616, Sept. 1983.

[DeM78] T. DeMarco. *Structured Analysis and System Specification.* Yourdon Press, New Jersey, 1978.

[Den77] E. Denert. Specification and Design of Dialogue Systems with State Diagrams. In *Proceedings of The International Computing Symposium*, pages 417–24, Munich, Germany, April 1977. North-Holland.

[DG82] V.A. Downes and S.J. Goldsack. *Programming Embedded Systems with ADA*. Prentice-Hall International, 1982.

[Dib82] R. Dibble. Software Design and Development Using MASCOT. In *AGARD Conference Proceedings: Software for Avionics*, number 330, pages 19/1–15, The Hague-Kijkduin, Netherlands, 6-10 Sept. 1982.

[DR79] A.M. Davis and T.G. Rauscher. Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications. In *Proceedings of The Conference on Specifications of Reliable Software*, pages 15–35, 1979.

[DS84] T. DeMarco and A. Soceneantu. SYNCRO: A Dataflow Command Shell for The Lilith/Modula Computer. In *Proceedings of The Seventh International Conference on Software Engineering*, pages 207–213, 1984.

[DT86] T. Docker and G. Tate. Executable Data Flow Diagrams. In P.J. Brown and D.J. Barnes, editors, *Proceedings of The BCS-IEE Software Engineering Conference*, Southampton, England, Sept. 1986.

[EFRV86] G. Estrin, R.S. Fenchel, R.R. Razouk, and M.K. Vernon. SARA: Modeling, Analysis and Simulation for Design of Concurrent Systems. *IEEE Transactions on Software Engineering*, SE-12(2):293–311, Feb. 1986.

[Est78] G. Estrin. A Methodology for The Design of Digital Systems - Supported by SARA at The age of one. *Proceedings of The National Computer Conference*, pages 313–24, 1978.

[FLL86] D.J. Flynn, P.J. Layzell, and P. Loucopoulos. Assisting The Analyst-The Aims and Approaches of The Analyst Assist Project. In D. Barnes P. Brown, editor, *Software Engineering 86*, pages 19–26. Peter Peregrinus, 1986.

[Fou84] R. Foulkes. A User's Experience with MASCOT. In *IEE Colloquium Digest No. 113*, 14^{th} Dec. 1984.

[FP86] A. Finkelstein and C. Potts. Structured common sense: The elicitation and formalization of system requirements. In D. Barnes P. Brown, editor, *Software Engineering 86*, pages 236–250. Peter Peregrinus, 1986.

[Fra85] B. Fraley. Design of Real-Time systems. In *Proceedings of The Third International Workshop on Software Specifications and Design*, pages 57–59, London, 26-27 August 1985. IEEE Computer Society Press.

[GGS89] A.N. Godwin, M.B. Gore, and D.W. Salt. A Comparison of JSD and DFD as Descriptive Tools. *The Computer Journal*, 32(3):202–211, June 1989.

[GH81] H. J. Genrich and K. Hautenbach. System Modeling with High-Level Petri Nets. *Theoretical Computer Science*, 13:109–136, 1981.

[Gil62] A. Gill. *Introduction to The Theory of Finite State Machines.* McGraw-Hill, New York, 1962.

[Gom84] H. Gomaa. A Software Design Method for Real-Time Systems. *Communications of The ACM*, 27(9):938–49, Sept. 1984.

[GS79] C. Gane and T. Sarson. *Structured Systems Analysis: Tools and Techniques.* Prentice-Hall International, 1979.

[Har80] D. Harel. On Folk Theorems. *Communications of The ACM*, 23(7):379–389, July 1980.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Programming*, 8:231–274, 1987.

[Har88] D. Harel. On Visual Formalisms. *Communications of ACM*, 31(4), April 1988.

[HLN$^+$88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtui-Trauing. STATEMATE: A Working Environment for The Development of Complex Reactive Systems. In *Proceedings of The 10th International Conference on Software Engineering*, April 1988.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of The ACM*, 21(8):666–677, Aug. 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, London, 1985.

[Hol87] M. Holcombe. Formal Methods in The Specification of The Human-Machine Interface. *International CIS Journal*, pages 24–34, July 1987.

[Hol88] M. Holcombe. X-Machines as a Basis for Dynamic System Specification. *Software Engineering Journal*, 3(2):69–76, March 1988.

[HP88] D.J. Hatley and I.A. Pirbhai. *Strategies for Real-Time System Specification.* Dorset House Publishing, New York, 1988.

[HPSS87] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On The Formal Semantics of Statecharts. In *Proceedings of The 2nd IEEE Symposium on Logic in Computer Science*, pages 54–64, 1987.

[HS87] A. Hecht and A. Simmons. The automation of Structured Analysis and Structured Design. In *Proceedings of The Sixth International Phoenix Conference on Computers and Communications*, pages 267–71, Scottsdale, AZ, 1987. IEEE Computer Society Press.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[II82] R.G. Babb II. Data-Driven Implementation of Data Flow Diagrams. In *Proceedings of The 6th Conference on Software Engineering*, pages 309–318, Tokyo, Japan, Sept. 1982.

[II85] R.G. Babb II. Programming with The HEP Large-Grain Data Flow Techniques. *Parallel MIMD Computation: HEP Supercomputer and its Application*, 1985.

[IOM+85] R.G. Babb II, K. Orr, A. Mili, S. Gearhart, and N. Martin. Proceedings of The Workshop On Models and Languages for Software Specification and Design. *Computer*, 18(3):103–8, March 1985.

[Irv84] K.W. Irvin. The MASCOT Environment. In *Application Development Tools: State of The Art Report*, pages 41–9. Pergamon Infotec Ltd., 1984.

[Jac75] M.A. Jackson. *Principles of Program Design.* Academic Press, London, 1975.

[Jac83] M.A. Jackson. *System Development.* Prentice-Hall Int., N.J., 1983.

[Jac84] K. Jackson. Introduction: Basic MASCOT Principles. In *IEE Colloquium Digest No. 113*, 14th Dec. 1984.

[Jen81] K. Jensen. Coloured Petri Nets and The Invariant-Method. *Theoretical Computer Science*, 14:317–336, 1981.

[JKM86] P. Jeremaes, S. Khosla, and T.S.E. Maibaum. A Modal (Action) Logic for Requirements Specification. In D. Barnes P. Brown, editor, *Software Engineering 86*, pages 269–294. Peter Peregrinus, 1986.

[KN88] J. Kramer and K. Ng. Animation of Requirements Specifications. *Software-Practice and Experience*, 18(8):749–774, August 1988.

[KNPW88] J. Kramer, K. Ng, C. Potts, and K. Whitehead. Tool Support for Requirements Analysis. *Software Engineering Journal*, 3(3):86–96, May 1988.

[Koo85] C.J. Koomen. From Specification Towards Implementation. *Methodologies for Computer System Design*, pages 105–121, 1985.

[KR89] D. Kalinsky and J. Ready. Distinctions Between Requirements Specification And Design of Real-Time Systems. In *Proceedings of The Second Int. Conference On Software Engineering for Real-Time Systems*, pages 26–30, The Royal Agricultural College, Cirencester, UK, 18-20 Sept. 89.

[Lev86] L.S. Levy. A Metaprogramming Method And Its Economic Justification. *IEEE Transactions on Software Engineering*, SE-12(2):272–277, Feb. 1986.

[LH87] M.D. Lubars and M.T. Harandi. Knowledge-Based Software Design Using Design Schemas. In *Proceedings of The 9th International Conference on Software Engineering*, pages 253–262, Monterey, Cal., March 1987. IEEE Computer Society Press.

[LL87] P.J. Layzell and P. Loucopoulos. *Systems Analysis and Development, 2nd Edition.* Chartwell-Bratt Studentlitteratur, Sweden, 1987.

[LM81] H. Ledgard and M. Marcotty. *The Programming Landscape.* Science Research Associates Inc., Chicago, 1981.

[LS87] J.H. Larkin and H.A. Simon. Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognative Science*, 11:65–99, 1987.

[Mai86] T.S.E. Maibaum. A Logic for Formal Requirements Specification of Real-Time Embedded Systems. Alvey Initiative FOREST Report R3, Department of Computing, Imperial College, London, UK, 1986.

[mas86] MASCOT Design Support Environment, Problem 2: Bank Autotellor Network, May 1986.

[McF82] W.S. McFadyen. A Cohesive Methodology for The Development of Large Real-Time Systems. *Journal of Telecommunications Networks(USA)*, 1(3):265–80, Fall 1982.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall International, 1988.

[Mil56] G.A. Miller. The Magical Number Seven Plus or Minus Two: Some Limits on our Capacity for Processing Information. *Psychological Review*, 63:81–97, 1956.

[Mil75] H.D. Mills. The New Math of Computer Programming. *Communications of The ACM*, 18(1):43–48, Jan. 1975.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[Mir89] E.L. Miranda. Specifying Control Transformations Through Petri Nets. *ACM SIGSOFT Software Engineering Notes*, 14(2):45–48, April 1989.

[MJAS85] T.J. McCabe, F.C. Joh Jr., K. Adams, and A.M. Sturgill. Structured Real-Time Analysis and Design. In *Proceedings of COMPSAC-85*, pages 40–52. IEEE, Oct. 1985.

[MM85] J. Martin and C. McClure. *Diagramming Techniques for Analysts and Programmers.* Prentice-Hall International, 1985.

[MP84] S.M. McMenamin and J.F. Palmer. *Essential Systems Analysis.* Yourdon Press, New Jersey, 1984.

[Mul84] G. Mullery. Requirements Overview Acquisition-Environment. *Advanced Courses On Distributed Systems-Methods and Tools for Specification*, April 1984.

[NSO90] M. Nejad-Sattary and P.E. Osmon. A Notation For Real-Time System Specification. In *Proceedings of The UKIT 1990 Conference*, University of Southampton, UK, 19-22 March 1990. An earlier version of this paper was presented at the Workshop On Real-Time Systems: Theory and Practice, University of York, Uk, 28-29 Sept. 89.

[NSO89] M. Nejad-Sattary and P.E. Osmon. On A Notation For Real-Time System Specification. In *Proceedings of The Second International Conference On Software Engineering for Real-Time Systems*, pages 31–35, The Royal Agricultural College, Cirencester, UK, 18-20 Sept. 89.

[ONR87] R.A. Orr, M.T. Norris, and C.D.V. Rouch. Complexity Control and Analysis of Real-Time Software Systems. *British Telecom Technology Journal*, 5(2):12–17, April 1987.

[OWW85] C. Olson, W. Webb, and R Wieland. Code Generation from Data Flow Diagrams. In *Proceedings of The Third International Workshop on Software Specifications and Design*, pages 172–176, London, 26-27 August 1985. IEEE Computer Society Press.

[PC86] D. Parnas and P.C. Clements. A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering*, SE-12(2):251–57, Feb. 1986.

[PCW85] D. Parnas, P.C. Clements, and D.M. Weiss. The Modular Structure of Complex Systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–66, March 1985.

[Pet77] J.L. Peterson. Petri Nets. *Computing Surveys*, 9(3):223–252, Sept. 1977.

[Pet81] J.L. Peterson. *Petri Net Theory and Modeling of Systems.* Prentice-Hall, 1981.

[PFAB86] C. Potts, A. Finkelstein, M. Aslett, and J. Booth. Structured Common Sense: A Requirement Elicitation and Formalization Method for Modal Action Logic. Alvey Initiative FOREST Report R2, Department of Computing, Imperial College, London, UK, 1986.

[PJ88] M. Page-Jones. *The Practical Guide to Structured Systems Design, 2nd Edition.* Prentice-Hall International, 1988.

[Rei82] W. Reisig. Petri Nets: An Introduction. In W. Brauer G. Rozenberg A. Salomaa, editor, *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1982.

[RJ77] D.T. Ross and K.E. Schoman Jr. Structured Analysis for Requirements Definition. *IEEE Transactions on Software Engineering*, SE-3(1):6–15, Jan. 1977.

[RPTU84] C.V. Ramamoorthy, A. Prakash, W.T. Tsai, and Y. Usuda. Software Engineering: Problems and Prospects. *Computer*, pages 191–209, Oct. 1984.

[RS82] A. Rockstrom and R. Saracco. SDL-CCITT Specification and Description Language. *IEEE Transactions on Communications*, COM-30(6):1310–1317, June 1982.

[Sal76] K.G. Salter. A Methodology for Decomposing System Requirements into Data Processing Requirements. In *Proceedings of The Second International Conference on Software Engineering*, pages 91–101, New York, 1976. IEEE Computer Society Press.

[San89] B. Sanden. An Entity-Life Modelling Approach to The Design of Concurrent Software. *Communications of The ACM*, 32(3):330–343, March 1989.

[Sim82] H.R. Simpson. MASCOT Developments to Improve Software Structure and Integrity. In *AGARD Conference Proceedings: Software for Avionics*, number 330, pages 5/1–14, The Hague-Kijkduin, Netherlands, 6-10 Sept. 1982.

[Sim84] H.R. Simpson. MASCOT 3. In *IEE Colloquium Digest No. 113*, 14^{th} Dec. 1984.

[SJ79] H.R. Simpson and K.L. Jackson. Process Synchronisation in MASCOT. *The Computer Journal*, 22(4):332–345, 1979.

[SJF88] H.R. Simpson, K. Jackson, and R. Foulkes. IEE Colloquium on Real Time Computing: The Future with MASCOT, Oct. 1988.

[SM88] S. Shlaer and S.J. Mellor. *Object-Oriented Systems Analysis: Modelling The World in Data.* Yourdon Press, 1988.

[SMC74] W.P Stevens, G.F. Myers, and L.C. Constantine. Structured Design. *IBM Systems Journal*, 13(2), 1974.

[Som89] I. Sommerville. *Software Engineering, 3rd Edition.* Addison Wesley, 1989.

[Sta88] J.A. Stankovic. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *Computer*, 21(10):10–19, Oct. 1988.

[Sut88] A. Sutcliffe. *Jackson System Development.* Prentice Hall International, London, 1988.

[Tay80] B. Taylor. A Method for Expressing The Functional Requirements of Real-Time Systems. In *Proceedings of Real-Time Programming*, pages 111–120, Liebnitz, Austria, 1980. IFAC.

[TB73] B.A. Trakhtenbrot and Y.M. Brazdin. *Finite Automata: Behaviour and Synthesis.* North-Holland, 1973.

[TCL89] K.P. Tan, T.S. Chua, and P.T. Lee. AUTO-DFD: An Intelligent Data Flow Processor. *The Computer Journal*, 32(3):194–201, June 1989.

[TI77] D. Teichroew and E.A. Hershey III. PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems. *IEEE Transactions on Software Engineering*, SE-3(1):41–48, Jan. 1977.

[TP89] T.H. Tse and L. Pong. Towards a Formal Foundation for DeMarco Data Flow Diagrams. *The Computer Journal*, 32(1):1–12, Feb. 1989.

[TRH87] W.R. Terry, H. Rao, and D.K. Handal. Computer-Aided Methodology for Development of Real-Time Control Systems for Synchronised Manufacturing. In *Proceedings of The 9th Annual Conference on Computers and Industrial Engineering*, pages 124–28, Atlanta, GA, 1987.

[Tse87a] T.H. Tse. The Identifications of Program Unstructuredness: a Formal Approach. *The Computer Journal*, 30(6):507–511, 1987.

[Tse87b] T.H. Tse. Towards a Single Criterion for Identifying Program Unstructuredness. *The Computer Journal*, 30(4):378–380, 1987.

[War86] P.T. Ward. The Transaction Schema: An Extension of The Data Flow Diagram to Represent Control and Timing. *IEEE Transactions on Software Engineering*, SE-12(2):198–210, Feb. 1986.

[War89] P.T. Ward. How to Integrate Object Orientation with Structured Analysis and Design. *IEEE Software*, pages 74–82, March 1989.

[Was80] A.I. Wasserman. Information System Design Methodology. *Journal of The American Society for Information Science*, Jan. 1980.

[Was87] A.I. Wasserman. CASE Environments: The Next Five Years. In *Proceedings of CASE '87*, Cambridge, MA, May 1987.

[WE82] J.W. Winchester and G. Estrin. Requirements Definition and its Interface to The SARA Design Methodology for Computer-Based Systems. In *Proceedings of The National Computing Conference*, pages 369–79, Arlington, Va., 1982. AFIPS Press.

[WFP83] A.I. Wasserman, P. Freeman, and M. Porcella. Characteristics of Software Development Methodologies. In T.W. Olle H.G. Sol J. Tully, editor, *Proceedings of CRIS II Conference: Information Systems Design Methodologies: Feature Analysis*, pages 37–57, York, UK, 1983. North-Holland.

[WHF82] A.T. Wood-Harper and G. Fitzgerald. A Taxonomy of Current Approaches to Systems Analysis. *The Computer Journal*, 25(1):12–16, 1982.

[Whi] S. White. Panel Problem: Software Controller for an Oil, Hot Water Home Heating System.

[Wil77] E.J. Wilkens. Finite State Techniques in Software Engineering. In *Proceedings of COMPSAC '77*, pages 691–697, Nov. 1977.

[Wil83] M.H. Williams. Flowchart Schemata and The Problem of Nomenclature. *The Computer Journal*, 26(3):270–276, 1983.

[WK87] P.T. Ward and D.A. Keskar. A Comparison of The Ward/Mellor And Boeing/Hatley Real-Time Methods. In *Proceedings of The Twelfth Structure Methods Conference*, pages 356–366, Chicago, August 1987.

[WL85] S.M. White and J.Z. Lavi. Embedded Computer System Requirements Workshop. *Computer*, 18(4):67–70, April 1985.

[WM86] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems*, volume 1, 2, & 3. Yourdon Press, New Jersey, 1986.

[WPSK86] A.I. Wasserman, P.A. Pircher, D.T. Shewmake, and M.L. Kersten. Developing Interactive Information Systems with User Software Engineering Methodology. *IEEE Transactions on Software Engineering*, SE-12(2):326–345, Feb. 1986.

[You89] E. Yourdon. *Modern Structured Analysis.* Prentice-Hall International, NJ, 1989.

[Zav82] P. Zave. An Operational Approach to Requirements Specification for Embedded Systems. *IEEE Transactions on Software Engineering*, SE-8(3):250–69, 1982.

[Zav85a] P. Zave. A Distributed Alternative to Finite State Machine Specifications. *ACM Transactions on Programming Languages and Systems*, 7(1):10–36, 1985.

[Zav85b] P. Zave. The Operational Versus The Conventional Approach to Software Development. *Communications of The ACM*, 27(2):104–118, Feb. 1985.

[ZS86] P. Zave and W. Schell. Salient Features of an Executable Specification Language and its Environment. *IEEE Transactions on Software Engineering*, SE-12(2):312–25, Feb. 1986.

Appendix A

Example Specifications

A.1 Overview

The petrol station exercise has been used for illustrating many aspects of the notation presented in this thesis. The notation was subjected to many other trial specifications. This Appendix presents the solutions to those exercises. Many of the exercises have been used in system design workshops to study the capabilities of a variety of specification approaches. Some of these studies have been used as a comparison basis for methodologies [Whi, HP88].

Each of the sections below starts by giving a textual specification of the example system's requirements. Many features of these requirements are typical of those usually produced by system users. They contain verbose descriptions, which contain duplications and implementation detail.

The diagrams of the specification for each system follows its textual requirements.

A.2 The Bottling System

This system consists of a number of bottle-filling [1] lines fed by a single vat containing a liquid to be bottled. Because of the single vat, the composition of the liquid being placed in the bottles is identical for all lines at a given time. However, the bottle size may differ from line to line. For example, at 7:30 one bottling line might be filling one-litre bottles and another might be filling five-litre bottles, but both lines would be using liquid maintained at the constant pH, say 6.52.

The tasks of the control system are to control the level and the pH of the liquid in the vat, to manage the movement and filling of bottles on vari-

[1]This exercise was adapted from the bottle-filling example in [WM86].

ous lines, and to exchange information with human operators working the individual lines and with an area supervisor monitoring the entire system.

The vat level control is accomplished by monitoring the level with a sensor and adjusting a liquid input valve accordingly. The requirement for controlling pH arrises because the liquid to be bottled reacts with its surroundings, causing the pH to "creep" over time. A constant pH is maintained by introducing, through a control valve, small quantities of a chemical that reverses the pH "creep". The addition rate of the pH-changing chemical depends both on the current pH in the vat (measured by a pH sensor) and on the rate of flow of liquid through the tank (measured by the liquid input valve control).

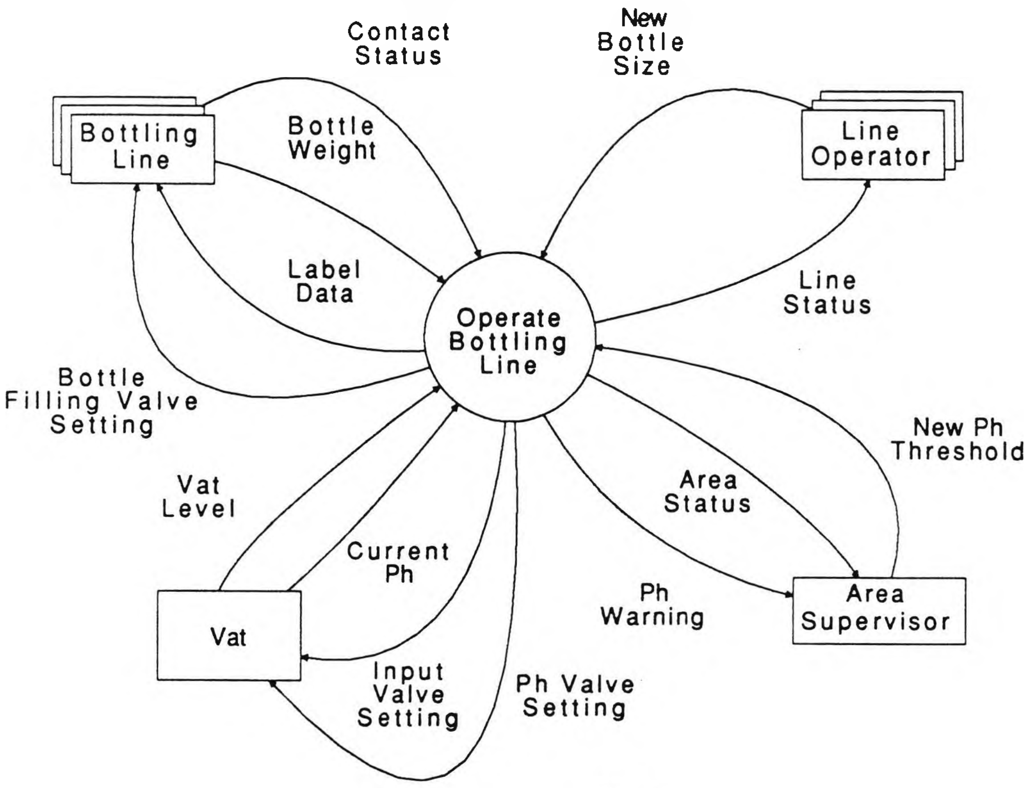
Bottles to be filled on a particular line are drawn one by one from a supply of bottles, as follows:

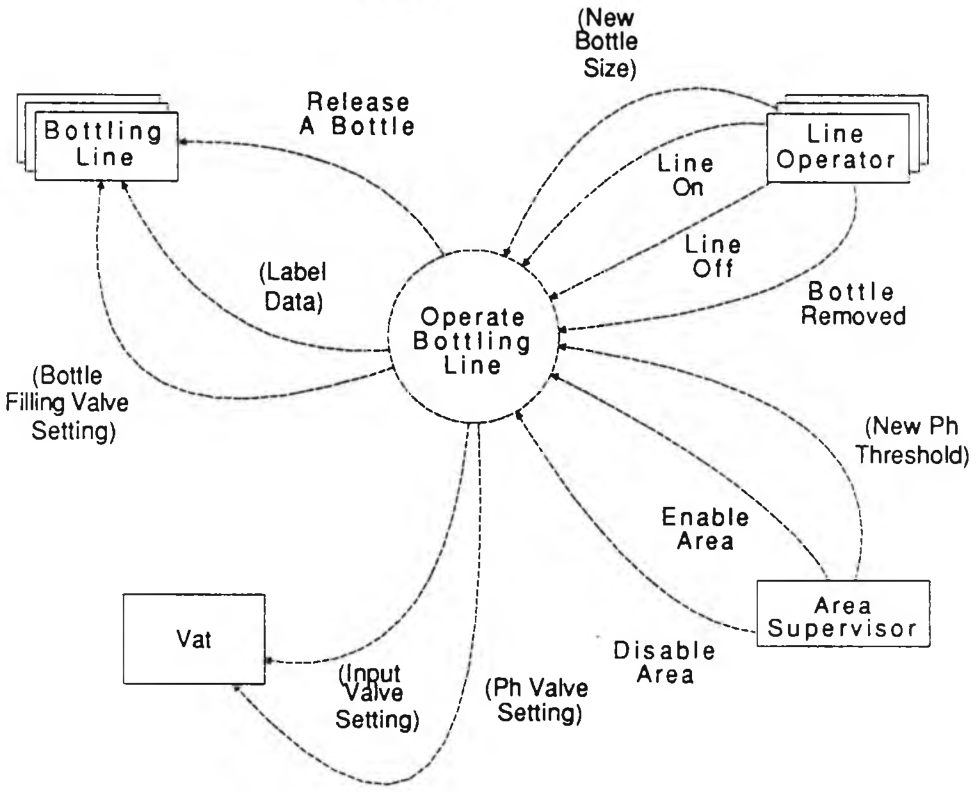
- A bottle is released from a gate and drops down onto a scale platform, at the same time depressing a bottle contact sensor.
- The bottle-filling valve is opened, and a measured amount of liquid is let into the bottle. (The scale platform measures the weight of the bottle plus it contents, and is used to determine when the bottle is full and to shut off the valve.)
- The filled bottle is labeled to show the actual pH when filled, and the nominal pH. The line operator caps and removes the filled bottle, and signals the system that the bottle has been removed. Removing the bottle releases the bottle contact sensor, removes the weight on the scale and allows the next bottle to be released from the gate.

The line operators can signal the system to start and stop individual lines, and the supervisor can signal the system to enable or disable overall operation of the set of lines. For a line to start operation from stopped status, both the area enable and the line start signal are necessary; in addition, the bottle contact must be off and the scale platform reading must be less than 0.1 gram. The line operators are given displays of the line status and are able to change bottle size for the line. The area supervisor is given a display of the current status of the system pH and vat levels and statuses of individual lines, and is able to change the pH of the bottled liquid by entering a new pH to be maintained.

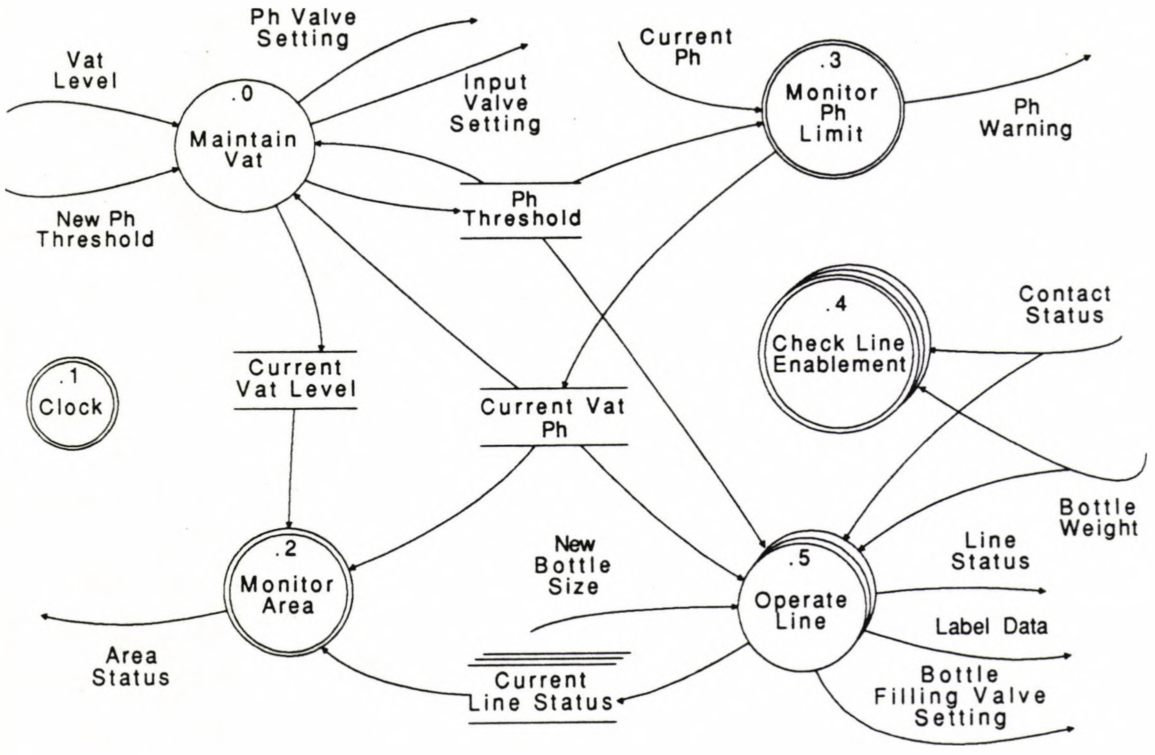
If, during operation of the system, the pH goes out of limits (>0.3 from the setpoint) all control actions are suspended. The vat pH is then stabilised manually. When the pH is back within limits, the system restarts automatically.

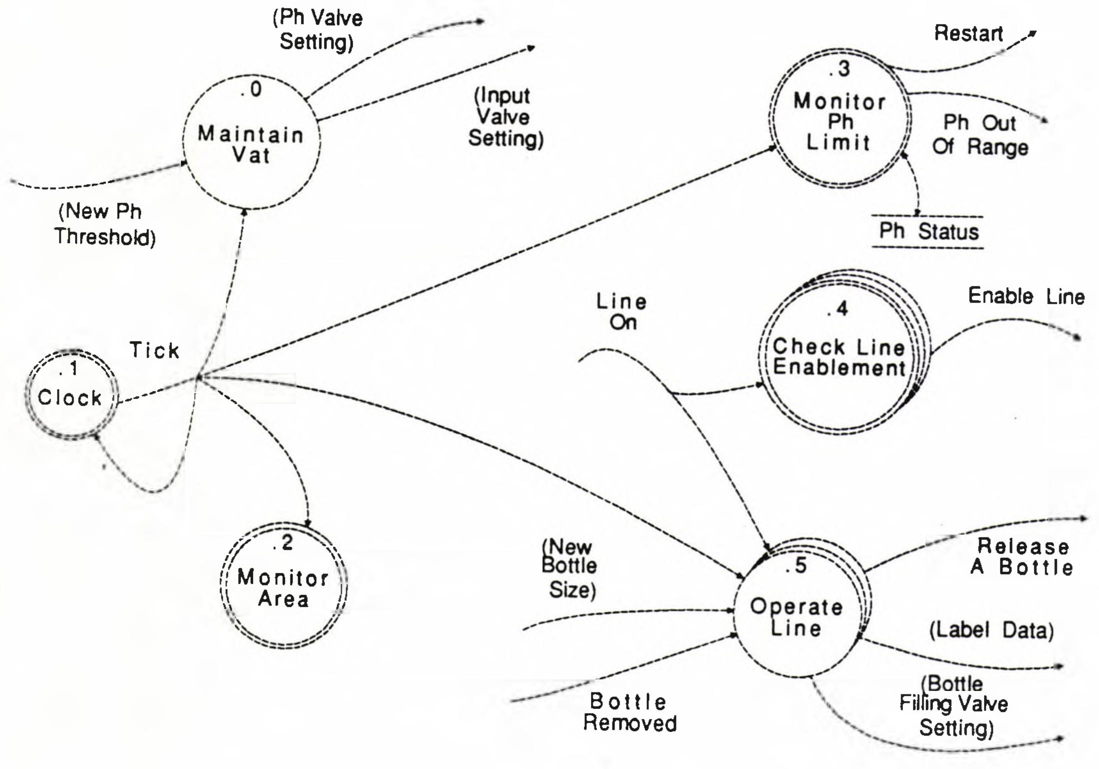
A.2.1 Specification Diagrams



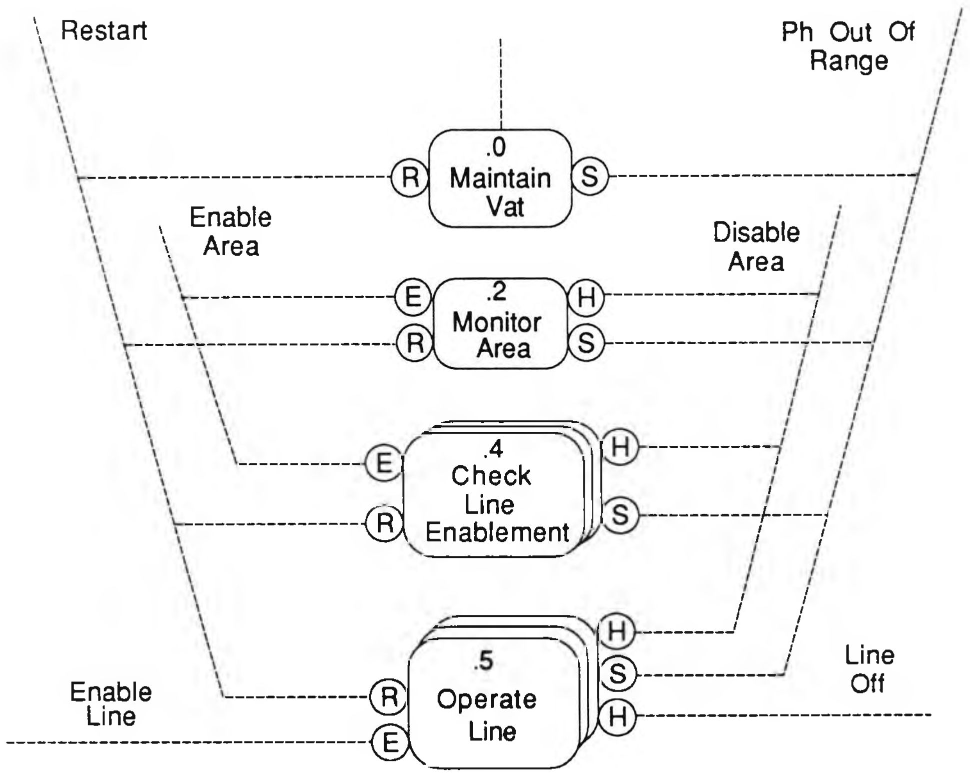


The Bottling System: Context Diagram

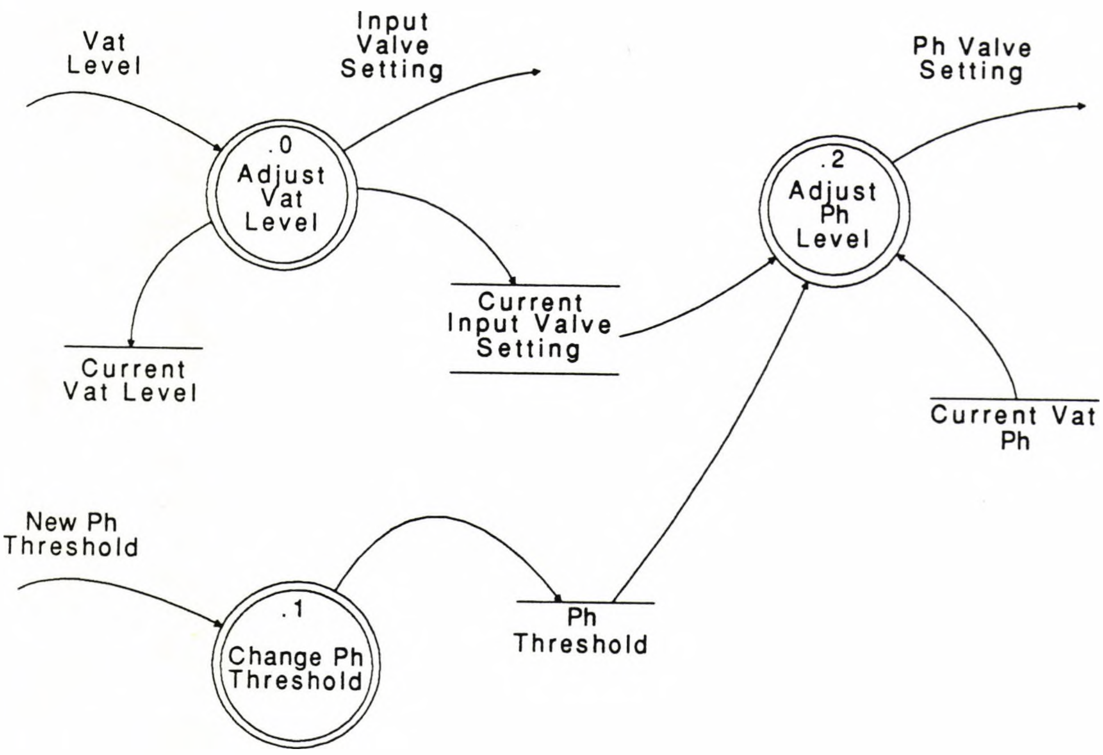


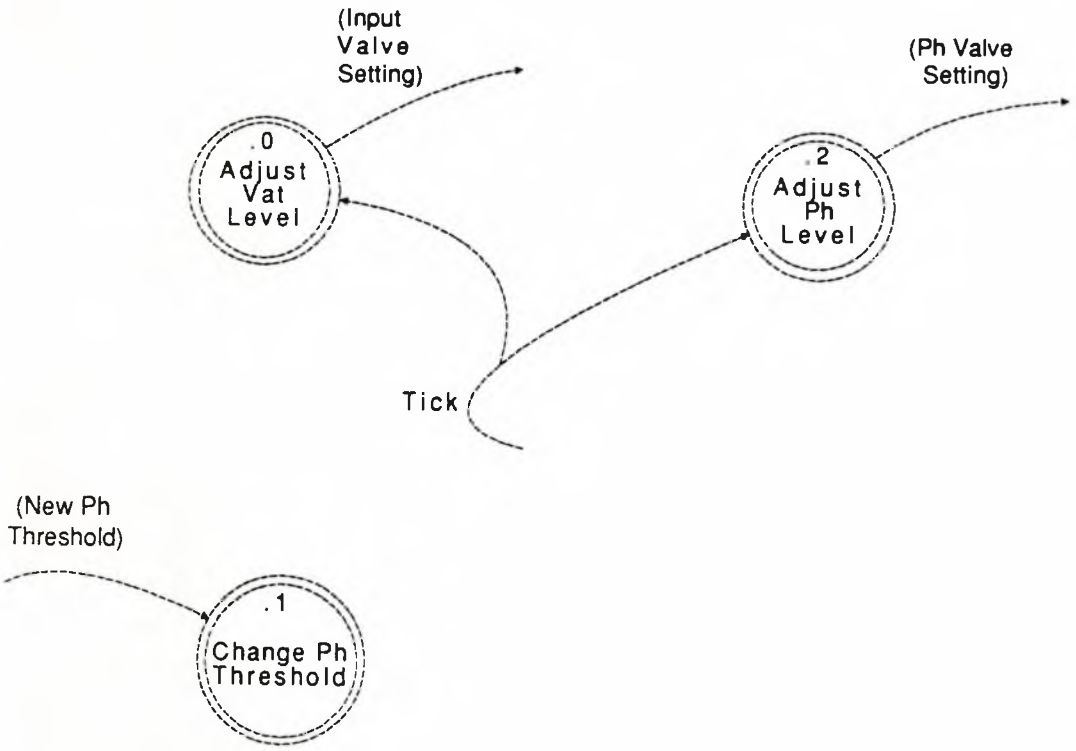


Operate Bottling Line (DFD/EFD)

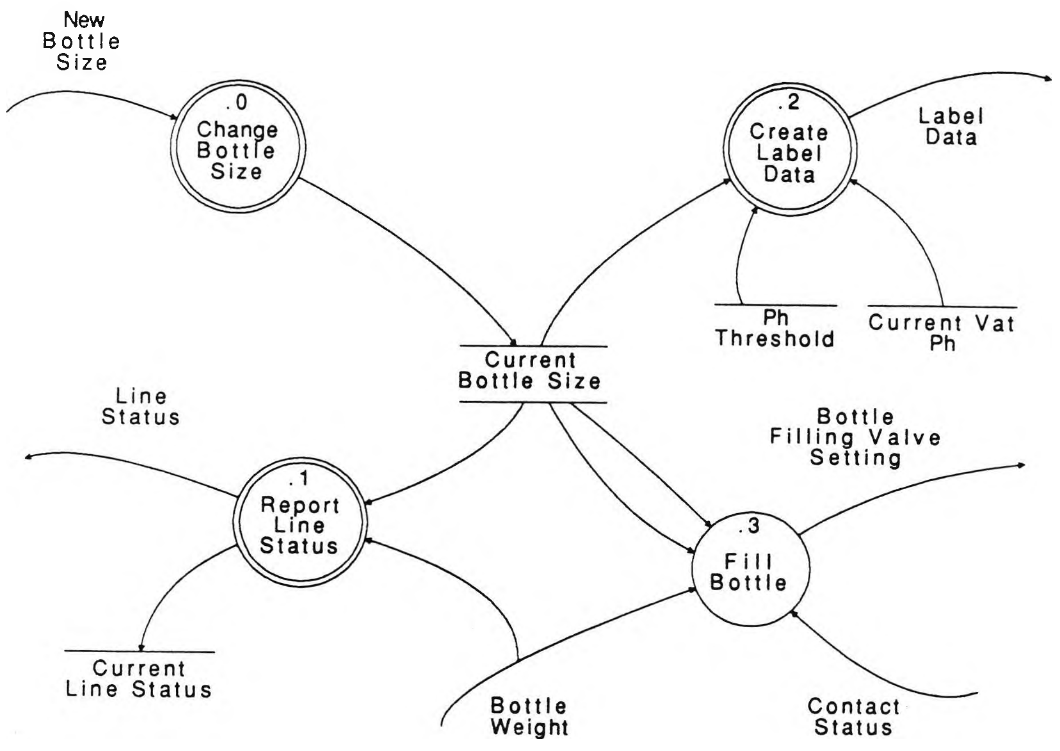


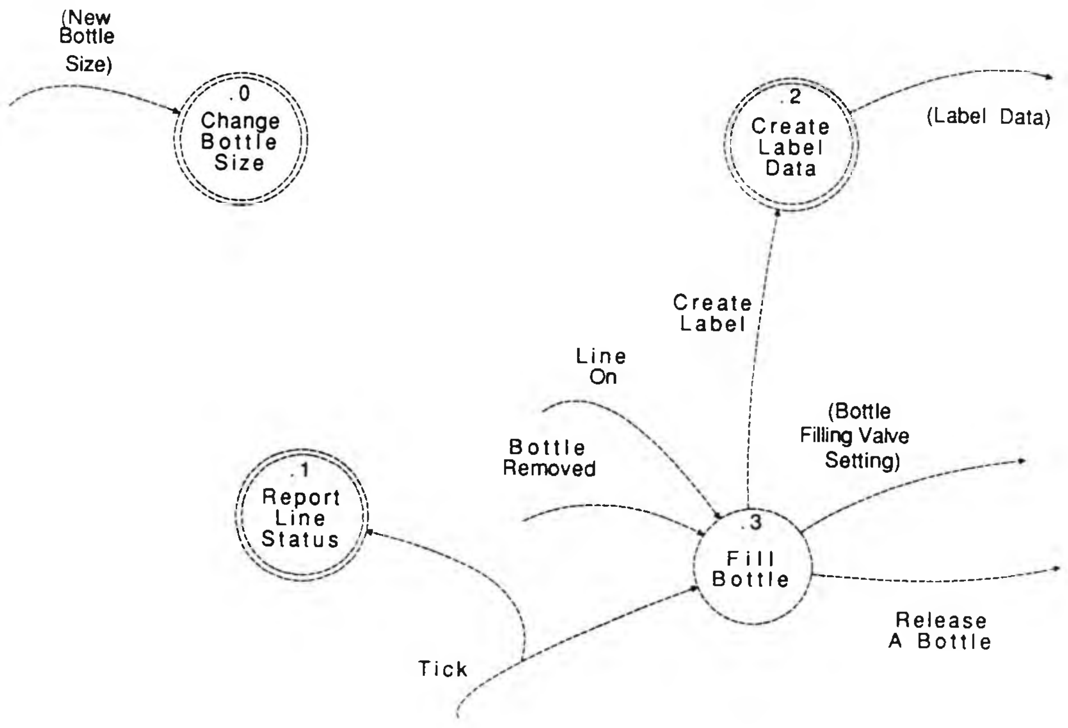
Operate Bottling Line (SCD)



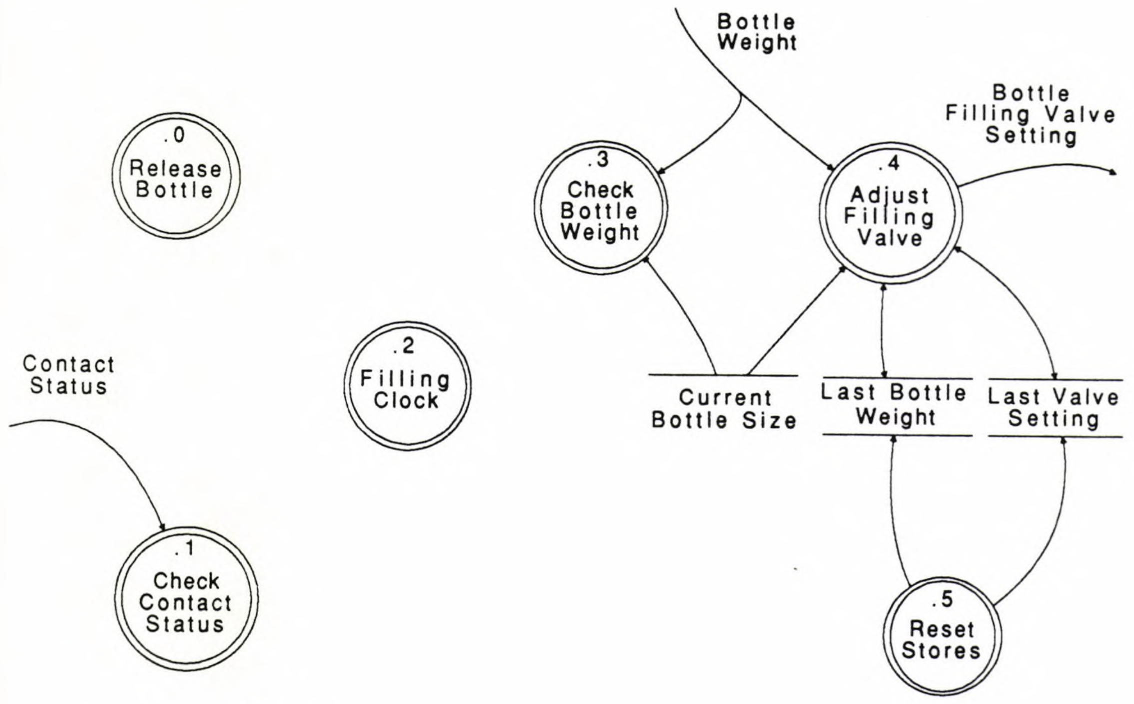


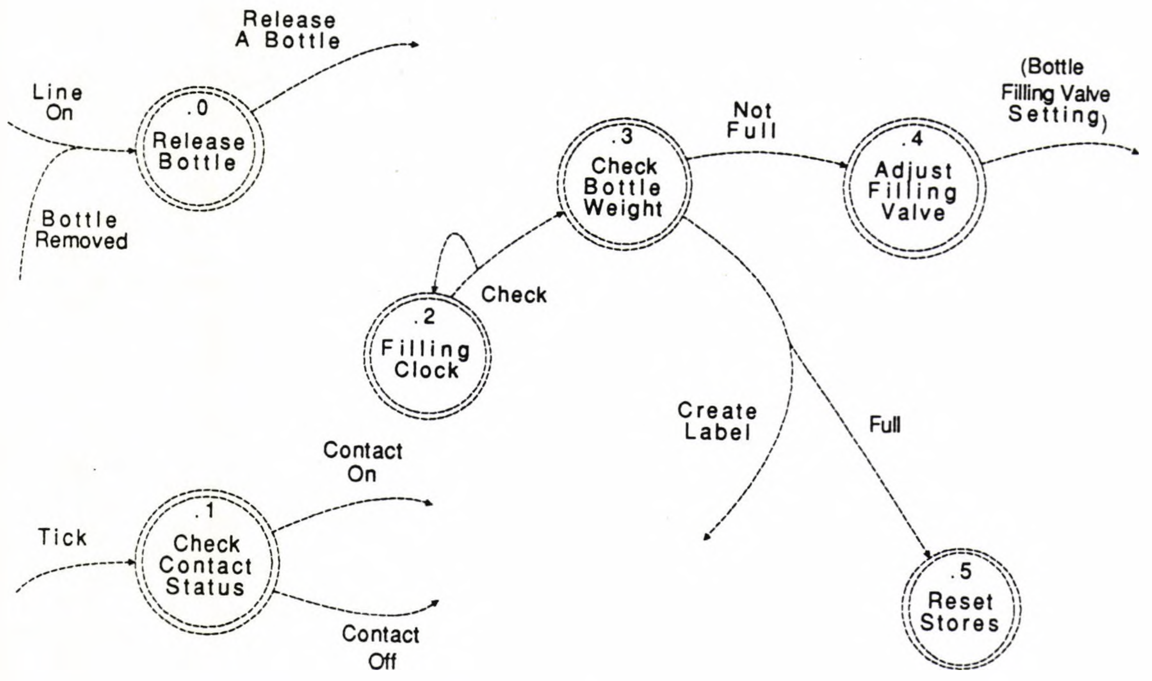
.0 Maintain Vat

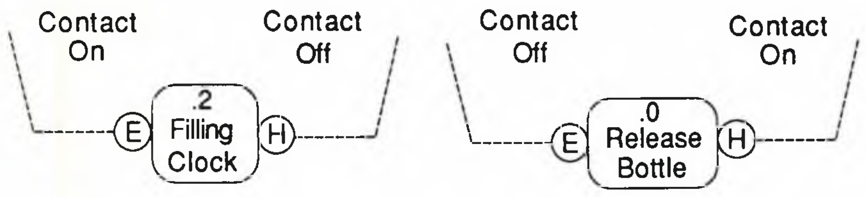




.5 Operate Line







.5.3 Fill Bottle

A.3 The Cruise Control System

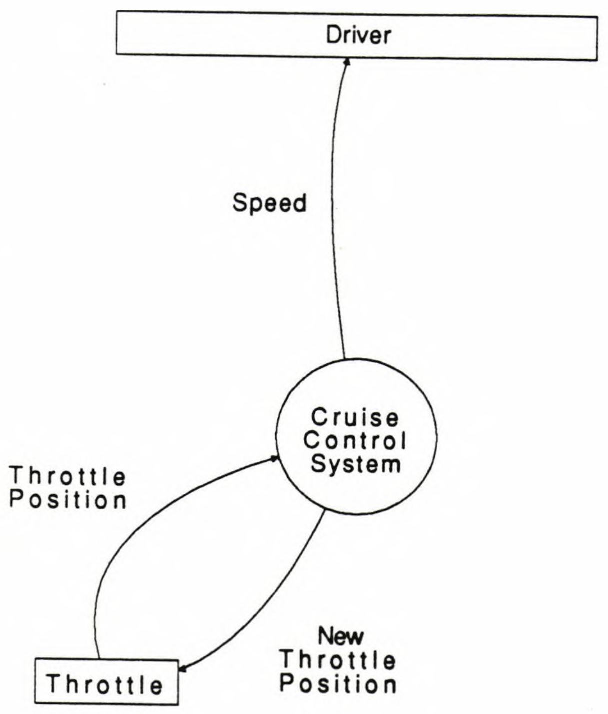
A cruise control system[2] relieves the car driver of the responsibility for maintaining speed by taking over the closed loop control. It operates only when the engine is running, and automatically sets to its "off" status when the engine is started. When the driver turns the system on, the speed at which the car is travelling at that instant is maintained. The system monitors the car's speed by sensing the rate at which the wheels are turning and maintains desired speed by maintaining and controlling the throttle position. The monitoring is accomplished by a sensor that produces a signal proportional to the throttle's position. The control is exercised by changing the degree of openness of a valve, which in turn operates a suction apparatus that draws on a chain to open the throttle. The throttle closes itself when not being actively controlled. After the system has been turned on, the driver may tell it to "start increasing speed", which causes the system to start increasing speed at a fixed rate. When the driver tells the system to "stop increasing speed", it will maintain the speed reached at that point.

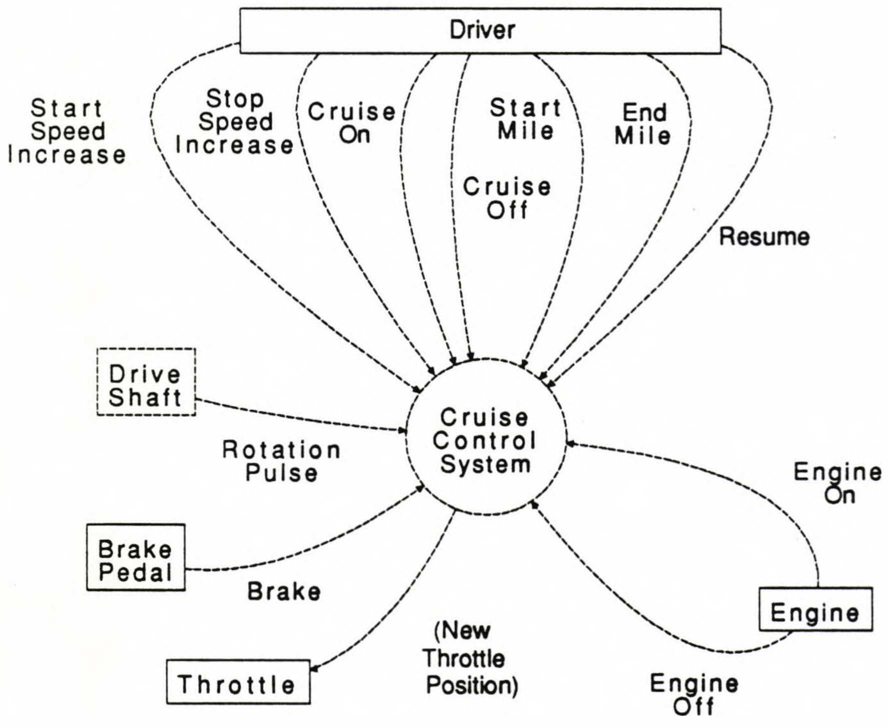
Of course, the driver may turn the system off at any time. In addition, the driver can override the system so as to increase speed simply by depressing the accelerator pedal. This causes the chain controlling the throttle to go slack. During the period of greater speed, the system continues to attempt to maintain the speed previously set, and the system will return the car to the previous speed when the driver releases the pedal. If the system is on and senses that the brake pedal has been pressed, it will cease maintaining speed but will not turn off. The driver may subsequently tell the system to resume speed (provided it hasn't been turned off in the interim), whereupon it will return at a fixed rate to the speed it was maintaining before braking and resume maintaining that speed.

The speedometers in many cars are inaccurate, and so this system incorporates its own speedometer. However, the speedometer must be calibrated when installed on a particular car. Since cars have tyres of various sizes, the mileage equivalent of one wheel rotation can vary. The system thus accepts "start measured mile" and "stop measure mile" instructions, and resets its conversion factors to correspond to the number of wheel rotations sensed within the time period of the measured mile. This can only be done when the cruise control is "off".

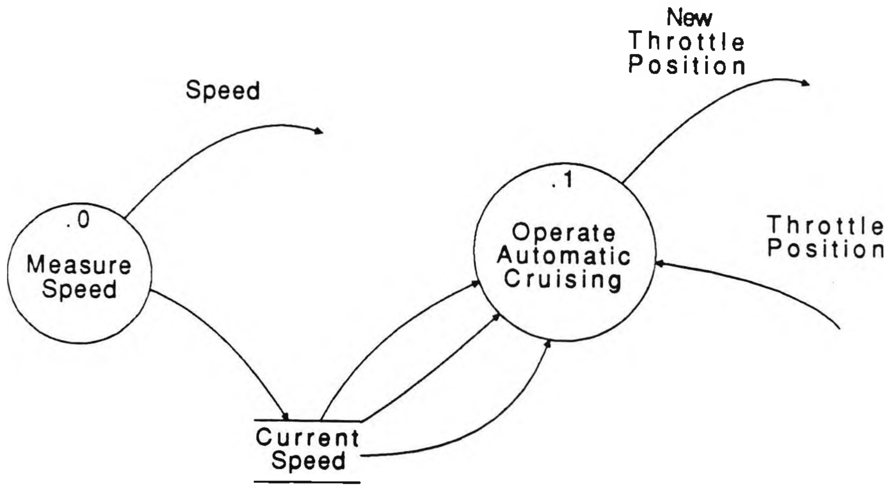
[2]This example was used in the July 1985 STARS Methodology Conference in Colorado as a basis for comparing several different development methods [HP88]. The version given here was adapted from that given in [WM86].

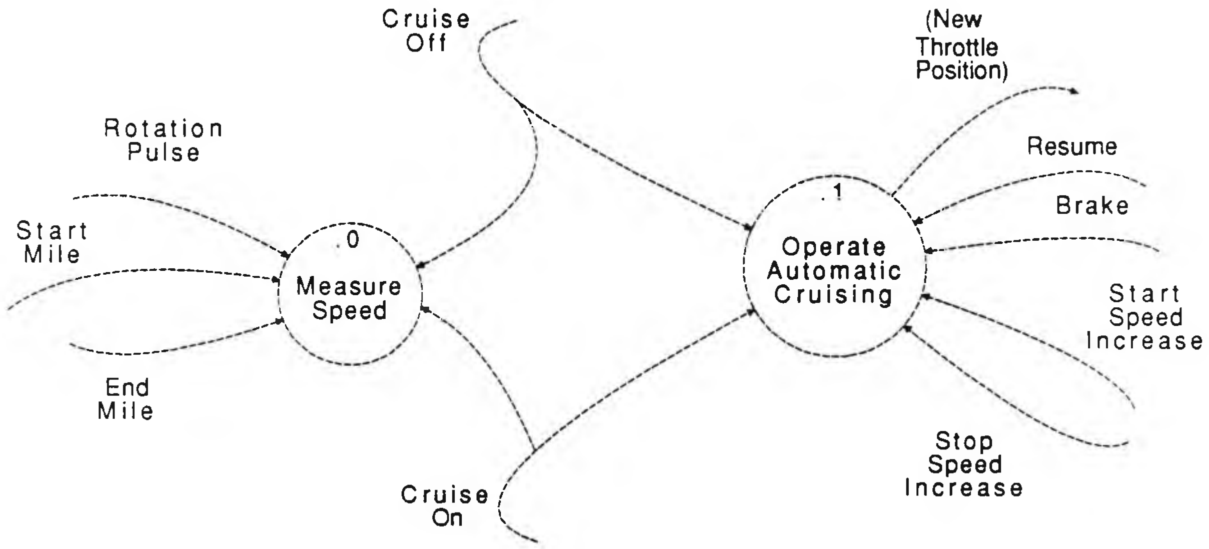
A.3.1 Specification Diagrams

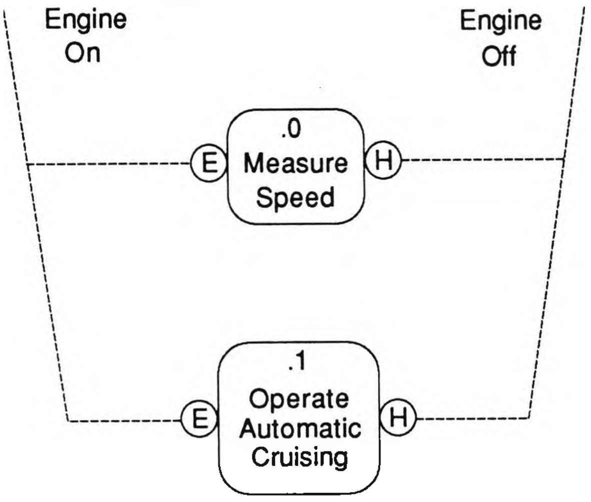




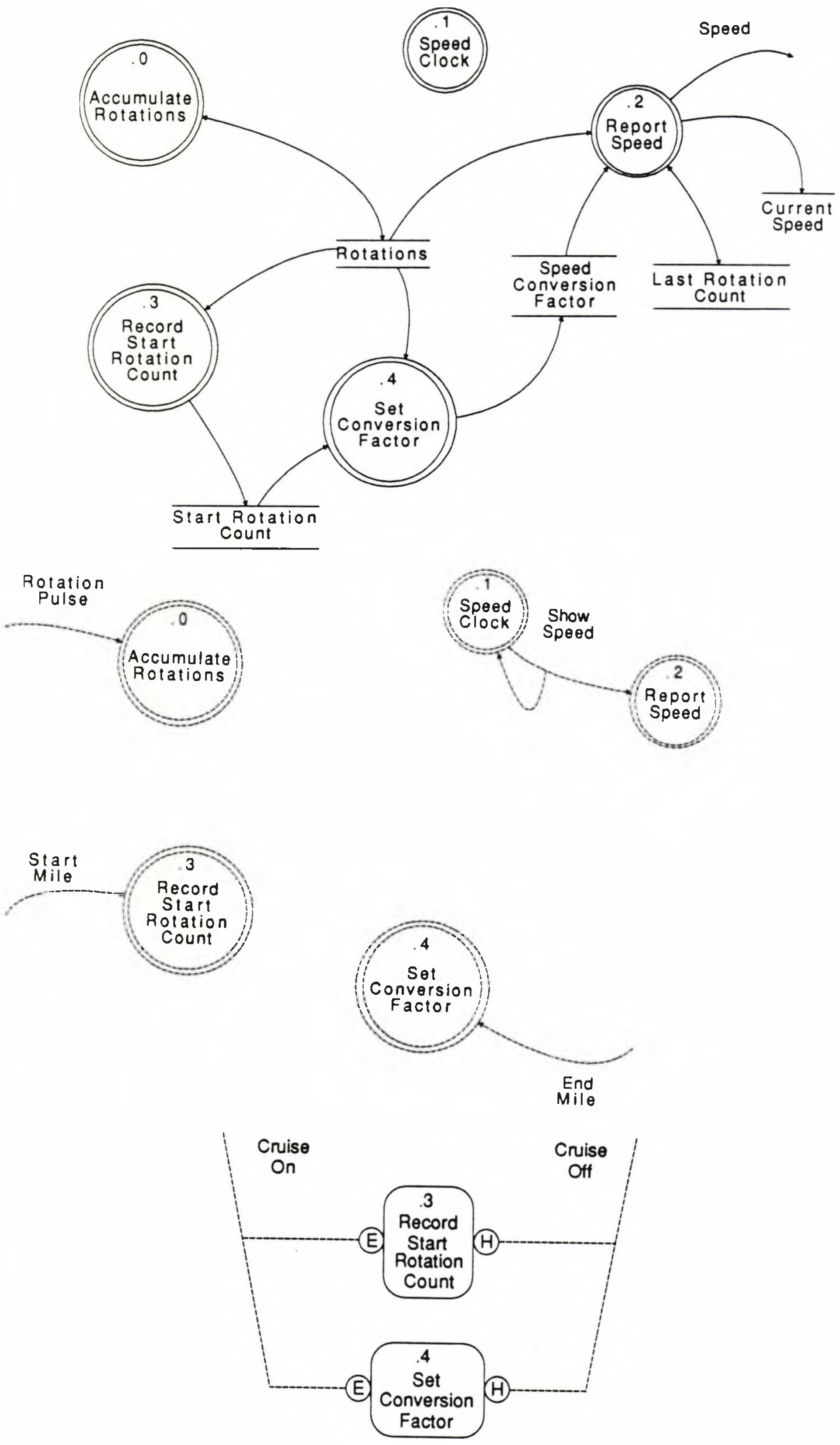
Cruise Control System: Context Diagram



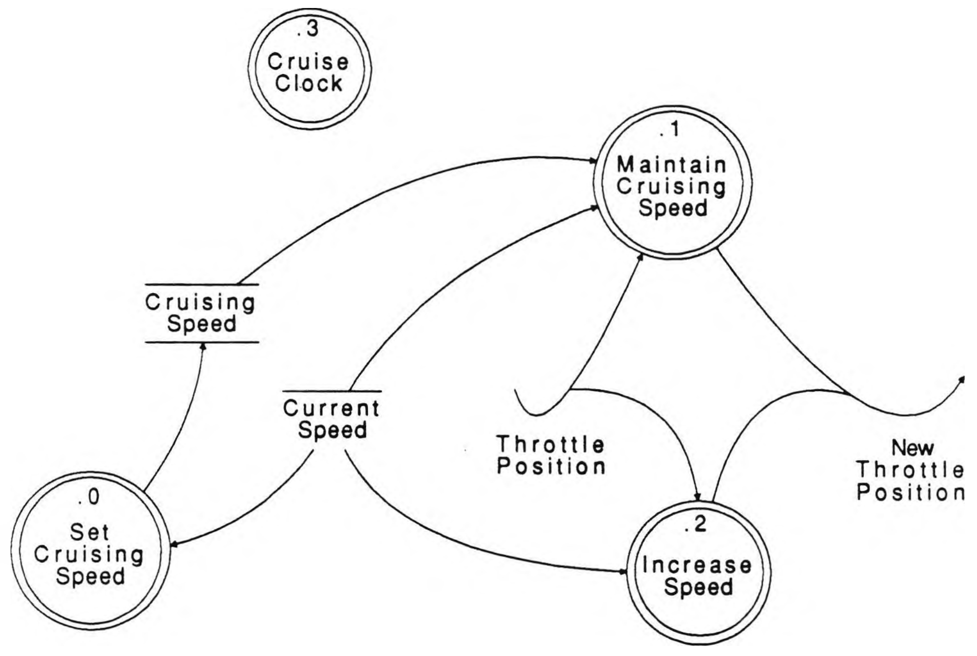


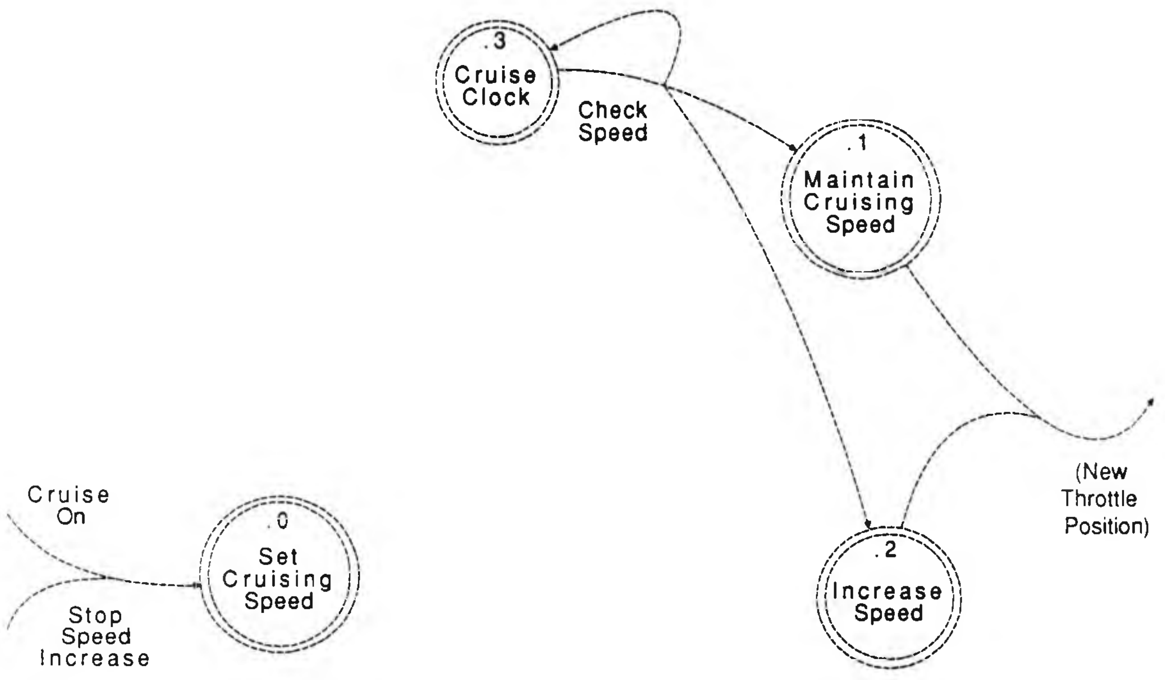


Cruise Control System

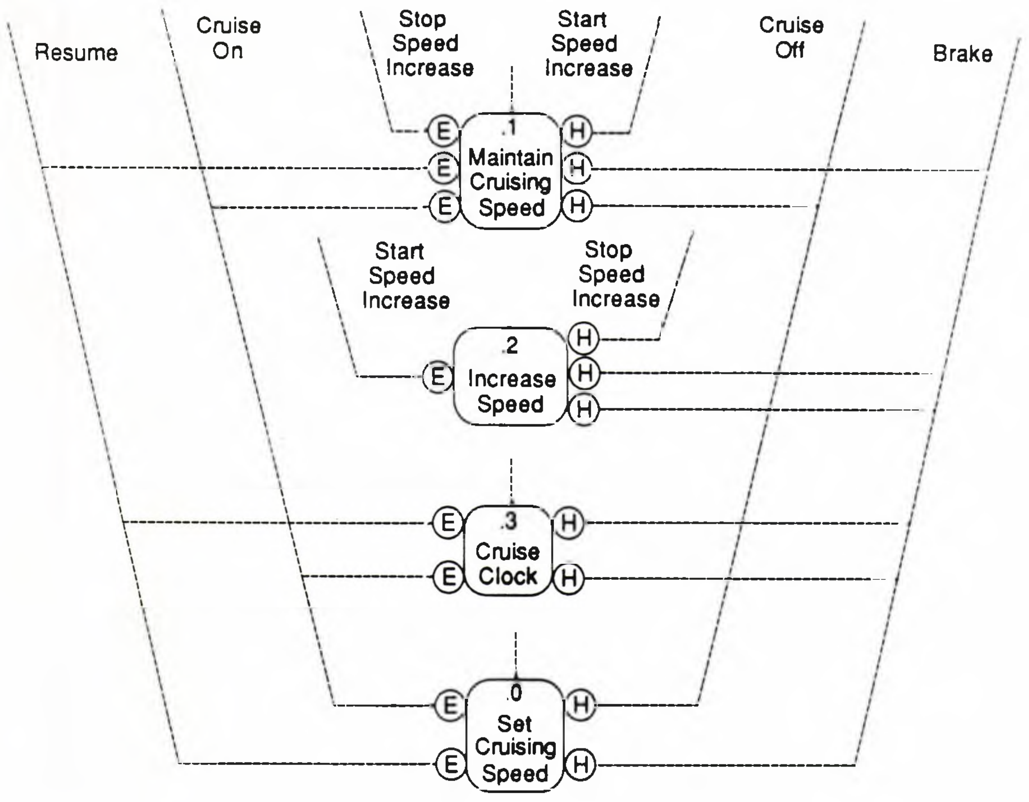


.0 Measure Speed





.1 Operate Automatic Cruising (DFD/EFD)



.1 Operate Automatic Cruising (SCD)

A.4 The Home Heating System

A computer system is to interact with a home heating system, which is equipped with a temperature sensing device and a furnace, to moderate the temperature of a house[3].

Heating System Overview

A temperature sensing device compares the difference between the temperature t_h, sensed in the house, and the reference temperature t_r, which is the desired temperature. The difference between these two, the error temperature, is measured and sent to the controller. The controller signals the furnace; the furnace produces heat, which is introduced to the house at rate Q_i; the house loses heat at the rate Q_o. If insufficient heat is supplied to the house, the temperature falls. If the amount of heat going into the house exceeds that flowing out by natural means, the temperature of the house rises. The purpose of the feedback mechanism is to keep the difference, t_r, between the reference temperature and the temperature of the house, within the desired limits if possible. A high outdoor temperature with the resultant heat flow into the house is possible, but no air conditioner is present in the current system.

Temperature Control Device

The computer system interacts with a temperature sensing device to control the desired temperature of the house. A master switch can be set at "HEAT" or "OFF". With a "HEAT" setting, the furnace will operate as in the description. With an "OFF" setting, the furnace will not operate. The homeowner is also allowed to select a desired temperature setting.

For purposes of comfort and furnace efficiency, the total change of temperature allowed will be 4 degrees. If a room temperature of 70 degrees is desired, the furnace must operate so that the temperature never falls below 68 degrees or rises above 72 degrees (unless the outside temperature is above 72 degrees).

Note that if the comfort interval (bandwidth) is too small, the frequency with which the furnace oscillates between ON and OFF will be too rapid to be efficient. If the bandwidth is too great, the house will sometimes be too cold, and sometimes too warm.

The temperature sensing device does not have great precision and accuracy. It will detect temperature variations of the order of magnitude of 1 degree. It also has a time lag of 1 minute.

The furnace Subsystem

[3]This example was used to compare several different real-time requirements methods at the 1986 COMPSAC Conference [HP88]. The version given here is adapted from that given in [Whi].

The oil furnace, which is used to heat the house, has a motor which drives a fan to supply combustion air, and also drives a fuel pump.

When the house gets too cold, the motor is activated. When the motor reaches normal operating speed, the ignition is activated and the oil valve is opened. The fuel is ignited at this time and the furnace begins to heat the water, which circulates through the house. A fuel flow indicator and an optical combustion sensor signal the controller if abnormalities occur.

The furnace is alternately activated and deactivated by the controller to maintain the temperature within the required limits. When the furnace is deactivated, first the oil valve is closed and, 5 seconds later (to allow for the valve lag time), the motor and ignition are deactivated. There is a three second lag time before the motor stops.

Controller

The inputs to the controller are:

- Heating system master switch setting which can be "OFF" or "HEAT".
- Error between the house temperature and temperature setting (t_r-t_h).
- Motor RPM.
- Combustion status.
- Fuel flow status.

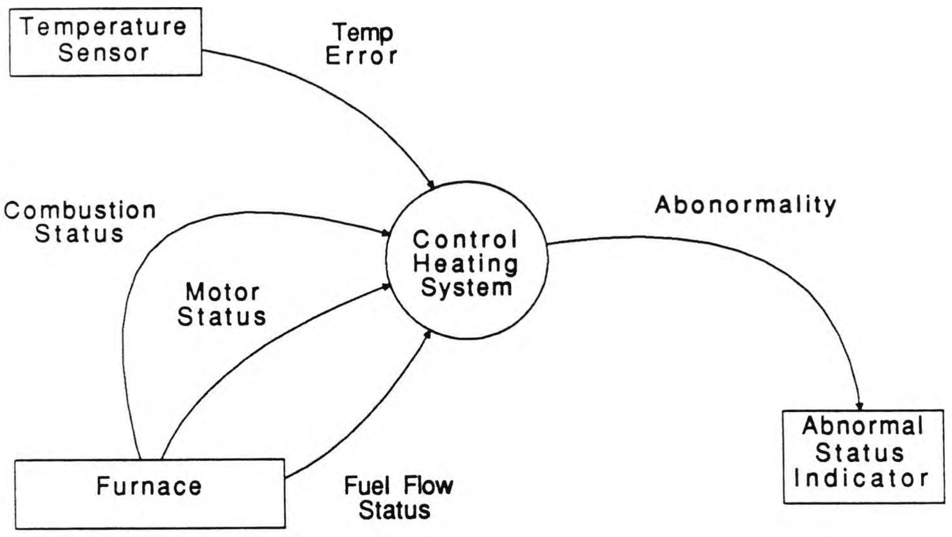
The outputs from the controller are:

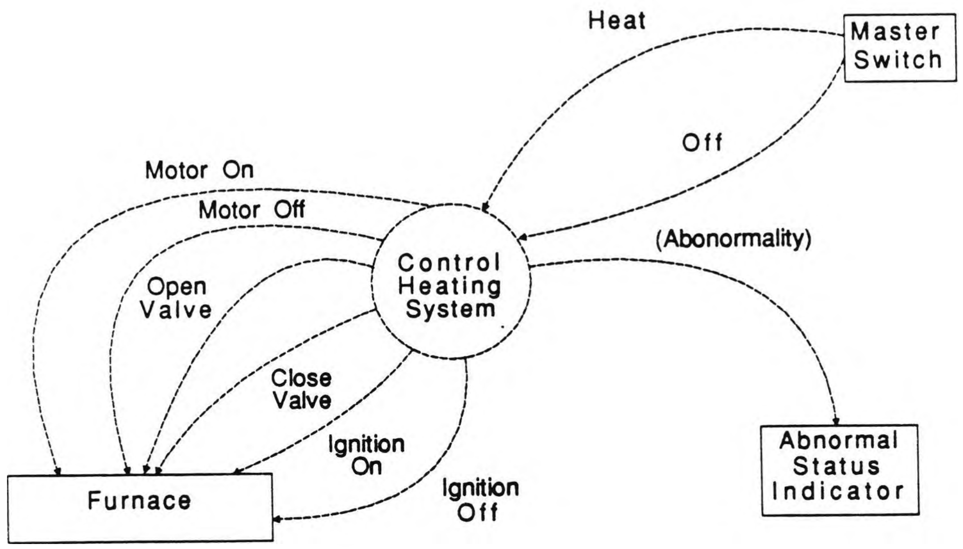
- Valve signal which is a discrete signaling the valve to open or close.
- Motor signal which is a discrete signaling the motor to start or stop.
- Signals to indicate abnormal status for combustion and fuel flow.

When the master switch is on and the outside temperature permits, the house temperature must be maintained within 2 degrees of the desired temperature. Furnace input controls shall be generated in a manner compatible with furnace operations described above. The minimum time for furnace restart after prior ON interval is 5 minutes. Furnace turn-off shall be initiated within 5 seconds after either the master switch is turned off, fuel flow rate falls below adequate levels, or the optical detector indicates the absence of combustion.

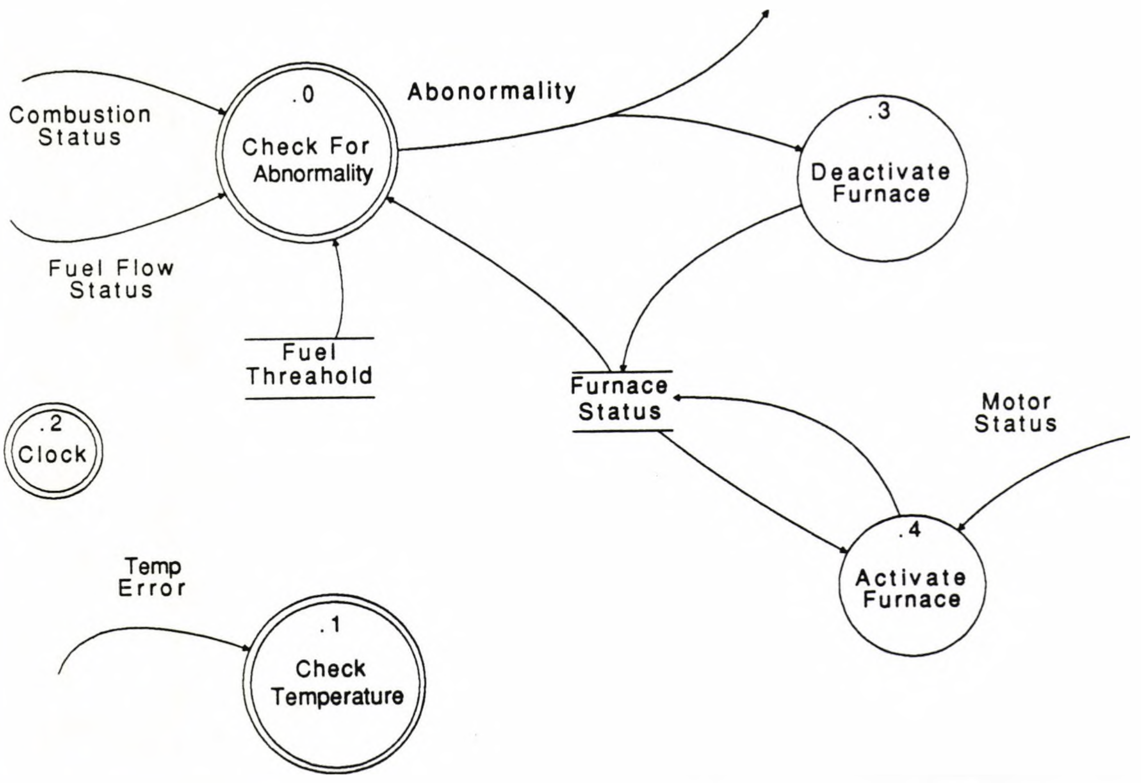
To minimise the extent of house temperature over-shoots and under-shoots beyond the desired limits, the timing of furnace signals initiating or terminating calls for heat shall be based on the rate of temperature change during the corresponding interval. The controller shall send signals to a status indicator device when abnormal conditions exist - inadequate fuel flow or lack of combustion.

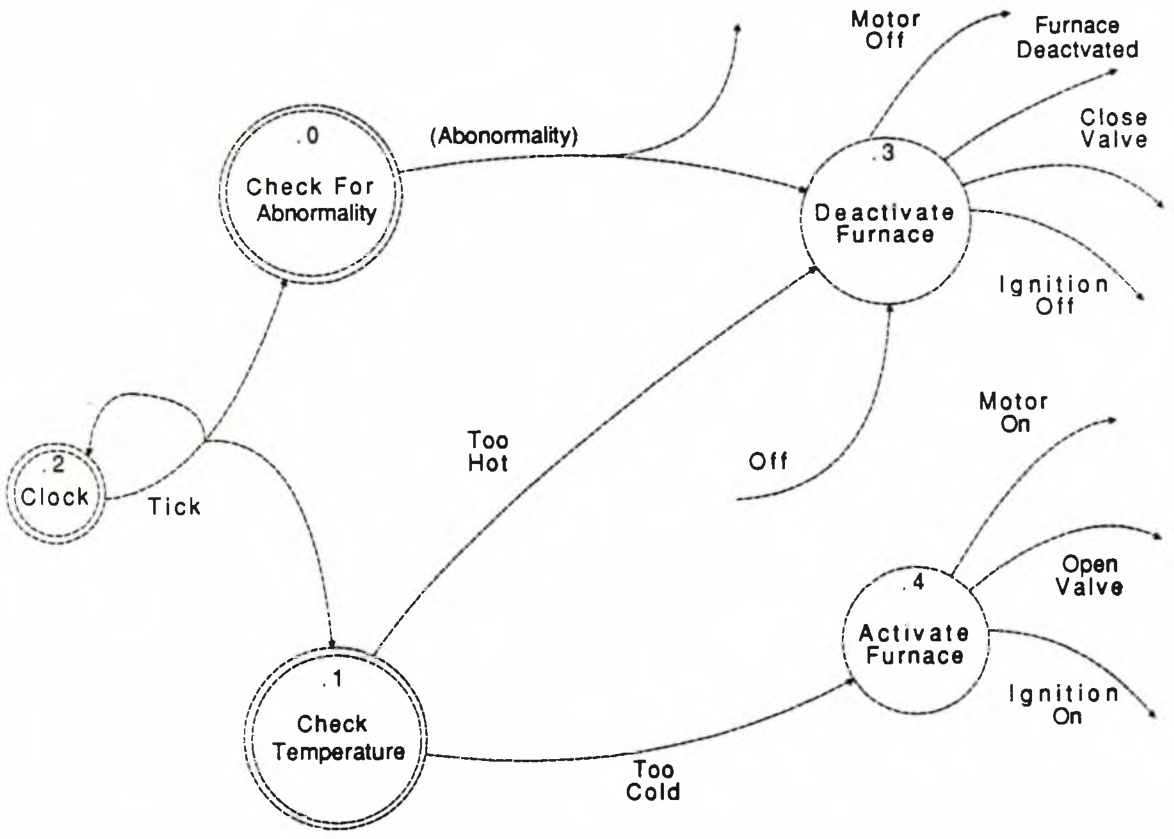
A.4.1 Specification Diagrams



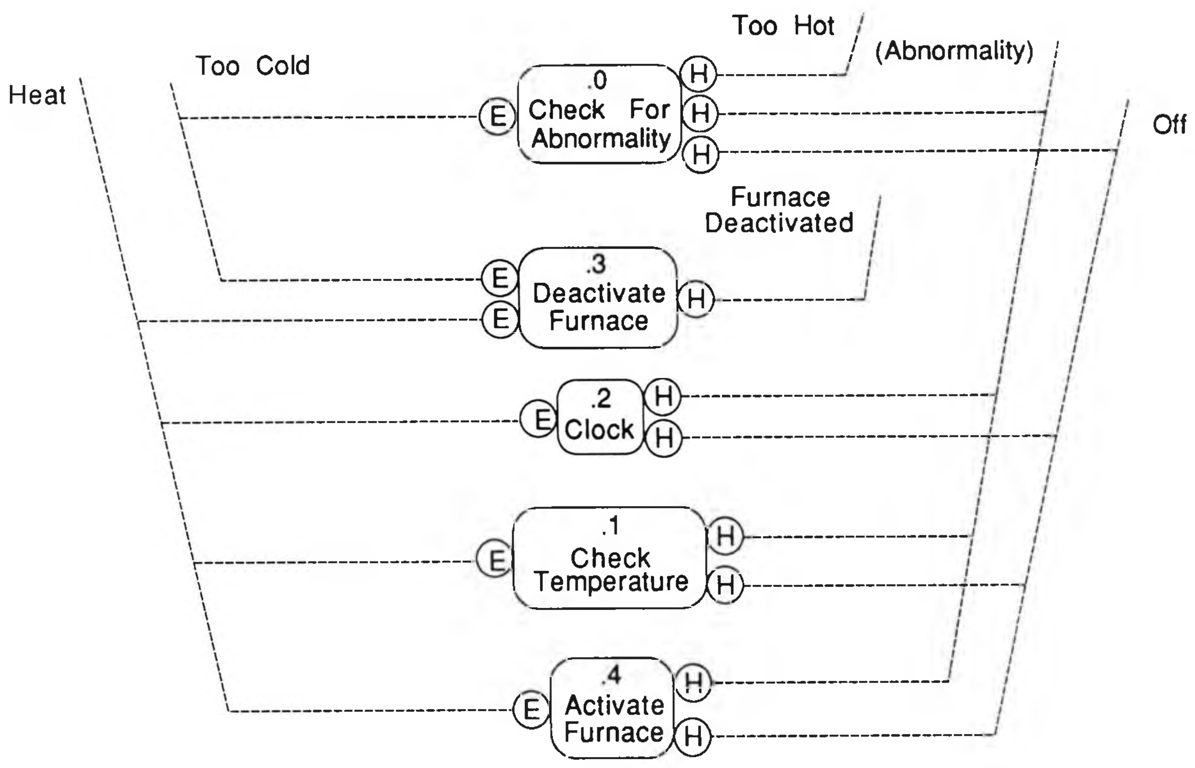


Control Heating System: Context Diagram

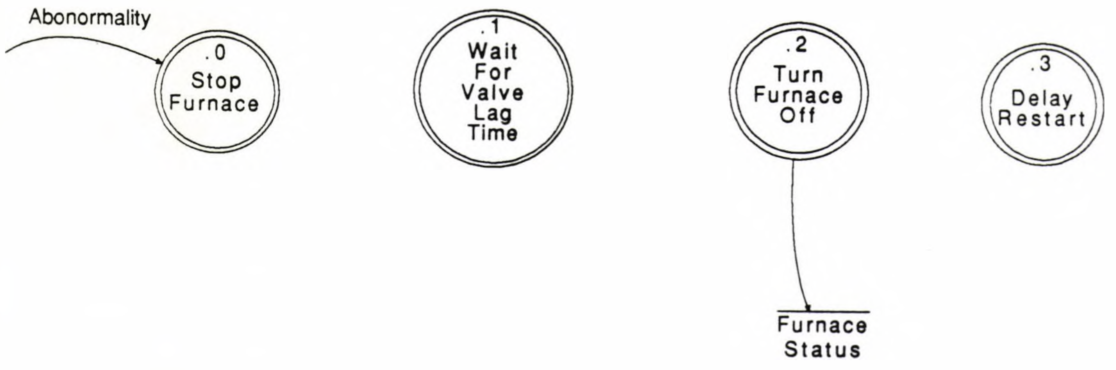


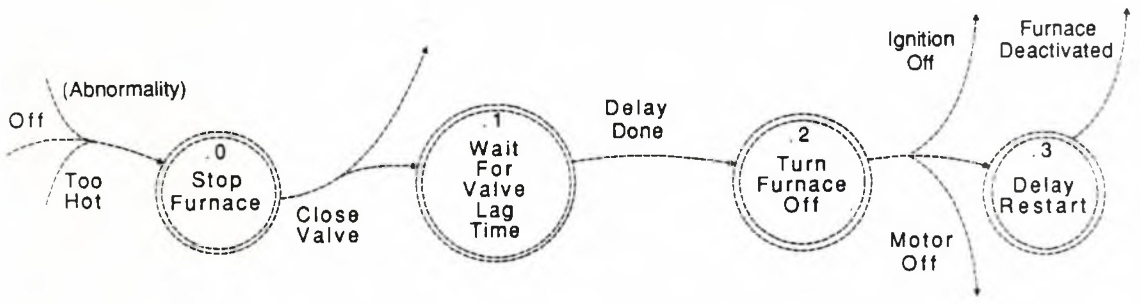


Control Heating System (DFD/EFD)

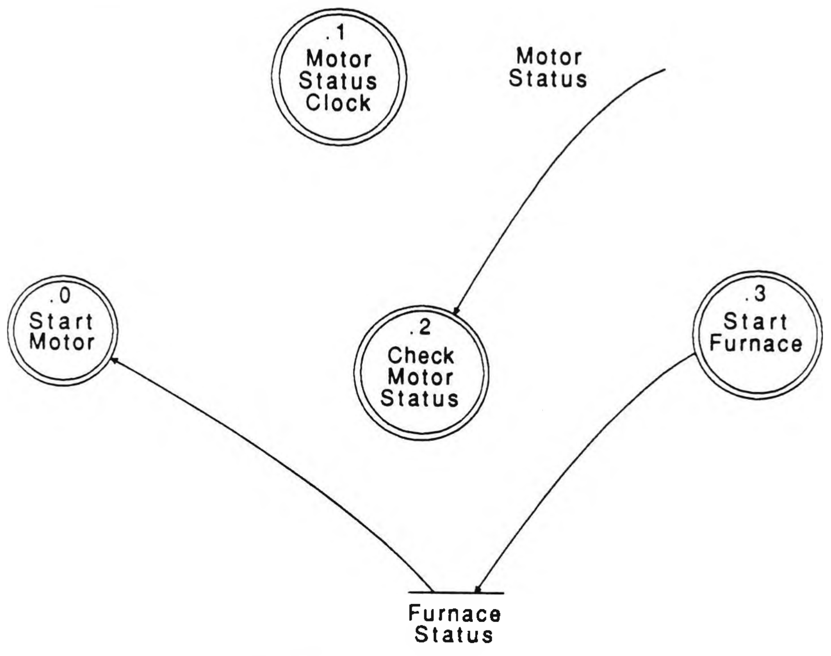


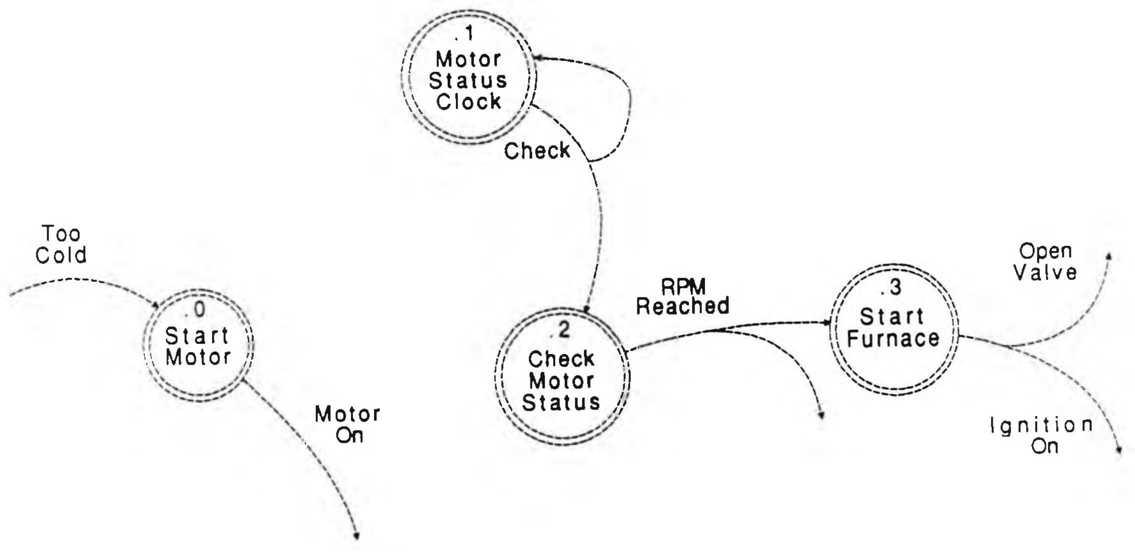
Control Heating System (SCD)

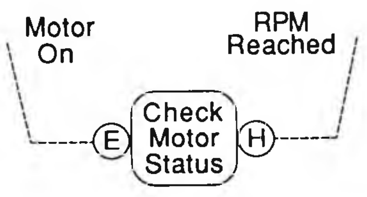




.3 Deactivate Furnace







.4 Activate Furnace

A.5 The Patient Monitoring System

A hospital has a cardiac surgery unit where open heart operations are performed on patients using the techniques of profound hypothermia[4]. After such a procedure there is considerable danger to the patient as his/her body readjusts to normal temperature control. In particular, post-operative patients have a large excess of body fluid. During the period of adjustment it is essential that the patient's body functions and vital signs be carefully monitored and, where possible, adjustments be made in time to preserve life.

The hospital is to install in its post-operative intensive care unit an on-line computer system which is used to monitor patients' life functions, such as blood pressure and heart rate, record data concerning the patients in the database, and raise an alarm when any of the monitored parameters lies outside acceptable critical limits.

The intensive care unit has a number of beds. Each bed is equipped with a visual display computer terminal (VDU) and a set of monitoring sensors that can be attached to the patient. There is also a central monitoring station where a VDU is provided for use by hospital staff. The system is controlled by a single processor with disc storage and a magnetic tape unit.

All interactions with the system are by hospital doctors and nurses and there are no special purpose computer staff employed, except for on-call maintenance engineers. The system is thus embedded within the normal functioning of the cardiac unit and has the real-time response problems of interacting with automatic monitors.

When a new patient is admitted to the unit a member of the medical staff is responsible for activating the monitoring system and initialising the patient's data. The first step is to prompt a beside VDU. This causes the system to respond with instructions which are followed by the hospital staff. Initially, the system prompts on the VDU for input data such as the patient's identity, the initial values of parameters that are not sensed directly and the acceptable upper and lower limits that are to be monitored on each parameter. After this phase, the sensors are connected by a nurse or doctor, the system responding after each connection to indicate either that the function is correct or that the sensor should be adjusted.

Once a patient has been successfully connected to the system, on-line monitoring of patient's vital signs begins, and data inputs and enquiries about the patient are accepted by the system. On-line monitoring continues until the patient is disconnected.

The monitoring data on patients is collected in two ways. Firstly, the sensors

[4]This example has been used to illustrate many programming concepts. It first appeared in [SMC74]. The version given here is adapted from the one in [DG82].

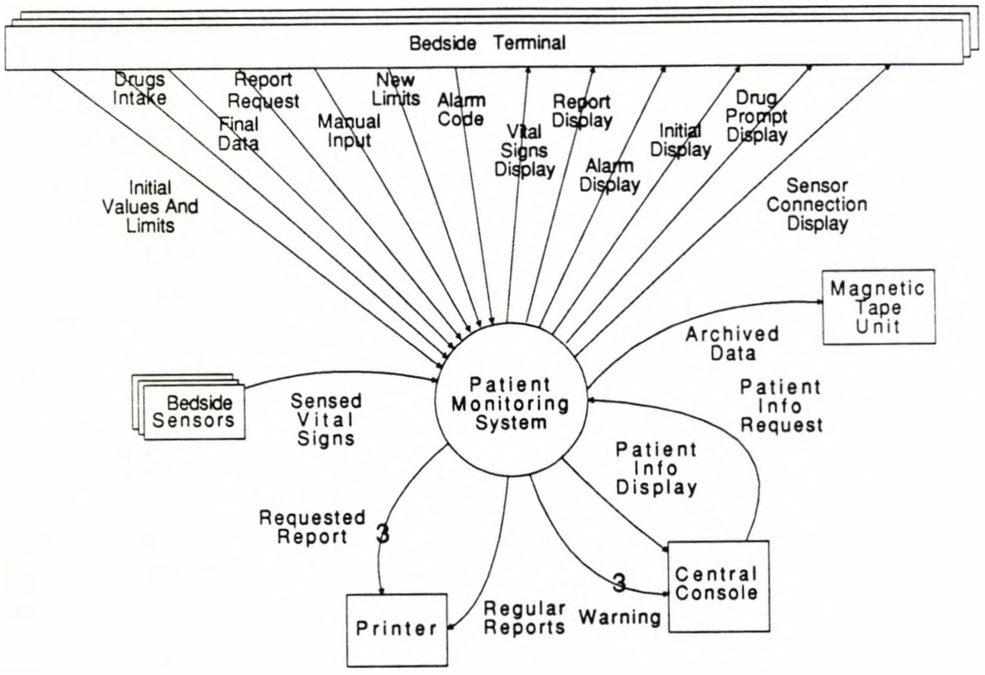
attached to each patient detect values such as blood pressure and temperature. These are polled at regular intervals and their readings recorded. Secondly, the VDU by each patient's bed is used for manual input of data by hospital staff. This data relates mainly to fluid inputs and outputs. From the measures given, the system calculates the fluid balance for the patient. Manual input can also be used to replace expected data from a failed sensor, an essential safety requirement. The upper section of the VDU screen is used to display the latest monitored readings. The system raises an alarm either when no reading is obtained, due to faults in the apparatus, or when readings show that the patient's vital signs have moved beyond acceptable limits.

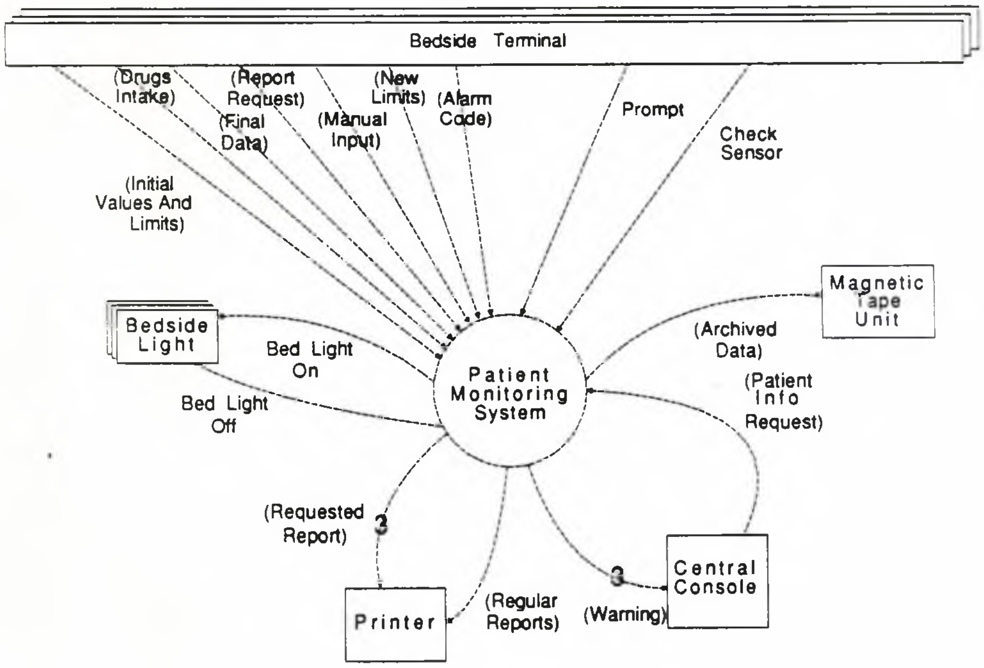
If the values detected by the system fall outside the limits set for a particular patient, then the system will activate a light over the patient's bed and a buzzer and light in a central monitoring booth. Information on the nature of the particular emergency will be displayed on the bedside VDU. These alarms will continue to function until a member of the medical staff types a code at the patient's VDU. During the alarm the system will still continue to record the patient's data for subsequent analysis.

In addition to the on-line monitoring, the system supports other functions. Hospital staff can use a bedside VDU to request that the system produce analysis of the historical data held on a patient. This can be displayed in a graphical form either on a VDU or using a hard copy printer. A doctor can note the drugs that a patient is to receive, and this information can be used to prompt the nursing staff who administer the drugs. Doctors can also alter the acceptable limits for the factors associated with a patient. There is one central VDU that is located at the nurse's monitoring station. This can be used to obtain information on any patient. The system also stores historical data on each patient which is used to produce regular report summaries.

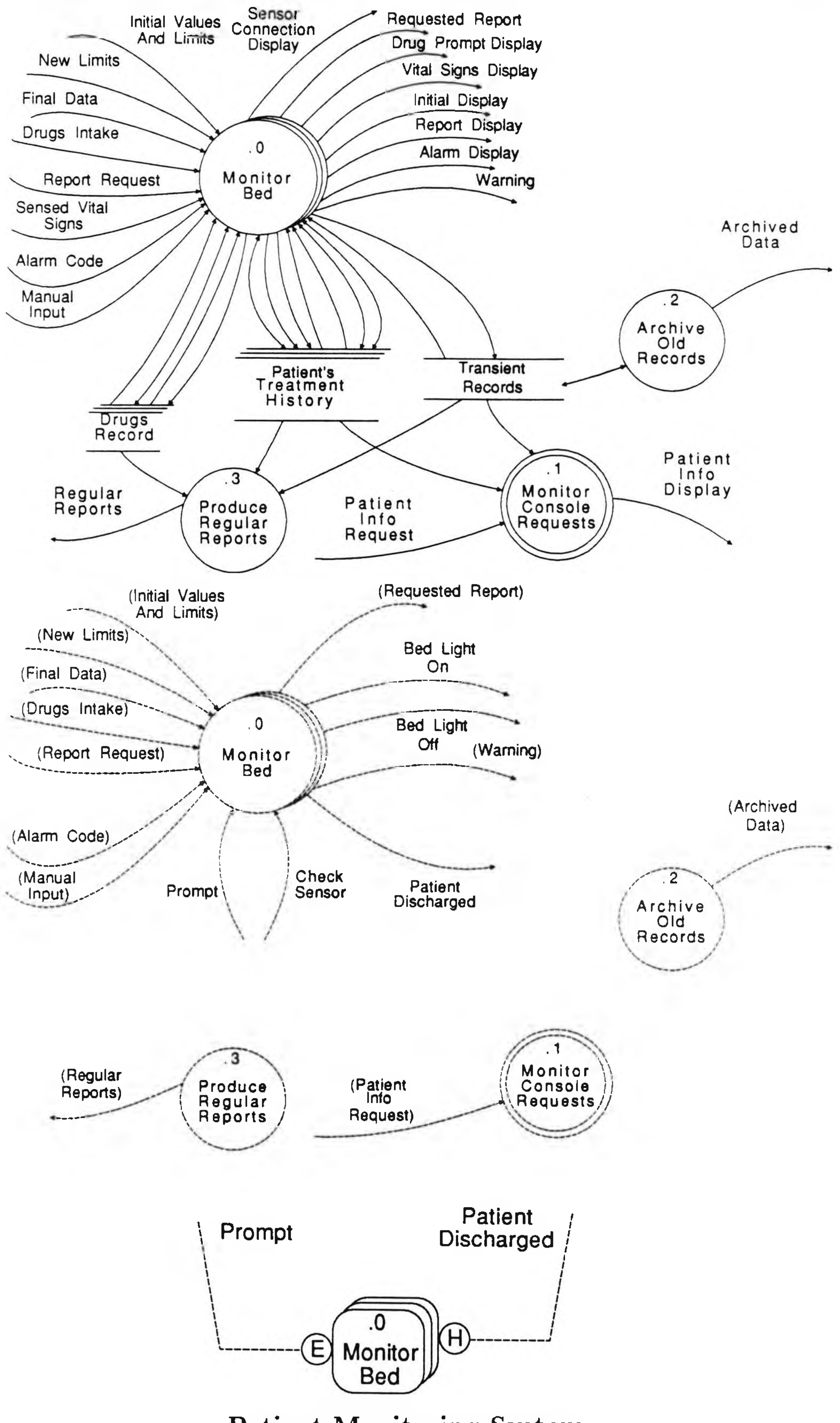
When a patient ends his stay in the unit, either by being discharged back to the ward or by dying, a member of hospital staff will disconnect his sensors and type any final data into the system using the bedside VDU. The patient's data will remain on-line for a further 48 hours, after which time it is archived onto tape and responsibility for it passed to the hospital's central data processing unit.

A.5.1 Specification Diagrams

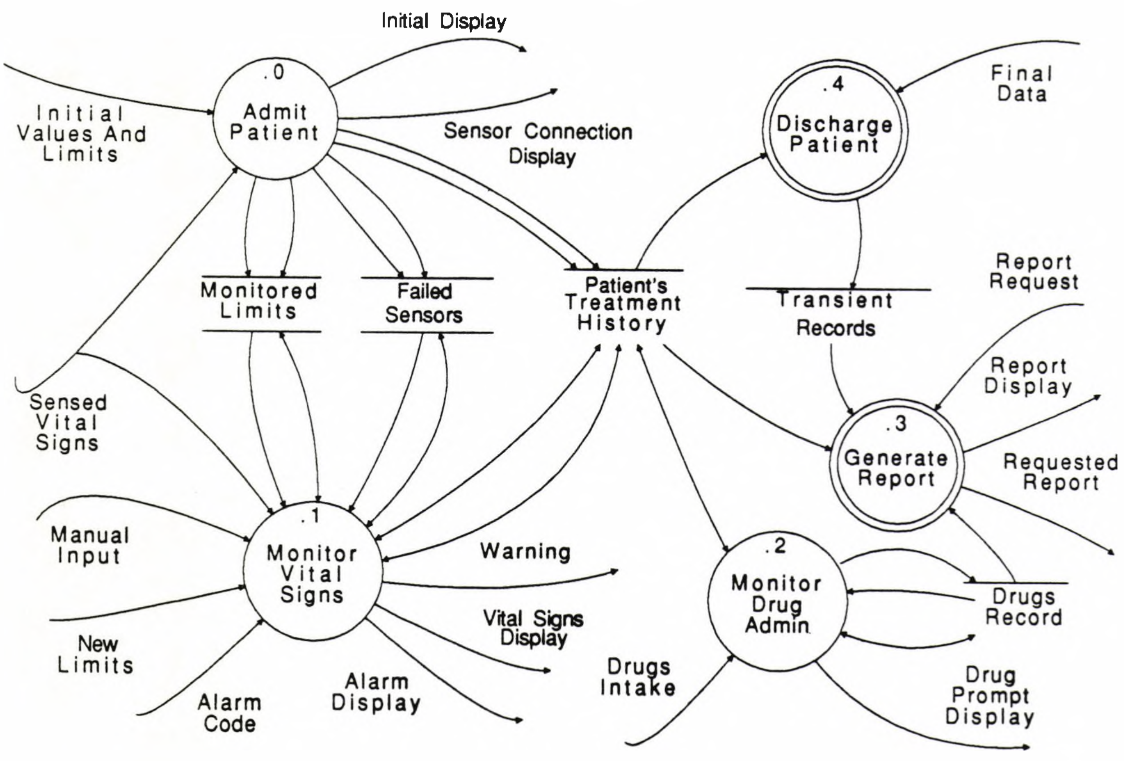


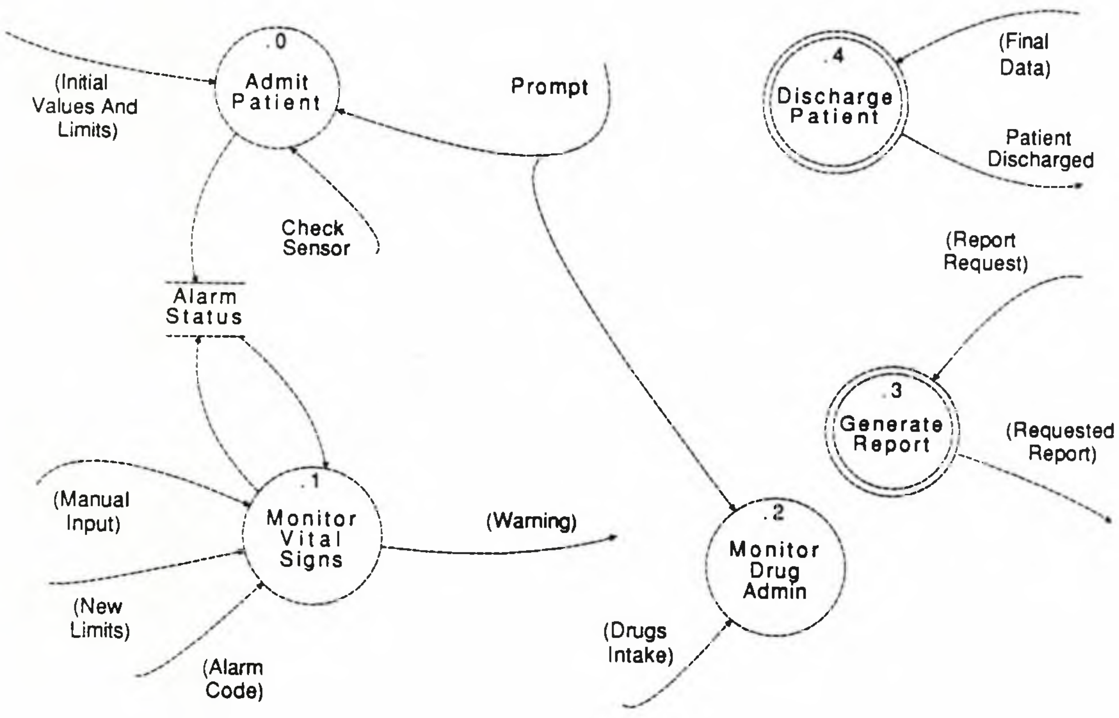


Patient Monitoring System: Context Diagram

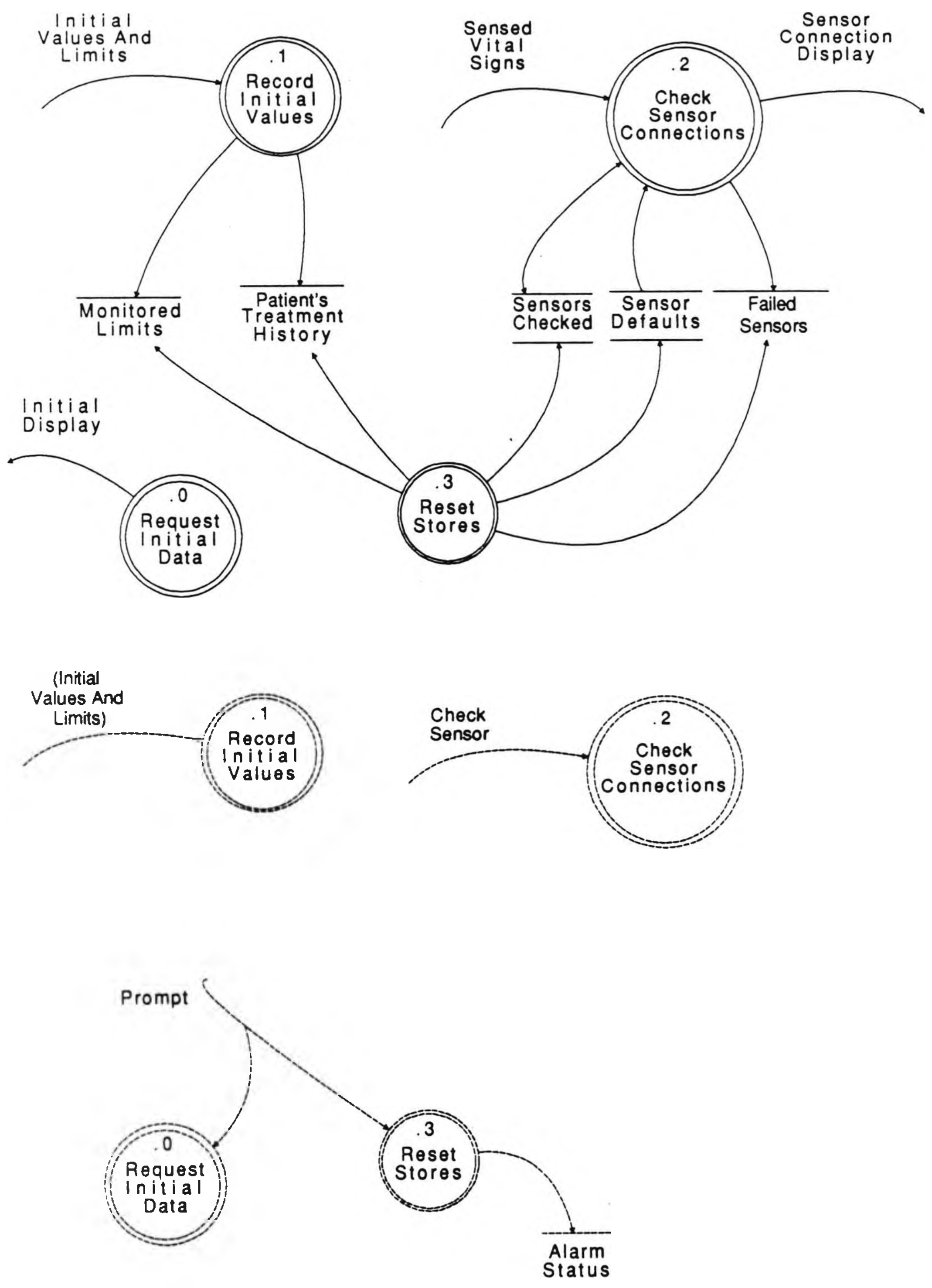


Patient Monitoring System

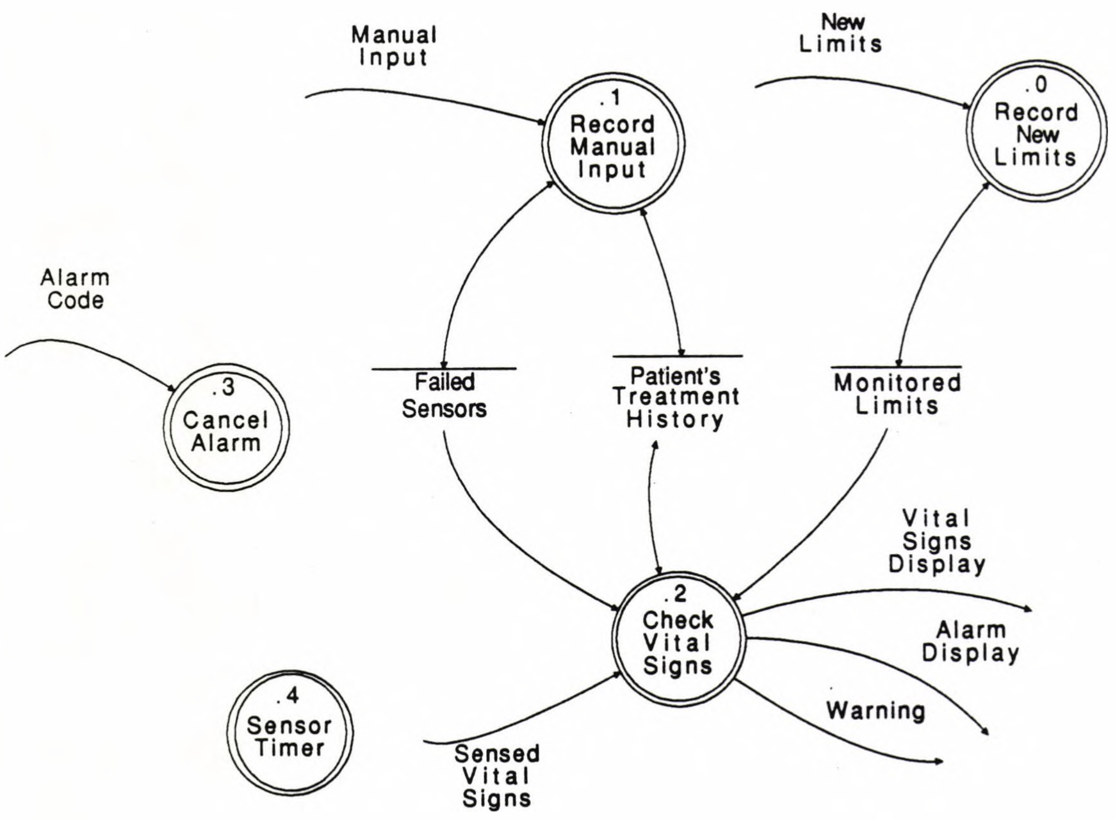


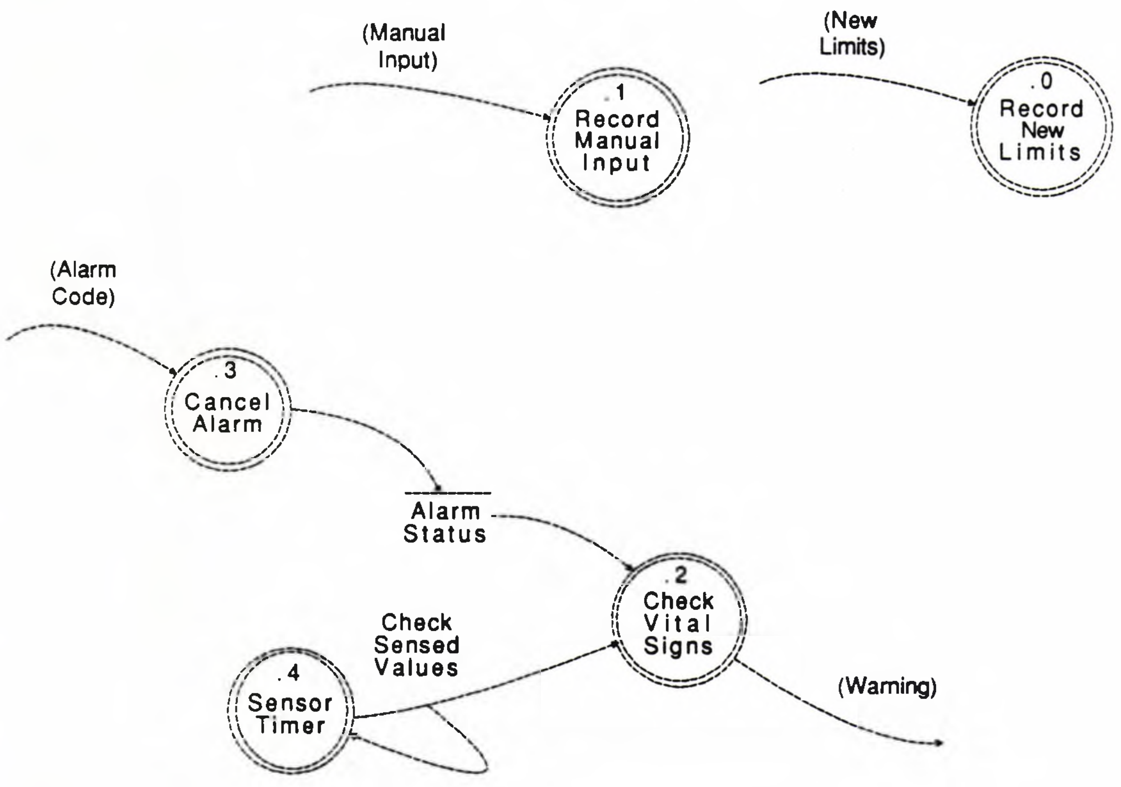


.0 Monitor Bed

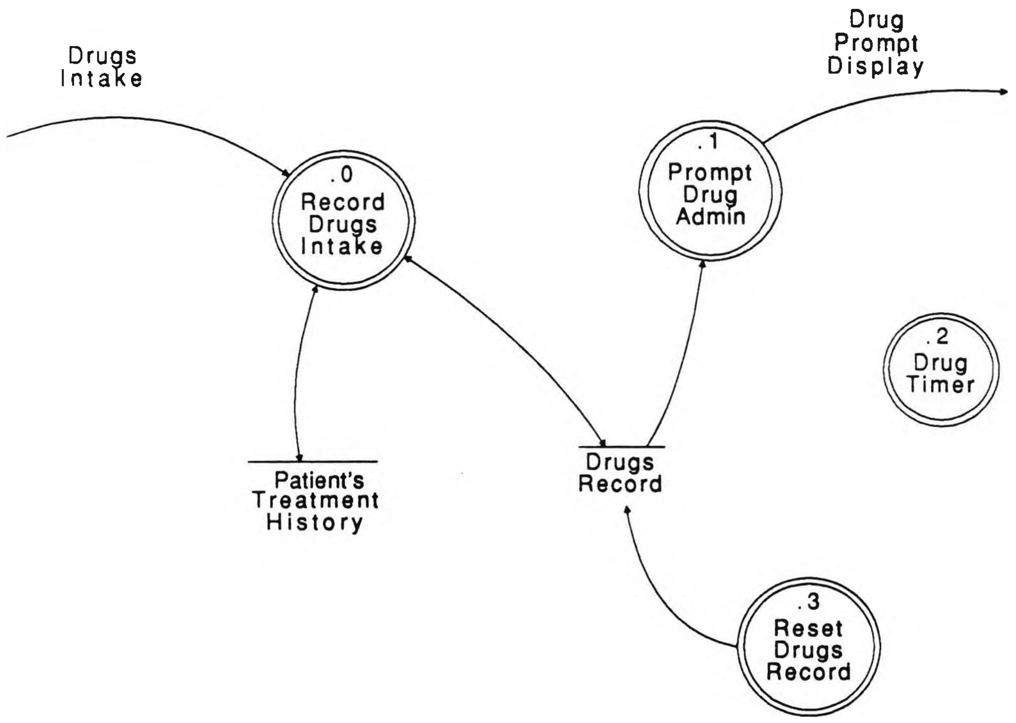


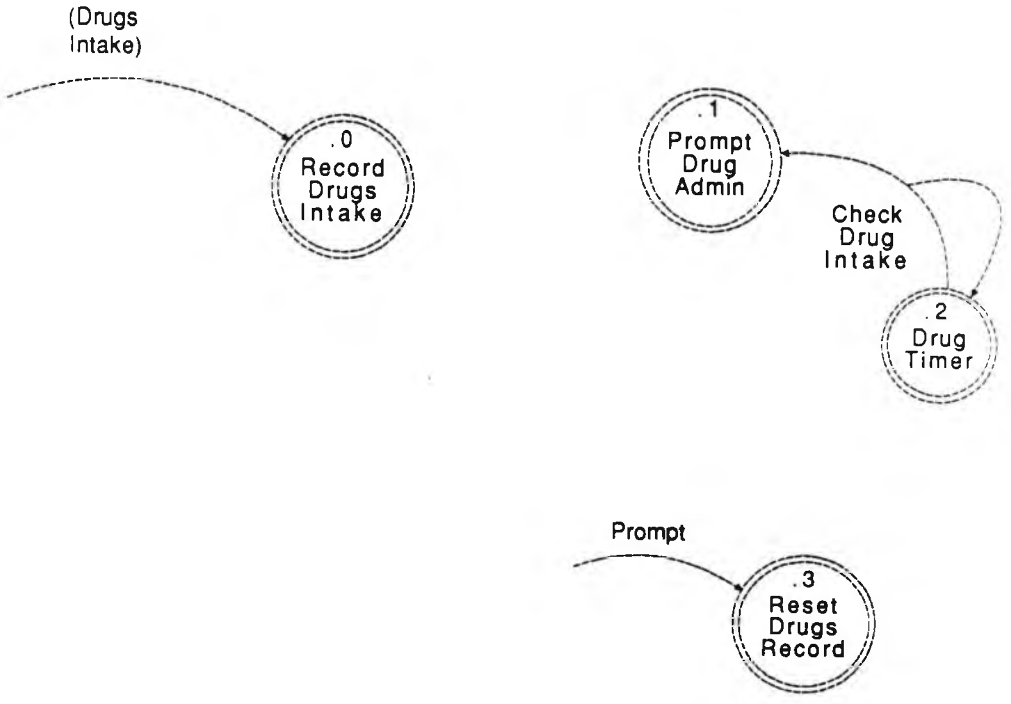
.0.0 Admit Patient



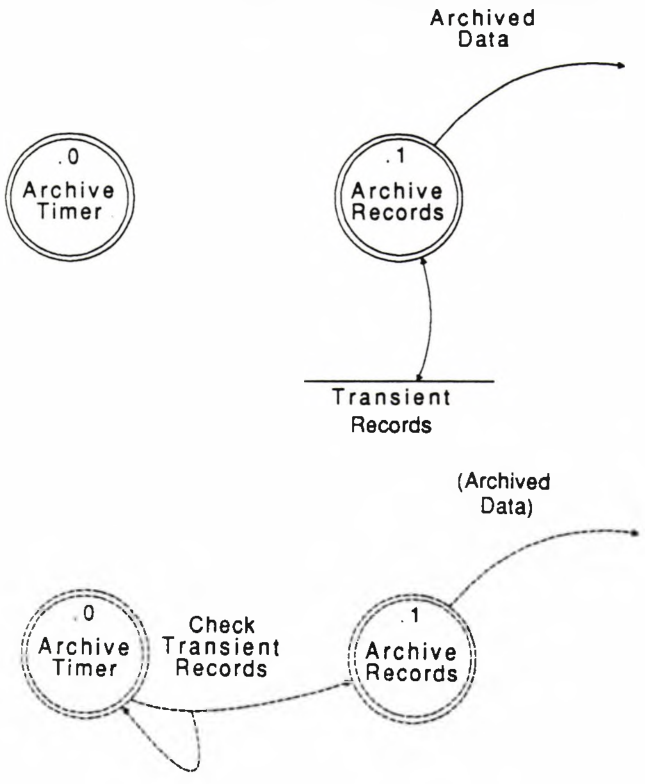


.0.1 Monitor Vital Signs

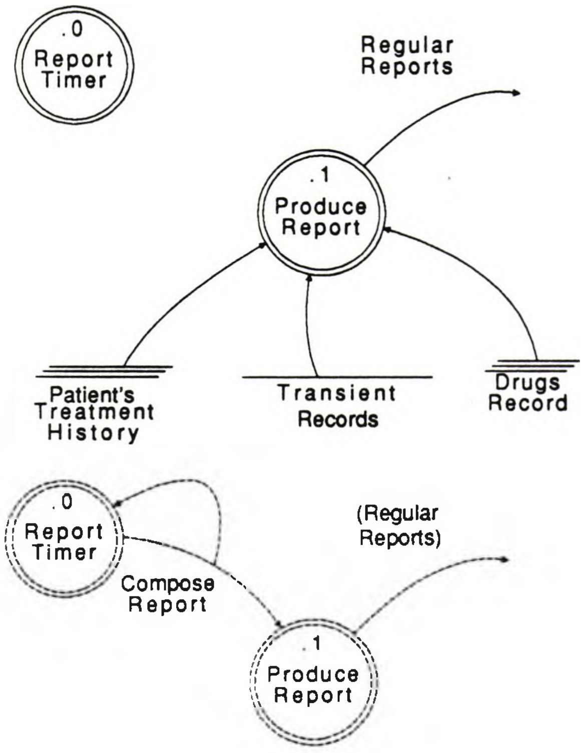




.0.2 Monitor Drug Administration



.2 Archive Old Data



.3 Produce Regular Reports

A.6 The Autoteller System

A bank is to equip its branches with a network of autotellers[5]. Each machine is to provide the following services:

- Dispense cash,
- Provide balance enquiry,
- Receive deposits, and
- Accept statement and cheque book request.

Each autoteller is equipped with a card reader, a keypad, a small screen, a printer, a deposit drawer, and a recording system. It operates as follows. When a customer inserts a card, the card reader decrypts the customer's identity from the magnetic tape strip on the card. If the wrong type of card is inserted or the card is inserted the wrong way, the card reader is capable of recognising this mistake. It ejects the card. Once a valid card has been inserted correctly, the card reader extracts the customer identity from the decrypted data. The card is then validated by consulting an area database, which is connected to the autoteller by telephone lines. The data returned by that database may indicate that the card is unrecognised, or has been reported missing or stolen. Unrecognised cards are rejected. Missing and stolen cards are transferred by the autoteller to an internal hopper. If the card status is OK, the customer is prompted for his personal ID number.

The personal ID is also checked with the area database for verification. The customer is allowed a number of retries for entering his personal ID number. If an invalid personal ID is entered on all retries, the card is again held by the autoteller, and transferred to its internal hopper. On successful entry of the personal ID number, the customer is presented with the choice of services offered by the autoteller.

If cash withdrawal is requested, then the customer is asked for the desired amount. Each customer is allocated a weekly limit. The amount requested is checked by enquiring the customer's limit from the area database. If the customer has already exceeded his/her limit, no cash is dispensed. If (s)he is allowed further withdrawals, which are smaller than the current choice, an amount re-entry is requested. When the requested amount does not violate the customer's limit, cash is dispensed along with a receipt.

To deposit money, the customer takes a numbered envelope from the drawer, and after sealing and replacing it in the drawer, enters the envelope number as well as the deposit details via the keypad. A balance enquiry causes the

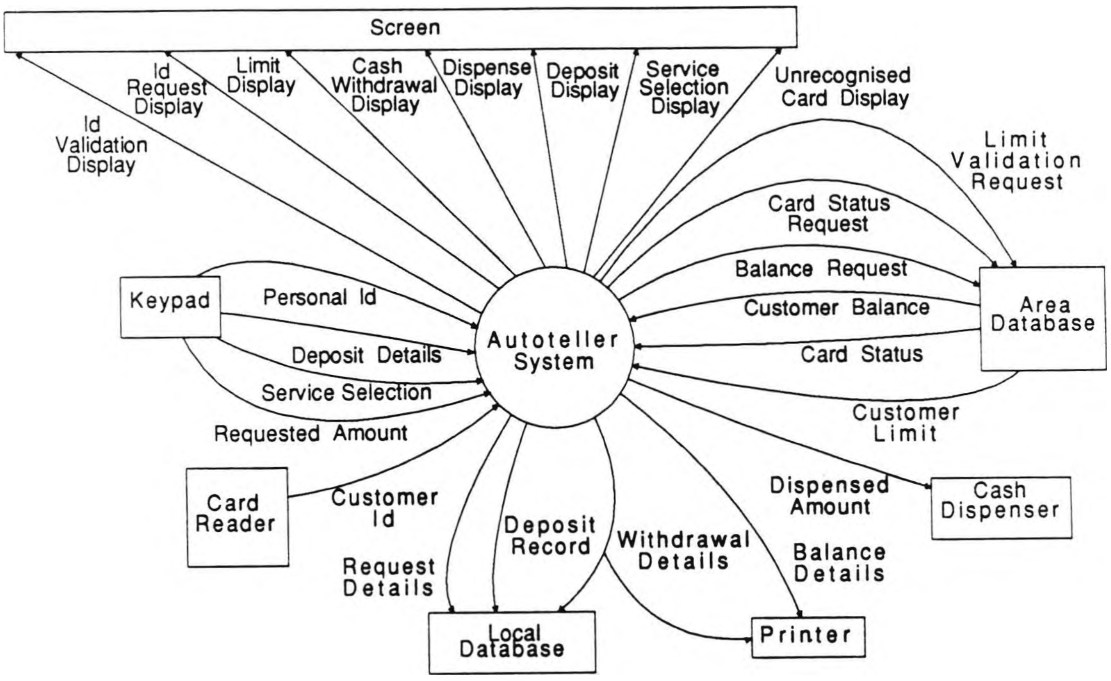
[5]This specification exercise is adapted from a MASCOT exercise set at the Computer Science Department of Stirling University [mas86].

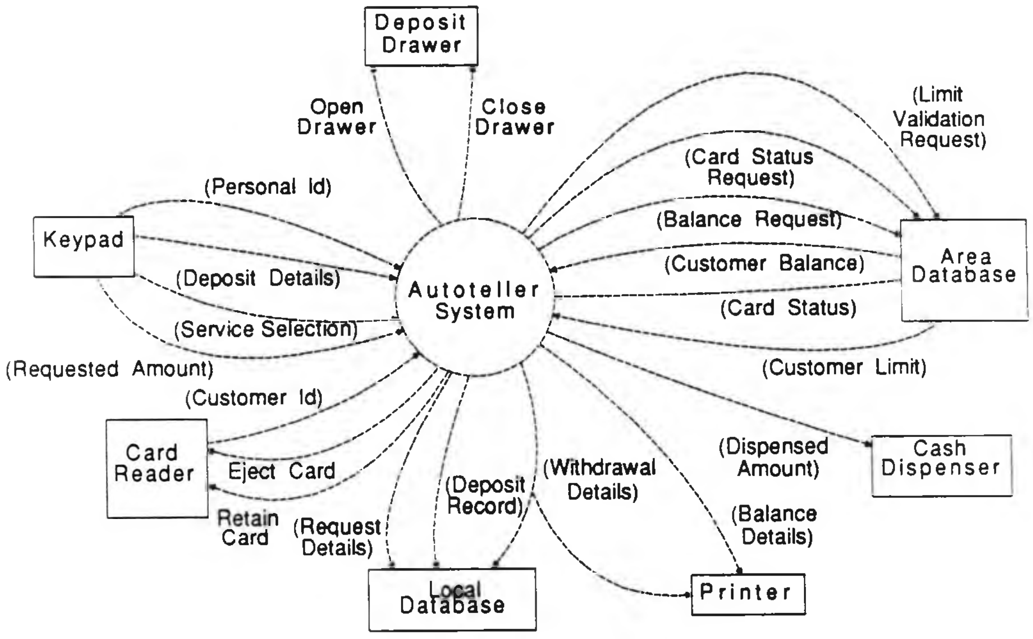
autoteller to forward a request to the area database, whose reply is printed on a slip.

All cash withdrawals are recorded locally for security reasons. They are also forwarded to the area database to allow calculation of weekly limits. The local database also stores deposit details for later manual handling by local staff. Statements and cheque book requests are also recorded locally and manually processed.

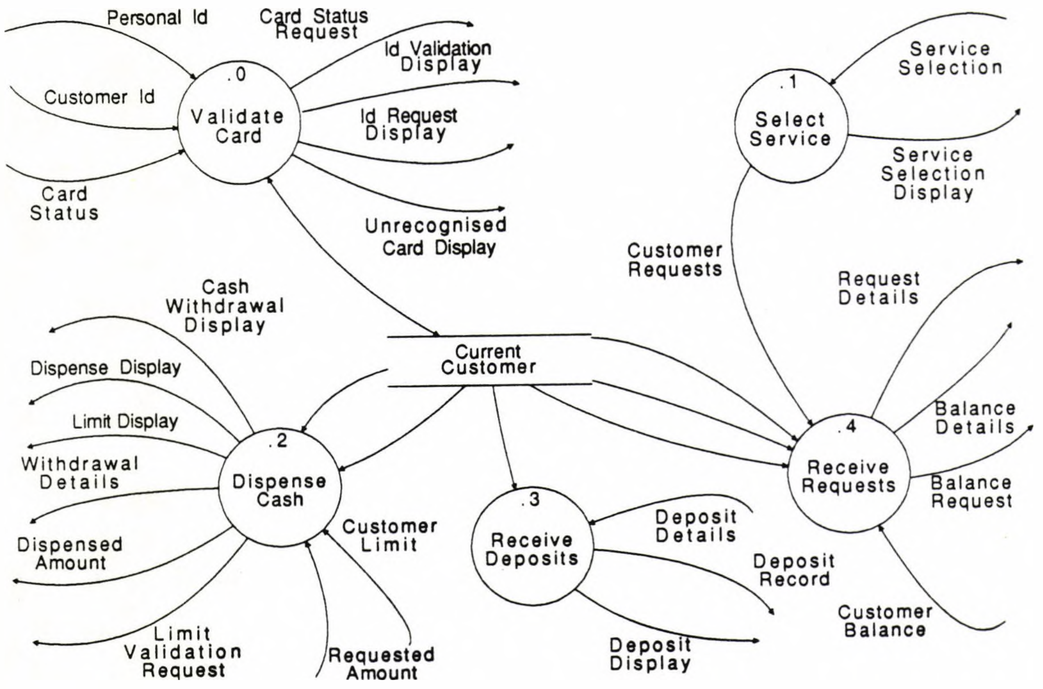
The customer is guided for input by various screen displays and should be allowed multiple choices from the services menu. (S)he terminates service selection by pressing an end key on the keypad. His/her card is then returned to him/her.

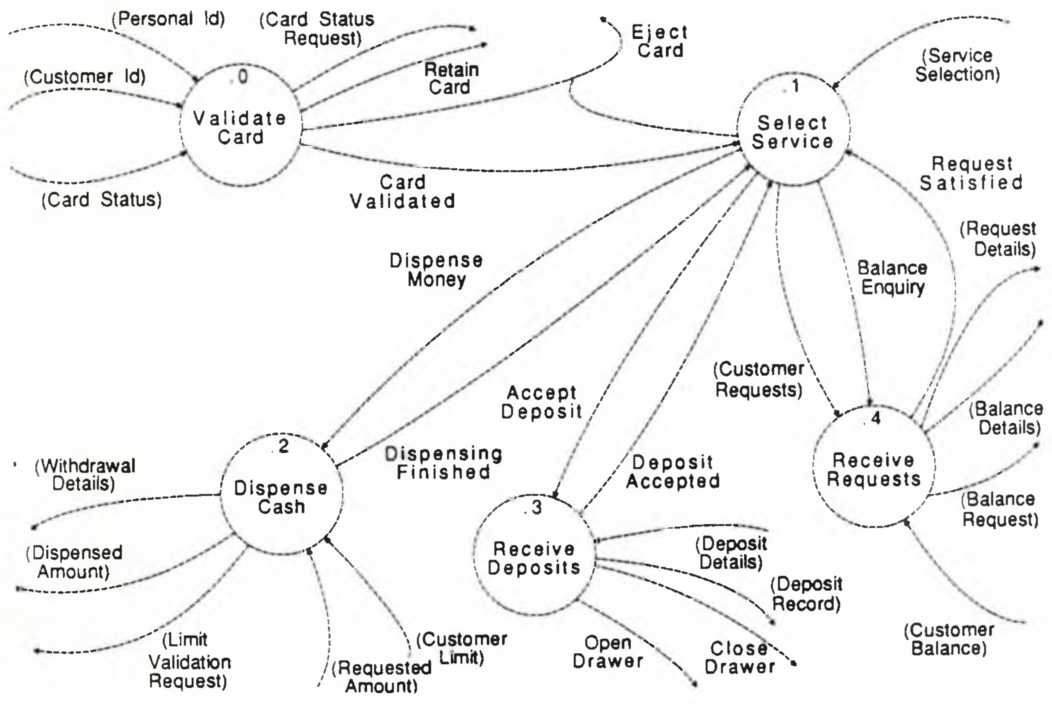
A.6.1 Specification Diagrams



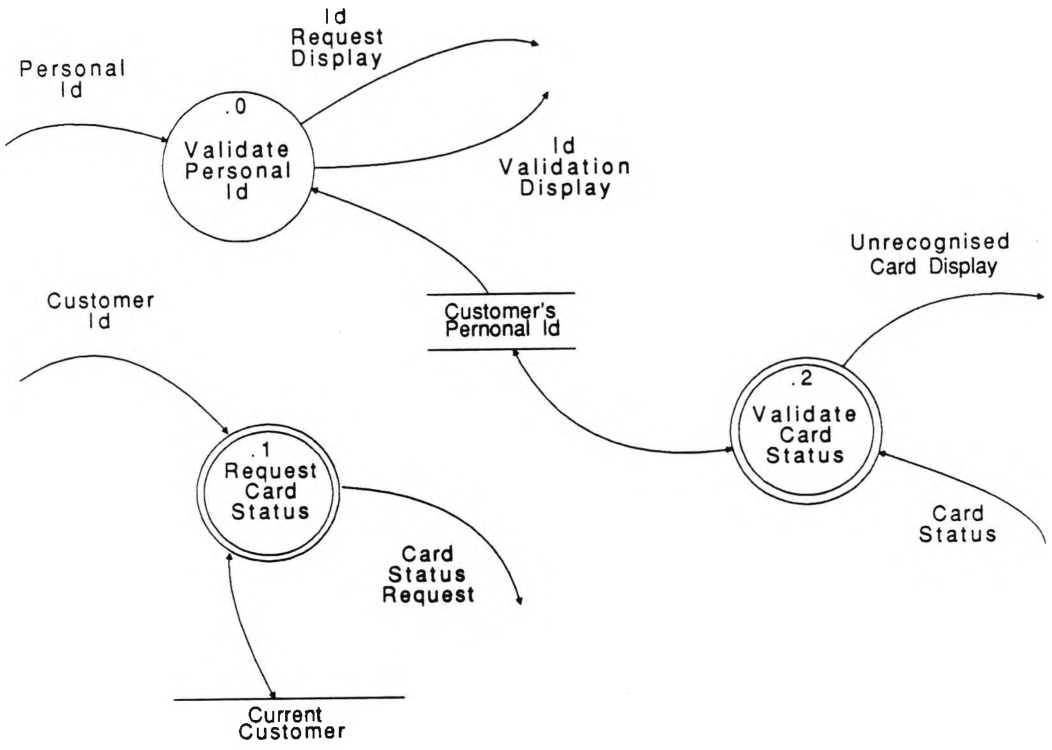


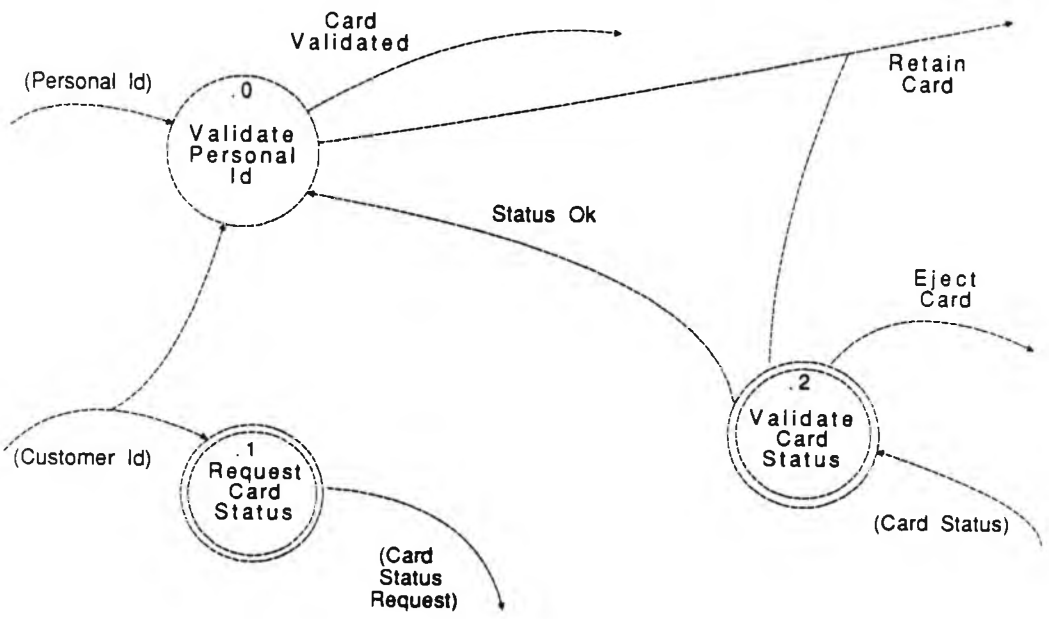
Autoteller System: Context Diagram



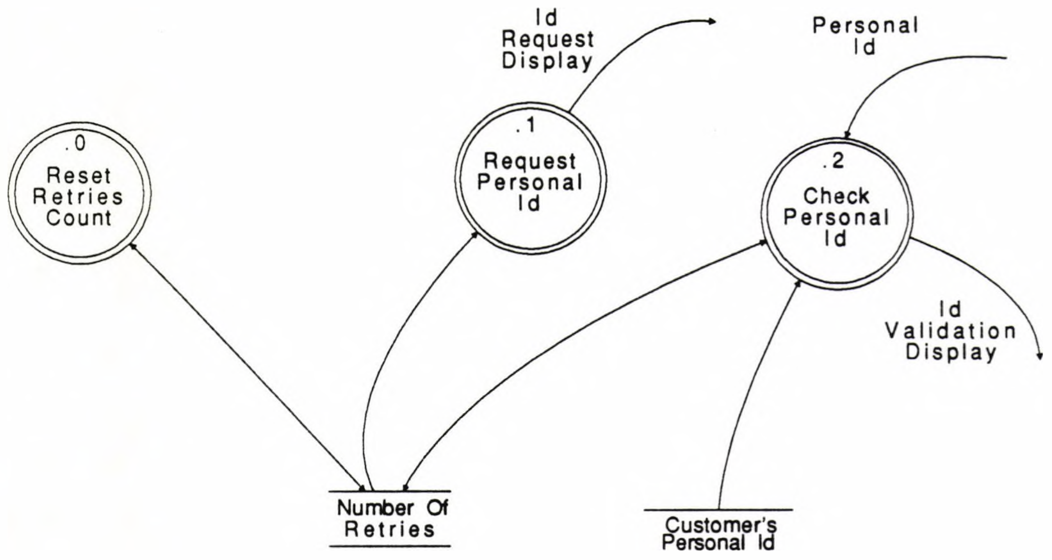


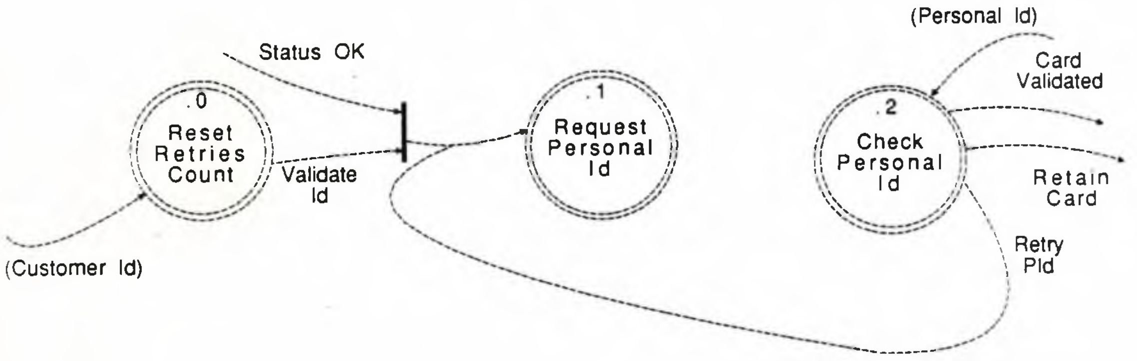
Autoteller System



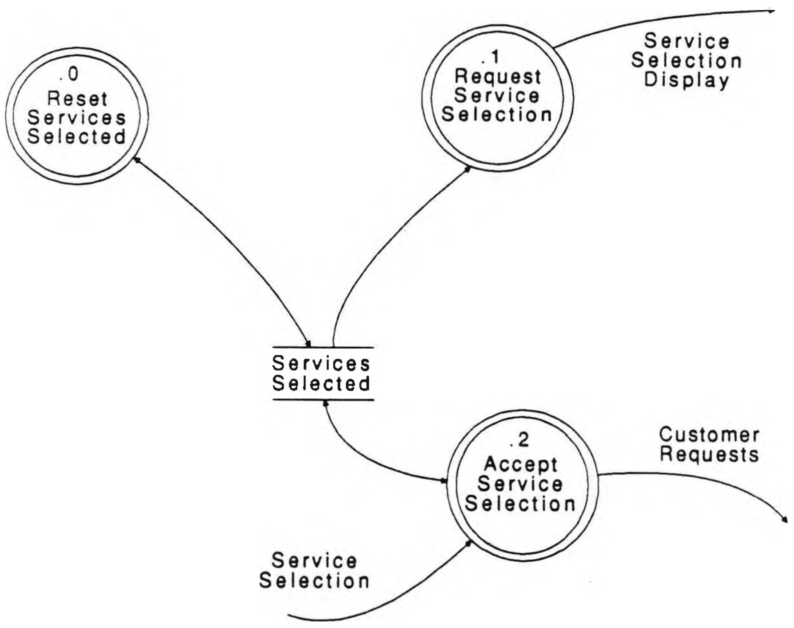


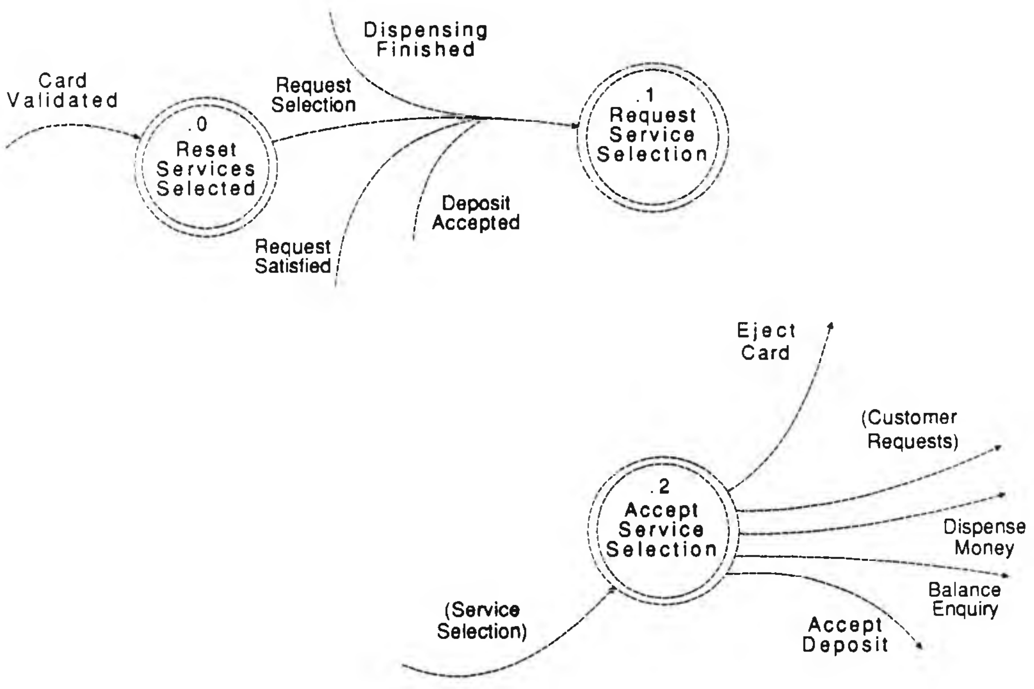
.0 Validate Card



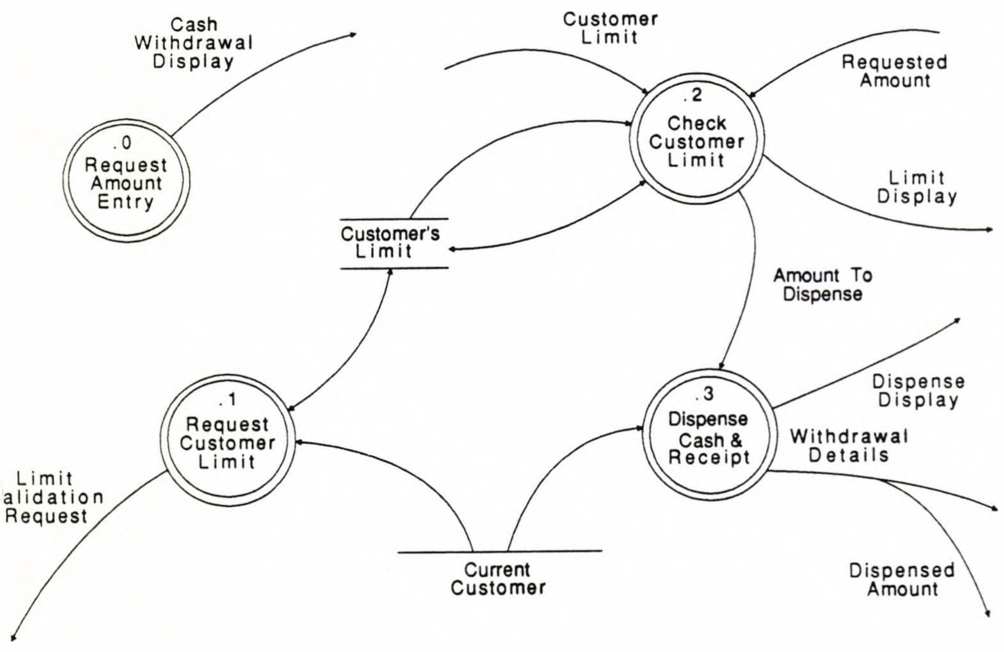


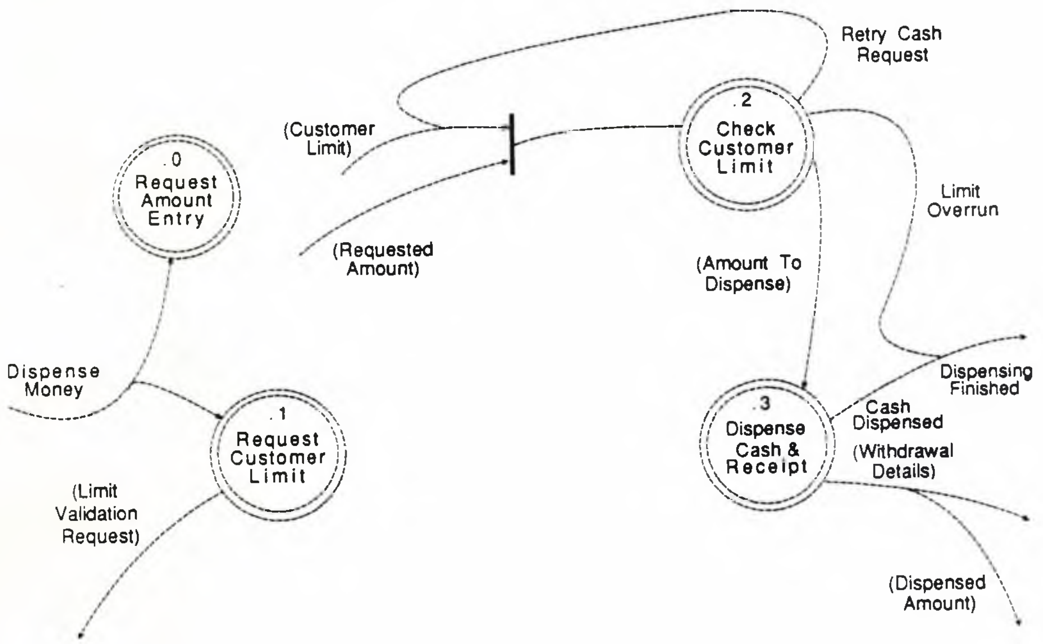
.0.0 Validate Personal Id



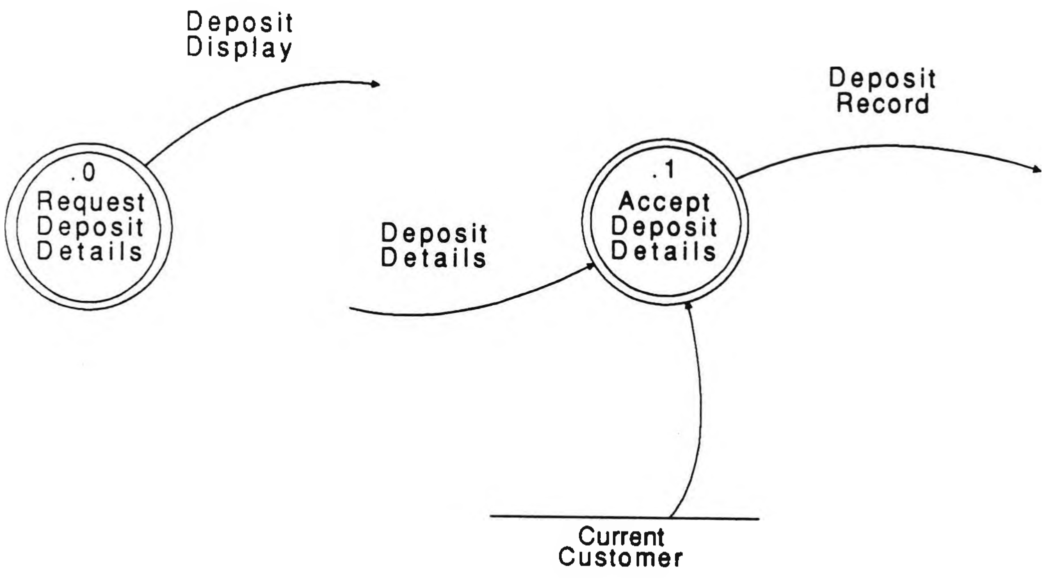


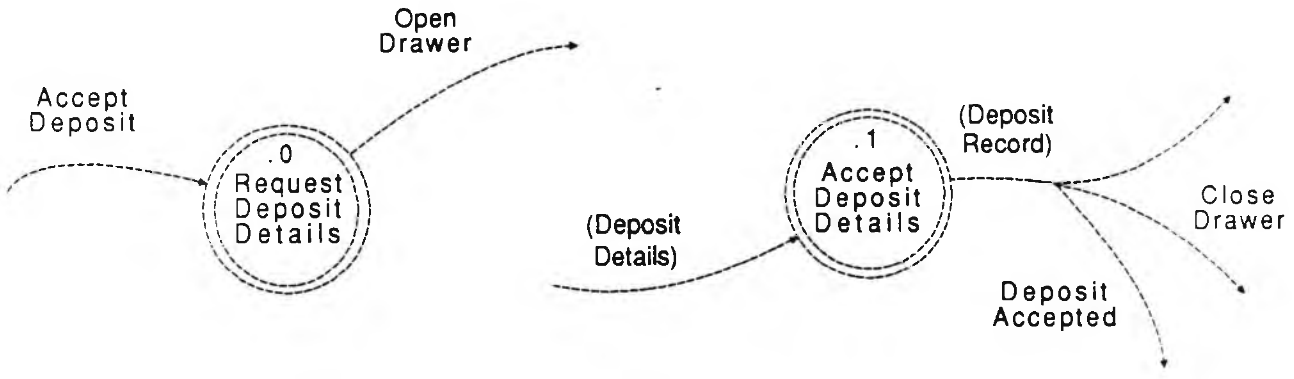
.1 Select Service



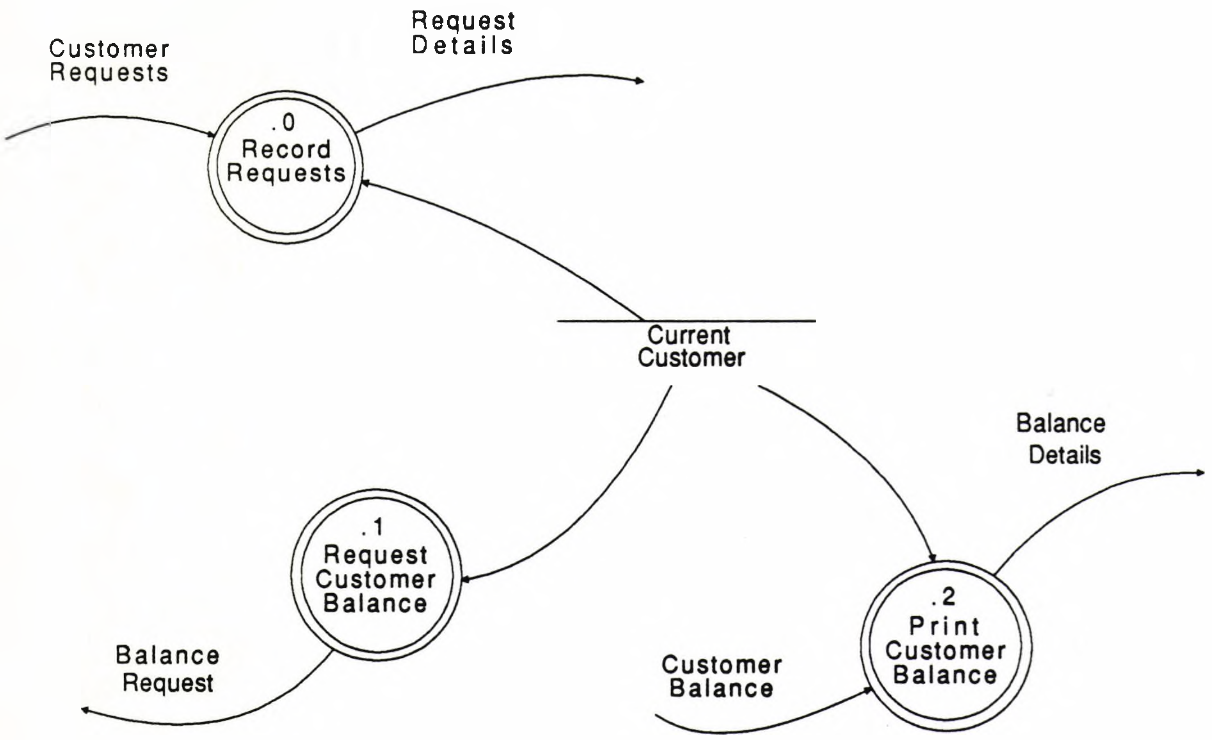


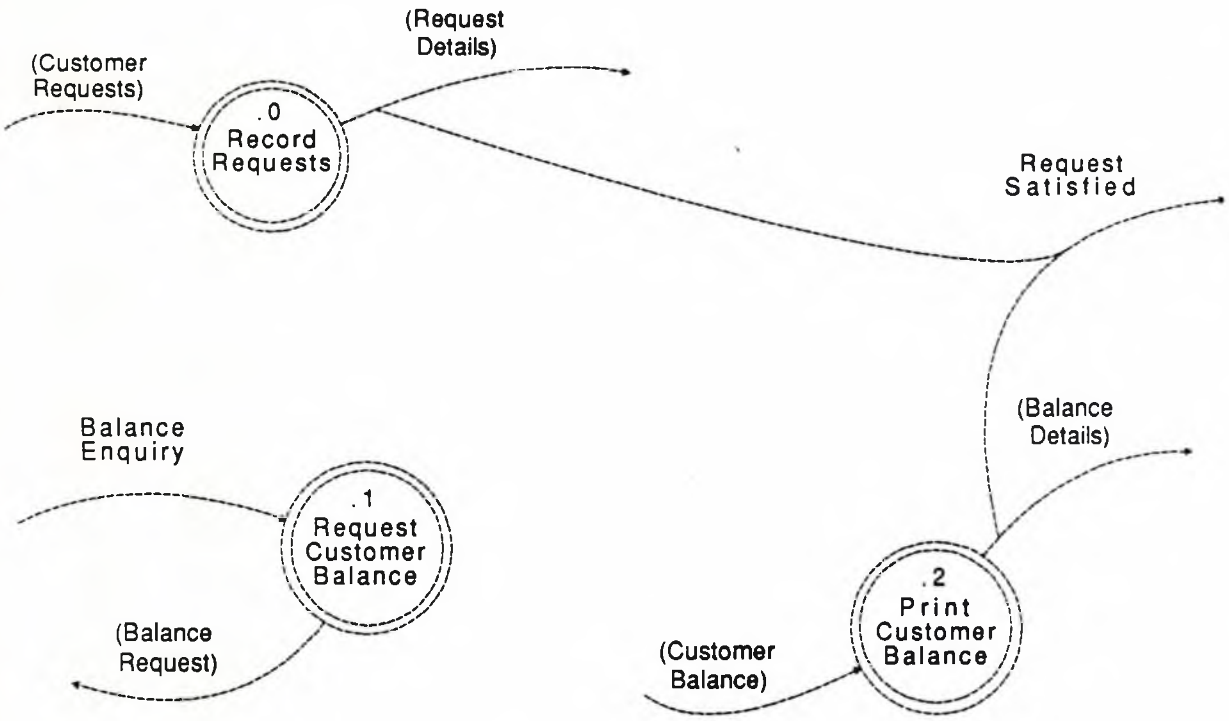
.2 Dispense Cash





.3 Receive Deposits





.4 Receive Requests

A.7 The Defect Inspection System

The purpose of the defect inspection system[6] is to chop rolls of metal foil into sheets and sort the sheets into two bins according to a preselected product standard. Those that meet the standard go into one bin. Those that do not go into another.

The system is run by a supervisor and a number of operators. The supervisor is responsible for the overall running of the system, including selecting product standards, configuring each of the production surfaces, and selecting sheet sizes.

The production surfaces are monitored by operators. They can start and stop production surfaces. They are also responsible for wheeling out full bins and replacing them with empty ones.

Each configuration surface is equipped with a scanner, a chopper, and two air jets. Any configuration of this equipment is workable, so long as both air jets follow the chopper. The supervisor tells the system which configuration has been set on each surface.

The scanner operates by reading the amount of light reflected from the foil. A large percentage of the reflected light for the squares scanned by the scanner must be between certain values, as defined by the product standard, for the foil to be deemed "good", otherwise the sheet must be rejected as "bad". Irregularities in the foil will tend to produce values outside the specified range. The scanner returns data for each of the squares by organising what it "sees" into lanes that run perpendicular to the foil's travelling direction. Data is produced for each square in the lane, preceded by the lane numbers.

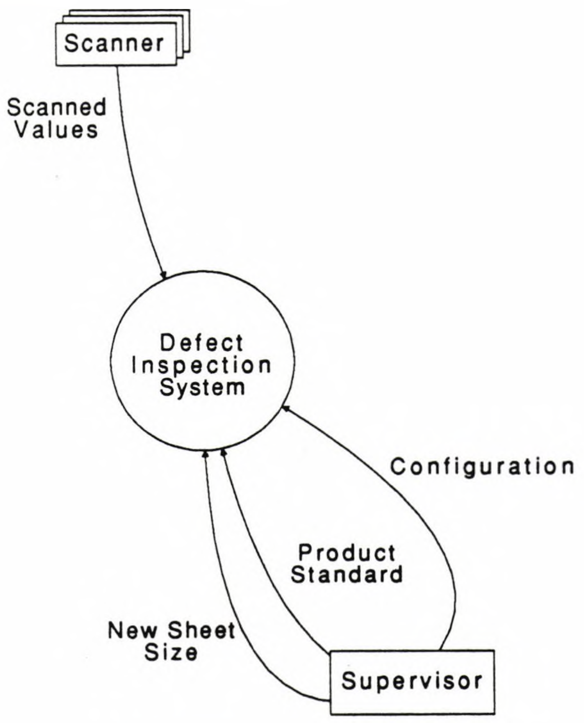
A chopper for each surface can be commanded to drop, thus cutting the foil into sheets. The chopper raises itself automatically once it has chopped the foil. The chopper must be controlled to chop the foil into sheets of constant size for a particular run. The foil may be chopped before it is scanned.

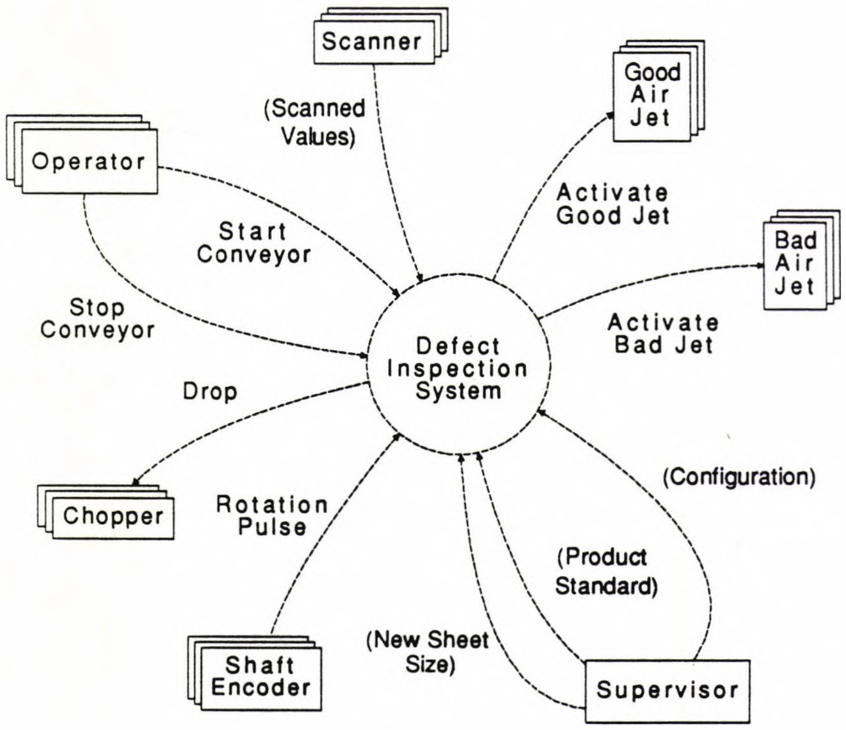
There are two air jets: one pushes the foil to the left, the other to the right. By custom good foil is thrown to the right.

The foil is moved along the production surface by a conveyor belt system that can be started and stopped by the operator (to start or stop the surface is, in fact, to start or stop the conveyor system). A shaft encoder is connected to the drive roll in the belt system. Each quarter of revolution of the drive roll will produce a pulse from the shaft encoder. The resolution of the system is sufficient to be able to cut sheets to lengths measured in units of shaft encoder pulses.

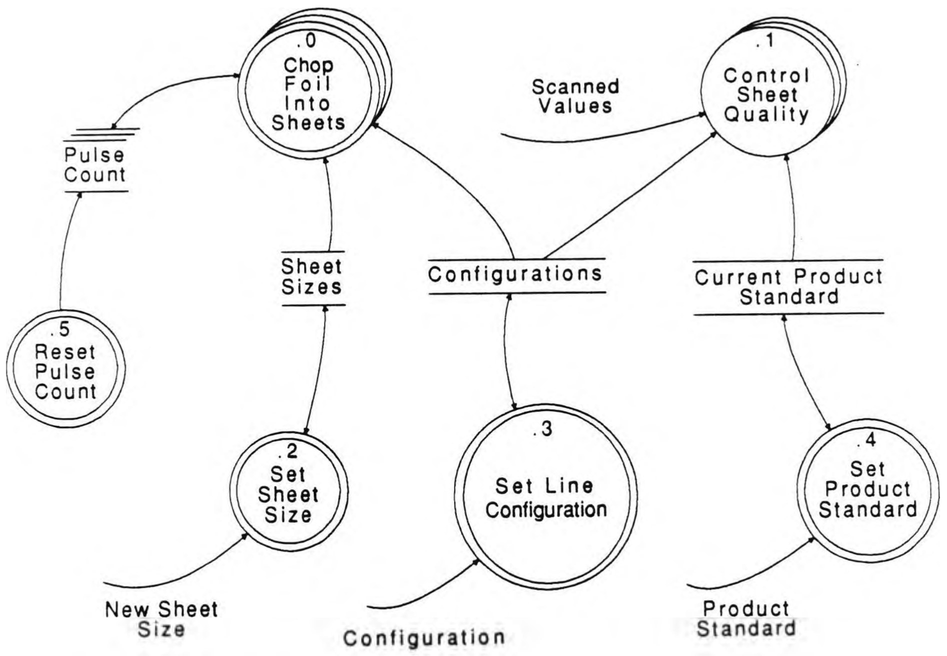
[6]This exercise was adapted from the Defect Inspection System in [WM86].

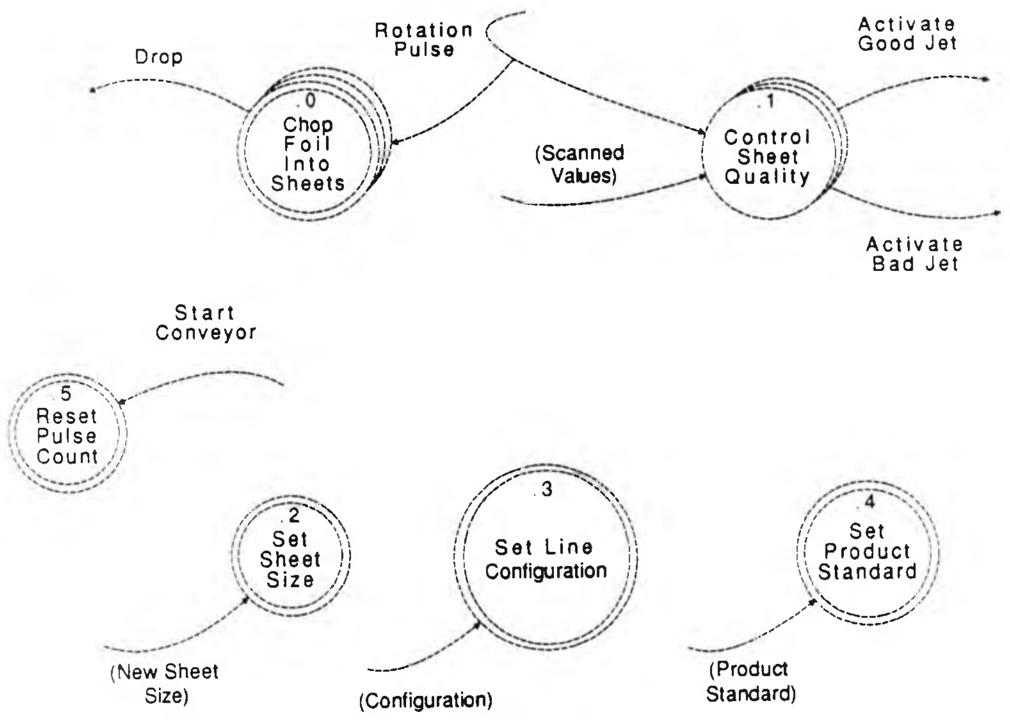
A.7.1 Specification Diagrams



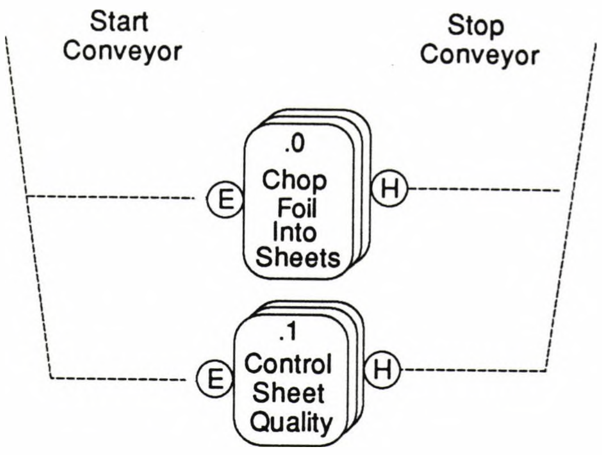


Defect Inspection System: Context Diagram

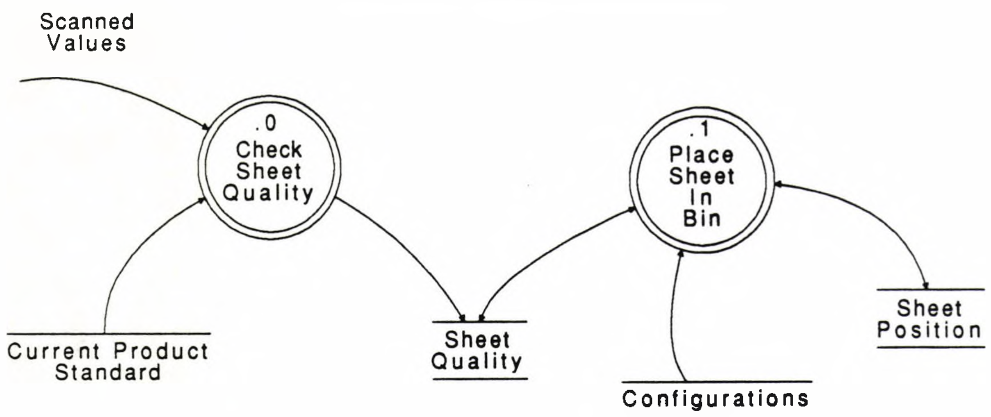


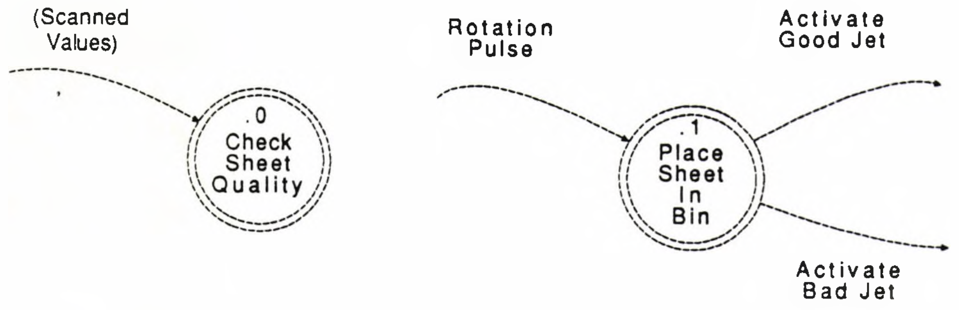


Defect Inspection System (DFD/EFD)



Defect Inspection System (SCD)





.1 Control Sheet Quality

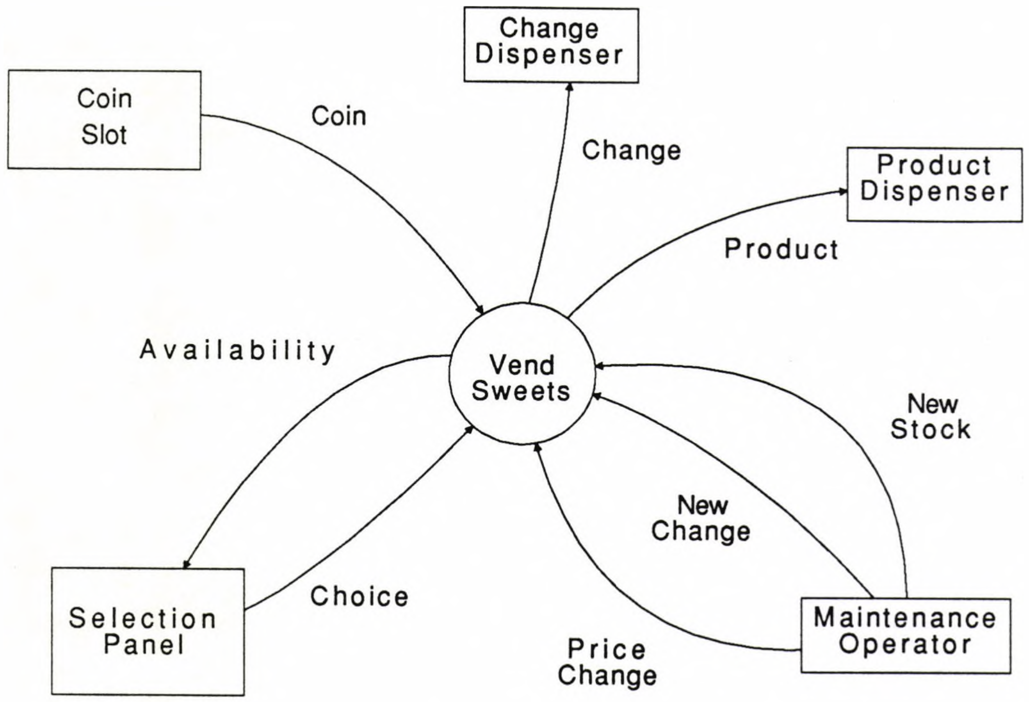
A.8 The Vending Machine

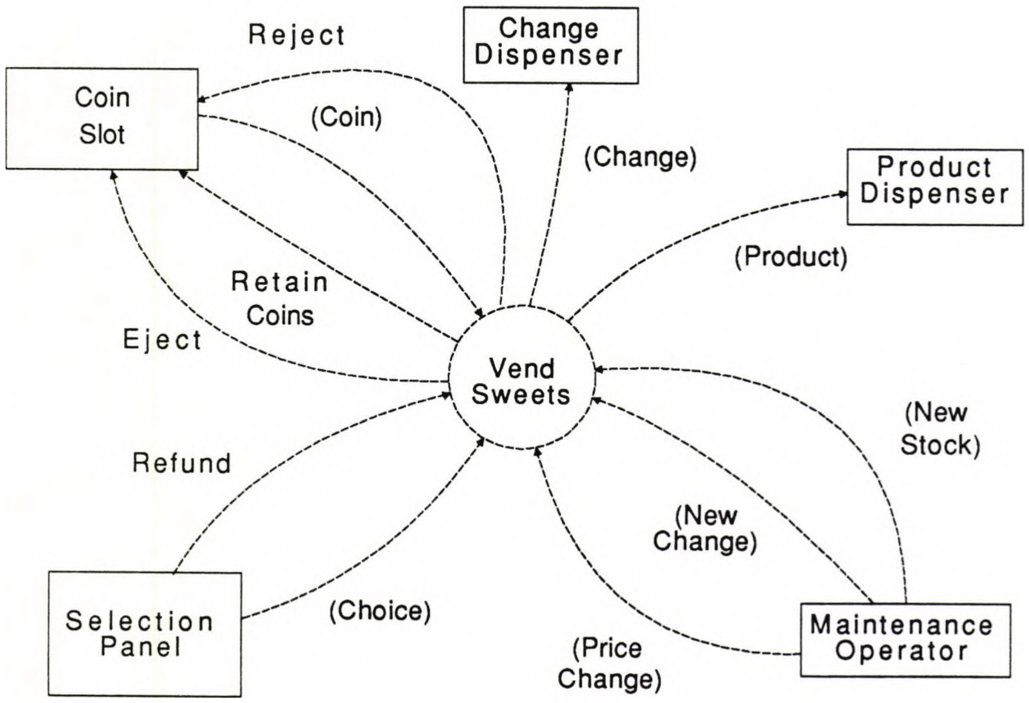
A vending machine[7] offers two kinds of product: chocolate and toffee. When the customer inserts coins, each coin is first validated. The customer then makes a choice, and provided (s)he has inserted sufficient coins, his/her choice is dispensed. The machine will return change if necessary.

Unrecognised coins are rejected. When the change dispenser runs out of change, the customer is notified. Similarly, the machine will notify lack of stock. Lights on the display panel indicate lack of product and change. Large coins are rejected when no change is available. All coins are ejected when a product is not available. Finally, a customer may request his/her money back without making a choice. A maintenance operator regularly visits the machine to refill it with products and change, and to empty the previous payments.

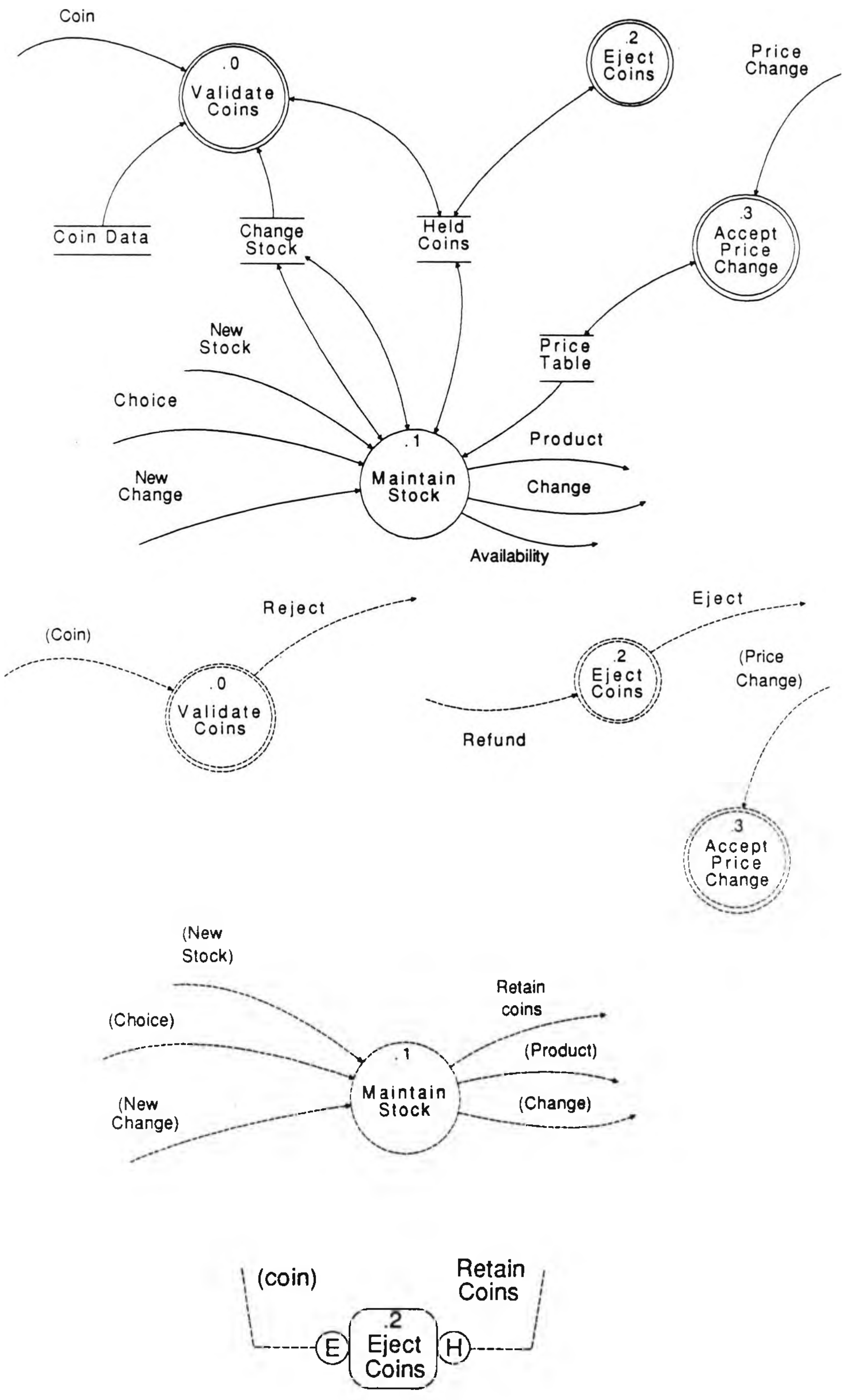
[7]This exercise is adapted from one of the examples in [Hoa85]

A.8.1 Specification Diagrams

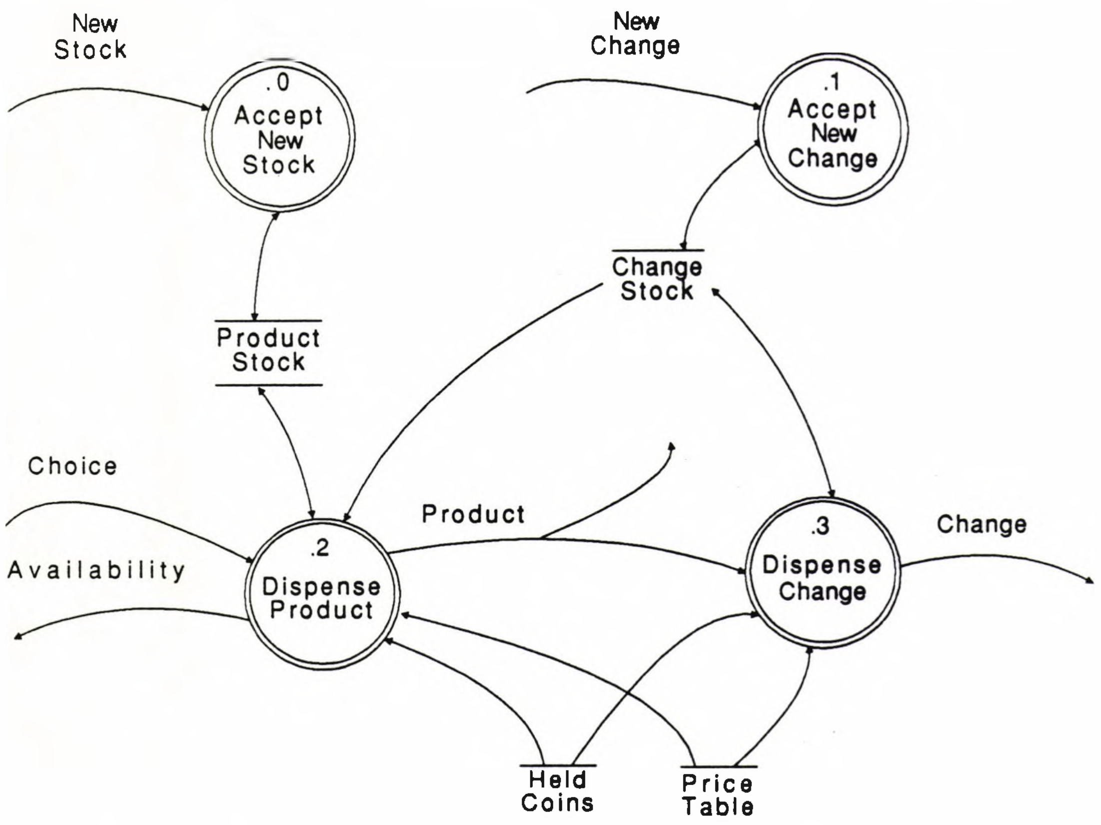


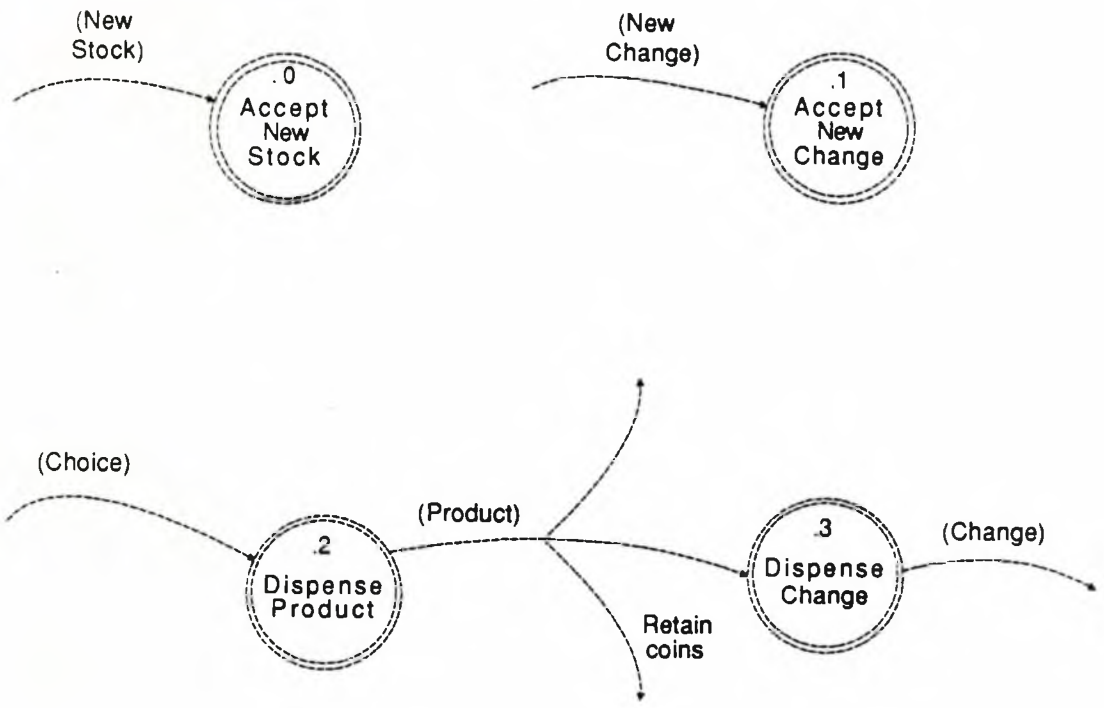


Vend Sweets: Context Diagram


Coin
.0 Validate Coins
.2 Eject Coins
Price Change
Coin Data
Change Stock
Held Coins
.3 Accept Price Change
New Stock
Price Table
Choice
.1 Maintain Stock
Product
New Change
Change
Availability
Reject
Eject
(Coin)
.0 Validate Coins
.2 Eject Coins
(Price Change)
Refund
.3 Accept Price Change
(New Stock)
Retain coins
(Choice)
(Product)
.1 Maintain Stock
(New Change)
(Change)
(coin)
Retain Coins
E
.2 Eject Coins
H

Vend Sweets





.1 Maintain Stock

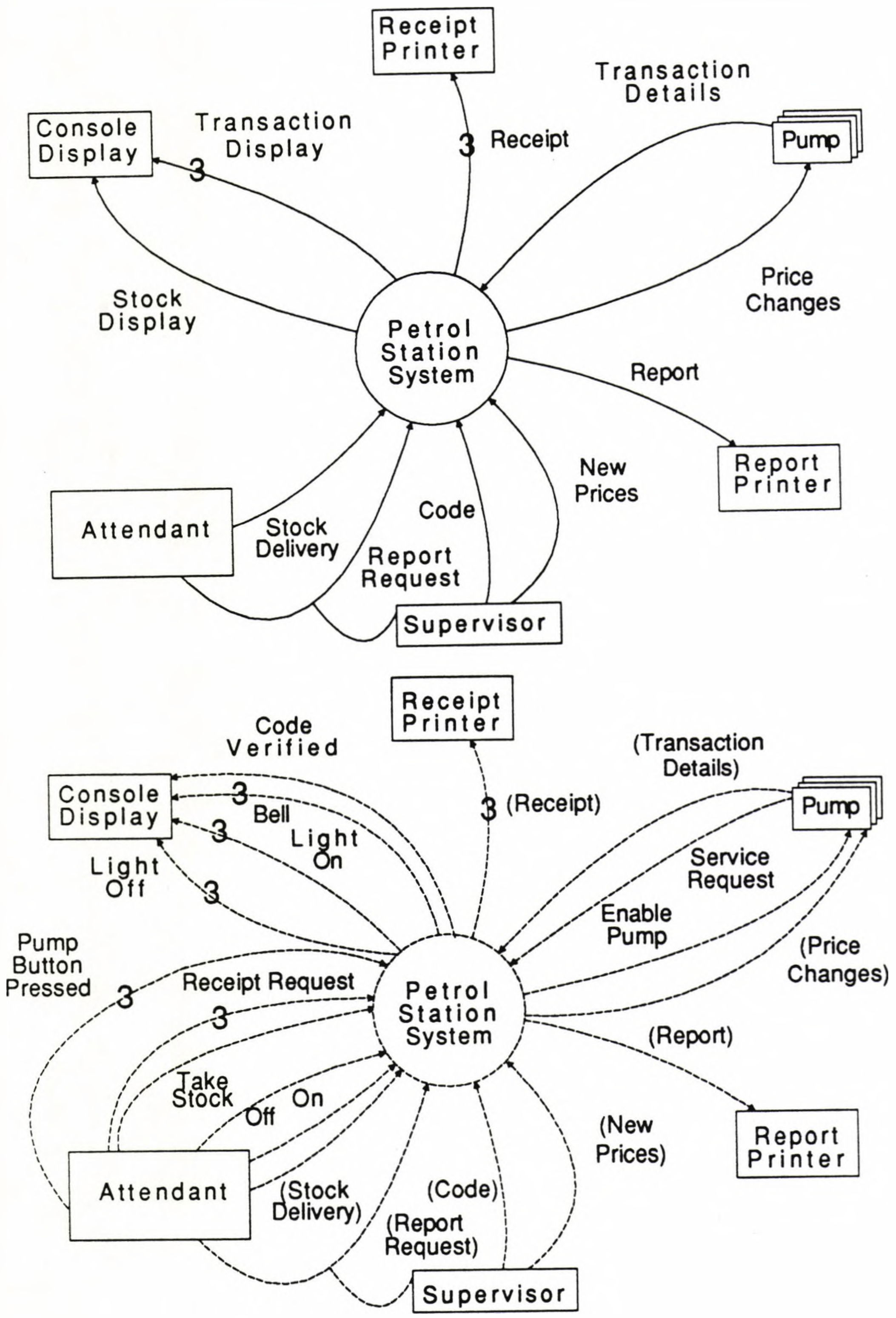
Appendix B

Design And Implementation

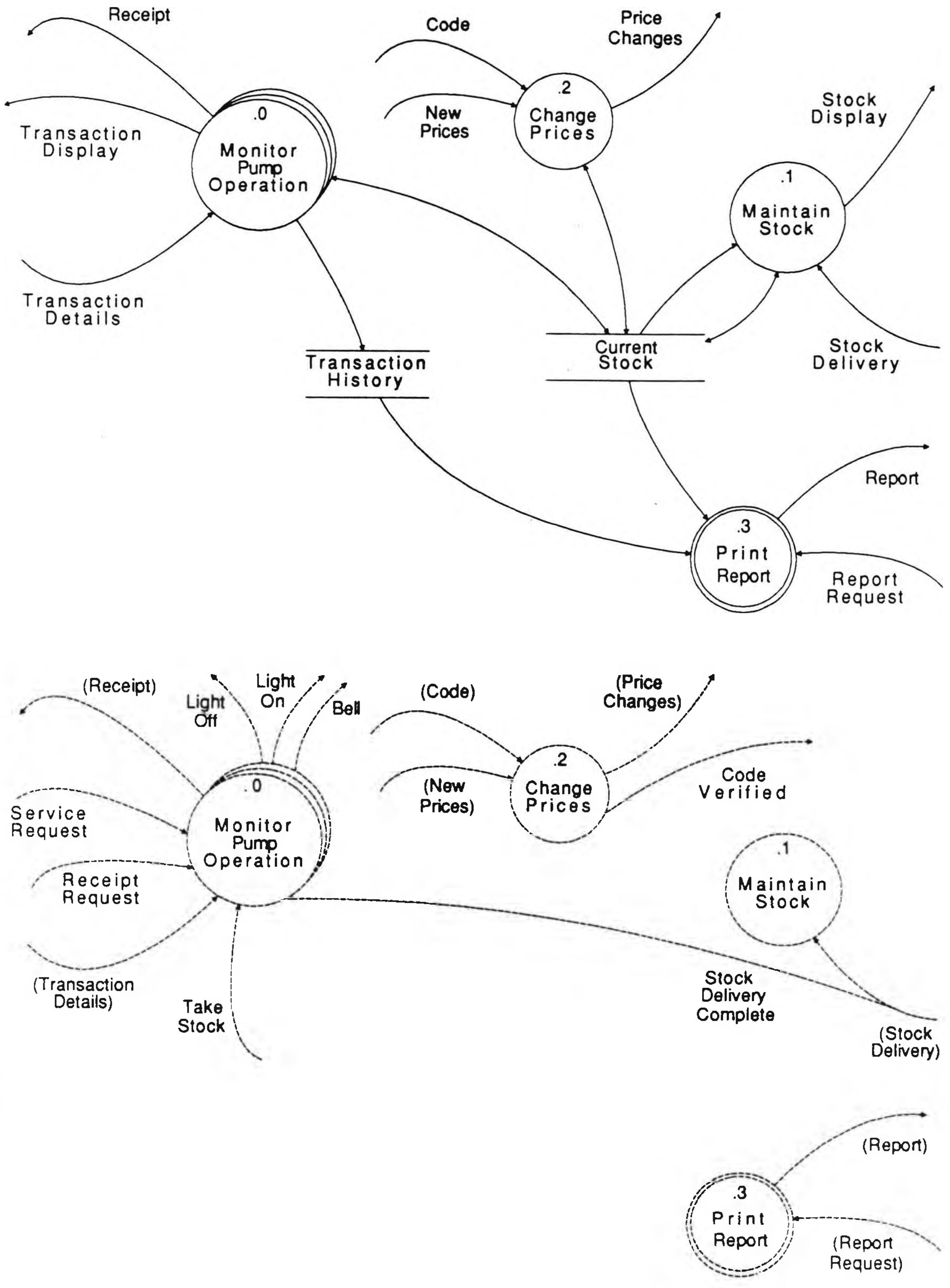
B.1 Overview

Chapter 4 included a discussion of the transition from a specification to design and implementation based on the experiences gained by implementing the Petrol Station example. The full specification and the implementation code for that system are given in this Appendix. The specification diagrams are repeated here for easy reference.

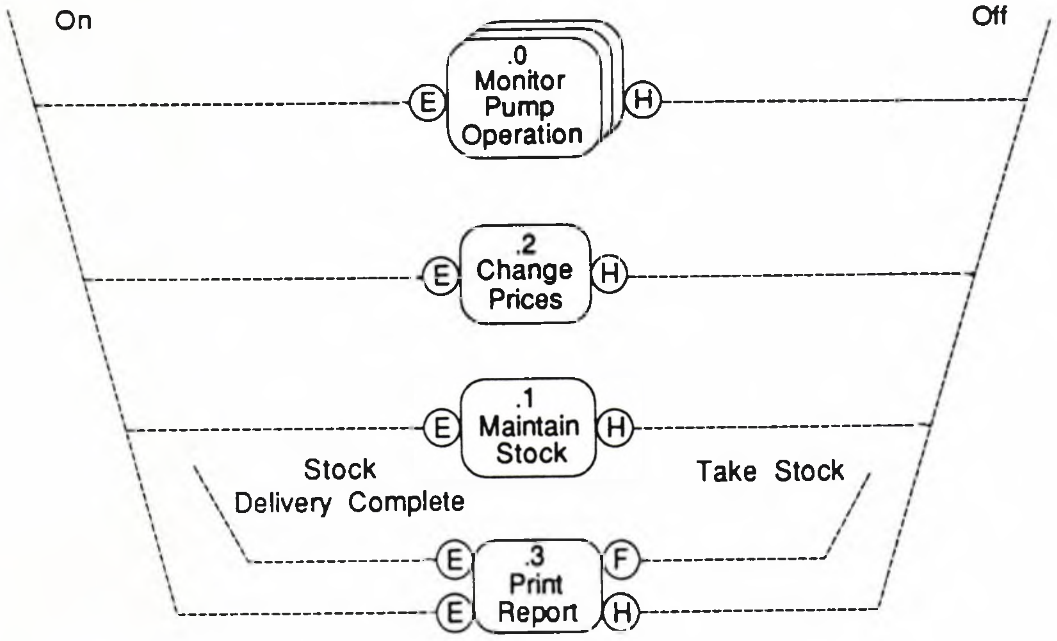
B.2 Specification of The petrol Station



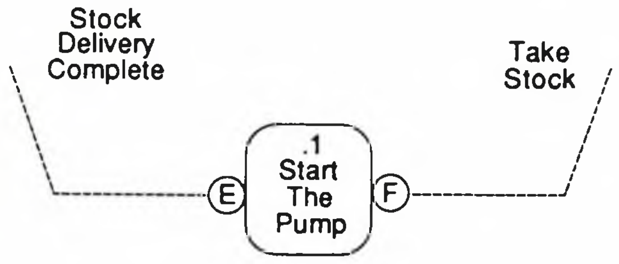
Petrol Station System: Context Diagram



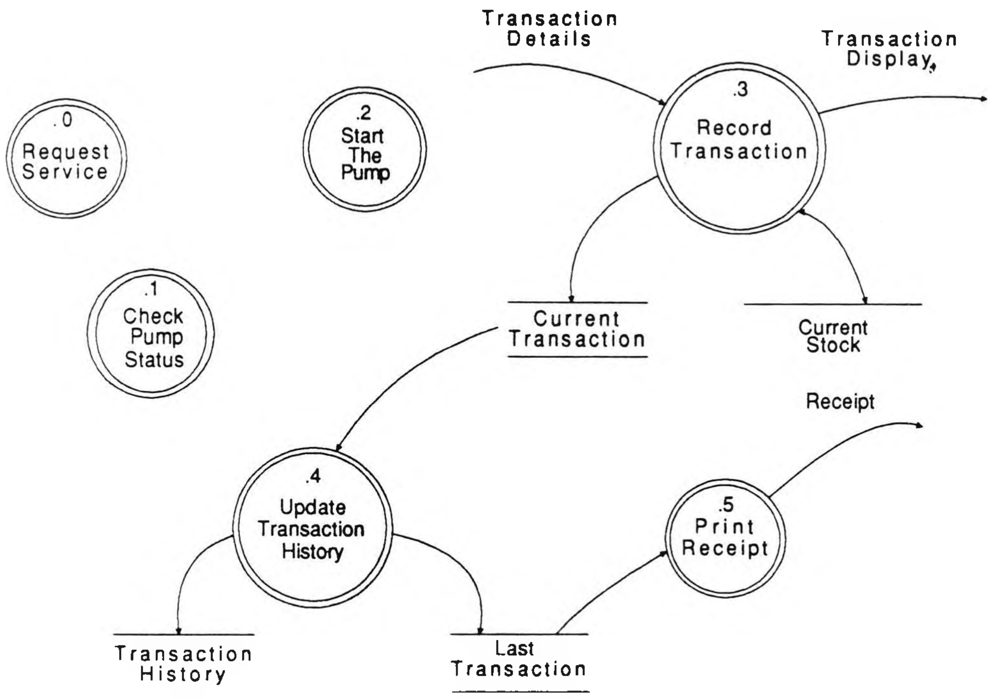
Petrol Station System (DFD/EFD)

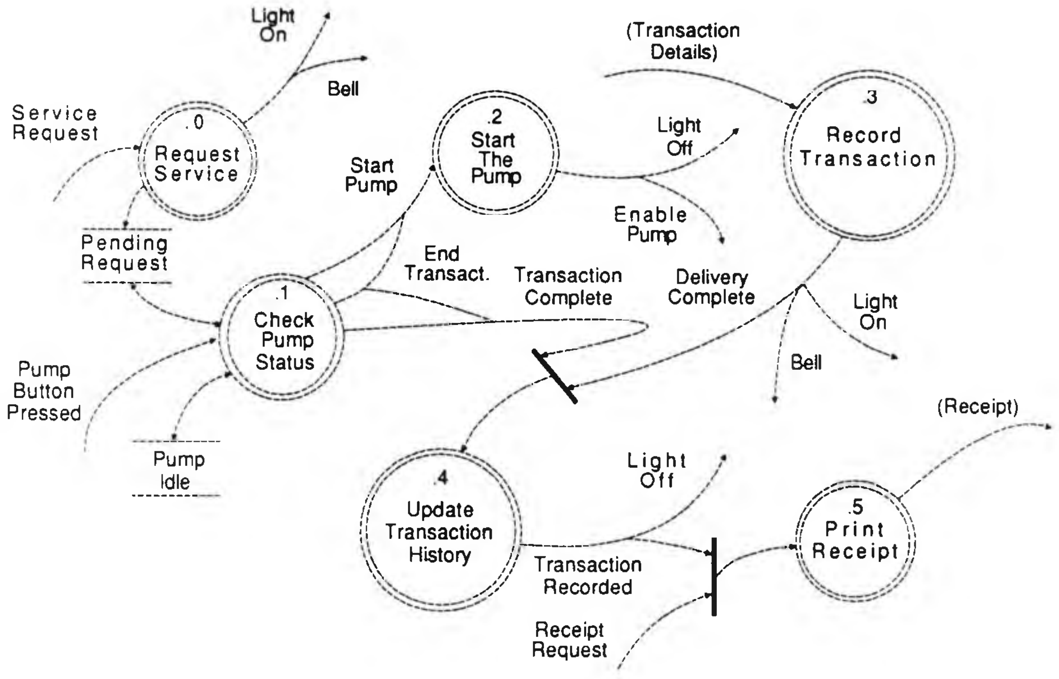


Petrol Station System(SCD)

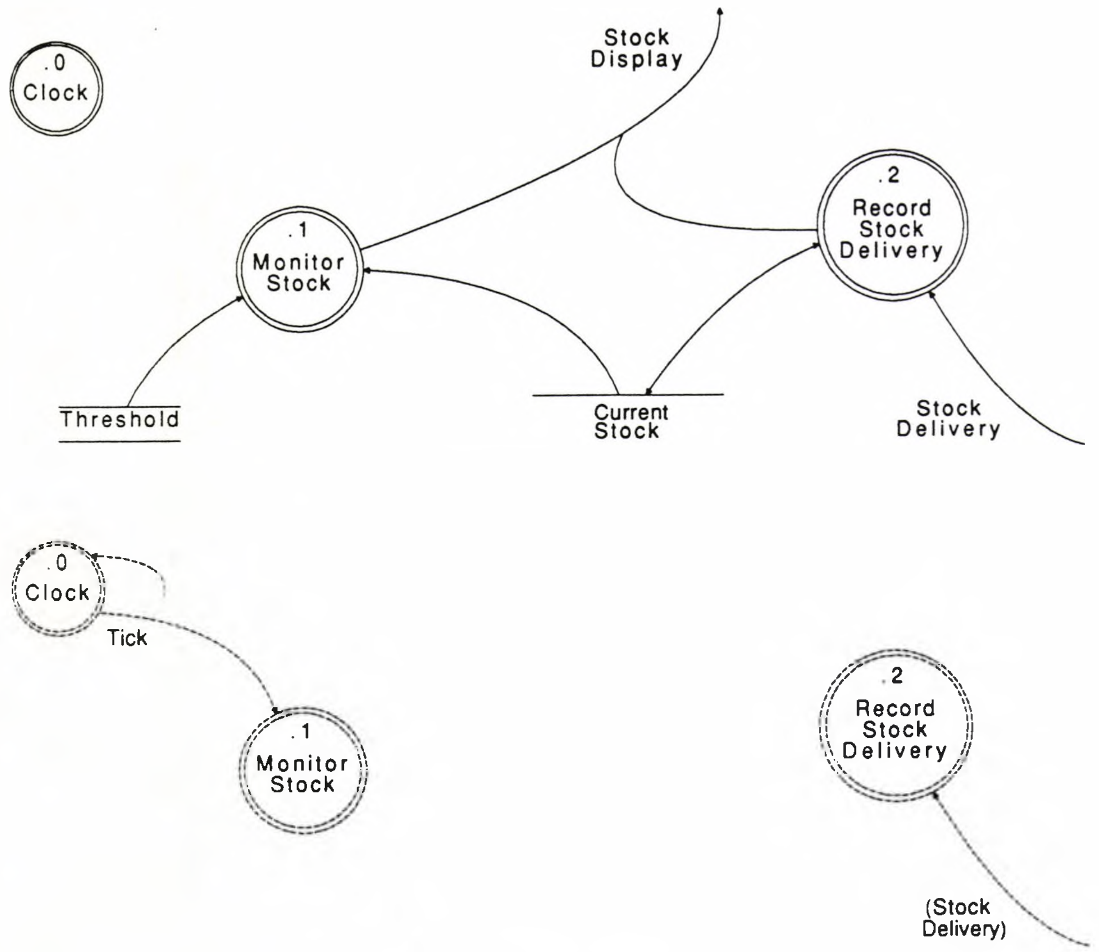


.0 Monitor Pump Operation(SCD)

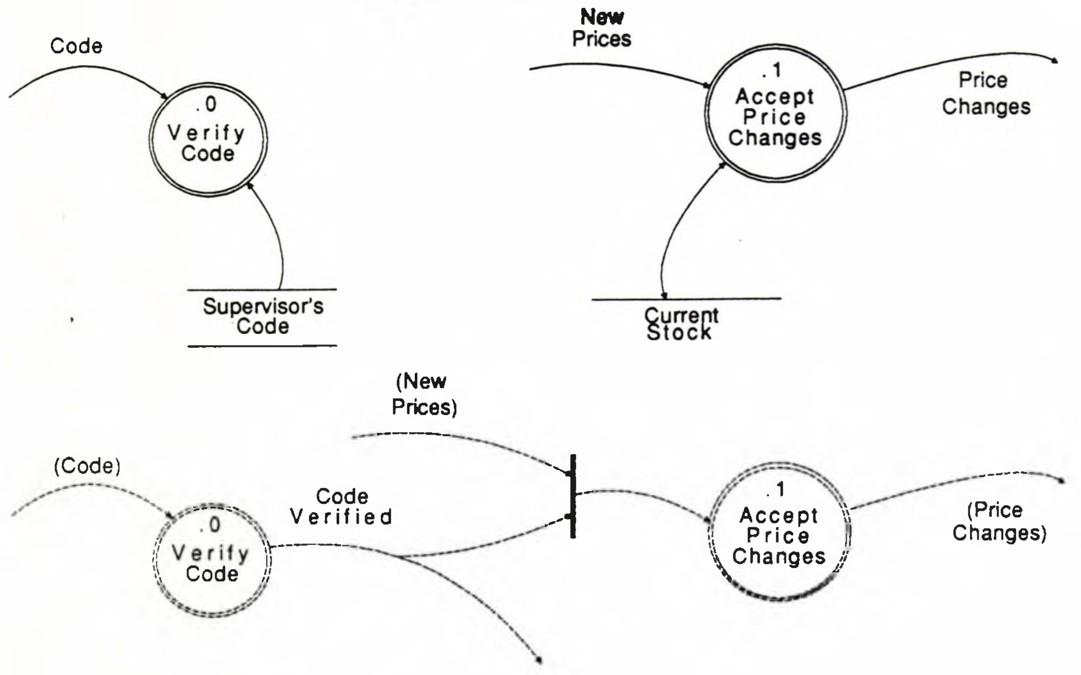




.0 Monitor Pump Operation (DFD/EFD)



.1 Maintain Stock



.2 Change Prices

B.2.1 The Data Dictionary

This section includes the data dictionary for the Petrol Station System. The entries follow the conventions of DeMarco style data dictionaries [DeM78].

Data Flows

Amount Delivered = Number

Code = String

Decimal Number = *

Full Report = Transaction Report + Stock Report

Full Report Request = Decimal Number

Grade Price = Number

Grade Stock = Number

Litres Delivered = Number

New Prices = {Petrol Grade + New Price}

Number = *

Petrol Grade = String

Price Changes = {Petrol Grade + New Price}

Price Paid = Number

Price To Pay = Number

Pump Id = Decimal Number

Receipt = Petrol Grade + Grade Price + Litres Delivered + Total Paid

Report Request = [Transaction Report Request | Stock Report Request | Full Report Request]

Report = [Transaction Report | Stock Report | Full Report]

Service Request Display = Pump Id + Bell

Signal = *

Stock Delivery = {Petrol Grade + Amount Delivered}

Stock Display = {Petrol Grade + Grade Price + Grade Stock + (Warning)}

Stock Report = {Petrol Grade + Grade Price + Grade Stock}

Stock Report Request = Decimal Number

String = *

Total Paid = Number

Transaction Details = Pump Id + Petrol Grade + Litres Delivered

Transaction Display = Pump Id + Petrol Grade + Litres Delivered + Price To Pay

Transaction Report = { Pump Id + Petrol Grade + Litres Delivered + Price Paid}

Transaction Report Request = Decimal Number

Warning = Signal

Data Stores

Current Stock = Petrol Grade + Grade Price + Grade Stock

Current Transaction = Pump Id + Petrol Grade + litres Delivered + Price To Pay

Last Transaction = Pump Id + Petrol Grade + litres Delivered + Price To Pay

Supervisor's Code = String # supervisor's security code #

Threshold = Number # Threshold for stock warning #

Transaction History = Pump Id + Petrol Grade + Litres Delivered + Price Paid

B.2.2 The Event Dictionary

This section gives the event dictionary for the Petrol Station System. The only dialogue sequence in the example system has been indicated by using the '$' symbol followed by a number, i.e. $1 is the first event in the dialogue sequence and $2 is the second.

Event Flows

Bell	# Sound the console bell #
Code Verified	# Code validation notification #
Delivery Complete	# Pump petrol delivery completion, $5 #
Enable Pump	# Enable a pump, $3 #
End Transact.	# End delivery and enable pump #
Light Off	# Turn off a pump's lights #
Light On	# Turn on a pump's lights #
Off	# System shutdown event #
On	# System powerup event #
Pump Button Pressed	# A pump button is pressed, $1 and $5#
Receipt Request	# Receipt is requested for a pump #
Service Request	# Service is requested by a pump #
Start Pump	# Start a a petrol delivery, $2 #
Stock Delivery Complete	# Alias for arrival of "Stock Delivery" #
Take Stock	# Attendant signals arrival of delivery tanker #
Tick	# Clock Tick #
Transaction Complete	# End a petrol delivery, $6 #
(Transaction Details)	# Pump sends transaction details, $4 #
Transaction Recorded	# Transaction database updated, $7 #

Event Stores

Pending Request	# Flag to indicate pending service requests #
Pump Idle	# Pump status flag #

B.2.3 Minispecifications of Atomic Processes

The following section gives the minispecs for the atomic processes of the Petrol Station System. The description of each atomic process is given in a pseudo code style, and is meant to be self explanatory. The description starts with the name of the process. Its digit string identifier, together with its firing event, follow in a C style comment (enclosed in /* and */). The body of the minispec is enclosed in a pair of braces; so are collections of statements within the body. The names of data flows, event flows, and fields of data stores and data flows are enclosed in speech marks. Local storage variables are given names in capital letters, and output to data flows is indicated by the "− >" symbol. The '+' symbol has been used to indicate both the addition of two numbers and the addition of a field to a record. The context should make the interpretation clear.

```
Accept Price Changes
/* .2.1: starts when "New Prices" has arrived and "code Verified" */
{
       For every record in "New Prices"
       {
              Get corresponding record from "Current Stock";
              "Grade Price" = "New Price";
              Put record to "Current Stock";
       }
       "New Prices" -> "Price Changes"
}

Check Pump Status
/* .0.1: fired by "Pump Button Pressed" */
{
       If ("Pump Idle") then
       {
              "Start Pump";
              Reset "Pump Idle";
       }
       Else
       {
              If ("Pending Request") then
              {
                     "End Transact." ;
                     Reset "Pending Request";
              }
              Else
              {
                     "Transaction Complete";
                     Set "Pump Idle";
              }
       }
```

```
}

Clock
/* .1.0: self perpetuating process */
{
       Every ten seconds "Tick";
}

Start The Pump
/* .0.2: fired by "Start Pump" */
{
       "Enable Pump" and "Light Off";
}

Monitor Stock
/* .1.1: fired by clock "Tick" */
{
       For every record in "Current Stock"
       {
              If("Grade Stock" < "Threshold")
                     SD = record + "Warning";
              Else
                     SD = record;
              SD -> "Stock Display";
       }
}

Print Receipt
/* .0.5: fired by "Transaction Recorded" and "Receipt Request" */
{
       Get "Last Transaction" record;
       record -> "Receipt";
}

Print Report
/* .3: starts when "Report Request" arrives */
{
        Case "Report Request" of
        {
              "Transaction Report Request" : Tr_Report;
              "Stock Report Request" : St_Report;
              "Full Report" : Tr_Report; St_Report;
       }
}
Tr_Report
/* Sub-procedure of .3 */
{
       For every record in "Transaction History"
```

```
              record -> "Report";
}
St_Report
/* Sub-procedure of .3 */
{
       For Every record in "Current Stock"
              record -> "Report";
}

Record Stock Delivery
/* .1.2: starts when "Stock Delivery" arrives */
{
       For every record in "Stock Delivery"
       {
           Get corresponding record from "Current Stock";
           "Grade Stock" = "Grade Stock" + "Amount Delivered";
           Put record to "Current Stock";
       }
}

Record Transaction
/* .0.3: starts when "Transaction Details" arrives */
{
       Get grade record from "Current Stock";
       "Grade Stock" = "Grade Stock" - "Litres Delivered";
       Put record to "Current Stock";
       PP = "Grade Price" * "Litres Delivered";
       TD = "Transaction Details" + PP;
       TD -> "Transaction Display";
       Put TD to "Current Transaction";
       "Delivery Complete" and "Bell" and "Light Off";
}

Request Service
/* .0.0: fired by "Service Request" */
{
       "Light On" and "Bell";
       Set "Pending Request";
}

Update Transaction History
/* .0.4: fired by "Transaction Complete" and "Delivery Complete" */
{
       Get record from "Current Transaction";
       Add record to "Transaction History";
       Put record in "Last Transaction";
       "Transaction Complete" and "Light Off";
}
```

```
Verify Code
/* .2.0: starts when "Code" arrives */
{
       Get "Supervisor's Code";
       If("Code" = "Supervisor's Code")
              "Code Verified";
}
```

B.3 Implementation Code

This section gives the full implementation code for the Petrol Station System. Where appropriate references to the diagrams and the minispecs have been given.

Makefile

```
CFLAGS = -O

CFILES = pump.c \
         mon_pump.c \
         mon_stk.c \
         main.c\
         display.c \
         report.c \
         pce_chge.c \
         rec_stk_del.c \
         misc.c

OFILES = pump.o \
         mon_pump.o \
         mon_stk.o \
         main.o\
         display.o \
         report.o \
         pce_chge.o \
         rec_stk_del.o \
         misc.o

XFILES = pssim\
         pump \
         mon_pump

HFILES = pump.h \
         files.h \
         display.h

all: pssim pump mon_pump mon_stk

pssim: main.o display.o report.o pce_chge.o rec_stk_del.o \
       misc.o $(HFILES)
        cc -O -o pssim main.o display.o report.o pce_chge.o  \
                rec_stk_del.o misc.o -lcurses -ltermcap

pump: pump.c pump.h
        cc -O -o pump pump.c

mon_pump: mon_pump.c $(HFILES)
        cc -O -o mon_pump mon_pump.c

mon_stk: mon_stk.c $(HFILES)
        cc -O -o mon_stk mon_stk.c
```

```
pce_chge.o : $(HFILES)

rec_stk_del.o : $(HFILES)

main.o : $(HFILES)

display.o : pump.h display.h

report.o : pump.h files.h

clean:
	rm -f $(OFILES) $(XFILES)
```

```
/* pump.h: header file for pump constant definitions */

#define YES         1
#define NO          0

#define PTM_PIPE    "/tmp/mns/ptm"  /* pump to mon. pipe */
#define PTP_PIPE    "/tmp/mns/ptp"  /* price change to pump */
#define MAX_PIP_NAM  12             /* max. length of pipe name */

#define IDLEN       2       /* length of pump id number */
#define GDLEN       2       /* length of grade id */
#define MAXPRICELEN  6      /* max. length of grade price */
#define MAXSTOCKLEN  8      /* max. length of on grade stock */
#define LTLEN        5      /* max. length of litres delivered */
/* max. message length from pump to monitor */
#define PTM_LEN      IDLEN+GDLEN+LTLEN
#define MAX_PLEN     7      /* max. length of price to pay */
#define MAXGDESTR    10     /* max. length of grade name */

/* maximum length of a price change message */
#define PCECHGLEN    NGRADES*(GDLEN+MAXPRICELEN)

/* max. length of current stock */
#define MAX_CSTK     NGRADES*(GDLEN+MAXPRICELEN+MAXSTOCKLEN+1)

/* max. length of transaction history record */
#define MAX_THRLEN   IDLEN+GDLEN+LTLEN+MAX_PLEN

#define NGRADES      3        /* number of available grades */
#define NPUMPS       3        /* default number of pumps */
#define MAX_PUMPS    9        /* maximum number of pumps (1-9) */
#define MAX_PIDLEN   2        /* maximum pump id (no) length */
```

/* display.h: header file for display definitions */

```
#define MTD_PIPE         "/tmp/mns/mtd"

#define ON        1
#define OFF       0

#define BEEP      007
#define ERASE     008

#define TRANS     0       /* transaction details */
#define RECPT     1       /* receipt */
#define LIGHT     2       /* pump light */
#define WARN      3       /* stock warning */
#define BELL      4       /* console bell */
#define CLRTR     5       /* clear last transaction display */
#define ACTVE     6       /* pump status */

typedef struct w {
                   int disp_type;
                   char buffer[1020];
                  } disp_prot;
```

/* files.h: header file for store names */

```
#define CURRENT_STOCK "/tmp/mns/curr_stock"  /* current stock */
#define TRANS_HIST    "/tmp/mns/trans_hist"  /* trans. hist */
```

/* main.c: main module. Forks all processes, sets up communication pipes, and monitors the keyboard and display pipe */

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <fcntl.h>
#include <unistd.h>
#include <curses.h>
#include <signal.h>
#include <string.h>
#include "pump.h"
#include "display.h"
#include "files.h"

#define forever        for(;;)

#define set_bit(bit, number)        number |= (1 << bit)
#define isset_bit(bit, number)      (number & (1 << bit))

static void get_choice(),
            terminate(),
            crt_pump_proc();

static int create_procs();

void get_gname(),
     display(),
     init_display(),
     verify_code(),
     record_stk_del();

void main(argc, argv) int argc;
                      char *argv[];
{
   int dpfd, pumps, nfds, rfds, i, smpid,
       monitors[MAX_PUMPS],
       pumpstats[MAX_PUMPS],
       error = NO;
   char command[70];

   if(argc < 2)                /* set the no of station pumps */
      pumps = NPUMPS;
   else
      if((pumps = atoi(argv[1])) < 1 || pumps > 9)
      {
         printf("Illegal number of pumps\n", argv[0]);
         printf("Reverting to default\n");
         pumps = NPUMPS;
      }
   /* catch SIGTERM and clean up */
   if(signal(SIGTERM, terminate) < 0)
      perror("termination signal");
```

```
(void)sprintf(command, "rm -f %s %s? %s?", MTD_PIPE
                                         , PTM_PIPE, PTP_PIPE);
system(command);

if(access(CURRENT_STOCK, F_OK) < 0)
{
   error = YES;
   printf("%s: current stock file is missing%c\n", argv[0], BEEP);
}

if(access(TRANS_HIST, F_OK) < 0)
{
   error = YES;
   printf("%s : transaction history file is missing%c\n", argv[0]
                                                    , BEEP);
}

if(error == YES)
   exit(1);

/* create display pipe and open it */
if(mknod(MTD_PIPE, 0666 | S_IFIFO, 0) < 0)
   perror("display pipe mknod");
if((dpfd = open(MTD_PIPE, O_RDWR|O_NDELAY)) < 0)
   perror("display pipe open");

if(setpgrp(0, getpid()) < 0)        /* set process group */
   perror("setgrp");

for(i=0; i<MAX_PUMPS; i++)  /* initialise pump statuses */
   pumpstats[i] = 0;

/* create processes */
smpid = create_procs(pumps, monitors);

init_display(pumps);      /* initialise display */

forever
{
   /* set appropriate fd bits */
   rfds = 0;
   set_bit(0, rfds);        /* include standard input */
   set_bit(dpfd, rfds);        /* include display pipe */

   nfds = select(32, &rfds, (int *)0, (int *)0,
                             (struct timeval *)0);

   switch(nfds)        /* select ready input */
   {
      case 1  : if(isset_bit(dpfd, rfds))
                   display(dpfd, pumps);
                 else
                   get_choice(pumps, monitors, pumpstats, smpid);
```

```
                     break;

          case 2  : get_choice(pumps, monitors, pumpstats, smpid);
                    display(dpfd, pumps);
                    break;

          default : perror("select");
                    break;
       }
   }
}

static void get_choice(pumps, monpids, pumpstats, smpid) int pumps,
                                                             monpids[],
                                                             pumpstats[],
                                                             smpid;
{
   char ch;
   static int last_pump = -1,  /* last completed trans. */
              delivery = NO;
   int index, i;

   ch = getch();
   mvaddch(pumps+11, 52, ch);    /* print ch */
   move(22, 40);                 /* clear precious warnings */
   clrtoeol();
   refresh();

   if(ch >= '1' && ch <= pumps+'0')
   {
      index = ch-'0'-1;         /* work out pump's table index */
      pumpstats[index] ^= 1;            /* change pump status */
      if(pumpstats[index] == 0)         /* update last_pump */
         last_pump = index;

      kill(monpids[index], SIGUSR2); /* send button to monitor */
   }
   else
   {
      switch(ch)
      {
         case 'r' :
         case 'R' : if(last_pump != -1)   /* catch errors */
                       kill(monpids[last_pump], SIGFPE);
                    break;

         case 'p' :
         case 'P' : if(delivery == NO)
                    {
                       pr_report();
                       touchwin(stdscr);
                    }
                    else
                       mvaddstr(22, 40 , "No reports during delivery");
                    break;

        case 'd' :
        case 'D' : if(delivery == NO)
                   {
                      for(i=0; i < pumps; i++)
                         if(kill(monpids[i], SIGTERM) < 0)
                            perror("disablement kill");
                      delivery = YES;
                   }
                   mvaddstr(21, 40, "Pumps disabled");
                   break;

        case 's' :
        case 'S' : if(delivery == YES)
                   {
                      record_stk_del();
                      /* inform stock monitor and pumps*/
                      if(kill(smpid, SIGUSR1) < 0)
                        perror("stock monitor kill");
                      for(i=0; i < pumps; i++)
                         if(kill(monpids[i], SIGTERM) < 0)
                            perror("enablement kill");
                      touchwin(stdscr);
                      /* clear stock warnings */
                      mvaddstr(pumps+6, 0, "                        ");
                      move(21, 40);
                      clrtoeol();
                      delivery = NO;
                   }
                   else
                   {
                      mvaddstr(22, 40, "Invalid choice");
                      addstr(": Pumps not disabled yet");
                   }
                   break;

        case 'c' :
        case 'C' : verify_code(pumps);
                   touchwin(stdscr);
                   break;

        case 'q' :
        case 'Q' : terminate();

        default  : mvaddstr(22, 40, "Invalid choice");
                   break;
      }

   refresh();        /* refresh changes to the screen */
   }
}

static int create_procs(pumps, monpids) int pumps, monpids[];
```

```
{
   int i, pid;
   char pip_name[MAX_PIP_NAM + MAX_PIDLEN + 1];

   if((pid = fork()) < 0)
      perror("stock monitor fork");

   if(pid == 0)
   {
      execl("mon_stk", "mon_stk", 0);
      perror("stock monitor exec");
   }
   else
   {
      for(i=1; i<=pumps; i++)        /* create pump processes */
      {
         /* make up pump pipe name and create it */
         (void)sprintf(pip_name, "%s%d", PTM_PIPE, i);
         if(mknod(pip_name, 0666 | S_IFIFO, 0) < 0)
            perror("pump pipe mknod");

         /* make up price change pipe name and create it */
         (void)sprintf(pip_name, "%s%d", PTP_PIPE, i);
         if(mknod(pip_name, 0666 | S_IFIFO, 0) < 0)
            perror("pump pipe mknod");

         crt_pump_proc(i, monpids); /* create pump procs */
      }

      return(pid);        /* return stock monitor's pid */
   }
}

static void crt_pump_proc(pid, monpids) int pid, monpids[];
{
   char ppid[15], pump_id[MAX_PIDLEN + 1];
   int pump_pid, mon_pid;

   (void)sprintf(pump_id, "%d", pid);    /* make up pump id no */

   if((mon_pid = fork()) < 0)        /* fork monitoring proc. */
      perror("pump monitor fork");
   if(mon_pid == 0)
   {
      if((pump_pid = fork()) < 0)   /* fork pump process */
           perror("pump fork");

      if(pump_pid == 0)
      {
         (void)sprintf(ppid, "%d", getppid()); /* get pump pid */
         execl("pump", "pump", pump_id, ppid, 0);
         perror("pump execl");
      }
      else
      {
          (void)sprintf(ppid, "%d", pump_pid); /* get pump pid */
          execl("mon_pump", "mon_pump", pump_id, ppid, 0);
          perror("pump monitor execl");
      }
   }
   else
      monpids[pid-1] = mon_pid;        /* pids idexing from 0! */
}

static void terminate()
{
   /* clean up and end if interrupted */
   endwin();        /* reset terminal attributes */

   /* kill children processes */
   if(setpgrp(0, 0) < 0)                /* reset process group */
      perror("setgrp");
   if(killpg(getpid(), SIGKILL) < 0)
      perror("killpg");

   system("clear");
   exit(0);
}
```

/* display.c: display module. */

```
#include <curses.h>
#include "pump.h"
#include "display.h"

#define LINE "--------------------------------------------------------"

static void disp_trans(),
            disp_light(),
            disp_recpt(),
            disp_warn(),
            disp_status(),
            clr_trans();

void init_display(pumps) int pumps;
{
   int i, pos;
   char term[10], buttons[20], *cptr, *getenv();

   cptr = getenv("TERM");                 /* get terminal type */
   (void)strcpy(term, cptr);
   setterm(term);                        /* set terminal type */

   initscr();                  /* initialise cureses */

   nonl();                          /* set interative mode */
   cbreak();
   noecho();

   clear();                  /* clear stdscr */

   /* draw pump display screen */
   addstr("Pump No      Petrol Grade      Litres Delivered      Total
Light    Active");
   for(i=1; i<=pumps; i++)
   {
      mvaddch(i+1, 4, (i + '0'));
      mvaddch(i+1, 76, 'N');
   }
   mvaddstr(++i, 0, LINE);

   /* print reciept and warning screens */
   mvaddstr(++i, 5, "Stock Warning Lights");
   mvaddstr(++i, 4, "---------------------");
   mvaddstr(++i, 4, "diesel 4 star unleaded");
   i=i+2;                          /* blank lines */
   mvaddstr(++i, 10, "Receipt");
   mvaddstr(++i, 10, "-------");
   mvaddstr(++i, 5, "MNS Petroleum Ltd.");
   mvaddstr(++i, 5, "Grade");
   mvaddstr(++i, 5, "Price");
   mvaddstr(++i, 5, "Litres");
   mvaddstr(++i, 5, "Total");
   /* print menu */
   pos = pumps + 3;
   (void)sprintf(buttons, "1..%d. Pump Buttons", pumps);
   mvaddstr(pos++, 40, buttons);
   mvaddstr(pos++, 40, "   C. Price Changes");
   mvaddstr(pos++, 40, "   D. Disable Pumps for Delivery");
   mvaddstr(pos++, 40, "   P. Request Report");
   mvaddstr(pos++, 40, "   Q. Turn System Off");
   mvaddstr(pos++, 40, "   R. Request Receipt");
   mvaddstr(pos++, 40, "   S. Stock Delivery Details");
   mvaddstr(++pos, 40, "Selection: ");

   refresh();                /* put stdscr on stdout */
}

void display(dpfd, pumps) int dpfd, pumps;
{
   disp_prot ws;

   if(read(dpfd, &ws, sizeof(disp_prot)) < sizeof(disp_prot))
      perror("display pipe read");

   switch(ws.disp_type)
   {
      case TRANS : disp_trans(ws.buffer);
                   break;
      case RECPT : disp_recpt(ws.buffer, pumps);
                   break;
      case LIGHT : disp_light(ws.buffer);
                   break;
      case WARN  : disp_warn(ws.buffer[0], pumps);
                   break;
      case BELL  : mvaddch(23, 40, BEEP);
                   break;
      case CLRTR : clr_trans(ws.buffer);
                   break;
      case ACTVE : disp_status(ws.buffer);
                   break;
      default      : printf("Display protocol error %d\n", ws.disp_type);
                   break;
   }

   refresh();                /* show the changes on the screen */
}

static void disp_trans(transaction) char *transaction;
{
   int pid, grade;
   char litres[LTLEN + 1],
        cost[MAX_PLEN + 1],
        gradestr[MAXGDESTR + 1];

   /* split transaction into its parts */
   (void)sscanf(transaction, "%d %d %s %s", &pid, &grade, litres, cost);

   get_gname(grade, gradestr);
   mvaddstr(pid+1, 14, gradestr);
   mvaddstr(pid+1, 36, litres);  /* display litres delivered */
   mvaddch(pid+1, 53, '$');
   mvaddstr(pid+1, 54, cost);        /* display total to pay */
}

static void disp_light(string) char *string;
{
   int pid, status;
   char ch;

   (void)sscanf(string, "%d %d", &pid, &status);
   if(status == ON)
      ch = '@';
   else
      ch = ' ';

   mvaddch(pid+1, 67, ch);
}

static void disp_warn(grade, pumps) char grade;
                                    int pumps;
{
   int pos=4;

   switch(grade)
   {
      case '0' : pos += 3;
                 break;
      case '1' : pos += 9;
                 break;
      case '2' : pos += 17;
                 break;
   }
   mvaddch(pumps + 4 + 2, pos, '@');
}

static void disp_recpt(receipt, pumps) char *receipt;
                                       int pumps;
{
   int pos, hpos, grade;
   float gde_price, atof();
   char price[MAXPRICELEN + 1],
        litres[MAX_PLEN + 1],
        cost[MAX_PLEN + 1],
        gradestr[MAXPRICELEN + 1];

   (void)sscanf(receipt, "%d %f %s", &grade, &gde_price, litres);

   pos = pumps + 4 + 7;                /* find correct line */
   get_gname(grade, gradestr);         /* print grade */
   hpos = 23 - strlen(gradestr);
   mvaddstr(pos++, hpos, gradestr);

   sprintf(price, "%.2f p", gde_price);        /* print price */
   hpos = MAXPRICELEN - strlen(price);
   mvaddstr(pos++, 17+hpos, price);

   hpos = LTLEN - strlen(litres);          /* print litres */
   mvaddstr(pos++, 18+hpos, litres);

   /* print total cost */
   (void)sprintf(cost, "%.2f", (gde_price * atof(litres) / 100.00));
   hpos = MAX_PLEN - strlen(cost);
   move(pos++, 14+hpos);
   printw(" $%s", cost);
}

static void disp_status(string) char *string;
{
   int pid, status;
   char ch;

   (void)sscanf(string, "%d %d", &pid, &status);

   if(status == YES)
      ch = 'Y';
   else
      ch = 'N';

   mvaddch(pid+1, 76, ch);
}

static void clr_trans(string) char *string;
{
   int pid;

   (void)sscanf(string, "%d", &pid);    /* get pump id */
   mvaddstr(pid+1, 14, "          ");   /* clear grade name */
   mvaddstr(pid+1, 36, "     ");        /* clear litres */
   mvaddstr(pid+1, 53, "         ");    /* clear total to pay */
}
```

/* pce_chge.c: price change module. */

```
#include <sys/file.h>
#include <fcntl.h>
#include <curses.h>
#include "pump.h"
#include "files.h"
#include "display.h"

#define SUPER_CODE      "mohammad"
#define CODE_LEN        8

void get_gname();

static void acc_pce_chage();

void verify_code(pumps) int pumps;
/* Diagram ref.: Verify Code, .2.0 */
{
   int i;
   char ch, code[CODE_LEN];
   WINDOW *pcescr;

   /* create price change screen and initialise it */
   pcescr = newwin(24, 80, 0, 0);
   wclear(pcescr);
   touchwin(pcescr);
   wmove(pcescr, 0, 34);
   waddstr(pcescr, "Price Change");
   wrefresh(pcescr);

   /* get code */
   wmove(pcescr, 1, 0);
   waddstr(pcescr, "Please enter code.");
   wrefresh(pcescr);
   for(i=0; (ch = getch()) != '\r'; )
      code[i++] = ch;
   code[i] = '\0';

   if(strcmp(code, SUPER_CODE) != 0)         /* verify code */
   {
      wmove(pcescr, 3, 0);
      waddstr(pcescr, "Invalid code! Type any key to continue.");
      wrefresh(pcescr);
      (void)getch();
   }
   else
   {
      wmove(pcescr, 1, 20);
      waddstr(pcescr, "Code validated");
      wrefresh(pcescr);
      acc_pce_chage(pcescr, pumps);
   }
}
```

```
static void acc_pce_chage(pcescr, pumps) WINDOW *pcescr;
                                         int pumps;
/* Diagram ref.: Accept Price Changes, .2.1 */
{
   int csfd, ppfd, i, j, grade, pos, line=2;
   float oldprice, newprice, oldstock, atof();
   char curr_stk[MAX_CSTK + 1],
        new_cstk[MAX_CSTK + 1],
        grade_records[NGRADES][GDLEN+MAXPRICELEN+MAXSTOCKLEN+1],
        gradestr[MAXGDESTR + 1],
        price[MAXPRICELEN + 1],
        pcr[GDLEN + MAXPRICELEN + 1],
        pip_name[MAX_PIP_NAM + 1],
        pcechge[PCECHGLEN + 1],
        ch, *cptr, *strnsp();

   for(i=0; i<PCECHGLEN; i++)
      pcechge[i] = '\0';

   if((csfd = open(CURRENT_STOCK, O_RDWR)) < 0)
      perror("current stock open");

   if(flock(csfd, LOCK_EX) < 0)        /* lock current stock  */
      perror("current stock flock");

   if(read(csfd, curr_stk, MAX_CSTK) < 0)   /* get records */
      perror("current stock read");

   for(cptr=curr_stk, i=0; i<NGRADES; i++) /* get grade records */
   {
      for(j=0; *cptr != '\n' && *cptr != '\0'; j++)
         grade_records[i][j] = *cptr++ ;
      grade_records[i][j++] = *cptr++;
      grade_records[i][j] = '\0';
   }

   wmove(pcescr, line, 0);
   waddstr(pcescr, "Please select grade: ");
   for(i=0; i<NGRADES; i++)
   {
      get_gname(i, gradestr);
      cptr = strnsp(gradestr);
      wmove(pcescr, line++, 21);
      wprintw(pcescr, "%d. %s", i+1, cptr);
   }
   wmove(pcescr, line++, 21);
   waddstr(pcescr, "Q. quit");
   wmove(pcescr, ++line, 0);
   waddstr(pcescr, "Selection: ");
   wrefresh(pcescr);

   do
   {
```

```
ch = getch();
wmove(pcescr, line, 11);        /* display ch */
waddch(pcescr, ch);
wrefresh(pcescr);

if(ch > '0' && ch <= NGRADES+'0')
{
   grade = ch-'0'-1;

   /* get new price */
   wmove(pcescr, line+2, 0);
   waddstr(pcescr, "Please enter new price: ");
   wrefresh(pcescr);
   for(pos=24, i=0; (ch = getch()) != '\r';)
   {
      if((ch>='0' && ch<='9') || ch=='.' || ch==ERASE)
      {
         if(ch != ERASE)
         {
            waddch(pcescr, ch);
            wrefresh(pcescr);
            price[i++] = ch;
            ++pos;
         }
         else
            if(i > 0)
            {
               waddch(pcescr, ch);
               wrefresh(pcescr);
               --i;
               --pos;
            }
      }
   }
   price[i] = '\0';

   if((newprice = atof(price)) <= 0.00 || newprice > 99.99)
   {
      wmove(pcescr, line+2, ++pos);
      waddstr(pcescr, "Invalid price.");
      waddstr(pcescr, "Type any key to continue");
      wrefresh(pcescr);
      (void)getch();
   }
   else
   {
      /* change old price */
      (void)sscanf(grade_records[grade], "%*d %f %f"
                                       , &oldprice, &oldstock);
      (void)sprintf(grade_records[grade], "%d %.2f %.2f\n",
                                  grade , newprice , oldstock);

      /* add new price record to pump message */
      (void)sprintf(pcr, "%d %.2f ", grade, newprice);
```

```
            (void)strcat(pcechge, pcr);
         }

         wmove(pcescr, line+2, 0);
         wclrtoeol(pcescr);
         wrefresh(pcescr);
      }
      else
         if(ch != 'q' && ch != 'Q')
         {
                 wmove(pcescr, line+2, 0);
                 waddstr(pcescr, "Invalid choice");
                 wrefresh(pcescr);
         }
   } while(ch != 'q' && ch != 'Q');

   delwin(pcescr);         /* delete price change window */

   for(i=0; i<MAX_CSTK; i++)        /* clear new_cstk */
      new_cstk[i] = '\0';
   for(i=0; i<NGRADES; i++)        /* copy new records */
      strcat(new_cstk, grade_records[i]);

   if((i = strlen(new_cstk)) < MAX_CSTK)
      while(i < MAX_CSTK)        /* pad new_cstk */
         new_cstk[i++] = ' ';
   new_cstk[i] = '\0';

   if(lseek(csfd, (long)0, 0) < 0)
      perror("current stock lseek");
   if(write(csfd, new_cstk, MAX_CSTK) < 0) /* write new records */
      perror("current stock write");

   if(flock(csfd, LOCK_UN) < 0)        /* unlock store */
      perror("current stock flock");

   if(close(csfd) < 0)
      perror("current stock close");

   if((i = strlen(pcechge)) < PCECHGLEN)
      while(i < PCECHGLEN)        /* pad pcechge */
         pcechge[i++] = ' ';
   pcechge[i] = '\0';

   for(i=0; i<pumps; i++)  /* send price changes to pump pipes */
   {
      (void)sprintf(pip_name, "%s%d", PTP_PIPE, i+1);
      if((ppfd = open(pip_name, O_RDWR|O_NDELAY)) < 0)
         perror("pump price change pipe open");
      if(write(ppfd, pcechge, PCECHGLEN) < 0)
         perror("pump price change pipe write");
      if(close(ppfd) < 0)
         perror("pump price change pipe close");
   }
```

```
}
```

/* report.c: report printing module */

```
#include <sys/file.h>
#include <fcntl.h>
#include <unistd.h>
#include <curses.h>
#include "files.h"
#include "pump.h"

static void trans_rep(),
            stock_rep();

void get_gname();

void pr_report()
/* Diagram ref.: Print Report, .3 */
{
   WINDOW *rep;
   char ch;

   rep = newwin(24, 80, 0, 0);
   wclear(rep);
   touchwin(rep);
   wrefresh(rep);

   waddstr(rep, "Please select report type: T. Transaction Report");
   wmove(rep, 1, 27);
   waddstr(rep, "S. Stock Report");
   wmove(rep, 2, 27);
   waddstr(rep, "F. Full Report");
   wmove(rep, 3, 27);
   waddstr(rep, "Q. Quit");
   wmove(rep, 4, 0);
   waddstr(rep, "Selection: ");
   wrefresh(rep);

   do
   {
      ch = getch();
      wmove(rep, 4, 12);
      waddch(rep, ch);
      wrefresh(rep);

      switch(ch)
      {
         case 't' :          /* transaction report */
         case 'T' : trans_rep(rep);
                    break;

         case 's' :          /* stock report */
         case 'S' : stock_rep(rep);
                    break;

         case 'f' :          /* full report */
         case 'F' : trans_rep(rep);
                    wmove(rep, 22, 0);
                    waddstr(rep, "Type any character to continue");
                    wrefresh(rep);
                    (void)getch();          /* wait for char */
                    stock_rep(rep);
                    break;

         case 'q' :
         case 'Q' : break;

         default  : wmove(rep, 6, 0);
                    waddstr(rep, "Invalid choice");
                    wrefresh(rep);
                    break;
      }
   }while(ch!='t' && ch!='T' && ch!='s' && ch!='S' &&
                    ch!='f' && ch!='F' && ch!='q' && ch!='Q');

   if(ch != 'q' && ch != 'Q') /* wait for report to be viewed */
   {
      wmove(rep, 22, 0);
      waddstr(rep, "Type any key to continue");
      wrefresh(rep);
      (void)getch();
   }

   delwin(rep);
}

static void trans_rep(rep) WINDOW *rep;
/* Minispecs ref.: Sub-procedure of .3 */
{
   int thfd, pid, grade, r, line = 2;
   float litres, cost, tot_litres = 0.00, tot_cash = 0.00;
   char trans_record[MAX_THRLEN + 1], gradestr[MAXGDESTR + 1];

   /* lock transaction history */
   if((thfd = open(TRANS_HIST, O_RDWR)) < 0)
      perror("transaction history open");
   if(flock(thfd, LOCK_EX) < 0)
      perror("transaction history flock");

   /* clear window and print headers */
   wclear(rep);
   wmove(rep, 0, 31);
   waddstr(rep, "TRANSACTION REPORT");
   wmove(rep, 1, 10);
   waddstr(rep, "Pump No   Litres Delivered   Petrol Grade   Total");
   wrefresh(rep);

   while((r = read(thfd, trans_record, MAX_THRLEN)) != 0)
   {
      if(r < 0)
```

```
         perror("transaction history read");

      (void)sscanf(trans_record, "%d %d %f %f", &pid, &grade
                                                , &litres, &cost);
      get_gname(grade, gradestr);

      tot_litres += litres;            /* accumulate litres */
      tot_cash += cost;                /* accumulate cash */

      /* print report record */
      wmove(rep, line++, 0);
      wprintw(rep, "%14d%19.2f%20s          $%.2f", pid, litres
                                                    , gradestr, cost);
      wrefresh(rep);

      if(line == 21)                /* screen full */
      {
         wmove(rep, 22, 0);
         waddstr(rep, "Type any character to continue");
         wrefresh(rep);
         (void)getch();                /* wait for char */
         line = 2;
         wmove(rep, 22, 0);
         wclrtoeol(rep);
         wrefresh(rep);
      }
   }

   if(flock(thfd, LOCK_UN) < 0)
      perror("transaction histroy flock");
   if(close(thfd) < 0)
      perror("transaction history close");

   /* print totals */
   wmove(rep, line++, 0);
   wprintw(rep, "%33s%36s", "-----", "------");
   wmove(rep, line++, 0);
   wprintw(rep, "%33.2f                          $%.2f", tot_litres
                                                      , tot_cash);

   for(;line < 23; line++)        /* delete left over lines */
   {
      wmove(rep, line, 0);
      wclrtoeol(rep);
   }
   wrefresh(rep);
}

static void stock_rep(rep) WINDOW *rep;
/* Minispecs ref.: Sub-procedure of .3 */
{
   int csfd, grade, i, j, line = 2;
   float price, stock;
   char grade_records[NGRADES][GDLEN+MAXPRICELEN+MAXSTOCKLEN + 1],
          curr_stk[MAX_CSTK + 1],
          gradestr[MAXGDESTR + 1],
          *cptr;

     /* lock current stock and read its records */
     if((csfd = open(CURRENT_STOCK, O_RDWR)) < 0)
        perror("current stock open");
     if(flock(csfd, LOCK_EX) < 0)
        perror("current stock flock");
     if(read(csfd, curr_stk, MAX_CSTK) < 0)
        perror("current stock read");
     if(flock(csfd, LOCK_UN) < 0)
        perror("current stock flock");
     if(close(csfd) < 0)
        perror("current stock close");

     /* clear window and print headers */
     wclear(rep);
     wmove(rep, 0, 34);
     waddstr(rep, "STOCK REPORT");
     wmove(rep, 1, 11);
     waddstr(rep, "Petrol Grade     Litres in stock       Price/Litre");
     wrefresh(rep);

     for(cptr=curr_stk, i=0; i<NGRADES; i++) /* get grade records */
     {
        for(j=0; *cptr != '\n' && *cptr != '\0'; j++)
           grade_records[i][j] = *cptr++ ;
        grade_records[i][j++] = *cptr++;
        grade_records[i][j] = '\0';

        (void)sscanf(grade_records[i], "%d %f %f", &grade, &price
                                                       , &stock);

        get_gname(grade, gradestr);
        wmove(rep, line++, 11);
        wprintw(rep, "%10s%21.2f             %.2f p", gradestr
                                                   , stock, price);

        wrefresh(rep);
     }
}
```

/* rec_stk_del.c: stock delivery module */

```
#include <sys/file.h>
#include <fcntl.h>
#include <curses.h>
#include "pump.h"
#include "files.h"
#include "display.h"

void get_gname();

void record_stk_del()
/* Diagram ref.: Record Stock Delivery, .1.2 */
{
   int csfd, i, j, grade, pos, line=2;
   float oldprice, newstock, oldstock, atof();
   char curr_stk[MAX_CSTK + 1],
        new_cstk[MAX_CSTK + 1],
        grade_records[NGRADES][GDLEN+MAXPRICELEN+MAXSTOCKLEN+1],
        gradestr[MAXGDESTR + 1],
        stock[MAXSTOCKLEN + 1],
        ch, *cptr, *strnsp();
   WINDOW *stkscr;

   /* create stock delivery screen and initialise it */
   stkscr = newwin(24, 80, 0, 0);
   wclear(stkscr);
   touchwin(stkscr);
   wmove(stkscr, 0, 33);
   waddstr(stkscr, "Stock Delivery");
   wrefresh(stkscr);

   if((csfd = open(CURRENT_STOCK, O_RDWR)) < 0)
      perror("current stock open");

   if(flock(csfd, LOCK_EX) < 0)        /* lock current stock  */
      perror("current stock flock");

   if(read(csfd, curr_stk, MAX_CSTK) < 0)   /* get records */
      perror("current stock read");

   for(cptr=curr_stk, i=0; i<NGRADES; i++) /* get grade records */
   {
      for(j=0; *cptr != '\n' && *cptr != '\0'; j++)
         grade_records[i][j] = *cptr++ ;
      grade_records[i][j++] = *cptr++;
      grade_records[i][j] = '\0';
   }

   wmove(stkscr, line, 0);
   waddstr(stkscr, "Please select grade: ");
   for(i=0; i<NGRADES; i++)
   {
      get_gname(i, gradestr);
      cptr = strnsp(gradestr);
      wmove(stkscr, line++, 21);
      wprintw(stkscr, "%d. %s", i+1, cptr);
   }
   wmove(stkscr, line++, 21);
   waddstr(stkscr, "Q. quit");
   wmove(stkscr, ++line, 0);
   waddstr(stkscr, "Selection: ");
   wrefresh(stkscr);

   do
   {
      ch = getch();
      wmove(stkscr, line, 11);          /* display ch */
      waddch(stkscr, ch);
      wrefresh(stkscr);

      if(ch > '0' && ch <= NGRADES+'0')
      {
         grade = ch-'0'-1;

         /* get new stock */
         wmove(stkscr, line+2, 0);
         waddstr(stkscr, "Please enter litres delivered: ");
         wrefresh(stkscr);
         for(pos=31, i=0; (ch = getch()) != '\r';)
         {
            if((ch>='0' && ch<='9') || ch=='.' || ch==ERASE)
            {
               if(ch != ERASE)
               {
                  waddch(stkscr, ch);
                  wrefresh(stkscr);
                  stock[i++] = ch;
                  ++pos;
               }
               else
                  if(i > 0)
                  {
                     waddch(stkscr, ch);
                     wrefresh(stkscr);
                     --i;
                     --pos;
                  }
            }
         }
         stock[i] = '\0';

         (void)sscanf(grade_records[grade], "%*d %f %f"
                                          , &oldprice , &oldstock);

         if((newstock = atof(stock)) <= 0.00
                                    || newstock+oldstock > 9999.99)
         {
            wmove(stkscr, line+2, ++pos);
            waddstr(stkscr, "Invalid delivery.");
            waddstr(stkscr, "Type any key to continue");
            wrefresh(stkscr);
            (void)getch();
         }
         else         /* change stock */
            (void)sprintf(grade_records[grade], "%d %.2f %.2f\n"
                                            , grade, oldprice
                                            , oldstock+newstock);
         wmove(stkscr, line+2, 0);
         wclrtoeol(stkscr);
         wrefresh(stkscr);
      }
      else
         if(ch != 'q' && ch != 'Q')
         {
            wmove(stkscr, line+2, 0);
            waddstr(stkscr, "Invalid choice");
            wrefresh(stkscr);
         }
   } while(ch != 'q' && ch != 'Q');

   delwin(stkscr);            /* delete stock change window */

   for(i=0; i<MAX_CSTK; i++)        /* clear new_cstk */
      new_cstk[i] = '\0';
   for(i=0; i<NGRADES; i++)        /* copy new records */
      strcat(new_cstk, grade_records[i]);

   if((i = strlen(new_cstk)) < MAX_CSTK)
      while(i < MAX_CSTK)        /* pad new_cstk */
         new_cstk[i++] = ' ';
   new_cstk[i] = '\0';

   if(lseek(csfd, (long)0, 0) < 0)
      perror("current stock lseek");
   if(write(csfd, new_cstk, MAX_CSTK) < 0) /* write new records */
      perror("current stock write");

   if(flock(csfd, LOCK_UN) < 0)        /* unlock store */
      perror("current stock flock");

   if(close(csfd) < 0)
      perror("current stock close");
}
```

/* mon_stk.c: process to monitor stock levels, and warn the operartor when any of the grades falls below a threshold value. */

```
#include <sys/file.h>
#include <signal.h>
#include <fcntl.h>
#include <unistd.h>
#include "files.h"
#include "pump.h"
#include "display.h"

#define THRESHOLD 100.00
#define forever          for(;;)

static void catch_del();

static int warned[NGRADES];

main()
/* Diagram ref.: Monitor Stock, .1.1 */
{
   int csfd, dpfd, i, j;
   float stock;
   char grade_records[NGRADES][GDLEN+MAXPRICELEN+MAXSTOCKLEN + 1],
        curr_stk[MAX_CSTK + 1],
        *cptr;
   disp_prot ds;

   if(signal(SIGUSR1, catch_del) < 0) /* catch delivery signal */
      perror("delivery catching signal");

   if((dpfd = open(MTD_PIPE, O_RDWR)) < 0) /* open display pipe */
      perror("display pipe open");

   for(i=0; i < NGRADES; i++)        /* initialise warnings */
      warned[i] = NO;

   forever
   {
      /* lock current stock and read its records */
      if((csfd = open(CURRENT_STOCK, O_RDWR)) < 0)
         perror("current stock open");
      if(flock(csfd, LOCK_EX) < 0)
         perror("current stock flock");
      if(read(csfd, curr_stk, MAX_CSTK) < 0)
         perror("current stock read");
      if(flock(csfd, LOCK_UN) < 0)
         perror("current stock flock");
      if(close(csfd) < 0)
         perror("current stock close");

      /* get grade records */
      for(cptr=curr_stk, i=0; i<NGRADES; i++)
      {
         for(j=0; *cptr != '\n' && *cptr != '\0'; j++)
            grade_records[i][j] = *cptr++ ;
         grade_records[i][j++] = *cptr++;
         grade_records[i][j] = '\0';

         (void)sscanf(grade_records[i], "%*d %*f %f", &stock);
         if(stock < THRESHOLD)
         {
            /* warn once only */
            if(warned[i] == NO)
              {
                 ds.disp_type = WARN;
                 (void)sprintf(ds.buffer, "%d", i);
                 if(write(dpfd, &ds, sizeof(disp_prot))
                                        < sizeof(disp_prot))
                    perror("display pipe write");

                 warned[i] = YES;
            }
         }
      }

      /* Diagram ref.: Clock, .1.0 */
      sleep(10);                /* every 10 seconds */
   }
}

static void catch_del()
{
   int i;

   signal(SIGUSR1, catch_del);

   for(i=0; i<NGRADES; i++)        /* reset warnings */
      warned[i] = NO;
}
```

/* mon_pump.c: process for monitoring pump operation. Receives the pump id and the pid for the pump simulator process. */

```
#include <sys/file.h>
#include <signal.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <curses.h>
#include "pump.h"
#include "files.h"
#include "display.h"

#define forever for(;;)

static int pend_req = NO,       /* event store for pending request */
           pump_idle = YES,     /* event store for pump status */
           del_comp = NO,       /* sync. flag for delivery complete */
           trans_rec = NO,      /* sync. flag for history updated */
           delivery = NO;       /* flag for delivery disbalement */

static int ppfd,                /* file descriptor for pump pipe */
           dpfd,                /* file descriptor for display pipe */
           sig_num,             /* signal identifier */
           mask,                /* singal mask */
           pumpidno,            /* pump identity number */
           pumppid;             /* pid for pump process */

static float gde_price;         /* grade price for receipt printer */

/* current transaction and last transaction stores */
static char curr_trans[MAX_THRLEN + 1],
            last_trans[MAX_THRLEN + 1];

static void catch_req(),
            req_serv(),
            catch_button(),
            get_button(),
            catch_receipt(),
            pr_receipt(),
            catch_delivery(),
            get_delivery(),
            check_pump_status(),
            rec_trans(),
            update_trans_hist();

void main(argc, argv) int argc;
                      char *argv[];
{
   char pin[MAX_PIP_NAM + MAX_PIDLEN + 1];

   if(argc < 3)
   {
      printf("%s: Too few arguments\n", argv[0]);
      exit(1);
   }

   if(signal(SIGUSR1, catch_req) < 0)      /* catch requests */
      perror("request catching signal");
   if(signal(SIGUSR2, catch_button) < 0)   /* catch button presses */
      perror("button catching signal");
   if(signal(SIGFPE, catch_receipt) < 0)   /* catch receipt request */
      perror("receipt catching signal");
   if(signal(SIGTERM, catch_delivery) < 0) /* catch deliveries */
      perror("delivery catching signal");

   /* set up signal mask */
   mask = 0;
   mask |= sigmask(SIGUSR1);
   mask |= sigmask(SIGUSR2);
   mask |= sigmask(SIGFPE);
   mask |= sigmask(SIGTERM);

   pumpidno = atoi(argv[1]);           /* get pump id no. */
   pumppid = atoi(argv[2]);            /* get pump simulator pid */

   sprintf(pin, "%s%s", PTM_PIPE, argv[1]); /* open pipe to pump */
   if((ppfd = open(pin, O_RDWR)) < 0)
      perror("pump input pipe open");

   if((dpfd = open(MTD_PIPE, O_RDWR)) < 0)  /* open disp. pipe */
      perror("display pipe open");

   forever
   {
      sigpause(0);                  /* wait for service request */
      switch(sig_num)
      {
         case SIGUSR1 : req_serv();
                        break;
         case SIGUSR2 : get_button();
                        break;
         case SIGFPE  : pr_receipt();
                        break;
         case SIGTERM : get_delivery();
                        break;
      }
   }
}

static void check_pump_status()
/* Diagram ref.: Check Pump Status, .0.1 */
{
   disp_prot ds;

   (void)sigblock(mask);               /* mask signals */
   if(pump_idle == YES)
```

```
{
   /* Diagram ref.: Start The pump, .0.2 */
   if(kill(pumppid, SIGUSR1) < 0)         /* start pump */
      perror("enable pump kill");

   /* turn pump light off */
   ds.disp_type = LIGHT;
   (void)sprintf(ds.buffer, "%d %d", pumpidno, OFF);
   if(write(dpfd, &ds, sizeof(disp_prot)) < sizeof(disp_prot))
      perror("display pipe write");

   /*change pump status */
   ds.disp_type = ACTVE;
   (void)sprintf(ds.buffer, "%d %d", pumpidno, YES);
   if(write(dpfd, &ds, sizeof(disp_prot)) < sizeof(disp_prot))
      perror("display pipe write");

   pump_idle = NO;
   pend_req = NO;
   rec_trans();
}
else
{
   update_trans_hist();

   if(pend_req == YES && delivery == NO)
   {
     /* Diagram ref.: Start The pump, .0.2 */
     if(kill(pumppid, SIGUSR1) < 0)     /* start pump */
        perror("enable pump kill");

     /*turn pump light off */
     ds.disp_type = LIGHT;
     (void)sprintf(ds.buffer, "%d %d", pumpidno, OFF);
     if(write(dpfd, &ds, sizeof(disp_prot)) < sizeof(disp_prot))
        perror("display pipe write");

     /*change pump status */
     ds.disp_type = ACTVE;
     (void)sprintf(ds.buffer, "%d %d", pumpidno, YES);
     if(write(dpfd, &ds, sizeof(disp_prot)) < sizeof(disp_prot))
        perror("display pipe write");

     pend_req = NO;
     rec_trans();
   }
   else
   {
     /*change pump status */
     ds.disp_type = ACTVE;
     (void)sprintf(ds.buffer, "%d %d", pumpidno, NO);
     if(write(dpfd, &ds, sizeof(disp_prot)) < sizeof(disp_prot))
        perror("display pipe write");
        pump_idle = YES;

        /* keep light on if pending request */
        if(pend_req == YES)
        {
           /*turn pump light on */
           ds.disp_type = LIGHT;
          (void)sprintf(ds.buffer, "%d %d", pumpidno, ON);
          if(write(dpfd, &ds, sizeof(disp_prot)) < sizeof(disp_prot))
             perror("display pipe write");
        }
      }
   }
}

static void rec_trans()
/* Diagram ref.: Record Transaction, .0.3 */
{
   char grade_records[NGRADES][GDLEN+MAXPRICELEN+MAXSTOCKLEN + 1],
        curr_stk[MAX_CSTK + 1],
        new_cstk[MAX_CSTK + 1],
        total[MAX_PLEN + 1],
        pumpid,
        *cptr;
   int csfd, pet_gde, i, j;
   float lit_del, oldstock;
   disp_prot ds;

   (void)sigblock(mask);                /* mask signals */
   /* clear curr_trans */
   for(i=0; i<PTM_LEN+MAX_PLEN+1; i++)
      curr_trans[i] = '\0';

   while(read(ppfd, curr_trans, PTM_LEN) < 0)    /* receive TD */
   {
      if(errno == EINTR)        /* continue after interupt */
         continue;
      else
         perror("pump pipe read");
   }
   sscanf(curr_trans, "%c %d %f", &pumpid, &pet_gde, &lit_del);

   if((csfd = open(CURRENT_STOCK, O_RDWR)) < 0)
      perror("current stock open");
   if(flock(csfd, LOCK_EX) < 0)         /* lock current stock  */
      perror("current stock flock");

   while(read(csfd, curr_stk, MAX_CSTK) < 0) /* get store records */
   {
      if(errno == EINTR)             /* continue after interupt */
              continue;
      else
              perror("current stock read");
   }
```

```
for(cptr=curr_stk, i=0; i<NGRADES; i++) /* get grade records */
{
   for(j=0; *cptr != '\n' && *cptr != '\0'; j++)
      grade_records[i][j] = *cptr++ ;
   grade_records[i][j++] = *cptr++;
   grade_records[i][j] = '\0';
}

(void)sscanf(grade_records[pet_gde],"%*d %f %f", &gde_price
                                                , &oldstock);
(void)sprintf(grade_records[pet_gde], "%d %.2f %.2f\n", pet_gde
                                       , gde_price , oldstock-lit_del);

for(i=0; i<MAX_CSTK; i++)        /* clear new_cstk */
   new_cstk[i] = '\0';
for(i=0; i<NGRADES; i++)        /* copy new records */
   strcat(new_cstk, grade_records[i]);

if((i = strlen(new_cstk)) < MAX_CSTK)
   while(i < MAX_CSTK)        /* pad new_cstk */
      new_cstk[i++] = ' ';
new_cstk[i] = '\0';

if(lseek(csfd, (long)0, 0) < 0)
   perror("current stock lseek");
if(write(csfd, new_cstk, MAX_CSTK) < 0) /* write new records */
   perror("current stock write");

if(flock(csfd, LOCK_UN) < 0)        /* unlock current stock */
   perror("current stock flock");

if(close(csfd) < 0)
   perror("current stock close");

/* concat. total to pay (in pounds) */
(void)sprintf(total, " %.2f", gde_price * lit_del / 100.00);
(void)strcat(curr_trans, total);

/* display current transaction */
ds.disp_type = TRANS;
(void)strcpy(ds.buffer, curr_trans);
if(write(dpfd, &ds, sizeof(disp_prot)) < sizeof(disp_prot))
   perror("display pipe write");


/* turn pump light on */
ds.disp_type = LIGHT;
(void)sprintf(ds.buffer, "%d %d", pumpidno, ON);
if(write(dpfd, &ds, sizeof(disp_prot)) < sizeof(disp_prot))
   perror("display pipe write");

/* sound console bell */
ds.disp_type = BELL;
```

```
   if(write(dpfd, &ds, sizeof(disp_prot)) < sizeof(disp_prot))
      perror("display pipe write");

   del_comp = YES;                /* delivery is complete */
}

static void update_trans_hist()
/* Diagram ref.: Update Transaction History, .0.4 */
{
   int thfd, i;
   disp_prot ds;

   (void)sigblock(mask);        /* block signals */

   del_comp = NO;                /* reset sync. flag */

   if((thfd = open(TRANS_HIST, O_RDWR)) < 0)
      perror("transaction history open");

   if(flock(thfd, LOCK_EX) < 0)    /* lock transaction history */
      perror("transaction history flock");

   if(lseek(thfd, (long)0, 2) < 0)        /* go to store end */
      perror("transaction history lseek");

   for(i=strlen(curr_trans); i<MAX_THRLEN - 1; i++)
      curr_trans[i] = ' ';
   curr_trans[i++] = '\n';
   curr_trans[i] = '\0';

   if(write(thfd, curr_trans, strlen(curr_trans)) < 0)
      perror("transaction history write");

   if(flock(thfd, LOCK_UN) < 0)    /* unlock transaction history */
      perror("transaction history flock");

   if(close(thfd) < 0)
      perror("transaction history close");

   (void)strcpy(last_trans, curr_trans);

   /*turn pump light off */
   ds.disp_type = LIGHT;
   (void)sprintf(ds.buffer, "%d %d", pumpidno, OFF);
   if(write(dpfd, &ds, sizeof(disp_prot)) < sizeof(disp_prot))
      perror("display pipe write");

   /*clear transaction display */
   ds.disp_type = CLRTR;
   (void)sprintf(ds.buffer, "%d", pumpidno);
   if(write(dpfd, &ds, sizeof(disp_prot)) < sizeof(disp_prot))
      perror("display pipe write");

   trans_rec = YES;        /* set synchronisation flag */
```

```
}

static void catch_req()
{
   if(signal(SIGUSR1, catch_req) < 0)
      perror("request catching signal");

   sig_num = SIGUSR1;
}

static void req_serv()
/* Diagram ref.: Request Service, .0.0 */
{
   disp_prot ds;

   pend_req = YES;
   if(pump_idle == YES)
   {
      /* turn pump light on */
      ds.disp_type = LIGHT;
      (void)sprintf(ds.buffer, "%d %d", pumpidno, ON);
      if(write(dpfd, &ds, sizeof(disp_prot)) < sizeof(disp_prot))
         perror("display pipe write");

      /* sound console bell */
      ds.disp_type = BELL;
      if(write(dpfd, &ds, sizeof(disp_prot)) < sizeof(disp_prot))
         perror("display pipe write");
   }
}

static void catch_button()
{
   if(signal(SIGUSR2, catch_button) < 0)
      perror("button catching signal");

   sig_num = SIGUSR2;
}

static void get_button()
{
   /* catch erronous button presses */
   if((pump_idle == YES && pend_req == YES && delivery == NO)
                        || (pump_idle == NO && del_comp == YES))
      check_pump_status();
}

static void catch_receipt()
{
   if(signal(SIGFPE, catch_receipt) < 0)
      perror("request catching signal");

   sig_num = SIGFPE;
}
```

```
static void pr_receipt()
/* Diagram ref.: .0.5 */
{
   disp_prot ds;
   char grade,
        litres[LTLEN + 1];

   if(trans_rec == YES)        /* synchronise and catch errors */
   {
      trans_rec = NO;     /* reset synchronisation flag */
      (void)sscanf(last_trans, "%*d %c %s %*s", &grade, litres);

      /* print receipt */
      ds.disp_type = RECPT;
      (void)sprintf(ds.buffer,"%c %.2f %s", grade, gde_price, litres);
      if(write(dpfd, &ds, sizeof(disp_prot)) < sizeof(disp_prot))
         perror("display pipe write");
   }
}

static void catch_delivery()
{
   if(signal(SIGTERM, catch_delivery) < 0)
      perror("delivery catching signal");

   sig_num = SIGTERM;
}

static void get_delivery()
{
   if(delivery == NO)   /* set delivery flag on 1st interrupt */
      delivery = YES;
   else                        /* unset it on the second */
      delivery = NO;
}
```

/* pump.c: process to simulate a pump operation. Receives the pump id and the pid of the pump monitoring process */

```
#include <fcntl.h>
#include <string.h>
#include <signal.h>
#include <errno.h>
#include "pump.h"

#define forever for(;;)

void catch_enb();

void main(argc, argv) int argc;
                      char *argv[];
{
   int i, send, pcge, litres, mpo_pid;
   unsigned short rand1[3], rand2[3];
   long nrand48();
   double erand48();
   char pip_name[MAX_PIP_NAM + MAX_PIDLEN + 1],
        pcechge[PCECHGLEN + 1],
        out_buff[PTM_LEN + 1];

   if(argc < 3)
   {
      printf("%s: Too few arguments\n", argv[0]);
      exit(1);
   }

   if(signal(SIGUSR1, catch_enb) < 0) /* catch enabling signal */
      perror("enablement catching signal");

   /* make up pipe to monitor name and open it */
   (void)sprintf(pip_name, "%s%s", PTM_PIPE, argv[1]);
   if((send = open(pip_name, O_RDWR)) < 0)
      perror("pump output pipe open");

   /* make up pipe from price change and open it */
   (void)sprintf(pip_name, "%s%s", PTP_PIPE, argv[1]);
   if((pcge = open(pip_name, O_RDWR|O_NDELAY)) < 0)
      perror("price change pipe open");

   for(i=0; i<3; i++)        /* initialise seed for nrand48 */
      rand1[i] = rand2[i] = atoi(argv[1]); /* seed id for pump */

   mpo_pid = atoi(argv[2])   /* get monitor process's pid */
   sleep((unsigned)5); /* wait for all procs to come to life */

   forever
   {
      /* wait for customer */
      sleep((unsigned) (nrand48(rand1) % 30));
      if(kill(mpo_pid, SIGUSR1) < 0)       /* request service */
         perror("pump to monitor kill");

      pause();                /* wait until service is granted */

      litres = (int) (nrand48(rand2) % 100);  /* max del. = 99 */
      if(litres < 2)                 /* min. delivery = 2 lit. */
         litres = 2;

      sleep(10);                /* wait for delivery */

      /* make up transaction details and send it */
      (void)sprintf(out_buff, "%s %d %.2f", argv[1], litres % NGRADES
                              , (float)(litres + erand48(rand2)));

      for(i=strlen(out_buff); i<PTM_LEN; i++)
         out_buff[i] = ' ';
      out_buff[i] = '\0';
      if(write(send, out_buff, PTM_LEN) < 0)        /* send TD */
         perror("pump pipe write");

      /* read price changes, if any */
      if(read(pcge, pcechge, PCECHGLEN) < 0)
         if(errno != EWOULDBLOCK)
            perror("pump price change pipe read");
   }
}

static void catch_enb()
{
   signal(SIGUSR1, catch_enb);
}
```

```
/* misc.c: module holding miscellaneous functions */

#include <string.h>

void get_gname(grade, string) int grade;
                              char *string;
{
   switch(grade)
   {
      case 0  : (void)strcpy(string, "  diesel");
                 break;
      case 1  : (void)strcpy(string, "  4 star");
                break;
      case 2  : (void)strcpy(string, "unleaded");
                break;
      default : printf("Grade string error\n");
                break;
   }
}

char *strnsp(string) char *string;
{
   while(*string == ' ' || *string == '\t' || *string == '\n')
      ++string;

   return(string);
}
```

Appendix C

Diagram Syntax Rules

C.1 Overview

Chapter 4 briefly discussed the syntax for the diagrams of the proposed new notation. The first section below gives an abstract syntax based on sets for this notation. The following section gives a concrete syntax in the form of a textual language for describing the diagrams. An example of a textual description is also included.

C.2 An Abstract Syntax

The abstract syntax of a set of specification diagrams can be described using sets. The set representation does not, however, include enough constraints on legal specifications. A set of logic predicates complement the sets to define a set of legal specification diagrams.

1. The diagram set of a specification, $DSPEC$, is a pair, consisting of the Context Diagram, CD, and a set of diagram networks, N,

$$DSPEC = (CD, N)$$

2. A Context Diagram consists of the Context Data Flow Diagram and the Context Event Flow Diagram,

$$CD = (CDFD, CEFD)$$

where

$$CDFD = (Sp, DT, CDF)$$

and

$$CEFD = (Sp, ET, CEF)$$

where Sp = The system process name,
DT = The set of data terminator names,
$CDF = \{CD : CD = (CDfs, CDfd, CDfl)\}$
where $CDfs$ = The name of the context data flow's source,
$CDfd$ = The name of the context data flow's destination, and
$CDfl$ = The context data flow's label,
ET = The set of event terminator names, and
$CEF = \{CF : CF = (CEfs, CEfd, CEfl)\}$
where $CEfs$ = The name of the context event flow's source,
$CEfd$ = The name of the context event flow's destination, and
$CEfl$ = The context event flow's label,

3. A diagram level, $DL \in N$, is a 4-tuple,

$$N = \{DL : DL = (Ln, DFD, EFD, SCD)\}$$

where Ln = The diagram level's name,
$DFD = (DP, DS, DF)$,
$EFD = (EP, ES, SC, EF)$, and
$SCD = \{SS : SS = (Sn, ON, OFF)\}$.

where DP = The set of DFD process names,
DS = The set of data store names,
$DF = \{D : D = (Dfs, Dfd, Dfl)\}$,
where Dfs = The name of the data flow's source,
Dfd = The name of the data flow's destination, and
Dfl = The data flow's label,
EP = The set of EFD process names,
ES = The set of event store names,
SC = The set of synchronisation symbol names,
$EF = \{E : E = (Efs, Efd, Efl)\}$,
where Efs = The name of the event flow's source,
Efd = The name of the event flow's destination, and
Efl = The event flow's label,
Sn = The subsystem name,
ON = The set of subsystem enablement event names, and
OFF = The set of subsystem disablement event names.

C.2.1 Rules For The Context Diagram

In the following predicates a sugared syntax is used, where
$\forall\, d \in DF(...)$ stands for $\forall\, d(d \in DF \implies ...)$, and
$\exists\, n \in N(...)$ stands for $\exists\, n(n \in N \wedge ...)$.

4. There is one and only one diagram in the set of diagram network named with the system process name,

$$\exists_1 n \in N(n.Ln = Sp)$$

where the n.Ln is used to denote the Ln member of the tuple n, and $\exists_1$ indicates there is *one and only one*.

5. There are some terminators which communicate with the system,

$$\neg(DT = \phi \wedge \ ET = \phi)$$

where ϕ denotes the empty set, and $\neg$ indicates negation.

6. A context data flow cannot connect a node to itself,

$$\forall d \in CDF(d.CDfs \neq d.CDfd)$$

7. A context data flow cannot directly connect two data terminators,

$$\forall d \in CDF\neg(d.CDfs \in DT \wedge \ d.CDfd \in DT)$$

8. Every context data flow connects the system process to a data terminator,

$$\begin{aligned} &\forall d \in CDF((d.CDfs \in DT \wedge \ dC.Dfd = Sp) \\ &\qquad\qquad \vee \ (dC.Dfs = Sp \wedge \ d.CDfd \in DT)) \end{aligned}$$

9. Every data terminator is connected to the system process,

$$\forall t \in DT(\exists d \in \ CDF(d.CDfs = t \vee \ d.CDfd = t))$$

10. All CDFD names are unique,

$$\begin{aligned} &\forall t \in DT \ \forall d \in CDF \forall d' \in CDF(t \neq Sp \wedge \ t \neq d.CDfl \\ &\qquad\qquad\qquad \wedge \ d.CDfl \neq Sp \wedge \ d.CDfl \neq d'.CDfl) \\ &\qquad\qquad\qquad) \end{aligned}$$

11. There is only one diagram labelled with the system process name, and all the data flows of the Context Diagram appear on the DFD of this diagram,

$$\begin{aligned} &\exists_1 n \in N(n.Ln = Sp) \wedge \forall d \in CDF(\\ &\qquad (d.CDfs \in DT \Longrightarrow \\ &\qquad\qquad \exists_1 d' \in n.DFD.DF \ \exists_1 p' \in n.DFD.DP \\ &\qquad\qquad\qquad (d' = (Inherited, p', d.CDfl)) \\ &\qquad) \\ &\qquad \wedge(d.CDfd \in DT \Longrightarrow \\ &\qquad\qquad \exists_1 d' \in n.DFD.DF \ \exists_1 p' \in n.DFD.DP \\ &\qquad\qquad\qquad (d' = (p', Inherited, d.CDfl)) \\ &\qquad) \\ &\qquad\qquad\qquad) \end{aligned}$$

where $n.DFD.DF$ and $n.dfd.DP$ denote the data flows and the processes, respectively, of the data flow diagram of network n; and the term *Inherited* is used to denote the unconnected end of an inherited data flow.

12. A context event flow cannot connect a node to itself,

$$\forall e \in CEF(e.CEfs \neq e.CEfd)$$

13. A context event flow cannot directly connect two event terminators,

$$\forall e \in CEF^{\neg}(e.CEfs \in ET \wedge\ e.CEfd \in ET)$$

14. Every context event flow connects the system process to a event terminator,

$$\begin{aligned}\forall e \in CEF((&e.CEfs \in ET \wedge\ e.CEfd = Sp)\\ &\vee (e.CEfs = Sp \wedge\ e.CEfd \in ET))\end{aligned}$$

15. Every event terminator is connected to the system process,

$$\forall t \in DT(\exists e \in\ CEF(e.CEfs = t \vee\ e.CEfd = t))$$

16. All CEFD names are unique,

$$\begin{aligned}\forall t \in ET\ \forall e \in CEF \forall e' \in CEF(&t \neq Sp \wedge\ t \neq e.CEfl\\ &\wedge e.CEfl \neq Sp \wedge e.CEfl \neq e'.CEfl)\\ &)\end{aligned}$$

17. There is only one diagram labelled with the system process name, and all the event flows of the Context Diagram appear on the EFD of this diagram,

$$\begin{aligned}&\exists_1 n \in N(n.Ln = Sp) \wedge \forall e \in CEF(\\ &\quad (e.CEfs \in ET \Longrightarrow\\ &\qquad \exists_1 e' \in n.EFD.EF\ \exists_1 p' \in n.EFD.EP\\ &\qquad\qquad (e' = (Inherited, p', e.CEfl))\\ &\quad)\\ &\quad \wedge(e.CEfd \in ET \Longrightarrow\\ &\qquad \exists_1 e' \in n.DFD.DF\ \exists_1 p' \in n.EFD.EP\\ &\qquad\qquad (e' = (p', Inherited, e.CEfl))\\ &\quad)\\ &\qquad)\end{aligned}$$

C.2.2 Rules For Hierarchy Levels

18. The name of a diagram, Ln, is the name of a (higher level) process (which can be the system process),

$$\forall n \in N(\exists n' \in N(Ln \in n'.DFD.DP) \vee\ Ln = Sp)$$

19. The process sets of the *DFD* and *EFD* for a level are identical, and non-empty,

$$\forall n \in N(n.DFD.DP = n.EFD.EP \land\ n.DFD.DP \neq \phi)$$

20. Every process must have at least one (data or event) input and one (data or event) output,

$$\begin{array}{l} \forall p \in DP(\\ \qquad\qquad (\exists d \in DF(d.Dfd = p) \lor\ \exists e \in EF(e.Efd = p)) \\ \qquad\qquad \land\ (\exists d \in DF(d.Dfs = p) \lor\ \exists e \in EF(e.Efs = p)) \\ \quad) \end{array}$$

21. The subsystems on a *SCD* are a subset of the processes on the corresponding *DFD*/*EFD*,

$$\forall ss \in SCD(ss.Sn \in DP)$$

22. The set of processes on the *DFD* and *EFD* of a diagram level can be further subdivided into atomic and expandable processes.

$$DP = ADP \cup\ EDP$$

$$ADP \cap\ EDP = \phi$$

$$EP = AEP \cup\ EEP$$

$$EDP \cap\ EEP = \phi$$

where ADP = The set of atomic data process names,
EDP = The set of expandable data process names,
ADP = The sèt of event atomic process names, and
EDP = The set of expandable event process names,

23. Every expandable process has an expansion in the set of diagram networks,

$$\forall p \in EDP(\exists_1 n \in N(n.Ln = p))$$

24. Every atomic process has *one and only one* active input event flow,

$$\forall p \in AEP(\exists_1 e \in EF(e.Efd = p \land\ e.Efs \notin ES))$$

25. In a similar fashion to its processes, the data stores, event stores, data flows, and event flows of a diagram level can each be divided into two sets,

$$DS = IDS \cup\ LDS$$

$$IDS \cap\ LDS = \phi$$

$$ES = IES \cup\ LES$$

$$IES \cap LES = \phi$$
$$DF = IDF \cup LDF$$
$$IDF \cap LDF = \phi$$
$$EF = IEF \cup LEF$$
$$IEF \cap LEF = \phi$$

where IDS = The set of inherited data store names,
LDS = The set of local data store names,
IES = The set of inherited event store names,
LES = The set of local event store names,
IDF = The set of inherited data flows,
LDF = The set of local data flows,
IEF = The set of inherited event flows, and
LEF = The set of local event flows

26. A data flow cannot connect a node to itself,

$$\forall d \in DF(d.Dfs \neq d.Dfd)$$

27. A data flow cannot directly connect two data stores,

$$\forall d \in DF \neg(d.Dfs \in DS \wedge d.Dfd \in DS)$$

28. All DFD names are unique,
$$\forall p \in DP\ \forall s \in DS\ \forall d \in DF \forall d' \in DF(s \neq p \wedge s \neq d.Dfl \\ \wedge d.Dfl \neq p \wedge d.Dfl \neq d'.Dfl) \\ )$$

29. Every data store and its connections to an expandable process appear on the DFD of that process's expansion,

$$\begin{aligned}&\forall s \in DS\ \forall d \in DF\ \forall p \in EDP(\\ &\quad ((d.Dfs = s \wedge d.Dfd = p) \Longrightarrow \\ &\qquad \exists_1 n \in N\ \exists p' \in n.DFD.DP\ \exists_1 d' \in n.DFD.DF \\ &\qquad\quad (s \in n.DFD.IDS \wedge d'.Dfs = s \wedge d'.Dfd = p') \\ &\quad) \\ &\quad \wedge ((d.Dfs = p \wedge d.Dfd = s) \Longrightarrow \\ &\qquad \exists_1 n \in N\ \exists p' \in n.DFD.DP\ \exists_1 d' \in n.DFD.DF \\ &\qquad\quad (s \in n.DFD.IDS \wedge d'.Dfs = p' \wedge d'.Dfd = s) \\ &\quad) \\ &)\end{aligned}$$

30. Conversely, every *Inherited* data store and its connections to the processes of a DFD are connected to the DFD's parent process,

$$\begin{aligned}&\forall s \in IDS\ \forall d \in DF(\\ &\quad (d.Dfs = s \Longrightarrow \exists_1 n \in N\ \exists_1 p \in n.DFD.DP\ \exists d' \in n.DFD.DF\end{aligned}$$

```
        $(p = Ln \wedge\ s \in n.DFD.DS \wedge\ d'.Dfs = s)$
    )
    $\wedge\ (d.Dfd = s \Longrightarrow\ \exists_1 n \in N\ \exists_1 p \in n.DFD.DP\ \exists d' \in n.DFD.DF$
        $(p = Ln \wedge\ s \in n.DFD.DS \wedge\ d'.Dfd = s)$
     )
                )
```

31. Every data flow that is connected to an expandiable process appears on the *DFD* of that process's expansion,

```
$\forall d \in DF\ \forall p \in EDP($
    $d.Dfs = p \Longrightarrow\ \exists_1 n \in N\ \exists p' \in n.DFD.DP\ \exists d' \in n.DFD.DF$
                $(n.Ln = p \wedge\ d' = (p', Inherited, d.Dfl))$
    $\wedge\ d.Dfd = p \Longrightarrow\ \exists_1 n \in N\ \exists p' \in n.DFD.DP\ \exists d' \in n.DFD.DF$
                $(n.Ln = p \wedge\ d' = (Inherited, p', d.Dfl))$
                )
```

32. Conversely, every *Inherited* data flow on a *DFD* is connected to its parent process (which can be the Context Diagram),

```
$\forall d \in IDF($
    $d.Dfs = Inherited \Longrightarrow$
        $(\exists_1 n \in N\ \exists p \in n.DFD.DP\ \exists_1 d' \in n.DFD.DF$
                $(p = Ln \wedge\ d'.Dfd = p \wedge\ d'.DFl = d.Dfl)$
        $\vee\ \exists d' \in CD.CDFD.CDF(d'.Dfd = Sp \wedge\ d'.Dfl = d.Dfl)$
        )
    $\wedge\ d.Dfd = Inherited \Longrightarrow$
        $(\exists_1 n \in N\ \exists p \in n.DFD.DP\ \exists_1 d' \in n.DFD.DF$
                $(p = Ln \wedge\ d'.Dfs = p \wedge\ d'.DFl = d.Dfl)$
        $\vee\ \exists d' \in CD.CDFD.CDF(d'.Dfs = Sp \wedge\ d'.Dfl = d.Dfl)$
        )
    )
```

33. An event flow *may* connect an atomic process to itself. Hence the first data flow rule does not apply to event flows. It must be modified for event flows: an event flow cannot connect an expandable process or a event store to itself,

$$\forall e \in EF\ \forall p \in EEP\neg(e.Efs = p \wedge\ e.Efd = p)$$

$$\forall e \in EF\ \forall s \in ES\neg(e.Efs = s \wedge\ e.Efd = s)$$

34. All EFD names are unique,

```
$\forall p \in EP\ \forall s \in ES\ \forall e \in EF \forall e' \in EF(s \neq p \wedge\ s \neq e.Efl$
                                        $\wedge\ e.Efl \neq p \wedge e.Efl \neq e'.Efl)$
                                    )
```

35. Every event store and its connections to an expandable process appear on the *EFD* of that process's expansion,

```
$\forall s \in ES\ \forall e \in EF\ \forall p \in EEP($
    $((e.Efs = s \wedge\ e.Efd = p) \Longrightarrow$
```

$$
\begin{array}{l}
\qquad\qquad \exists_1 n \in N\ \exists p' \in n.EFD.EP\ \exists_1 e' \in n.EFD.EF \\
\qquad\qquad\qquad (s \in n.EFD.IES \wedge\ e'.Efs = s \wedge\ e'.Efd = p') \\
\quad) \\
\quad \wedge\ ((e.Efs = p \wedge\ e.Efd = s) \implies \\
\qquad\qquad \exists_1 n \in N\ \exists p' \in n.EFD.EP\ \exists_1 e' \in n.EFD.EF \\
\qquad\qquad\qquad (s \in n.EFD.IES \wedge\ e'.Efs = p' \wedge\ e'.Efd = s) \\
\quad) \\
\qquad\qquad)
\end{array}
$$

36. Conversely, every *Inherited* event store and its connections to the processes of a EFD are connected to the EFD's parent process,

$$
\begin{array}{l}
\forall s \in IES\ \forall e \in EF(\\
\quad (e.Efs = s \implies \exists_1 n \in N\ \exists_1 p \in n.EFD.EP\ \exists e' \in n.EFD.EF \\
\qquad (p = Ln \wedge\ s \in n.EFD.ES \wedge\ e'.Efs = s) \\
\quad) \\
\quad \wedge\ (e.Efd = s \implies \exists_1 n \in N\ \exists_1 p \in n.EFD.EP\ \exists e' \in n.EFD.EF \\
\qquad (p = Ln \wedge\ s \in n.EFD.ES \wedge\ e'.Efd = s) \\
\quad) \\
\qquad)
\end{array}
$$

37. The event flow balancing rules are slightly different to those for data flows. Every event flow output from an expandable process appears on that process's EFD expansion; and every event flow input to an expandiable process appears either on the EFD or the SCD of that process's expansion,

$$
\begin{array}{l}
\forall e \in EF\ \forall p \in EEP(\\
\quad e.Efs = p \implies \exists_1 n \in N\ \exists p' \in n.EFD.EP\ \exists e' \in n.EFD.EF \\
\qquad\qquad\qquad (n.Ln = p \wedge\ e' = (p', Inherited, e.Efl)) \\
\quad \wedge\ e.Efd = p \implies \\
\qquad (\exists_1 n \in N\ \exists p' \in n.EFD.EP\ \exists e' \in n.EFD.EF\ \exists ss \in n.SCD \\
\qquad\qquad (n.Ln = p \wedge\ (e' = (Inherited, p', e.Efl) \\
\qquad\qquad\qquad \vee\ (e.Efl \in ss.ON \bigtriangledown\ e.Efl \in ss.OFF) \\
\qquad\qquad) \\
\qquad\qquad)
\end{array}
$$

where $\bigtriangledown$ indicates an *exlusive or.*

38. Conversely, every *Inherited* input event flow on an EFD is connected to its parent process (which can be the Context Diagram); and every *Inherited* output event flow is either connected to its parent process (which can be the context diagrsm) or appears on the current SCD,

$$
\begin{array}{l}
\forall e \in IEF(\\
\quad e.Efs = Inherited \implies \\
\qquad (\exists_1 n \in N\ \exists p \in n.EFD.EP\ \exists e' \in n.EFD.EF \\
\qquad\qquad (p = Ln \wedge\ e'.Efd = p \wedge\ e'.EFl = e.Efl) \\
\qquad \vee\ \exists_1 e' \in CD.CEFD.CEF(e'.Efd = Sp \wedge\ e'.Efl = e.Efl) \\
\qquad)
\end{array}
$$

$$
\begin{array}{l}
\quad \wedge\ e.Efd = Inherited \Longrightarrow \\
\qquad ((\exists_1 n \in N\ \exists p \in n.EFD.EP\ \exists e' \in n.EFD.EF \\
\qquad\qquad (p = Ln \wedge\ e'.Efs = p \wedge\ e'.EFl = e.Efl) \\
\qquad\ \vee\ \exists_1 e' \in CD.CEFD.CEF(e'.Efs = Sp \wedge\ e'.Efl = e.Efl) \\
\qquad\ ) \\
\qquad \vee\ \exists ss \in SCD.SS(e.Efl \in ss.ON\ \triangledown\ e.Efl \in ss.OFF) \\
\qquad) \\
\qquad)
\end{array}
$$

39. A synchronisation symbol must have only one output and more than one input,

$$
\begin{array}{l}
\forall sc \in SC(\\
\qquad \exists_1 e \in EF\ \exists p \in AEP(e.Efs = sc \wedge\ e.Efd = p) \\
\qquad \wedge\ \exists e_1 \in EF\ \exists e_2 \in EF(e_1.Efd = sc \wedge\ e_2.Efd = sc \wedge e_1 \neq e_2) \\
\qquad)
\end{array}
$$

40. Every subsystem on a SCD must have a transition connected to it,

$$\forall ss \in SCD\neg(ON = \phi \wedge\ OFF = \phi)$$

41. Every transition on a SCD is either connected to the SCD's parent process, or is output from one of the processes on the current EFD,

$$
\begin{array}{l}
\forall ss \in SCD\ \forall tr \in ss.ON(\\
\qquad\qquad \exists_1 n \in N\ \exists p \in n.EFD.EP\ \exists e \in n.EFD.EF \\
\qquad\qquad\qquad\qquad (p = Ln \wedge\ e.Efd = p) \\
\qquad\qquad \vee\ \exists p \in EP\ \exists e \in EF(e.Efd = p) \\
\qquad\qquad\quad) \\
\forall ss \in SCD\ \forall tr \in ss.OFF(\\
\qquad\qquad \exists_1 n \in N\ \exists p \in n.EFD.EP\ \exists e \in n.EFD.EF \\
\qquad\qquad\qquad\qquad (p = Ln \wedge\ e.Efd = p) \\
\qquad\qquad \vee\ \exists p \in EP\ \exists e \in EF(e.Efd = p) \\
\qquad\qquad\quad)
\end{array}
$$

42. A subsystem cannot be enabled and disabled by the same event.

$$\forall ss \in SCD\ \forall tr_1 \in ss.ON\ \forall tr_2 \in ss.OFF(tr_1 \neq tr_2)$$

C.3 An Alternative Syntax

An alternative representation of a set of specification diagrams of the new notation can be given by using a textual language. Such a language can be formed by flattening the two dimensions of the diagrams into a (one-dimensional) text description. This language must be as clear as possible in its description of a particular set of diagrams. In order to achieve this clarity, the language constructs can simply divide each diagram type into its constituent symbol types, listing each instance of each symbol in a grouping. Each grouping can then be given a heading to make it easier to identify.

The following section gives the BNF description for such a language. This is followed by an example (textual) specification in this language. Note that this language can be used as a textual interface to an analysis tool as an alternative to a diagrammatic editor. Also note that the given BNF description gives a context free description of the language. The sentences of the language have to be parsed to ensure that a given specification conforms to the rules for a well formed set of diagrams. Context sensitive aspects of a given specification can be tested by applying the rules given in the predicates of the set representation.

The Language Syntax

```
< Spec. Diagrams >       ::= < Context Diagram > < Proc. Hierarchy >
< Context Diagram >      ::= Context Diagram : < Context DFD >
                             < Context EFD >
< Context DFD >          ::= Context DFD < Sys. Proc. Name >
                             Data Sources/Sinks : ( < Src/Sink List > )
                             Context Data Flows : ( < Context DF List > )
< Sys. Proc. Name >      ::= < Name String >
< Src/Sink List >        ::= < Optional Rep > < Src/Sink Name >
                          |  < Optional Rep > < Src/Sink Name > ,
                             < Src/Sink List >
< Src/Sink Name >        ::= < Name String >
< Optional Rep >         ::= < Empty >
                          |  < Positive Number >
< Context DF List >      ::= < Context DF >
                          |  < Context DF > , < Context DF List >
< Context DF >           ::= < Optional Rep > < One Way DF >
                          |  < Optional Rep > < Copied DF >
                          |  < Optional Rep > < Merged DF >
< One Way DF >           ::= ( < DF Source Name > , < DF Dest Name > ,
                             < DF Label > )
< Copied DF >            ::= ( < DF Source Name > , ( < DF Dest List > ) ,
                             < DF Label > )
< Merged DF >            ::= ( ( < DF Source List > ) , < DF Dest Name > ,
                             ( < DF Src Labels > ) )
< DF Source Name >       ::= < Name String >
< DF Dest Name >         ::= < Name String >
< DF Label >             ::= < Name String >
< DF Dest List >         ::= < DF Dest Name >
                          |  < DF Dest Name > , < DF Dest List >
< DF Source List >       ::= < DF Source Name >
                          |  < DF Source Name > , < DF Source List >
< DF Src Labels >        ::= < DF Label >
```

```
                            |    < DF Label > , < DF Src Labels >


< Context EFD >            ::=  Context EFD < Sys. Proc. Name >
                                Event Sources/Sinks : < Src/Sink List >
                                Context Event Flows : ( < Context EF List > )
< Context EF List >        ::=  < Context EF >
                            |    < Context EF > , < Context EF List >
< Context EF >             ::=  < Optional Rep > < One Way EF >
                            |    < Optional Rep > < Copied EF >
                            |    < Optional Rep > < Merged EF >
< One Way EF >             ::=  ( < EF Source Name > , < EF Dest Name > ,
                                < EF Label > )
< Copied EF >              ::=  ( < EF Source Name > , ( < EF Dest List > )
                                , < EF Label > )
< Merged EF >              ::=  ( ( < EF Source List > ) , < EF Dest Name >
                                , ( < EF Src Labels > ) )
< EF Source Name >         ::=  < Name String >
< EF Dest Name >           ::=  < Name String >
< EF Label >               ::=  < Name String >
                            |    < Bracketed Name >
< Bracketed Name >         ::=  [ < Name String > ]
< EF Dest List >           ::=  < EF Dest Name >
                            |    < EF Dest Name > , < EF Dest List >
< EF Source List >         ::=  < EF Source Name >
                            |    < EF Source Name > , < EF Source List >
< EF Src Labels >          ::=  < EF Label >
                            |    < EF Label > , < EF Src Labels >
< Proc. Hierarchy >        ::=  < Hierarchy Level >
                            |    < Hierarchy Level > < Proc. Hierarchy >


< Hierarchy Level >        ::=  < Level Name > < Data Flow Spec >
                                < Event Flow Spec >
< Level Name >             ::=  < Name String >
< Data Flow Spec >         ::=  Data Flow Diagram Processes : (
                                < Process List > )
                                < Data Store List >
                                Data Flows : ( < DFD DF List > )
< Process List >           ::=  < Optional Rep > < Process >
                            |    < Optional Rep > < Process > ,
                                < Process List >
< Process >                ::=  ( < Process Name > , < Process Number > ,
                                < Process Type > )
< Process Name >           ::=  < Name String >
< Process Number >         ::=  < Positive Number >
< Process Type >           ::=  Atomic
                            |    Expandable
```

```
< Data Store List >       ::= < Empty >
                            |   Data Stores : ( < Data Stores > )
< Data Store >            ::= < Optional Rep > < Data Store Name >
                            |   < Optional Rep > < Data Store Name > ,
                                < Data Stores >
< Data Store Name >       ::= < Name String >
< DFD DF List >           ::= < Data Flow >
                            |   < Data Flow > , < DFD DF List >
< Data Flow >             ::= < DFD One Way DF >
                            |   < DFD Two Way DF >
                            |   < DFD Combined DF >
                            |   < DFD Merged DF >
                            |   < DFD Copied DF >
                            |   < DFD Split DF >
< DFD One Way DF >        ::= < Store DF >
                            |   < Proc. 1-Way DF >
< Store DF >              ::= < TO Store DF >
                            |   < From Store DF >
< TO Store DF >           ::= ( < Process Name > , < Data Store Name > )
< From Store DF >         ::= ( < Data Store Name > , < Process Name > )
< Proc. 1-Way DF >        ::= ( < DFD DF Src Name > ,
                                < DFD DF Dst Name > ,
                                < DFD DF Label > )
< DFD DF Src Name >       ::= Inherited
                            |   < Name String >
< DFD DF Dst Name >       ::= Inherited
                            |   < Name String >
< DFD DF Label >          ::= < Name String >
< DFD Two Way DF >        ::= ( 2 , < Process Name > ,
                                < Data Store Name > )
< DFD Combined DF >       ::= ( ( < DFD DF Sources > ) ,
                                < DFD DF Dst Name > ,
                                ( < DF Src Labels > ) , < DF Dest Label > )
< DFD DF Sources >        ::= < DFD DF Src Name >
                            |   < DFD DF Src Name > , < DFD DF Sources >
< DF Src Labels >         ::= < DFD DF Label >
                            |   < DFD DF Label > , < DF Src Labels >
< DF Dest Label >         ::= < Name String >
< DFD Merged DF >         ::= ( ( < DFD DF Sources > ) ,
                                < DFD DF Dst Name > ,
                                < Merged DF Dest > )
< Merged DF Dest >        ::= < DFD DF Dst Name >
                            |   ( < DF Src Labels > )
< DFD Copied DF >         ::= ( < DFD DF Src Name > ,
                                ( < DFD DF Dests > ) ,
                                < DFD DF Label > )
< DFD DF Dests >          ::= < DFD DF Dst Name >
                            |   < DFD DF Dst Name > , < DFD DF Dests >
```

```
< DFD Split DF >        ::= ( < DFD DF Src Name > , (
                            < DFD DF Dests > ) ,
                            ( < DF Dest Labels > ) )
< DF Dest Labels >      ::= < DFD DF Label >
                        |   < DFD DF Label > , < DF Dest Labels >


< Event Flow Spec >     ::= Event Flow Diagram Processes :
                            ( < Process List > )
                            < Event Store Lst >
                            < Synch Lits >
                            Events Flows : ( < EFD EF List > )
                            < Optional SCD >
< Event Store Lst >     ::= < Empty >
                        |   Event Stores : ( < Event Stores > )
< Event Stores >        ::= < Optional Rep > < Event Store Nam >
                        |   < Optional Rep > < Event Store Nam > ,
                            < Event Stores >
< Event Store Nam >     ::= < Name String >
< Synch List >          ::= < Empty >
                        |   Synchs : ( < Synch List > )
< Synch List >          ::= < Optional Rep > < Synch Name >
                        |   < Optional Rep > < Synch Name > ,
                            < Synch List >
< Synch Name >          ::= Synch < Positive Number >
< EFD EF List >         ::= < Event Flow >
                        |   < Event Flow > , < EFD EF List >
< Event Flow >          ::= < EFD One Way EF >
                        |   < EFD Two Way EF >
                        |   < EFD Copied EF >
                        |   < EFD Merged EF >
< EFD One Way EF >      ::= < Store EF >
                        |   < Proc. 1-Way EF >
                        |   < Synch 1-Way EF >
< Store EF >            ::= < To Store EF >
                        |   < From Store EF >
< To Store EF >         ::= ( < Process Name > , < Event Store Nam > )
< From Store EF >       ::= ( < Event Store Nam > , < Process Name > )
< Proc. 1-Way EF >      ::= ( < EFD EF Src Name > ,
                            < EFD EF Dest Nam > ,
                            < EFD EF Label > )
< Synch 1-Way EF >      ::= < To Synch EF >
                        |   < From Synch EF >
< To Synch EF >         ::= ( < EFD EF Src Name > , < Synch Name > ,
                            < EFD EF Label > )
< From Synch EF >       ::= ( < Synch Name > , < Process Name > )
                        |   ( < Synch Name > , < Process Name > ,
                            < EFD EF Label > )
< Synch Name >          ::= < Name String >
< EFD EF Src Name >     ::= Inherited
                        |   < Name String >
< EFD EF Dest Nam >     ::= Inherited
                        |   < Name String >
< EFD EF Label >        ::= < Name String >
                        |   < Bracketed Name >
< EFD Two Way EF >      ::= ( 2 , < Process Name > , < Event Store Nam > )
< EFD Copied EF >       ::= ( < EFD EF Src Name > , ( < EFD EF Dests > ) ,
                            < Copied EF Label > )
< EFD EF Dests >        ::= < EFD EF Dest Nam >
                        |   < EFD EF Dest Nam > , < EFD EF Dests >
< Copied EF Label >     ::= < CEF Src Label >
                        |   < CEF Src Label > , ( < CEF Dest Labels > )
                        |   ( < CEF Dest Labels > )
< CEF Src Label >       ::= < EFD EF Label >
< CEF Dest Labels >     ::= < EFD EF Label >
                        |   < EFD EF Label > , < CEF Dest Labels >
< EFD Merged EF >       ::= ( ( < EFD EF Src List > ) ,
                            < EFD EF Dest Nam > ,
                            ( < Merged EF Labels > ) )
< EFD EF Src List >     ::= < EFD EF Src Name >
                        |   < EFD EF Src Name > , < EFD EF Src List >
< Merged EFD Labels >   ::= < EFD EF Label >
                        |   < EFD EF Label > , < Merged EFD Labels >


< Optional SCD >        ::= < Empty >
                        |   Subsystems : ( < Subsys List > )
< Subsys List >         ::= < Subsystem >
                        |   < Subsystem > , < Subsys List >
< Subsystem >           ::= < Optional Rep > ( < Subsystem Name > ,
                            < Subsystem No > ,
                            Enabled : ( < Enable Events > )
                            Disabled : ( < Disable Events > )
< Subsystem Name >      ::= < Name String >
< Subsystem No >        ::= < Positive Number >
< Enable Events >       ::= < Enable Event >
                        |   < Enable Event > , < Enable Events >
< Enable Event >        ::= ( < Event Name > , < Enablement Type > )
< Event Name >          ::= < Name String >
                        |   < Bracketed Name >
< Enablement Type >     ::= E
                        |   R
< Disable Events >      ::= < Disable Event >
                        |   < Disable Event > , < Disable Events >
< Disable Event >       ::= ( < Event Name > , < Disablement Type > )
< Enablement Type >     ::= H
```

```
                              |    F
                              |    S
< Name String >               ::=  < Name >
                              |    < Name > < Name String >
< Name >                      ::=  < Letter > < Name Tail >
< Name Tail >                 ::=  < Empty >
                              |    < Letter > < Name Tail >
                              |    < Digit > < Name Tail >
                              |    _ < Name Tail >


< Letter >                    ::=  a
                              |    b
                              |    c
                              |    d
                              |    e
                              |    f
                              |    g
                              |    h
                              |    i
                              |    j
                              |    k
                              |    l
                              |    m
                              |    n
                              |    o
                              |    p
                              |    q
                              |    r
                              |    s
                              |    t
                              |    u
                              |    v
                              |    w
                              |    x
                              |    y
                              |    z
                              |    A
                              |    B
                              |    C
                              |    D
                              |    E
                              |    F
                              |    G
                              |    H
                              |    I
                              |    J
                              |    K
```

```
                        |  L
                        |  M
                        |  N
                        |  O
                        |  P
                        |  Q
                        |  R
                        |  S
                        |  T
                        |  U
                        |  V
                        |  W
                        |  X
                        |  Y
                        |  Z
< Digit >               ::= 0
                        |  1
                        |  2
                        |  3
                        |  4
                        |  5
                        |  6
                        |  7
                        |  8
                        |  9
< Positive Number >     ::= < Digit >
                        |  < Digit > < Positive Number >
< Empty >               ::=
```

An Example Specification

This section gives an example specification in the above language for the petrol station system.

```
Context Diagram :
 Context DFD
   Petrol Station System
   Data Sources/Sinks : ( Console Display, Receipt Printer,
                          Attendant, Supervisor, Report Printer,
                          3 Pump
                        )
   Context Data Flows : (
        (Attendant, Petrol Station System , Stock Delivery),
        ( (Attendant, Supervisor), Petrol Station System,
          Report Request
        ),
        (Supervisor, Petrol Station System, Code),
        (Supervisor, Petrol Station System, New Prices),
        (Petrol Station System, Report Printer, Report),
        (Petrol Station System, Pump, Price Changes),
        (Pump, Petrol Station System, Transaction Details),
        3 (Petrol Station System, Receipt Printer, Receipt),
        3 (Petrol Station System, Console Display,
           Transaction Display
          ),
        (Petrol Station System, Console Display,
         Stock Display
        )
                      )
Context EFD
  Petrol Station System
  Context Event Flows : (
        (Attendant, Petrol Station System ,
         [Stock Delivery]
        ),
        ((Attendant, Supervisor), Petrol Station System,
         [Report Request]
        ),
        3 (Attendant, Petrol Station System, Button Pressed),
        3 (Attendant, Petrol Station System,
           Receipt Request
          ),
        (Attendant, Petrol Station System, Take Stock),
        (Attendant, Petrol Station System, On),
        (Attendant, Petrol Station System, Off),
        (Supervisor, Petrol Station System, [Code]),
        (Supervisor, Petrol Station System, [New Prices]),
        (Petrol Station System, Report Printer, [Report]),
        (Petrol Station System, Pump, [Price Changes]),
        (Pump, Petrol Station System, [Transaction Details]),
        (Petrol Station System, Pump, Enable Pump),
        (Pump, Petrol Station System, Service Request),
        3 (Petrol Station System, Receipt Printer,
           [Receipt]
          ),
        (Petrol Station System, Console Display,
         Code Verified
        ),
        3 (Petrol Station System, Console Display, Bell),
        3 (Petrol Station System, Console Display, Light On),
        3 (Petrol Station System, Console Display,
           Light Off
          )
```

```
                        )


Petrol Station System
 Data Flow Diagram
   Processes : ( 3 (Monitor Pump Operation, 0, Expandable),
                 (Monitor Stock, 1, Expandable),
                 (Change Prices, 2, Expandable),
                 (Print Reports, 3, Atomic)
               )
   Data Stores : (Transaction History, Current Stock)
   Data Flows : (
         (Monitor Pump Operation, Inherited, Receipt),
         (Monitor Pump Operation, Inherited, Transaction Display),
         (Inherited, Monitor Pump Operation, Transaction Details),
         (Monitor Pump Operation, Transaction History),
         (2, Monitor Pump Operation, Current Stock),
         (Maintain Stock, Inherited, Stock Display),
         (Inherited, Maintain Stock, Stock Delivery),
         (Current Stock, Maintain Stock),
         (2, Maintain Stock, Current Stock),
         (Inherited, Change Prices, Code),
         (Inherited, Change Prices, New Prices),
         (Change Prices, Inherited, Price Changes),
         (2, Change Prices, Current Stock),
         (Print Report, Inherited, Report),
         (Inherited, Print Report, Report Request),
         (Current Stock, Print Report),
         (Transaction History, Print Report)
               )
Event Flow Diagram
   Processes : ( 3 (Monitor Pump Operation, 0, Expandable),
                 (Monitor Stock, 1, Expandable),
                 (Change Prices, 2, Expandable),
                 (Print Reports, 3, Atomic)
               )
   Event Flows : (
         (Monitor Pump Operation, Inherited, [Receipt]),
         (Inherited, Monitor Pump Operation, [Transaction Details]),
         (Monitor Pump Operation, Inherited, Bell),
         (Monitor Pump Operation, Inherited, Light On),
         (Monitor Pump Operation, Inherited, Light Off),
         (Inherited, Monitor Pump Operation, Service Request),
         (Inherited, Monitor Pump Operation, Receipt Request),
         (Inherited, Monitor Pump Operation, [Transaction Details]),
         (Inherited, Monitor Pump Operation, Take Stock),
         (Inherited, (Monitor Pump Operation, Maintain Stock),
          [Stock Delivery], (Stock Delivery Complete)
         ),
         (Inherited, Change Prices, [Code]),
         (Inherited, Change Prices, [New Prices]),
         (Change Prices, Inherited, [Price Changes]),
         (Change Prices, Inherited, Code Verified),
         (Print Report, Inherited, [Report]),
         (Inherited, Print Report, [Report Request])
                 )
 Subsystem Control Diagram
   Subsystems  : ( 3 (Monitor Pump Operation, 0,
                           Enabled : ( (On, E) )
                           Disabled : ( (Off, H) )
                      ),
                    (Maintain Stock, 1,
                           Enabled : ( (On, E) )
                           Disabled : ( (Off, H) )
                    ),
                    (Change Prices, 2,
                           Enabled : ( (On, E) )
                           Disabled : ( (Off, H) )
                    ),
                    (Print Report, 3,
                           Enabled : ( (On, E),
                                       (Stock Delivery Complete,
                                        E
                                       )
                                     )
                           Disabled : ( (Off, H) ,
                                        (Take Stock, F)
                                      )
                    ),

Monitor Pump Operation
 Data Flow Diagram
   Processes : ( (Request Service, 0, Atomic),
                 (Start The Pump, 1, Atomic),
                 (Record Transaction, 2, Atomic),
                 (Update Transaction History, 3, Atomic),
                 (Print Receipt, 4, Atomic),
                 (Check Pump Status, 5, Atomic)
               )
   Data Stores : (Transaction History, Current Stock,
                  Current Transaction, Last Transaction)
   Data Flows : (
         (Inherited, Record Transaction, Transaction Details),
         (Record Transaction, Inherited, Transaction Display),
         (Record Transaction, Current Transaction),
         (2, Record Transaction, Current Stock),
         (Current Stock, Update Transaction History),
         (Update Transaction History, Transaction History),
         (Update Transaction History, Last Transaction),
         (Print Receipt, Inherited, Receipt),
         (Last Transaction, Print Receipt)
                )

Event Flow Diagram
   Processes : ( (Request Service, 0, Atomic),
                 (Start The Pump, 1, Atomic),
                 (Record Transaction, 2, Atomic),
                 (Update Transaction History, 3, Atomic),
                 (Print Receipt, 4, Atomic),
                 (Check Pump Status, 5, Atomic)
               )
   Event Stores : (Pending Request, Pump Idle)
   Synchs : (Synch 1, Synch 2)
   Event Flows : (
         (Inherited, Request Service, Service Request),
         (Request Service, (Inherited, Inherited),
          (Light On, Bell)
         ),
         (Request Service, Pending Request),
         (Start The Pump, (Inherited, Inherited),
          (Light Off, Enable Pump)
         ),
         (Inherited, Record Transaction, [Transaction Details]),
         (Record Transaction, (Inherited, Inherited,
          Synch 2
         ),
          (Light On, Bell, Delivery Complete)
         ),
         (Synch 1, Update Transaction History),
         (Update Transaction History, (Inherited, Synch 2),
          Transaction Recorded, (Light Off)
         ),
         (Synch 2, Print Receipt),
         (Print Receipt, Inherited,[Receipt]),
         (Inherited, Synch 2, Receipt Request),
         (Inherited, Check Pump Status, Pump Button Pressed),
         (Check Pump Status, Start The Pump, Start Pump),
         (Check Pump Status, (Start Pump, Transaction Complete),
          Start New Transaction
         ),
         (Check Pump Status, Synch 1, Transaction Complete),
         (2, Check Pump Status, Pending Request),
         (2, Check Pump Status, Pump Idle)
                )
```

```
 Subsystem Control Diagram
   Subsystems : ( (Start The Pump, 1,
                    Enabled : ( (Stock Delivery Complete),
                                E)
                              )
                    Disabled : ( (Take Stock, F) )
                )

Maintain Stock
 Data Flow Diagram
   Processes : ( (Clock, 0, Atomic),
                 (Monitor Stock, 1, Atomic),
                 (Record Stock Delivery, 2, Atomic)
               )
   Data Stores : (Current Stock, Threshold)
   Data Flows : (
         ((Monitor Stock, Record Stock Delivery), Inherited,
          Stock Display
         ),
         (Threshold, Monitor Stock),
         (Current Stock, Monitor Stock),
         (Inherited, Record Stock Delivery, Stock Delivery),
         (2, Record Stock Delivery, Current Stock)
                )
 Event Flow Diagram
   Processes : ( (Clock, 0, Atomic),
                 (Monitor Stock, 1, Atomic),
                 (Record Stock Delivery, 2, Atomic)
               )
   Event Flows : (
         (Clock, (Clock, Monitor Stock), Tick),
         (Inherited, Record Stock Delivery, [Stock Delivery])
                 )

Change Prices
 Data Flow Diagram
   Processes : ( (Verify Code, 0, Atomic),
                 (Accept Price Changes, 1, Atomic)
               )
   Data Stores : (Current Stock, Supervisor's Code)
   Data Flows : (
         (Inherited, Verify Code, Code),
         (Supervisor's Code, Verify Code),
         (Inherited, Accept Price Changes, New Prices),
         (Accept Price Changes, Inherited, Price Changes),
         (2, Accept Price Changes, Current Stock)
                )
 Event Flow Diagram
```

```
Processes : ( (Verify Code, 0, Atomic),
              (Accept Price Changes, 1, Atomic)
            )
Synchs : (Synch 1)
Event Flows : (
      (Inherited, Verify Code, [Code]),
      (Verify Code, (Synch 1, Inherited), Code Verified),
      (Inherited, Synch 1, [New Prices]),
      (Synch 1, Accept Price Changes),
      (Accept Price Changes, Inherited, [Price Changes])
              )
```