Chapter 2

Business processes, problems and solutions

Summary: Problems and solutions, design methods, business process models, information systems, requirements capture, system specification, dynamic models, the problem of real time, examples of X-machine specifications, microprocessors. Implementation.

With the ever expanding areas of applications of computers today it is often easy to loose sight of the essential similarities that exist between all of these systems. The common theme in many endeavours is one of a problem with a number of important and critical features and a proposed solution constructed on the basis of a computer programme or suite of programs with the desired behaviour. The critical issue, as we have mentioned in the introduction, is the relationship between the proposed solution and the understanding of the “problem’s originator” as to whether the proposed solution does, in fact, provide the desired answer.

We will refer to the person with the problem as the *client*, he or she has a problem to be solved - and the solution provider, the *designer*, must endeavour to achieve an *acceptable* solution. What exactly these terms mean will be the subject of much what will follow. In the course of examining these issues we will explore some of the myths, beliefs and established wisdom about the design of software systems that percolate through the subject.

2.1 Problems and solutions

Our starting point is the following: the clients (and the prospective users of any solution) are the *most important individuals* in the exercise. It is only by understanding *their needs* and *their objectives* that the designers can achieve their’s - which should be client satisfaction and high user productivity.

In order to support this we must ensure that the communication between clients, users and designers is such that all can contribute to the construction of an appropriate solution in a cost effective way. It has been widely commented that one of the key causes of the failure of a software design project is that lack of understanding by the designers of the clients requirements. It could also be mentioned that the client may not be fully appreciative of their problem, let alone a possible solution. We are thus brought to the position where we must find some effective mechanism to enable us to describe and understand the client’s problem before we can even contemplate the construction of a solution. Our first starting point is to examine the context of the client’s problem area and to build a *simple business process model*. In our main example it is clearly oriented towards a common type of business application that may be solved by constructing a suitable information system of the type commonly encountered in almost every office. Later in the chapter we will use similar techniques to look at a different type of problem domain where *time* is a much more critical aspect of the situation and there is a less obvious business dimension.

2.2 Design methods - old and new

Much has been written about business processes and business process models. Various notations, languages and tools have been proposed to try to capture the essence of the business process and these have often been related to design methodologies and analysis techniques to better enable software systems to be built for business applications. Many of these techniques and notations have their origin in software engineering, they are the attempts of computer professionals to try to describe the essentials of a business process in their own terms and notations. The supposed benefits of this approach is that the information so gleaned is then suitable for use in the “traditional” software design approaches, (e.g.[2]) including, in recent years the object oriented approach to software production ([12]).

There is a continuing problem with building software systems that provide what the client wants. A great deal of effort has been expended on developing requirements capture notations, methods and tools without much impact on the problem as a whole which is primarily one of communication between the client and his or her business and the designer and implementer. There are a number of reasons for this. One is the conceptual gap that develops not only between the client and the designer but also between the designer and the developer and the tester or quality assurance process. Many of the design methods that are used reinforce this latter gap in unnecessary ways. For example, the common design methods and tools focus on data and processes in a loose way and ignore the issue of algorithm. Thus the opportunity for automatic generation of code for a thoroughly analysed design are limited. To overcome this problem some tools, for example CASE tools [2], [44], allow the designer to annotate processes with code to represent the algorithm, which seems to defeat the object since full analysis is then still not possible with our current techniques. Furthermore, without the algorithm being made specific the opportunities to carry out detailed and well-founded testing are reduced.

Even attempts to improve the matter through the use of formal methods and notations, [16], [17], [18] often suffer from similar problems. These focus on notations and concepts that could provide a vehicle for describing and analysing systems mathematically and thus rigorously. It is a commonly expressed philosophy amongst formal methods practitioners that one should not consider any implementation issues and that implicit specification is preferred to an explicit algorithm. An algorithm does not mean an implementation or a piece of code, it is a mathematical formula which describes in terms of the declared data types the outcome of the process in terms of its inputs. This is something that can be used for analysis, automatic test generation and code construction.

All computer software systems are examples of computational models. If the design languages do not create such models as a by-product of their use then there will be left a significant barrier that needs to be negotiated before the successful construction of a working system. In its simplest form this barrier is represented by the need to construct an algorithm that will implement the function defined during the data analysis and functional decomposition stages of the analysis and design process. Most object oriented design methods also meet with this obstacle. What this means is that the construction of the algorithms has been separated from the rest of the design process and left to programmers who may not fully understand the problem. The programmers will have to develop code that seems to implement the informal description that they have been given but this is a weak link in the chain, as well as being unnecessary. As part of our integrated approach to design and test we will be putting the algorithm back into the heart of the activity. Before we do, however, we will consider some of the approaches currently favoured for software engineering.

Traditional design approaches.

Many traditional design methods are based on data analysis. This involves identifying the way that data flows through the system, which process needs which data and how the data needs to be organised to enable efficient access and query systems. The methodologies have involved diagramming techniques such as data flow diagrams; entity relationship diagrams, which try to describe the interactions between different types of data; entity life histories, which show how data elements are created and destroyed during the operation of the system and finite state machines. In none of these is it possible to construct a coherent, integrated model of the system that could be analysed and used as a basis for full code generation or functional test set construction. The structured analysis and design approach, in its many variants and embedded into many different software life cycle models ranging from the waterfall, through incremental delivery [7] to Rapid Application Development [45] have failed to deliver the quality of solution that is required. The costs of developing systems using these methods has become uneconomic, especially now that amount of effort needed for testing has escalated to between 50% and 90% of many projects. Couple this with the observation that even if the systems more or less work they may not be the solution to the problem - which may have changed during the long slow process of construction anyway.

The classical approach of *relational data analysis*, [46] while serving an important purpose in the days where all applications were extending the capabilities of slow computers with small areas of storage, is probably no longer necessary. For many applications nowadays the capabilities of the modern microprocessor and the availability of cheap and fast storage result in their being much less of a speed or memory problem. It is possible to organise the data as a flat or linear file, nowadays, without any serious loss of performance in searching and querying operations. It is the limitations of the human operator in terms of speed of reaction in carrying out these operations that we are now reaching. There are also considerable benefits in relaxing the *relational tyranny* in the design of databases when it comes to one of the major challenges of software engineering - the easy maintenance and redevelopment of systems that may have been solutions of problems that time has rendered obsolete.

How can businesses keep paying for their databases to be redesigned whenever their business environment changes? One solution may be to empower the businessman or woman so that he/she can configure and reconfigure their system to their needs without having to explain their requirements to a software analyst. But how could this be done? We cannot expect them to learn how to program, they must be given sensible metaphors, notations and tools that will help them to model their business processes in a convenient way and which will generate correct implementations, painlessly. This is an important issue which we will return to.

Another possible approach, related to the philosophy outlined later in relation to software systems in general, is that of components. Why cannot we have modular database systems where new standard database components can be bolted on in a robust way when the need arises without having to to go through an expensive reanalysis and design process?

Object oriented design.

Object oriented programming languages, [13, 14], were invented by programmers who were looking for a mechanism that saved them from having to keep writing very similar code for different applications. It has caught the imagination of many and has lead to the introduction of a number of design methodologies to support it. One of the driving forces behind it is the idea of *software reuse*. Sadly this is a poorly understood phenomenon. The principal is that the generic framework of commonly used software segments can be identified as a system of classes that collaborate to provide some generic system architecture. A *class*, on the other hand, is a description of a collection of specific *objects* that carry out some coherent software function. These objects are convenient packages of data and process (methods) that provide a controlled mechanism for communicating and manipulating the data. Further, these classes can be placed in libraries and reused for similar purposes in different applications. The ideas have been taken further with the concept of *inheritance* whereby some of the features that belong to one class can be used to define other classes and in *polymorphism* which involves the generalisation of some functions to apply to many different types of data. The object metaphor enables the programmer to collect together into convenient bundles the data and its related processing functions (algorithms) in such a way that prevents the programmer from interfering with the internal workings of the object. This is seen as a good thing since one of the big problems with programming - this distinguishes software engineering from all other types of engineering[1] - is the ability the programmer has to change any part of the programme at any time prior to delivery thus creating the potential for disaster. Unfortunately the unstated implication of the reuse of classes was that this would reduce the amount of testing needed, for, after all, if the classes had been used previously with success in some other system then they would be successful in other applications. This is a very dangerous myth which we will come back to. Suffice it to say that many experts, for example, [15], comment that testing object oriented systems is considerably harder and more time consuming than testing traditional systems.

With the development of object oriented languages came a plethora of object oriented design methodologies, [47], [48]. Here the claim was that the use of objects was a natural human process and this could be utilised when it came to the design of systems and in the description of requirements. The client and the designer would find that they shared a common language which would assist in the understanding of the problem. In fact this is probably a case of *after the event justification* because the origins of the object are to be found in the theoretical work on abstract data types [49], an attempt to bring some mathematical order to the programming chaos of the 1970s, and in a more cognitive analysis of the ways humans approach the understanding of a problem using object-like metaphors such as an icon, window, stream etc. [50]. Over the subsequent years these ideas have crystalised into the object-oriented paradigms of today. At no stage was there any emphasis on the development of models of business processes and problems. Because of its origins in programming rather than in applications we are in danger of recycling object oriented solutions to the wrong problems. In [52] Sommerville and Sawyer remark on the use of object models in

1. In other types of engineering there is fabrication phase where the designer hands over the design to technicians who then built the artifact, this does not happen in software where the engineer can be involved all the way from initial concept to the installation of the final product.

requirements capture and system modelling: “... because they (object models) are based on independent objects, the models do not give a clear picture of end-to-end processing in the system.” They go on: “.. if you do not develop object-oriented programs, we do not recommend the use of object models. Making the transition from an object-oriented model to a system based on functions is not easy.”

With so many different approaches to design notations there has been an attempt to integrate some of the more popular techniques into a Unified Modelling Language [52] but the result looks more like a classic committee compromise solution than a coherent and usable mechanism for building a bridge between the problem and a solution. It is not a methodology, as such and much remains to be done before something emerges that can satisfy all the challenges that we are faced with. Existing attempts at constructing design methods still suffer from the problems of *algorithm neglect*, although things are beginning to change and the work of Selic [53] does address the issue of algorithm description through the use of state machines within objects.

Formal methods.

The issues of quality and correctness have also been approached through the use of *formal methods*. Here the solution is to develop rigorous techniques based on mathematical notations and to use these to specify the required system in a precise way which could then be analysed formally. It had a lot of good things going for it but has rarely delivered in practice. The problems arise in a number of ways. The notations were often outside the experience of many programmers and engineers who may have not studied discrete mathematics. There was something unfriendly about the languages that was to prove an enormous obstacle to their use in industry. The notations were just that - notations, they were not explicit methods and many engineers who managed to learn the notations could not build specifications using the languages successfully because they did not know how to organise the task effectively. The notations were supported by some of the poorest software tools one is likely to meet, tools which were built by formal methods practitioners who did not practice what they preached, the user interfaces, in almost all cases[1], were and are appalling! There was also the belief that by formally specifying the system and transforming this specification, using

1. The next generation of tools may be better, for example the VisualZ concept of an icon based Z specification environment inspired by some of the visual programming languages, has demonstrated that it is possible [54].

some mathematically based transformation mechanisms, into code would result in a correct system. This is, of course, nonsense, it is impossible to formalise *fully* all the aspects of the operating environment of the system, the operating system functions, the hardware etc. so as to ensure that it functions correctly. *Testing still has to be carried out,* an important factor at the heart of our approach.

One of the unfortunate developments in the subject is the philosophy that the specification must not be influenced by the likely implementation. The reason given is that this would become a distraction from the purity of thought required to construct a correct specification. So the style tends to be one of defining, in an implicit way, the properties of what a function should do rather than trying to construct a definition of the function. This has a number of consequences, firstly it is erecting another barrier between the specification and the implementation that has to be overcome, possibly with difficulty and secondly there may be important limitations that the implementation environment may impose on the solution, limitations that are not recognised in the specification. An example of this will be seen later.

The most positive aspects of the use of formal methods is that a specification, if correctly captured from the requirements, does offer a formal basis for analysing and understanding the proposed system better. It can be the basis for the construction of test sets but this can still be a problem as we will see later. The most damning indictment of formal methods is the difficulty of applying them to industrial sized projects despite the fact that during the last 10 years almost all of the country's computing graduates have been taught formal methods and large amounts of research money has been spent on "industrial demonstrator projects". There have been some successes but there must still be something wrong with the approach if it has had such a small impact.

We will now turn to trying to overcome some of these problems and describe how it is possible to put the client and his/her problem (including the users) at the centre of the stage and introduce a formal mechanism for the capturing of the problem, the analysis of the problem, the construction and the delivery of a *correct* solution. We will also lay the foundations for a genuine component based approach which does not suffer from the excessive testing demands of object oriented approaches, which is firmly based on formal computational modelling and is yet practical and effective.

There are a number of important criteria that need to be considered when thinking about requirements modelling, analysis and design, in order to ensure that we develop correct systems - systems which work and solve the right problem. We will approach these criteria from the point of view of what sort of product a problem and system modelling method must produce.

Principles for correct systems design.

The full model of the system or subsystem must be:

sufficiently close to the client's/user's view of the business process that misunderstandings can be avoided;
easily modified and developed in the light of changing requirements;
amenable to mathematical analysis;
suitable for the automatic generation of code;
suitable for the automatic functional test set generation.

Finally we should ask ourselves whether a fully computerised solution is actually needed. For example, in one problem brought to us by a client who owned a chain of retail shops the most obvious solution was not necessarily the best. The issue was concerned with keeping a record of the hours worked by a large number of part-time sales assistants. Many of these were of a mature age and were not familiar with computers, there was also a very high staff turnover. The initial solution of a program which each assistant was to use to log their hours worked relied on the assumption that they could be trained to use the system. Subsequent analysis rejected a computerised interface in favour of one involving a pencil and paper and the use of computer readable cards for the key data entry.

2.3 Business process models

There are many ways in which a business may be modelled some of which have been made into methods supported by tools of various types, e.g. [55]. There is, however, a lack of precision about these techniques which makes them unsuitable for the sort of analysis that we wish to carry out. We will introduce an alternative and simple method which can be used to derive detailed models of the systems that we wish to build to support the business process. Many processes share the following common features:

there is some *receipt* of information or data necessary for the process to operate;
there are some aspects of the context of the business, internal knowledge of some sort;
the process then *evaluates* the received data in the light of this context and produces or *generates* some resulting observable output, information or other product;

the context is *revised* or updated ready for the next time that the process operates.

Examples of this might include the process of updating a customer's records, whereby the new information about the customer is fed into the process, the database provides the context and the output is the confirmation through printout or screen display that the record has been updated. The database updating is the context being revised in the light of the process operating. Another, higher level, example might be a factory receiving orders and raw materials, producing some artefact and delivering this to a customer together with an invoice. The context will include information about the way the factory operates and all the planning, management and production control systems that are involved.

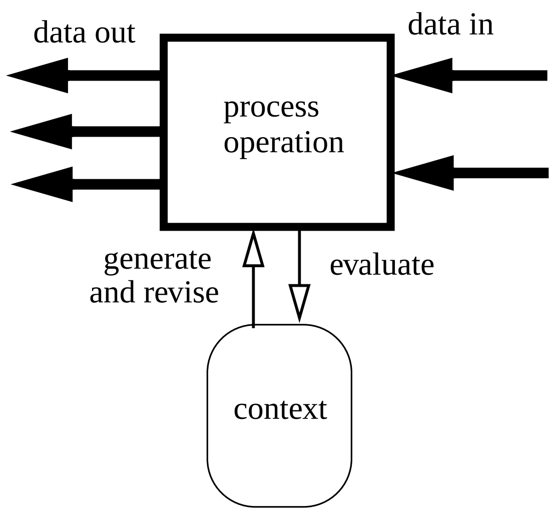

Fig. 2.1 A business process.

If we can identify a set *Input* of data that it is possible to supply the process, a set *Output* of possible output data and a set *Memory* consisting of the internal context or data associated with the process then we can model the business process as a *partial* function of the form:-

process: $Memory \times Input \to Output \times Memory$

In the cases we consider here the sets *Memory, Input* and *Output* are all discrete, and with the possible exception of the *Memory,* are finite.

Definition. 2.3.1 A **business process** is a partial function of the form:

$$\mathbf{process}: \mathit{Memory} \times \mathit{Input} \rightarrow \mathit{Output} \times \mathit{Memory}$$

where the sets *Memory, Input*, *Output* are defined for that specific process.

Integrating business processes.

A complete business will be made up of many interacting business process and so we have to construct a more general model that will integrate all of these functions. The organisation will be such that at any one time there will be some, many or no processes operating. in some cases one particular process must be completed before another may begin and so the complete model must reflect this. The way to do this is to arrange the processes into a dynamic network where the interrelationships between them can be described. Before we can do this, however, we need to normalise the processes into a more uniform structure.

Suppose that there are a number of processes

$$\mathbf{process}_\mathbf{i}: \mathit{Memory}_i \times \mathit{Input}_i \rightarrow \mathit{Output}_i \times \mathit{Memory}_i$$

for i = 1,...n

We first consider the memory sets and orthogonalise them by forming the union of any two that overlap until we have a collection $\{\mathit{Memory'}_j\}$ for j = 1,...,m satisfying the property that the intersection of $\mathit{Memory'}_j$ and $\mathit{Memory'}_k$ is empty for $j \neq k$. Such a collection is called an *orthogonal memory set*.

Now we form the set $\mathit{Memory'} = \mathit{Memory'}_1 \times \mathit{Memory'}_2 \times .... \times \mathit{Memory'}_m$.

We also construct the set $\mathit{Input'} = \mathit{Input}_1 \cup \mathit{Input}_2 \cup ..... \cup \mathit{Input}_n$
and

$$\mathit{Output'} = \mathit{Output}_1 \cup \mathit{Output}_2 \cup .... \cup \mathit{Output}_n$$

Now each process function process_i can be transformed into a function

$$\mathbf{process'}_\mathbf{i}: \mathit{Memory'} \times \mathit{Input'} \rightarrow \mathit{Output'} \times \mathit{Memory'}$$

by extending the definition to be

$$\mathbf{process'_i}((m'_1, ...m'_i,...m'_n), input_i) =$$
$$(\mathbf{proj_1}(\mathbf{process_i}((m_i,), input_i))), (m'_1, ...\mathbf{proj_2}(\mathbf{process_i}((m_i,), input_i),...m'_n))$$

where $\mathbf{proj_1}$ and $\mathbf{proj_2}$ are the projection functions.

We can attempt to integrate all these process functions into an X-machine. This is dependent on the degree of parallelism and synchronisation that there is amongst the business processes. In the simplest case the functions become the basic functions of an X-machine model. We will call this an Enterprise model.

Definition 2.3.2 An **Enterprise model** is an X-machine whose basic functions are business processes. Clearly in this situation we require the *Memory*, *Input* and *Output* sets to be defined for the complete enterprise, which will be achieved, in many cases, by forming a global product from amongst the individual business process sets.

It is not always a simple task to put together a collection of business process functions into a coherent and coordinated enterprise machine model. In some complex cases it is better to construct a collection of enterprise models of subsystems of the complete organisation and then to construct a network of collaborating X-machine models. These may or may not synchronise easily with one another. This analysis may identify problems with the way the organisation operates and could suggest a mechanism for improving matters. The communication and synchronisation models built may be analysed to solve some of these problems. It might, for example, be necessary to restrict when some process can operate, perhaps they should be shared by several enterprise machines, to ensure that the overall enterprise is working effectively. There are many complex issues in this area and it is not the focus for this particular investigation. We will assume that we are dealing with a coherent enterprise model, that is one which can be represented as an X-machine.

Note that a business process can itself be representing an X-machine so we have a hierarchical situation which allows us to consider an enterprise as a collection of component business processes which are, in turn, enterprises and so on. This allows us to move between the macroscopic and the microscopic views of organisations thus providing us with a coherent and flexible mechanism for modelling almost any organisation and its behaviour. So, we are contemplating, the main ingredients of a business model from the perspective of computational processes and a potential computer based solution. The use of computational models is crucial, if we cannot model some business process or enterprise as a computational model we will not be able to implement the any solution relevant to the enterprise *faithfully.*

There are many aspects of any organisation or application that can be addressed in this way. In *all* cases there is data associated with the situation, and this can take many forms. This data is being processed in some way, when and how, is dependent on a number of environmental and other considerations which can change during time. It is important that we recognise this as far as possible. Many of the processes will involve and depend on the activities of people in the business. We must think about the people who might be using the system, they will need to understand what is going on and what they are expected to do.

It may come as a surprise that this approach might also be applicable to applications in, for example, process and control engineering and in the design of microprocessors, but there are enough similarities to enable a common approach to work. We will call the modelling of the requirements and the environment of all these applications, *business process modelling* (control engineering applications are as much as part of business as the banking and insurance industry!) and use the metaphors that we defined above in all of them. The next task is to try to see how the business processes and enterprise models might be constructed for a specific application.

One approach to identifying what the business process is is to try to identify scenarios of activities that take place during the operation of the business or application. For example the creation of a new customer account in a sales office, the setting of an alarm clock as part of setting up a hi-fi system, the loading of a 32 bit data value in the register of a microprocessor, all involve the creation and manipulation of data that can be done at certain moments and contexts within the system. We will look at some simple examples of these to see how they might be dealt with.

2.4 A simple business process case study.

We will consider the development of a solution to the following problem[1]:

1. This example is a simplification of a real business application that we managed.

The client has a business selling boxes and other products manufactured from metal. Customers supply a blueprint or detailed design drawings of the required items and the company then build them to this specification. The problem that the client identified was the estimation of the cost of the customer's job. This was done by hand and involved the client's manager studying the blueprints, working out the amount and type of sheet metal and other components required, together with the time and labour needed for the different types of manufacturing process - metal cutting, welding, polishing, painting etc. The estimate was achieved by looking up various tables containing the prices of the raw materials and the labour costs of the processes involved and formulating a total price for the job. The problems with the system was that it was very time-consuming, if the customer wanted any changes - for example modifications to reduce the cost - then the process had to start again and there was no connection with the production line management system or the stock control system - in both cases the data for an estimate that had been accepted (and an order placed) had to be typed in by hand. The objective of the project was to build a simple to use system that would automate this process as far as possible and act as a database that could be connected to other systems, specifically the accounts, production management and stock control systems.

There were a number of non-functional requirements that had also to be considered, one being that the company had a PC network running Windows NT and the software needed to be compatible with this. This immediately restricted the implementation language options - as did the time available for building the solution. We will see later the effect that this had, suffice it to say that it is another indication that the purist approach to formal specification - which demands that all implementation issues must be ignored during specification - is untenable in the real world for many applications. Other requirements were concerned with the user friendliness and efficiency of the tool. These sorts of requirements are often difficult to incorporate into the software development process and yet without achieving them the solution will be flawed - it cannot be correct if it fails to satisfy, for example the criterion of user friendliness! However we must consider how we might define these requirements in a precise way. This we will do shortly.

To make further progress we need to consider some of the typical scenarios that the estimator will be involved in. The following represent a number of business processes, note that they are identified, initially, by the business person's view of the data, in general the equivalent of the sort of forms that may need to be filled in or changed, and the type of processing that these forms undergo.

2.4.1 New customers arrive with orders to estimate. A record therefore needs to be made of who the customer is, their address, phone numbers etc. and, usually, unique customer reference number will be generated. Old customers may change their address, etc. and this also needs to be recorded.
2.4.2 The prices of sheet metal and other components as well as the cost of different production processes have to be recorded and updated from time to time, with a record kept of past prices.
2.4.3 The estimates for possible new jobs need to be created and, from time to time, updated.
2.4.4 A comprehensive help facility is also desired. This corresponds to the process of seeking assistance concerning the other processes.

So let us try to identify the sorts of data and processes that might be involved here. Traditional structured systems analysis and design methods tend to start with an analysis of the data using techniques such as data flow diagrams - which model process communication and entity relationship diagrams - which describe the relationships between different types of data. In both cases the choice of which processes and data are regarded as the important aspects is a key one and it is this that can introduce problems in the communication with the client. The client may not look at the business in the same way. In early object oriented analysis the designer might start with a class of objects called **customers**, a class called **suppliers** and a class called **estimates**. These would then be defined with respect to the types of data and the methods that they might contain as a separate activity. By doing this we have already lost sight of the person at the heart of the process, the intended user of the system, something we want to avoid. We have also started looking at the system from a disembodied form as a collection of parts rather than as a coherent whole. The dynamic aspects of the system are missing and users will usually see their interaction with the business as being involved in tasks of a particular type at an appropriate time. In other words we have sunk to too low a level too early in the game.

Modern Object oriented design techniques attempt to overcome these problems by introducing diagrams to help relate the class structures and through the use of use-case descriptions, these are like scenarios. This is an improvement but there are problems due to the nature of objects and how we tend to think about them. If really we want to reuse components we need to get the component idea really well defined. A component must be fully specified and tested if it is going to be useful. This is rarely the case with classes and frameworks at present and the embedding of a specific object from one application or library into another application is fraught and full of difficulties. The integration of the objects into the system is only loosely specified through the use-case information and it is precisely this integration that needs to be precisely described at an early stage.

We want to emphasise the integrated nature of the system, the control structure of it and to approach it from the perspective of the user at all times. This will assist us in extracting and confirming the requirements of the system. The enterprise model of the system will thus emphasise the business processes we have identified together with their relationships. In a first model we would expect each business process to be carried out separately and integrated through a single access process. We will try to look at these processes and their possible implementation through the eyes of the client or user. We are thus interested in what will be appearing on the screen during the operation. We will, first, design the screens to provide the client with a clear and concrete understanding of what is going on in the system. These screens will give us a “handle” on the processes that the client wishes to support with the solution. If we build the interface screens and produce a prototype of how the screens might appear and how the different parts of the system relate to one another we are more likely to be able to communicate effectively with the client and understand the business process better.

For the **Estimator** system this will entail developing the welcome screen to provide access to the different parts of the business processes. These have been identified from the scenarios as being the *creation and updating of the customer records*, the *creation and updating of the prices of the raw material and labour* and the *creation and updating of the individual orders*. There is also the need to introduce a *help system* at the beginning, we will restrict the help system to providing help for the customer records updating area only, it is easy enough to extend this to other parts of the system. Each of these modes of the system will be accessed from the welcome screen by clicking on a suitably labelled button and this will determine the layout of the initial screen display and its role in the control of the process model and the system. This is illustrated in Fig. 2.2.

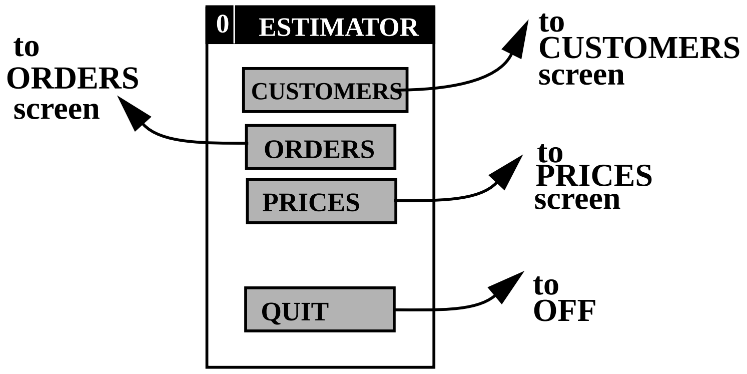

Figure 2.2 The MAIN-MENU screen.

Once a particular form (screen) has been accessed there will be specific types of processing applicable to each form. The more detailed models will reflect this. Thus the enterprise X-machine model might be represented by the “state diagram” illustrated in Figure 2.3 where the circles indicate states and “super-states” which will be refined to a more detailed model, later. Thus the **edit_cust, edit_order, edit_price** and **help_cust** operations represent business processes which will be refined into subsidiary models (X-machines) involving more detailed screens at a later stage. The diagram in Figure 2.3 is not yet a fully defined X-machine but it is part of the process of developing a model of the business process as interpreted through a computer oriented context.

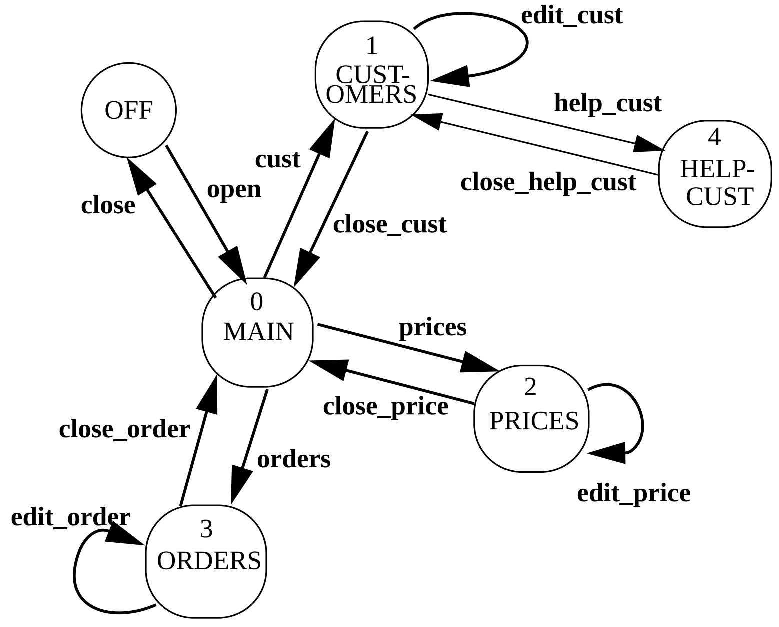

Figure 2.3 The Estimator - top level system design.

The interpretation of this diagram is that when in a given state, for example MAIN, the operation of the function **cust** will cause a state change to occur which will result in a new screen which will provide the (business) processing required to deal with the creation or updating of the customer records. Initially this screen may contain blank entries coupled with a question as to whether the user wishes to open an existing customer record or create a new one. Perhaps something like Figure 2.4 where there is a “pop-up” menu under the name field to enable the user to choose between a new entry or the revision of a previous entry would be appropriate.

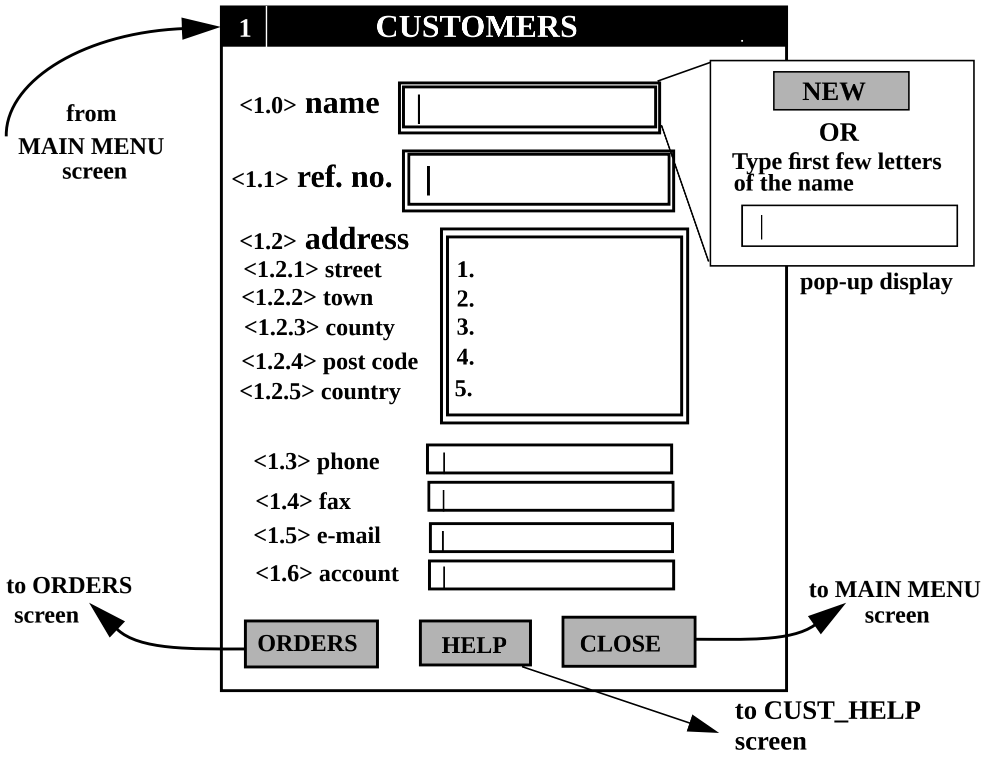

Figure 2.4 The CUSTOMERS edit screen.

The numbers in ” <, > ” brackets are data labels used in the specification and do not appear on the screen displays. Thus labels of the form < *1.n* >, < *1.n.m* > indicate data values that are provided in screen *1* at level *n* and sublevel *m* respectively. This enables us to keep track of where data comes from during the design and analysis. Some of this data will be displayed on other screens, for example < 1.0 > will appear on screen 3 also during the compilation of an order, see Figure 2.5. The critical point to remember is that each screen will have a set of data fields of given types associated with it.

The information contained in these fields will be part of the internal memory that the system in this superstate can have access to. While that screen is active this information can be changed or enhanced using the appropriate functions defined by the X-machine that is representing this super state.

This screen now identifies the type of data that is required for a customer entry. Each data entry field will have a type associated with it, the complete set of data entry fields will form the structure of part of the internal memory of the system.

Note that the input required of the user is of two main types, the input of information to be stored or used in the processing and the input used for navigating through the system. So that pressing the button marked *CUSTOMERS* in Figure 2.1 is a navigation (or control) input entry whereas the entry of "*J. Smith (Containers)*" is a data entry for placing a specific type value in the < 1 > field of the customer records. Nearly all applications will involve these two different types of input. Such inputs require the existence of trustworthy low level functions that can read suitable input events - a button click, mouse movement, typed string of keyboard characters or a value supplied by an external device such as a sensor and either use this event to generate a state transition or insert the information reliably into part of the internal memory. Since most programming languages offer well tested and used functions, procedures or objects to do this we will not concern ourselves with this level of detail in future. However, it is possible to build much lower level models to describe these operations using the methodology if one so desires, perhaps in a highly critical application this may be necessary.

The output from the system also needs some explanation. In many cases the outputs will be conveyed to the user by means of a screen display. The structure of this can be defined abstractly using a suitable convention which defines the required graphical elements of the display. In some applications the outputs will be commands or messages which are used to control other devices, for example in process control applications. Thus there are two types of common outputs met - informational screen displays and control commands.

Figure 2.4 illustrates a possible first draft of a screen display for the customer edit screen. It is meant to be as similar to the "paper" version that the client might be used to as possible.

The types of data that might be entered in the slots provided need to be defined and named. This will help us to define the structure of our internal memory and thus the underlying database. The types of the various data entry values can be defined once we have introduced some standard notation, which we will keep as simple as possible. We need some basic types to start with:

CHAR is the set of all alpha-numeric keyboard characters.

CHAR$^{@}$ is the set *CHAR* together with the symbol “@”.

DIGIT is the finite set {*0, 1, 2,3, 4, 5, 6, 7, 8, 9*}.

LETTER is the set of letters, both upper case and lower case.

NUMBER is the set of all non-negative (i.e. natural) numbers, *0, 1, 2,* .....

INTEGER is the set of all integers, *0, +1, -1, +2, -2,* ....

REAL is the set of all real numbers (in practice rational numbers to some suitable level of accuracy.

BOOLEAN is the set { true, false}

From these we can construct a number of further types:

CHAR(20) is the set of all sequences of *CHAR*s of length less than or equal to 20.

CHAR$^{@}$(20) is the set of all sequences of *CHAR*$^{@}$s of length less than or equal to 20

DIGIT(10) is the set of all sequences of *DIGIT*s of length less than or equal to 10.

LETTER(20) is the set of all sequences of *LETTER*s of length less than or equal to 20.

To relate the data values to the screens that they are associated with we introduce a Screen/Data table.

Table 1: A Screen/Data table for the customer screen.

reference	name	type	conditions	screens
< 1.0 >	*Customer_name*	*CHAR*(20)	N/A	1, 3
< 1.1 >	*Customer_code*	*DIGIT*(10)	length = 10	1, 3
< 1.2 >	*Customer_address*	compound	N/A	1
< 1.2.1 >	*number_and_street*	*CHAR*(20)	N/A	1
< 1.2.2 >	*town*	*LETTER*(20)	N/A	1
< 1.2.3 >	*county*	*LETTER*(20)	N/A	1
< 1.2.4 >	*postcode*	*CHAR*(20)	N/A	1
< 1.2.5 >	*country*	*LETTER*(20)	N/A	1
< 1.3 >	*phone*	*DIGIT*(20)	N/A	1
< 1.4 >	*fax*	*DIGIT*(20)	N/A	1

Table 1: A Screen/Data table for the customer screen.

reference	**name**	**type**	**conditions**	**screens**
< 1.5 >	*e-mail*	$CHAR^{@}(20)$	only one @	1
< 1.6 >	*account*	*CHAR*(20)	N/A	1

The names in the table will be used in the definition of the *customer* X-machine.

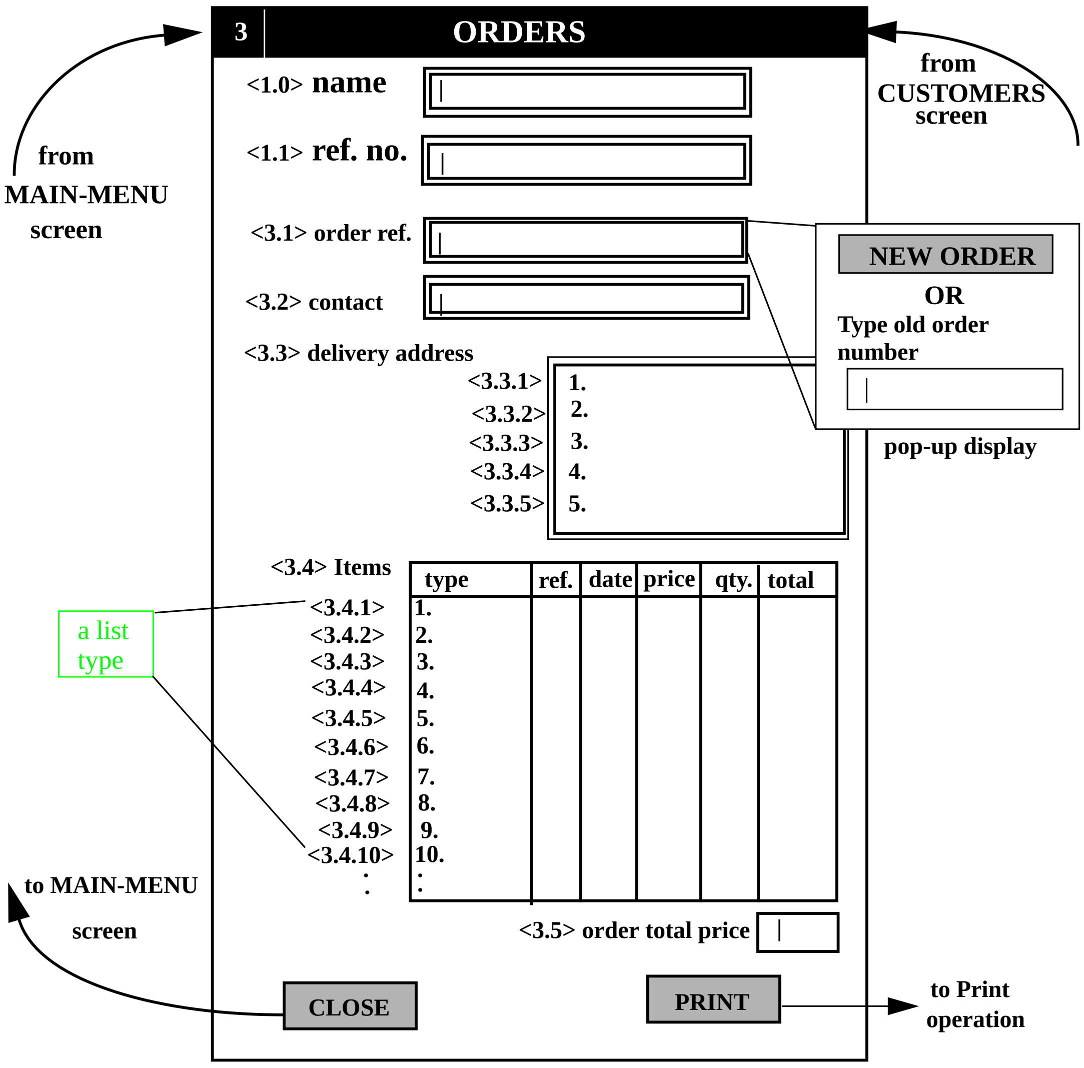

Figure 2.5 The ORDERS edit screen.

Now, in many approaches to design, one might be expected to carry out a data analysis which could culminate in the development of a relational data model in a suitable normal form. We will not do this here for a number of reasons. We question whether the benefits are worth the effort, as mentioned before the improvements in speed and storage probably mean that the benefits are marginal with the possible exception of massive distributed databases,

[46], which need special treatment anyway!

But another reason is the need to keep the model sufficiently simple to allow for its updating, the business is going to change frequently and so the business model will also change. It has been noted in many places that the key to success in the modern world is the ability to manage change effectively. The world is a highly dynamic, if not chaotic, organism and the ability to update our models simply and effectively is vital if we are not to build obsolescence into our methods and systems.

Our data model is as follows:

STORAGE = CUSTOMER_STORE × *PRICES_STORE* × *ORDERS_STORE*
where:

CUSTOMER_STORE =

<1.0> × <1.1> × <1.2> × <1.3> × <1.4> × <1.5> × <1.6>

and

<1.2> = <1.2.1> × <1.2.2> × <1.2.3> × <1.2.4> × <1.2.5>

We will reserve a special value of this type : blank_customer_store to represent a blank form on the screen for a new customer.

the other components are:

ORDERS_STORE = <1.0> × <1.1> × <3.1> × <3.2> × <3.3> × <3.4> × <3.5>
(similarly blank_order represents a blank orders screen),
(see Figure 2.4 for details) and *PRICES_STORE* which will not be considered further here.

Suppose that we have to revise the data model in a significant way at a later date, for example the customer component may have to record whether the company was VAT registered[1]. It is a simple matter to update the model and the screen to allow for a new parameter to represent this information. We just append to the product set of *CUSTOMER_STORE* the

1. VAT or Value Added Tax is a tax that has to be paid by all but the smallest company in the European Union.

appropriate data type to cater for the new requirement. This is a simple example which does not create a major problem but more complex changes can also be dealt with in a similar manner without having to comprehensively redesign the relational or object data model of the business. For example, a more drastic change may occur if the client company decides to expand into exporting their products and the treatment of overseas customers requires different procedures and records. Perhaps a new screen for exports could be designed and accessed from the MAIN state which would then be interfaced with the current system and the current database in a simple way, an issue we shall examine shortly.

The next stage is to assemble an X-machine description of the system. If we look at the top level model state diagram (Figure 2.3) we can identify 6 states :- OFF, MAIN, CUSTOMERS, HELP_CUST, PRICES, ORDERS. The processing functions are:- **open**, **cust**, **close_cust**, **help_cust**, **prices**, **orders**, **edit_cust**, e**dit_price**, **edit_order, close_price, close_order, close_help_cust**. The question we have to answer now is how do we define these functions?

Each function has a declaration which must be of the form:

function: $Memory \times Input \rightarrow Output \times Memory$

where *Memory, Input* and *Output* are defined suitably.

The *Input* set is relatively simple to define, we need to ask what are the inputs that make sense when considering the state diagram.

There is an input which corresponds to loading the program - let us denote that by the label *on*.

There is an input corresponding to the pressing of the QUIT button to close the program, we call this input *off*.

There is an input corresponding to pressing the CUSTOMERS button and we will call that input *1*.

There is an input corresponding to pressing the PRICES button which we will denote by *2*.

Similarly we have inputs: *3, help_cust, close_1, close_2, close_3, cust_details, edit_price, edit_order, help, close_h*.

The “**edit_cust, edit_order, edit_price**” business processes will not be described in detail until we reach the next level of the design they are considered as ”atomic” operators at this stage.

So

INPUTS = {*1, 2, 3, off, on, close_1, close_2, close_3, cust_details, edit_price, edit_order, help, close_h*}

What is the memory at this stage? We will construct the memory from two components - *STORAGE* and *SCREENS*, the latter component being used primarily to refresh the outputs - this is necessary because the stream X-machine model does not allow for outputs to be used as an input parameter by a processing function.

The *STORAGE* component has been described above and comprises a set of linear records describing the customers, the prices and the orders organised as a linear collection of linear records.

The *SCREENS* component is made up of the five screen displays with associated *slots* for presenting output information - the data types associated with the screens. We will call these, for the time being, *screen*(CUST), *screen*(PRICES), *screen*(MAIN), *screen*(HELP_CUST), *screen*(ORDERS) and deal with the data output later.

So *SCREENS*~={*screen*(CUST), *screen*(PRICES), *screen*(MAIN), *screen*(HELP_CUST), *screen*(ORDERS)} [1]

Thus $MEMORY = STORAGE \times SCREENS$

The outputs will be $SCREENS \times DATA$ where *DATA* describes the format of the printed output from screens such as the ORDERS screen.

1. The notation ~= indicates an incomplete definition.

Now we can define the functions, recall the notation we use:

function_name(*memory, input*) = (*output, memory'*) or in this case:

function_name((*storage, screens*), *input*) = ((*screens, data*), (*storage', screens'*))
and the convention that " - " is used as a "leave alone" symbol when the component is a memory value and Λ is a null output.

A diagrammatic notation like the following may be preferred:

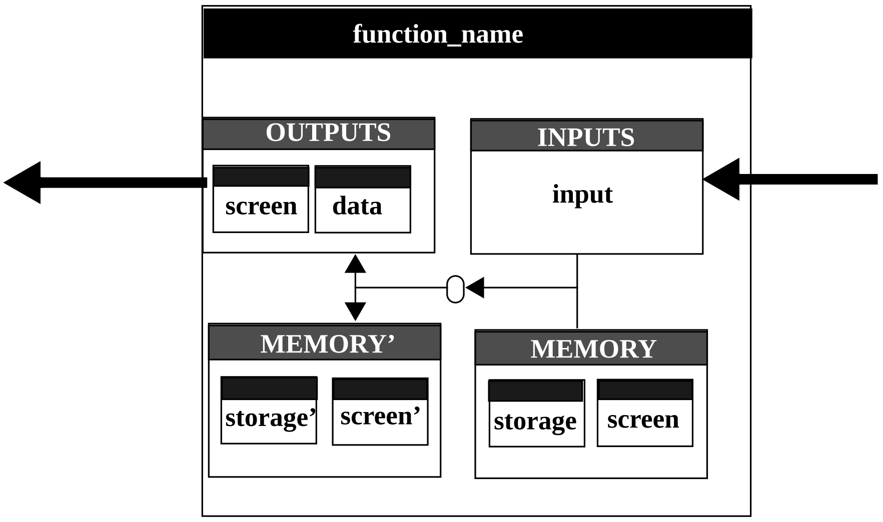

Figure 2.6 Schematic for describing basic functions.

Thus:

STATE-0 FUNCTIONS
open((-, -), *on*) = ((*screen*(MAIN), Λ), -);

The **open** function simply sets up the system and displays the MAIN screen.

cust ((-, *screen*(MAIN)), *1*) ~=
((*screen*(CUST),Λ), (*blank_customer, screen*(CUST));

The **cust** function responds to the *customers* menu selection, the input *1*, and displays the blank CUSTOMERS screen with the database ready to accept the details input.

orders((-, *screen*(MAIN)), *3*) ~= ((*screen*(ORDERS),Λ), (*blank_order*, *screen*(ORDERS));

The **orders** function responds to the *orders* menu choice (input *3*) and causes the ORDERS screen to be displayed ready for the input of an order.

prices ((-, *screen*(MAIN)), *2*) ~= ((*screen*(PRICES),Λ), (*blank_price*, *screen*(PRICES));

The **prices** function does the same for the prices subsystem.

help_cust ((-, *screen*(MAIN)), *help_cust*) ~=
((*screen*(HELP_CUST),Λ), (*blank_help*, screen(HELP_CUST));

The **help_cust** function displays the HELP_CUST screen. Other help functions could be defined in a straightforward way, if required.

close((-, -), *off*) ~= ((-, -), -)

This function closes the system.

We cannot yet develop the other functions that loop back to their states until more details of the state screens are available. We can therefore look at the **edit_cust** function as representing the overall operation of the CUSTOMERS screen, etc..

The next stage is to develop the state diagram for these screens. We will present the diagram for a slight simplification of the CUSTOMERS screen as an example (some of the data entry variables have been removed to prevent the process seeming too complicated at this stage).

The diagram of the simplified screen in Figure 2.7 has a number of features that should be commented upon. The address data entry has been described at a higher level since it needs to be broken down into 5 specific data entry actions. This can be done through the expansion of the **address_edit** function as a subsidiary X-machine with five states in a similar way to the way that we decomposed the **edit_cust** function.

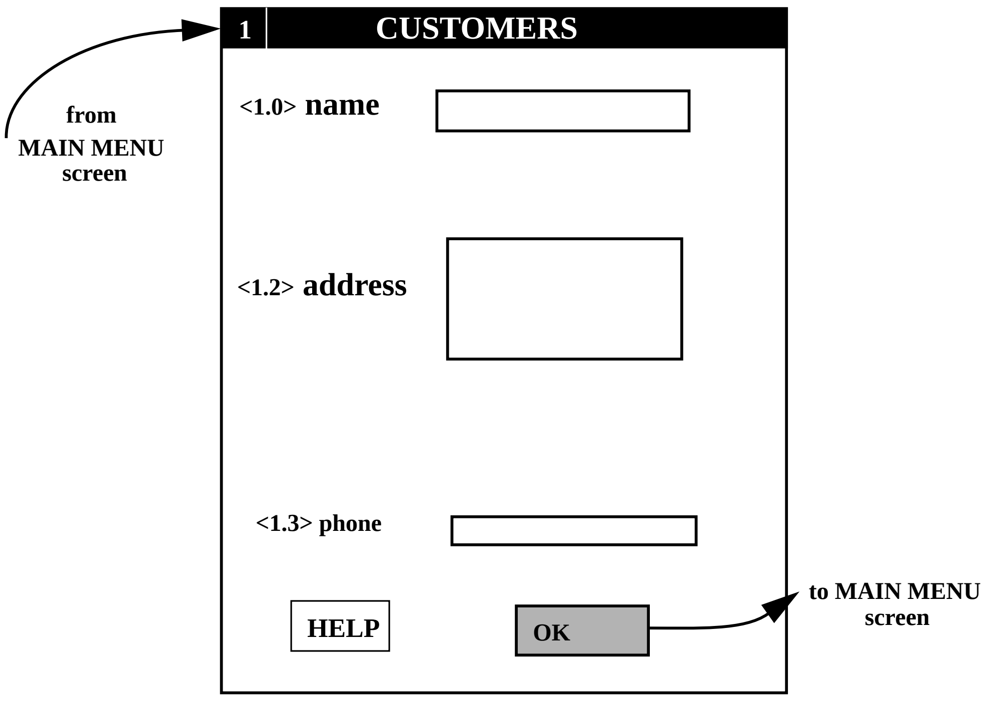

Figure 2.7 The simplified CUSTOMERS edit screen.

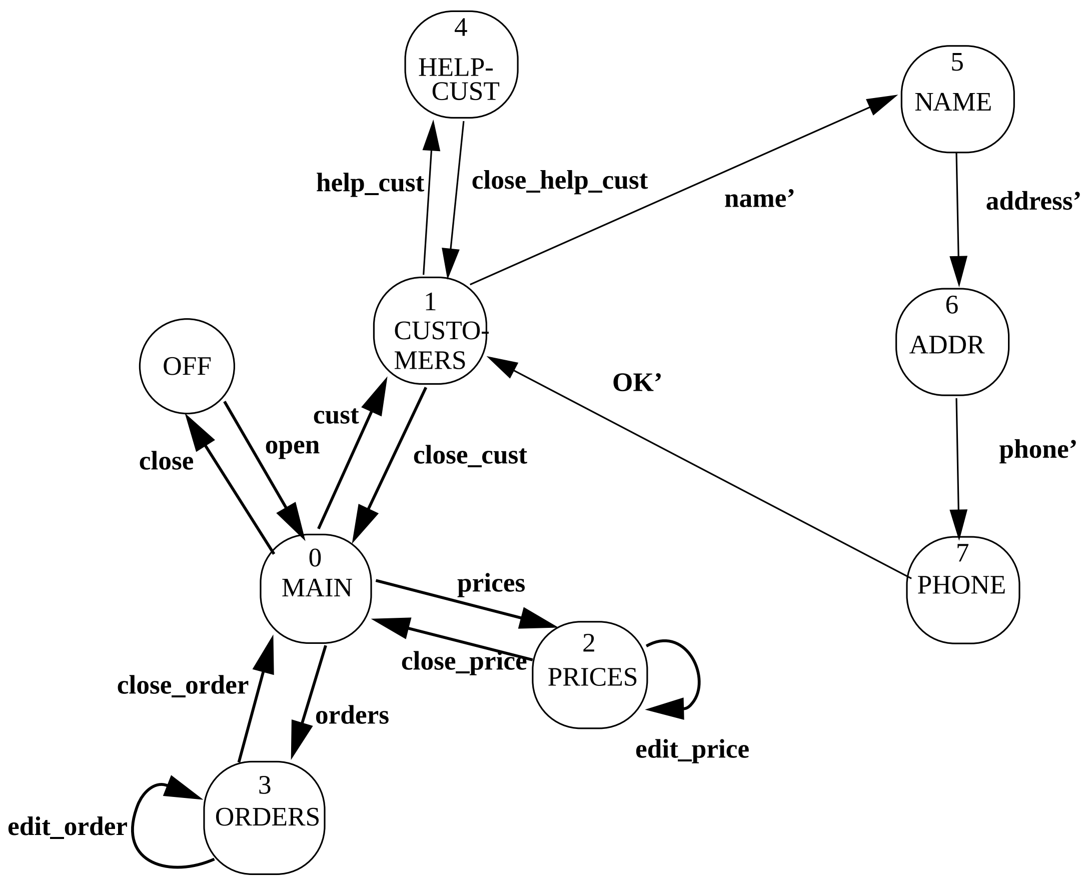

Figure 2.8 A view of the Estimator system with edit_cust refined into a submachine.

In Figure 2.8 we have expanded part of the system to reflect the activity in the CUSTOMER state. This is an example of model refinement which we will look at in Chapter 4.

2.5. Another example of a business process and enterprise model.

The previous system was based on a simple office process, the next model will introduce the issue of time. We will consider a simple electronic alarm clock illustrated in Figure 2.9.

There are two modes of behaviour, user independent time display and the user dependent setting of the current time or the alarm time. These correspond to the 4 business processes described below.

The clock has five buttons, one to set the time, one to set the alarm, one to increment the hours, one to increment the minutes and one to switch the alarm bell off.

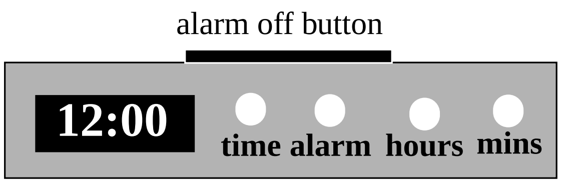

Figure 2.9 An alarm clock.

We consider the basic scenarios or processes.

2.5.1 The clock automatically updates the time each second displaying the current time at each update.

2.5.2. The user sets or resets the current time by accessing the time setting mechanism (by pushing the “time” button) and setting the hours and the minutes, then switching back to automatic timekeeping mode by releasing the “time” button.

2.5.3. The user sets or resets the next alarm time by accessing the time setting mechanism through the “alarm” button and setting the hours and the minutes, then switching back to automatic timekeeping mode.

2.5.4 The clock, on reaching the alarm set time rings until the user cancels the alarm by pressing a suitable button.

These are the processes we have to integrate into an enterprise model. Note that one of these processes (2.5.1) involves a regular or *periodic process*, namely the updating of the clock at each "tick". Such a periodic process usually needs to be synchronised with all the other processes. Another similar example, that of a video cassette recorder, will be found in [39].

We need to consider, as before the inputs, the outputs and the memory associated with these processes.

We will need to identify a data type to represent the time, which will be represented at this level by integers, and could be defined at a lower level by a model of the 24 hour clock.

So we define *TIME* as the basic type to represent time. The main operations that we can apply to this type is to increment it by one second according to the laws of the 24 hour clock, so we indicate the increment by x + 1.

For the inputs we have to have buttons that allow the user to access the different modes of the clock. Thus there is:

the time setting mode (call this T_SET) which is accessed when the input *time_button_on* occurs;

the alarm setting mode (A_SET) accessed when the *alarm_button_on* event occurs;

the actual time changing process which involves feeding in a new time which will be any value *x* in the set *TIME*;

the button that cancels the alarm bell, the *cancel_alarm event*;

together with an *on* switch and an *off* switch.

It also turns out that we need to inform the clock of when we have finished setting the time and the alarm, these will be *time_button_off* and *alarm_button_off* events (inputs). Finally the passing of time, the ticking of the clock must be represented by an automatically generated action or input called *tick*. This will update the memory, the display and cause the alarm to sound when appropriate.

So *INPUTS* = {*time_button_on, alarm_button_on, cancel_alarm, on, off, time_button_off, alarm_button_off, x* | where *x* is any value of *TIME* }.

The outputs are the display (in hours and minutes) of the current time and the display of the set alarm time and the alarm ringing. We write *display*(*x*) to indicate that the time value *x* from *TIME* is displayed on the clock face.

So *OUTPUTS* = { *display*(*x*), *alarm* | where *x* is any value of *TIME*}.

The memory needs to keep a record of the current time and the set alarm time.

Thus *MEMORY* = *CURRENT_TIME* $\times$ *ALARM_TIME*

where *CURRENT_TIME* = *TIME* and *ALARM_TIME* = *TIME*.
So *MEMORY* = { (*x*, *y*) | where *x* and *y* are any values of *TIME*

The top level state diagram is shown in Figure 2.10.

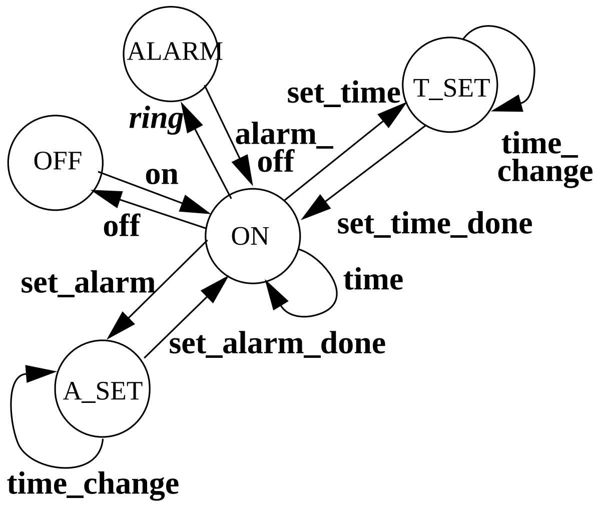

Figure 2.10 Alarm clock top level state diagram.

The functions are defined in the following format:

function_name (*memory, input*) = (*output, memory*);

i.e. **function_name** ((*x, y*), *input*) = (*output*, (*x', y'*));
Some specific examples are:

time((*x, y*), *tick*) = (*display* (*x* + 1), (*x* + 1, *y*));
the passing of time is modelled by the time function which is driven by the tick event.

set_time((*x, y*), *time_button_on*) = (*display* (*x* + 1), (*x* + 1, *y*));
the time setting mode is accessed by the *time_button_on* event and this is assumed to operate over a period of one second.

time_change((*x*, *y*), *x’*) = (*display* (*x’*), (*x’*, *y*));
the time setting operation is achieved, once we are in the T_SET mode, by inputting the correct time *x’*, this is also assumed to take a second - an assumption that can be relaxed when the lower level details of this operation are considered.

set_time_done((*x*, *y*), *time_button_off*) = (*display* (*x* + 1), (*x* + 1, *y*));
the conclusion of the time setting process is modelled by the release of the time set button, i.e. the event *time_button_off*.

The other functions can be defined in a similar way.

2.6 A microprocessor example.

A common example of a simple microprocessor is the system described by Gordon [56] we will show how this same system can be specified very simply using X-machines, see Bogdanov *et al* for further details [57]. Harman and Tucker [58] also looked at this example and describe an alternative model based on synchronous algorithms and stream functions.

The approach taken in [57] went through a series of refinements, an issue that we will consider in a later chapter. Here we will describe the process model requirements and consider an initial model.

The system will be considered from the point of view of the microprogrammer. It consists of a program counter, *PC*, an accumulator, *ACC*, a set, *OP*, of instructions and a set of 16 toggle switches which define the data to be processed and a push (on/off) button and an internal memory store. There is a dial with 4 settings and a memory address register, *MAR*. A collection of 13 lights are used to display the contents of *PC*, 16 lights to display the contents of *ACC* a ready light and an idle light. The dial has 4 positions: *LoadPC*, *LoadACC*, *Store*, *Run*. A simple, conceptual diagram is shown in Figure 2.11. As before we try to envisage a concrete representation of the system to provide a basis for the client to think about the model.

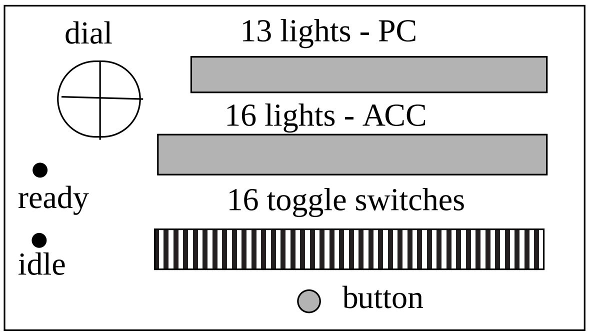

Figure 2.11 A simple microprocessor control panel

The data values involved are 13 bit numbers for the contents of the *PC*, 16 bit numbers for the contents of the *ACC* and the memory store.

The *LoadPC* dial setting causes, when the push button is pressed, the binary value represented by 13 right most toggle switch settings to be stored in the *PC*. The *LoadACC* setting places the 16 values from the switches into the *ACC* when the button is pressed. The Store setting replaces the contents of the memory location specified by the *PC* with the contents of the *ACC* when the button is pressed. The *Run* setting starts the execution of the program at the memory location specified by the *PC* when the button is pressed.

There are a number of instructions carried out by the computer as described in the following table:

Instruction	**Code**	**Effect.**
HALT	000	Stops execution
JMP L	011	Jumps to memory location *L*
JZERO L	010	Jumps to memory location *L* if *ACC* is zero
ADD L	011	Add contents of memory location *L* to *ACC*
SUB L	100	Subtract contents of memory location *L* from *ACC*

Instruction	**Code**	**Effect.**
LD L	101	Load contents of memory location *L* into *ACC*
ST L	110	Store contents of *ACC* in memory location *L*
SKIP	111	Skip to next location

There are two main business processes involved, writing a program and storing it in the memory and running a stored program. When the machine is switched on the dial can be set to a value and the toggle switches operated.

The scenarios:

Part A Writing a program. This is achieved by setting the toggle switches and then using these and the dial functions to place values in suitable places.

2.6.1 If the dial is set to *LoadPC* then the value represented by the toggle switches is placed in the *PC* when the button is pressed.
2.6.2 If the dial is set to *LoadACC* then the value represented by the toggle switches is placed in the *ACC* when the button is pressed.
2.6.3 If the dial is set to *Store* then the value represented by the toggle switches is placed in the PC.

Part B Running a program.

2.6.4 If a program is stored in the memory then it may be executed by setting the dial to *Run* and pressing the button. The process then goes through a process of:

fetching an instruction;
carrying it out and returning for the next instruction;
until it reaches the halt instruction or the emergency halt button (*B_halt*) is pressed.

A simple first stage enterprise model is illustrated in Figure 2.12.

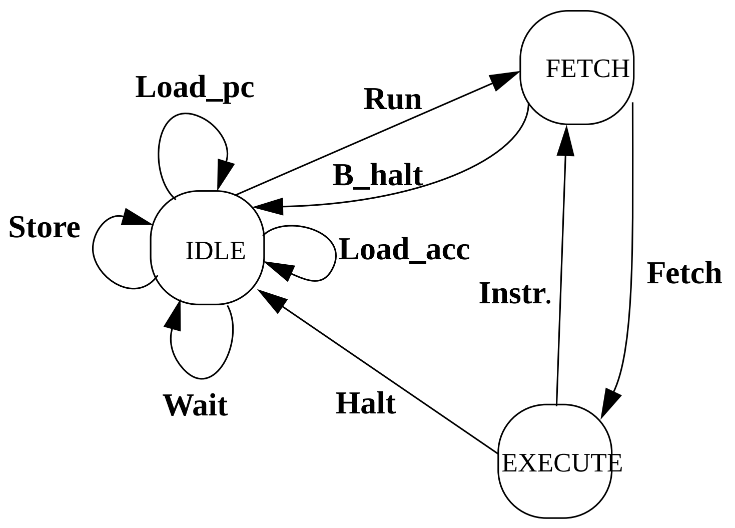

Figure 2.12 A microprocessor.

In this model we have slightly expanded the process 2.6.4 to show some extra internal structure which describes the basic fetch-execution cycle of the system since this was detailed in the process description.

The process functions can be defined once we have specified appropriate *Input*, *Memory* and *Output* sets.

$$Memory = PC \times ACC \times READY \times IDLE \times MAR \times MEM \times OP$$

where *PC* is the set of 13 bit words giving the contents of the program counter;
ACC is the set of 16 bit words, the accumulator contents, l(*acc*) means the 3 leftmost bits of the word *acc* and r(*acc*) is the 13 rightmost bits of $acc \in ACC$;
OP is the set of 3 bit words representing instructions;
READY = { 0, 1 } where 1 means the ready light is on and 0 means it is off;
IDLE = { 0, 1 } where 1 means the idle light is on and 0 means it is off;
MAR is the set of 13 bit words which can be put in the memory address register;

MEM is the set of functions from *MAR* to *ACC*;

$INPUT = SWITCHES \times DIAL \times BUTTON$

where *SWITCHES* is the set of 16 bit words obtained from the 16 switches on the console;
DIAL = { 1,2,3,4 } with 1 meaning *Load-acc*, 2 is *Load-pc*, 3 is *Store* and 4 is *Run*;
BUTTON = { 0, 1 } where 1 signifies that the button is pressed and 0 is when it isn't.

$OUTPUT = PC \times ACC \times ROUT \times IOUT$

where *ROUT* describes the appearance of the *READY* light and *IOUT* that of the *IDLE* light.

Two sample process functions are given next, one related to the activity of writing a program and the other describes how a specific operation will work when a program is executed.

Load_pc((*pc, acc, 1, 1, addr, mem, op*), (*sw*, 2, 1)) =
((r(*sw*), *acc*, 1, 1), (r(*sw*), *acc*, 1, 1, *addr, mem, op*));
Thus the **Load_pc** functions takes the rightmost 13 values from the switch input and places it in the *PC* with the rest of the memory unaffected and outputs the information through the lights.

Sub((*pc*, *acc*, 0, 0, *addr, mem, SUB*), (*sw, d*, 0)) =
((*pc*+1, *acc-mem*(r(*mem*(*addr*))), 1, 0), (*pc*+1,*acc-mem*(r(*mem*(*addr*))),1,0,*pc*,*mem*, *SUB*))
The **Sub** function takes the value of the memory at the address given by the 13 rightmost digits obtained from the element of memory at the address given by *addr* and subtracts this from the current value of the *ACC* placing this new value in the *ACC*. The program counter is incremented by one.

2.7 Implementation.

The effort made to construct a formal computational model of the system is not in vain. We will understand the business processes and the potential solutions much better once this has been done. We also have a precise statement of what the system should be, something that will be invaluable when it comes to establishing the correctness of any implementation. Not only can we use it as a reference point for evaluating particular implementations but, as we shall see in the next chapter, this description can be used as a basis for a powerful testing strategy. In fact, the complete test sets can be generated automatically from the X-machine description.

That is not all, however. The specification can also be turned into a operational implementation very easily. Eventually it will be possible to build a suite of tools that can be used to create and edit X-machine specifications, generate complete functional test sets for the specification and to generate executable implementations automatically. (In fact prototype tools already exist to do this.) If we examine how an implementation can be built we will see how valuable it was to think about algorithms at an early stage. Depending on the programming language chosen, the task of developing the code can be straightforward.

If the implementation is in a standard imperative language such as Pascal, the basic functions can be implemented more or less directly from the function definitions as procedures that produce the required outputs and memory updates. The states are implemented as procedures with `IF` statements corresponding to the transitions from that state.

In an object oriented language such as C^{++} the memory is implemented as a set of global variables accessible by all the basic functions. These, also, were simply implemented using the definitions. The explicit control structure of the model allows for a straightforward architecture which can be built with little effort and a very high level of confidence.

A number of systems have been built using the technique in several different languages. The effort required to complete the implementation, given the specification, was quite slight. In Chapter 5 we will look at one of these case studies in more detail, the system there was built and installed for a client’s business, a high quality user interface was required and a very robust system was developed in a relatively short time.