17/3/98

PART ONE

CHAPTER 1

Models of computer based systems.

Summary. Modelling systems, finite state machines, statecharts, X-machines.

The fundamental desire of all those involved in the development of new computer systems, whether business software solutions, real-time control systems or hardware devices is that the final product behaves correctly. In traditional computer science research this means that some formal mathematical proof must exist that establishes the logical equivalence of the implementation with some mathematical definition or specification of what the system should be like. This has always been a very difficult task that is rarely achieved except with very small systems and under very restricted conditions. The practising software and hardware designer rarely considers even attempting to do this - even supposing that they knew how to. The existence of such a formal verification is insufficient to guarantee the system is correct, anyway.

We take a different approach. Our starting point, as we will see in Chapter 2, is the client and the client's needs. A problem exists, this problem needs expressing and analysing. Possible solutions can then be investigated. At this stage it is vital that we do not loose contact with the client's perspective, otherwise we might find that our potential or actual "solution" is a *solution to the wrong problem*! No matter how much mathematical analysis and formal verification that has been carried out on the system if it is the wrong system *it cannot be correct*!

So, our interpretation of a correct system is one that can demonstrably solve the required problem within the constraints agreed with the client. In order to do this we will be investigating ways in which we can communicate with the client using a mixture of easily understood diagrams and carefully worded text which contains a fully formal and precise model of the system and a potential solution. This formal structure can be analysed by an engineer in a number of ways. It is possible to "check" the model by establishing that certain necessary properties are satisfied by it. It is possible to use such a model to generate, automatically or semi-automatically, an implementation (either software or a hardware design in a suitable technology) and also to develop a powerful complete functional testing method that will provide a rigorous and searching examination of any final system in its operating environment.

If the system that we finally construct is such that:

i) the initial model of the requirements has the required functionality in the opinion of the client;
ii) the model can be shown to satisfy any necessary properties;
iii) the implementation is generated directly from the model using reliable techniques and tools;
iv) the implementation passes ALL the tests constructed using the complete functional test generation model;

then we may be justified in calling the system a *correct system*.

In the preceding discussion we used the word *model* in several places. The process of designing correct systems depends on building accurate and useful models of the processes that are being investigated, and eventually of the proposed solution. There are, however, many types of models that are proposed in computer science. Some of these are useful for specific types of application, many of them are either too restrictive in their scope or very difficult to use. We have to be realistic. If we wish to encourage engineers to use a formal model in a coherent and productive way we must ensure that it is usable and sufficiently powerful for the job in hand. In the first section we will be looking at the sort of models that might be useful and, eventually, concluding that one particular model is the most appropriate. This model is a full computational model, something that we believe is vital if a complete model of a system is to be built. It is also a new type of model and the claim that we make is that it is the first truly integrated and general modelling language that has been developed which is usable by any proficient engineer.

We begin by reviewing some of the previous attempts to construct formal modelling languages for the specification of computing systems.

1.1. Introduction to modelling systems

A major thrust of software system specification, in recent years, has centered around the use of *models of data types*, either functional or relational models such as Z [16] or VDM [17] or axiomatic ones such as OBJ [18]. This identification of data types and the definition of algorithms as functions or relations on these mathematical entities has led to some considerable advances in software design, and has even given rise to attempts to specify the human-computer interface of an interactive system using this ideas [19]. However, one basic problem with these abstract specification methodologies, whether Z, VDM, OBJ or whatever, is that lack the ability to express the dynamics of the system in an amenable manner. It is, for example, very hard to describe the menu structure of a user interface in an intuitive way using Z or VDM. They are also quite difficult to use and have failed to penetrate the industry to any significant extent, despite the fact that many computing undergraduates have attended courses in them during their degrees.

Generally, all these specification methods can describe is what the system does rather than how it does it. In fact, part of the culture of these languages is to avoid any tarnishing of the specification with information that relate to the way the system might be implemented. If the system has to perform many “basic” operations in order to attain the final configuration, it is usually quite hard to describe the system evolution in an intuitive way using such a specification method. From a theoretical point of view one can argue that at the specification phase all we need to know is what the system is supposed to do and that the algorithm to be used is left to the implementation. This is obviously true when the task in hand is quite simple and there is no doubt as to how it will be implemented. If the system to be specified is, for example, a simple program that finds the maximum of two numbers it is very unlikely that anyone will require the corresponding algorithm. However, if the program is more complex - it sorts the elements of an array, for example - the declarative specification might not be considered to be sufficient and the algorithm might also be needed - in this case the choice of algorithm can affect the effectiveness of the program, for example. In some instances, the choice of algorithm may even effect the final result - if the program to be specified solved a polynomial equation then the results would depend on the method used to approximate the solutions.

As we have noted, there is a tradition within the formal methods community for trying to avoid any consideration of the way in which a program is to be implemented. This encourages implicit styles of specification. The problem with this approach is that it succeeds in erecting barriers between the formal specification - the *what* question and the possible implementations – the *how* questions. To overcome these barriers is not always straightforward and, despite the achievements of modern theories of refinement, the practical obstacles to transforming an implicit formal description into an effective working system seem to be huge. Furthermore, we can use Z or a similar methods to specify something that cannot be implemented on a computer system, i.e. there is no algorithm that produces the result given in the specification. This is due to the fact that Z is not based on a *computational model* and so non-computable functions can be specific within the language.

Another issue that needs to be addressed is that of the scope of a formal specification language in describing dynamic aspects of a system. The user interface, for example, is a vital and complex part of most of today's software systems, trying to specify this in an intelligible and useful way with a language like Z is very difficult. In safety-critical systems the software is usually an example of a *reactive system*, [20] where both the dynamics and the timing of events and computations is of vital importance. It is, again, difficult to use a language such as Z to deal with these dimensions.

Although it might appear that we are belittling the effectiveness of these abstract specification methodologies this is not the intention. Clearly the use of these methods has resulted in an important advance in the specification of systems, especially for safety-critical applications. However, we believe that there is a need for an unified specification methodology (or a meta-method) that can be used to describe both the data processing and the dynamic information of the system. Such a methodology would have to provide means for specifying what the system does as well as how it does it. Here, when we say "how it does it" we refer to an algorithm that can run on a computer system and which produces the desired result. If we describe the dependency between the inputs and the outputs of the system as a function, then we say that the function is *computable by an algorithm* or simply *computable* [21].

What we are advocating here is the idea that the process of software design, including within that activity all phases of requirements capture, specification, design, prototyping, analysis, implementation, validation, verification and maintenance is one that should be oriented around the construction of *computational* solutions to specific problems. When we are constructing a software system (this also applies to hardware) we are attempting to construct something that will, when operating, carry out some computable function. On the other hand, computable functions have been identified as functions computed by some algebraic models called machines, Turing machines or similar alternative models [22]. Generally, the Turing machine is accepted as the main model of computation. However, although the Turing machine has received much study in a variety of theoretical areas, it is not used by software engineers for the specification and design of systems, the principle reason being that it is based at a very low level of abstraction and is not very amenable to analysis and/or system development.

Less general models, such as finite state machines [23] and Petri nets [24], however, are the basis of many system specification and development methodologies. We will consider finite state machines in the next section. Petri nets have a substantial literature and have evolved a number of different variations and generalizations to try to make them more expressible. Although they have a role in system design and the modelling of concurrent systems they are not easy to analyse in a general way. We will not focus on them here.

1.2. Finite state machines.

In this section we will consider ways of modelling a variety of systems using the ideas of *machine theory.* Basically a machine is a system that interacts in some way with its environment. There are many areas of applications for these ideas ranging from mathematical models of computer hardware systems to models of power station generators, from the biochemical activity of a living cell to the definition of a compiler for a programming language.

The purposes of studying these machines range from attempts to simulate the behaviour of a complex system to the specification in a formal way of some system by modelling it with a suitable machine in order to analyse it and then to use this model to develop an implementation of the system. These two applications of *simulation* and *design* are of great importance in many practical areas and we shall look at some examples soon. There are also good reasons for studying the mathematical and philosophical basis for computation using the models of machine theory.

There are two fundamental concepts associated with any *dynamic* or *reactive* system which is situated in and reacting with some environment:

1) the environment itself must be defined in some precise, mathematical way, this will usually involve identifying the important aspects of the environment and the way in which they may change in accordance with the activities of the system.

2) the system will be responding to environmental changes by changing its basic parameters and possibly affecting the environment as well. Thus there are two possible ways in which the system reacts:

   (i) it undergoes internal changes;

   (ii) it produces outputs that affect the environment.

We can display the situation as follows:

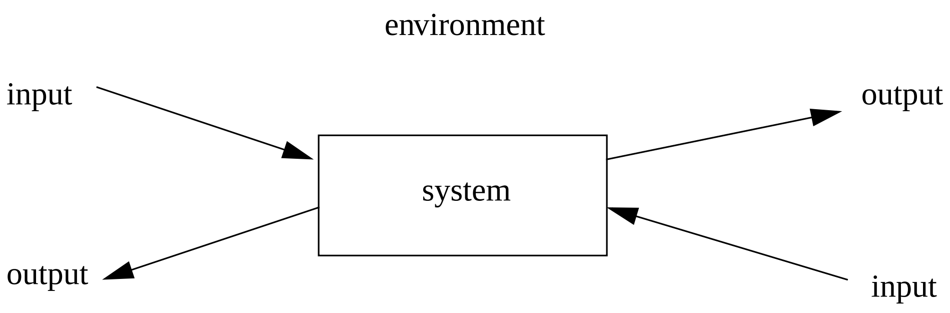

Figure 1.1.

The finite state machine is a mathematical model of a system with discrete inputs and outputs. The system can be in any of a *finite* number of internal states. The state of the system summarises the information concerning past inputs that is needed to determine the behaviour of the system on subsequent inputs. To describe these ideas mathematically we introduce suitable sets and functions that will form the basis of a machine description of the system and its environment.

We will describe the environment first. The important aspect is the elements of the environment that directly affect the system, these elements will be collected together to form an *input* set which we will denote by the symbol *In* with *in* representing a typical input element. It is important that we keep an open mind about the sort of candidates that exist for the input set, and to emphasize this we will examine a few widely different examples showing that machines can arise naturally in so many situations. In many situations there will also be an output generated by the system and this will affect its environment. Initially, however, we will only consider systems which react to inputs and do not generate outputs.

It is thus emerging that, for some dynamic systems, there are three essential factors to consider:

(i) a set of appropriate environmental stimuli or *inputs*;

(ii) a set of suitable internal *states* of the system;

(iii) a *rule* of some sort that relates the two and tells us what the system state will change to if a particular input arrives while the system is in a particular state.

There are many different types of models of systems based on these considerations, some more sophisticated than others. We will consider a variety of these, they all have their uses and often the secret of good design and systems analysis is the correct choice of the type of model for the problem in hand.

For the moment we will look at the basic ideas presented here in more detail. Note that we have described some of the essential features in terms of two sets the input set and the state set. For much of what we will be doing it is sensible to assume that both these sets are finite. The essential feature that we are trying to capture is that of *change*. How do changes in the environment affect the system's internal state?

Consider a system with state set *Q* existing in an environment with input set *Input*. Let the system be in state *q* and receive input *input* at time t_0. The next state which the system "jumps" to at time t_0+1 will be determined by a function (or partial function) which describes the behaviour of the system as the environment changes, that is, as inputs occur. The form of this function, called the *next state function,* is

$$\mathbf{F} : Q \times Input \rightarrow Q$$

What this is saying is that the function, named **F**, requires two values, a State value, (a member of *Q*) and an *Input* value and it will then return a State value. Within the collection of internal states we will identify an *initial* state, in which the system will always start up, and a set of *terminal* states, which may be different or which may include the initial state. If the function F is defined for every possible combination of a state and an input then it will be called a *complete* function. Under these conditions the behaviour of the system is fully determined. If there are situations where the function is not defined when in a specific state and receiving a specific input then the system will simply halt and no further behaviour is possible, unless the system is reset in some way.

We now proceed to the formal definition.

Definition 1.2.1.
Let *Input* be a finite alphabet. A *finite state machine* (or *finite automaton*) over *Input* consists of the following components (see [25]):

1. A finite set *Q* of elements called *states*.
2. A subset *I* of *Q* containing the *initial states*.
3. A subset *T* of *Q* containing the *terminal states*.
4. A finite set of *transitions* that give, for each state and for each letter of the input alphabet, which state (or states), if any, to go to next. This can be represented as a, possibly partial, function

$$\mathbf{F}: Q \times Input \rightarrow P\ Q,$$

where **F**(*q, input*) contains the states the machine is allowed to go to from state *q* when it receives the input *input* (recall that *P Q* denotes the set of all subsets of the set *Q*, usually referred to as the *power set of Q*).

The set *Input* can involve many different things, it could be specific user keyboard actions or commands, perhaps mouse clicks or movements or the receipt of a signal from a sensor or other monitoring device.

The set of states, *Q*, could be indicated by the values of certain important system variables, by the mode of behaviour of the system and so on. In a user interface the sort of screen that is visible could be a state, in a control application the state might indicate whether some device is active or not.

The transitions could be such that in a given state and under the influence of a particular Input value the system could respond with several possible operations, in other words the next state could be a set of possible states and there is no information as to which one it might be. This is described by the term *nondeterminism*.

Each triple $(q, input, q') \in Q \times Input \times Q$, where $q' \in \mathbf{F}(input,q)$, is called an *arc* of the automaton.

We frequently use the notations

$$q \xrightarrow{input} q'$$

and

$$input\colon q \rightarrow q'$$

to indicate the edge. If $\mathbf{F}(input,q)$ is a set of values, say $\{q_1, q_2, ...q_n\}$then the picture could be:

$$\begin{tikzcd}
 & q_1 \\
q \arrow[ur, "input"] \arrow[r, "input"] \arrow[dr, "input"'] & q_2 \arrow[d, dotted, no head] \\
 & q_n
\end{tikzcd}$$

A sequence of arcs is called a *path*.

$$\begin{tikzcd}
q \arrow[r, "input_1"] & q_1 \arrow[r, "input_2"] & q_2 \arrow[r, "input_3"] & q_3 \arrow[r, dashed, "input_n"] & q_n \arrow[r, dashed, no head] & {}
\end{tikzcd}$$

Here a path starting from state q is labelled by the inputs: $input_1$, $input_2$, $input_3$, ..., $input_n$,... and passes through the states q_1, q_2, q_3 ,...., q_n and so on.

The automaton, or machine is presented with an input string of letters (or characters) representing the abstraction of the set of actual inputs that it reads letter by letter starting at the leftmost letter. Beginning at one of the initial states, the letters determine a sequence of states along a path. The sequence may be interrupted when there is no edge corresponding to the letter read and, in this case, the input string is rejected. Otherwise, the sequence ends when the last input letter has been read. If the last state of the sequence is a terminal one, the input string is accepted, otherwise it is rejected. A sequence of inputs may determine a *set* of paths if there are situations where several transitions with the same label leave a particular state.

The set $L \subseteq Input^*$ of all the strings accepted is called the *language accepted* (or *recognised*) by the automaton.

Note: $Input^* = Input^+ \cup \{\varepsilon\}$, where $Input^+$ is the set of all finite sequences of characters from *Input* and < > is the empty sequence. In some cases we also refer to $Input^*$

by the formula seq(*Input*) and the empty sequence is sometimes denoted in some works by the symbol 1 , λ or ε.

If $input_1$, $input_2$, $input_3$,....,$input_n \in$ *Input* then the sequence consisting of these elements assembled in this order will be written as:

$$input_1 :: input_2 :: input_3 ::....:: input_n$$

which is an element of seq(*Input*).

The automaton is called *deterministic* if there is only one initial state (i.e. $I = \{q_0\}$) and for each state and each letter there is at most one single next state (i.e. **F** can be regarded as a partial function **F**: $Q \times Input \rightarrow Q$). Thus all the arcs leaving a given state must have different input labels. Otherwise, the automaton is called *nondeterministic*. Obviously, the path through a deterministic automaton is fully determined by the input string, whereas for the nondeterministic case, the choices of the initial state and the next state selected for each letter and current state are made randomly from a set of possible options. In this latter case, a string is accepted if at least one path determined by it through the machine ends in a final state. Although nondeterministic automata appear to have extra power over the deterministic ones, it has been proved that any language accepted by a nondeterministic automaton can be also accepted by a deterministic one, [22].

The transition diagram of such a machine is described using circles to indicate the states, arrows to indicate the transitions and labels on the arrows to indicate the transition labels.

In the following example there are three states called *a*, *b*, and *c*. There are two possible inputs, namely *0* and *1*. The diagram describes the function **F**: $Q \times Input \rightarrow P\, Q$ which in this case can be replaced by a function **F**: $Q \times Input \rightarrow Q$ which is partial since **F**(*b*,*0*) is undefined.

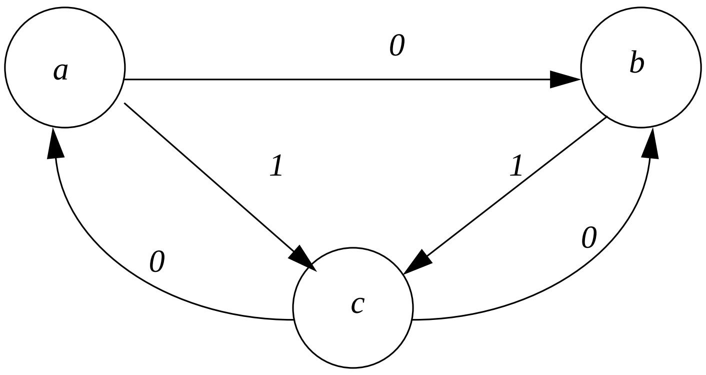

Figure 1.2. A simple finite state machine.

In Figure 1.2 we have a deterministic machine with 3 states, $Q = \{a, b, c\}$ and two possible inputs, $Input = \{0, 1\}$ with the function being indicated by a set of labelled arrows. Let us make *a* the initial state and {*b*} the set of terminal states. If the system is in state *a* and an input *1* is received then the next state is *c*; if the input had been *0*, however, the next state would have been *b.* When in state *b* an input *0* sends the system to state *b* (no change in state) whereas an input of *1* would cause the system to change to state *c*, and so on.

So $\mathbf{F}(a, 0) = b$; $\mathbf{F}(a, 1) = c$; $\mathbf{F}(c,0) = \{a, b\}$; etc. Also $(a, 0, b)$ and $(a, 1, c)$ are arcs.

Now we see that the following strings of inputs will be recognised by the machine since there is a path leading from state *a* to state *b* labelled by the input sequence:
0; *0*::*1*::*0*; *0*::*1*::*0*::*0*; etc. there are many more. Note that *0*::*1*::*0* defines two paths, one through *a, b, c, b* and the other through *a, b, c, a* only the former leads to the terminal state. The sequence *0*::*1*::*0*::*0* defines a path through the states: *a, b, c, a, b* and a path through *a, b, c, b,* ? which halts before the last input is read.

However the strings *1*::*1*::*0*::*1*; *1*::*0*::*1* are not recognised, the first because it cannot complete the process and the second because it ends in a state which is not a terminal state.

The machines considered above simply react to the environmental inputs by changing state. Some machines, however, do more than this, they generate some message, command or other *output*. These will be called *finite state machines with output.*

Definition 1.2.2
A *finite state machine with output* consists of the following components (see [25]):
1. A finite set *Q* of elements called *states*.
2. A subset *I* of *Q* containing the *initial states*.
3. A set *Output* of possible outputs.
4. A finite set of *transitions* that give, for each state and for each letter of the input alphabet *Input*, which state (or states), if any, to go to next. This can be represented as a partial function
   $\mathbf{F}: Q \times Input \rightarrow P\ Q,$
5. A function $\mathbf{G} : Q \times Input \rightarrow Output$ which determines for each state and input received the output delivered. Note that **F** or **G** could be partial functions.

We will refer to the machine by its tuple $(Q, I, Input, Output, \mathbf{F}, \mathbf{G})$.

Example 1.2.2

Consider the following machine, illustrated in Figure 1.3 where:

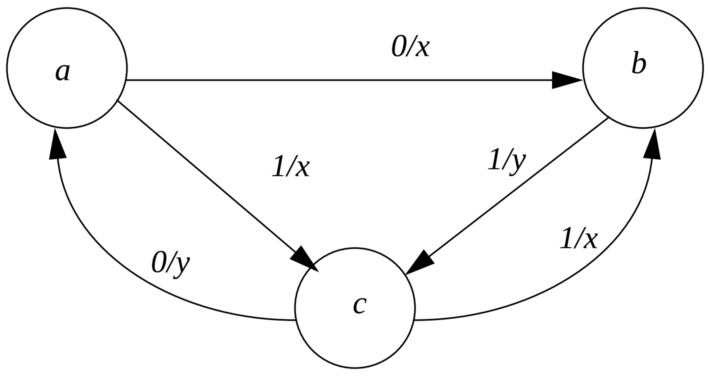

Figure 1.3 A simple finite state machine with output.

Q = {*a*, *b*, *c*} is the set of states, there are two possible inputs, *Input* = {*0, 1*} and two possible outputs - *Output* = { *x, y* }. The function **F** is given in Figure 1.3 as is the function **G** which is defined by:

G(*a,0*) = *x*; **G**(*a,1*) = *x*, **G**(*b,1*) = *y*; etc.

An arc of the form:

$$q \xrightarrow{in/out} q'$$

means that **F**(*q, in*) = *q'* and **G**(*q, in*) = *out*.

Now consider starting the machine is started in state *a* and receives the input sequence *0*::*1*::*0* then the sequence of state is *a, b, c, b* and as it traverses this path the outputs *x, y,* and *x* are produced in this order. Thus the input sequence *0*::*1*::*0* produces a corresponding output sequence *x*::*y*::*x* , which is an element of seq(*Output*). If we were to start the machine off in a different state it will, in general, produce a different output sequence. If we use another input sequence then another output sequence will be generated.

Given a particular finite state machine with outputs, assuming that we always start off the machine in the same initial state, then each input sequence will generate an output sequence or it will halt immediately because the state transition function is not fully defined for the first input. So the machine will act as a translation function from input sequences to output sequences.

The usage of finite state machines is a widely followed practice, particularly in sequential hardware design [26]. In some types of software system design, they are also used, for example in user interface design [27]. To adequately design and test user interfaces we need a sort of formalism for modelling both the computer system and the user. Since many models of human behaviour involve the concept of state, it is natural to try to represent the interaction between the system and its users in this way. Let us illustrate the use of finite state machines for modelling user interfaces with the following example.

Example 1.2.3
The system specified here is a very simple word processor that allows the following operations:

type a character;
insert a mathematical symbol (we call this operation *insert_symbol*);
insert a drawing (we call this operation *insert_drawing*);
delete a character, a drawing or a symbol;
move the cursor to the right or to the left using the left and right arrows - we call the operations *move_right* and *move_left*.
select (highlight) a part of the document; this can be done as follows:
- the word processor starts selecting a part of the document when a certain input is received - let us call this operation *activate_select*.
- characters, drawings or mathematical symbols in the document are selected by moving the cursor to the right or to the left (we call these operations *move_right* and *move_left* respectively)
- the action ends when a certain input is received (we call this operation *deactivate_select*) when a character is inserted (*type*), when a character, symbol or drawing are deleted (*delete*) or when the main menu of the processor is selected (*select_menu*).

cut a character, a drawing or a symbol.
copy a character, a drawing or a symbol.
paste a character, a drawing or a symbol.

We assume that the word processor has a menu with the following options: *drawing*, *symbol*, *cut*, *copy* and *paste*. The first two options, if selected, allow the insertion of drawings or mathematical symbols respectively into the document.

We can represent the machine in a simple diagram (Figure 1.4) which shows how these different functions are inter-related. So whilst typing it is possible to select the menu which provides access to the functions for drawing diagrams, typing formulae, cutting and pasting etc.. The diagram helps us to organise these functions in a sensible way, to restrict access some some functions when it is inappropriate to use them and to *control* the entire program.

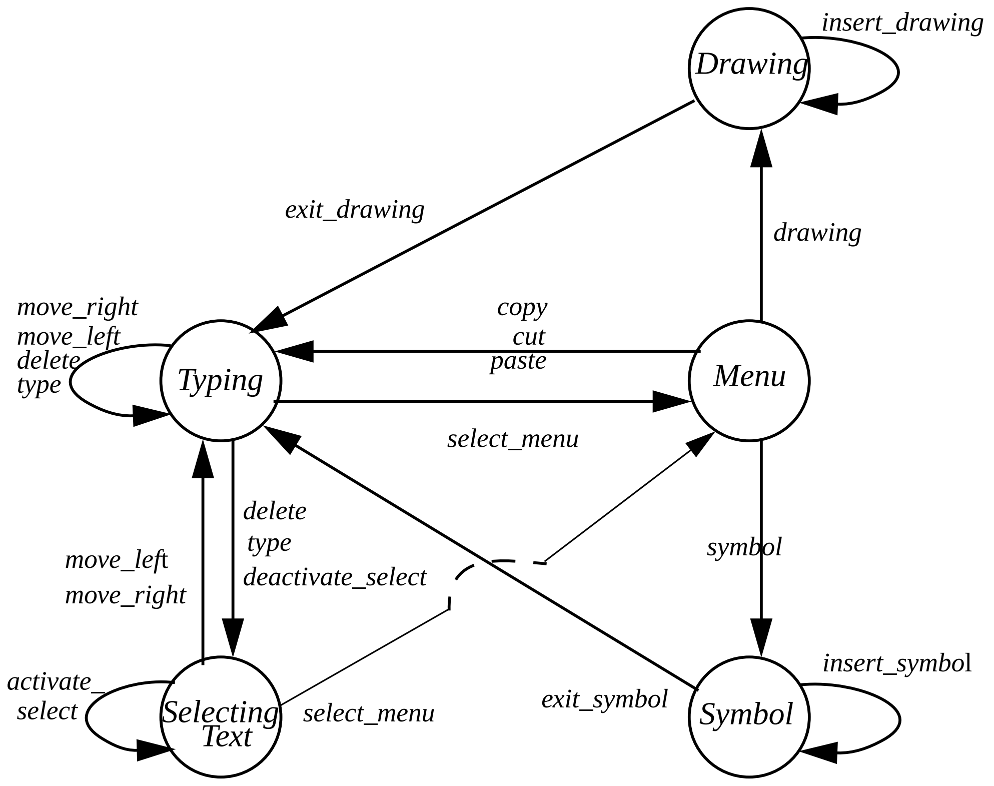

Figure 1.4 A simple finite state machine model of an interface.

It is quite natural to model an user interface in this way, quite complex user interfaces can be modelled similarly. The state transition diagram gives a clear and intuitive description of the evolution of the system and the operations that are allowed in any state. The current state of the system can be very easily traced using the sequence of operations that the system has already performed. Important properties of the model can be deduced from the diagram, for example if there are states that cannot be reached, or states from which there is no exit. Such information is very important in the analysis of user interfaces. The main drawback is that the modelling of large and complex machines is difficult, in other words the technique does not scale very well without introducing a more structured model, such as the statechart (see the next section, 1.3).

It is relatively easy to create an implementation once a state machine description is available, or, at least, in software terms the outline control structure of the software can be generated. State machines do not provide a sensible model of the data and the detailed processing so there are limitations to how far this can be taken. In hardware design, simple sequential systems are often implemented directly from state diagrams and software is available to do this. The solutions may not be optimal but, again, optimising software can be used to overcome this.

The state machine model can be enhanced, thus giving rise to more complex, but similar, models that can be used to reduce the number of states of a state diagram or when the standard finite state machine is too limited to cope with the required application.

These have been extensively studied and include pushdown machines, stack machines, tree automata and many more, [22], [28]. Rather than cover them all we will briefly look at another extended state machine. Later on we will introduce our main model which will be seen to be a generalisation of all of these types of machine. Such a model is, for example, the *traceable finite state machine* (or *traceable automaton*); here we have an additional stack that keeps a record of the states the system has visited up to the current moment.

Definition 1.2.3.
More precisely, a *deterministic traceable automaton* [29] over a finite alphabet *Input* consists of the following components:

1. A finite set of states Q.
2. An initial state $q_0 \in Q$.
3. A set of terminal states $T \subseteq Q$.
4. A *state stack* $S \subseteq Q^*$.
5. A partial function $\mathbf{F_t}$: $Q \times Input \rightarrow Q \cup \{Trace\}$.

The state stack is a simple memory device that keeps a record of what the last state was in case it is needed. It is represented by a sequence of values from the set of states, Q. We can only access the stack through the topmost value. Under certain conditions, when the *Trace* operation is called, the top of the stack is read and the system returns to the state that that names.

Initially then, the automaton is in one of its initial states and the state stack is empty. The function F_t determines the next state according to the current state q and the input symbol *in*, read by the automaton.

a) if $F_t\ (q, in) = q'$, then q is pushed to the top of the stack, the control goes to the state q'.

b) if $F_t\ (q, in) = Trace$ and the stack is not empty then the control goes to p which is the state on top of the stack and p is popped from the stack. If the stack is empty then the machine stops and the input string is rejected.

Using this device we can produce a simpler and more powerful model of the word processor, illustrated in Figure 1.5.

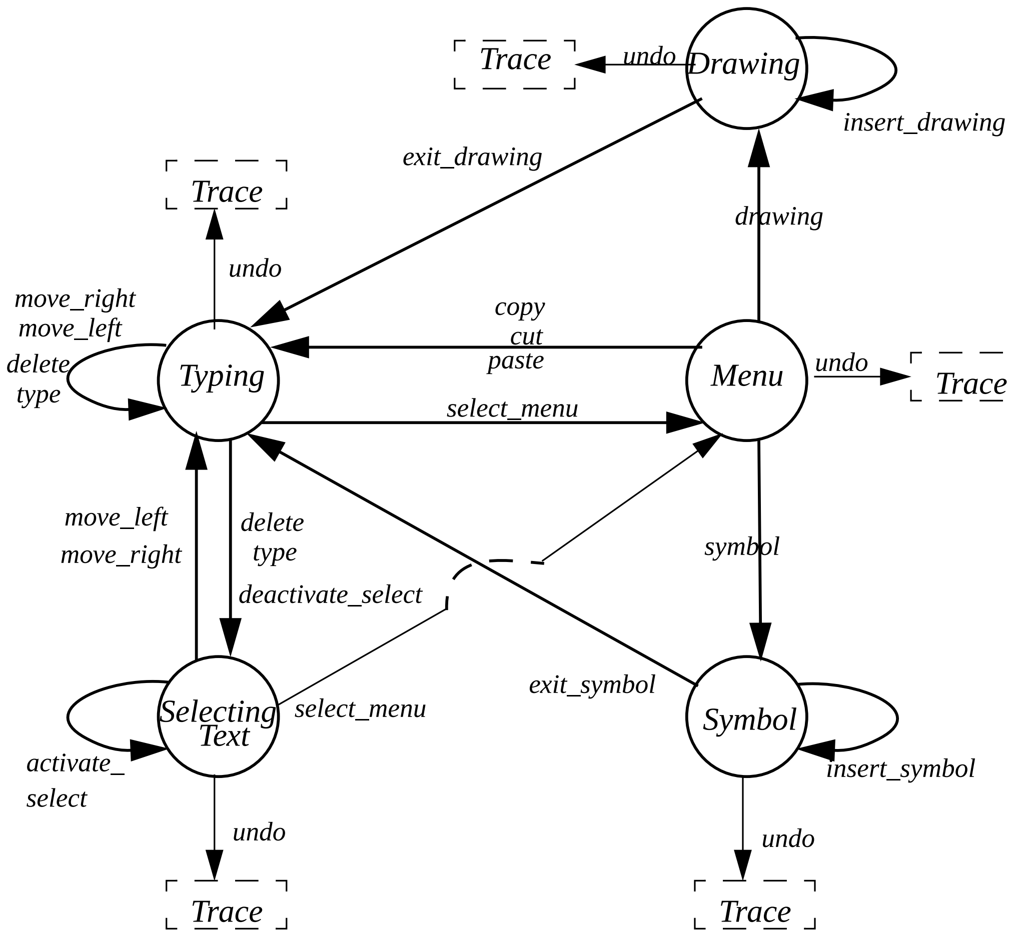

Figure 1.5 A traceable machine model of a word processor..

Traceable automata are more powerful then standard finite state machines (see [30]); they can be used to describe user interfaces - especially their menus - that cannot be handled by standard finite state machines. For example let us consider that the word processor has an *undo* option that cancels the last operation performed and takes the system back to the previous state. If this is to be described in the specification we have to keep track of the operations performed by the word processor up to the current moment and there is no way we can do that using a standard finite state machine. However, a traceable automaton can describe this extra feature very easily, the *Trace* transitions are added to the initial model.

However, finite state machines alone - standard, traceable or other variants - cannot describe completely a system, since there is no reference to the internal data of the system (i.e. a word document in the above example or the records in a database) and as to how this data is affected by each operation in the state transition diagram. Thus, other specification methods such as Z or VDM have to be used for modelling the data aspects of the system. Furthermore there needs to be a mechanism (or a meta-model) that unifies the two views of the specification. Without such a mechanism, the two models of the system - one representing the system control and the other the system data may be inaccurate since there is no way of capturing the dependency between the two models - a transition in the control structure of a problem may affect the program data and vice-versa. For example, the *copy* and *cut* transitions from the *Menu* state in the example above will only be enabled if part of the document has already been selected. It is possible to describe the state transitions in a language like Z but it is a clumsy and unsatisfactory way to do it.

1.3. Statecharts

Finite state machines with output are also the basis of more complex specification languages. *Statecharts*, [31], for example, adapt the state machine formalism by allowing nested states, i.e. a state of a chart can be represented itself as a chart at a lower level. In this way the complexities of a large state diagram can be managed. The formalism introduces a structuring facility and a refinement process. When building state machine models we can identify a type of state (called a *superstate*) from which a lot of information is abstracted out and this can be replaced at a later stage by a more detailed part of the model. Thus a superstate can be replaced later with a submachine or a collection of submachines either running in parallel or a which allow a degree of choice.

Thus, there are several types of states in statecharts:

(i) OR states. If a system is in this state it can only be in one of its sub-states. Therefore the sub-states of an or-state behave like a finite state machine.

(ii) AND states. When a machine is in this state, it is in all of its sub-states which are “running concurrently”.

(iii) BASIC states. This is a state with no sub-states.

If a statechart is in an OR state, some state of the chart within it has to be active. Conversely, if a chart is in some state, the “parent” OR state has to be active. For an AND state, all its sub-states have to be active. Such restrictions select sets of states called configurations from all possible set of states a chart can be in.

A general label of a transition in a statecharts specification has the form $e[c]/a$, where e is the event expression (conjunction of events or negated events) that triggers the transition, c is a condition (boolean expression) that has to be satisfied for the transition to fire and a is the resulting action. All the transition label components are optional (e.g. no transitions in Figure 1.6 contain the ’$[c]$’ component and only two have the ’a’ component). Besides allowing actions to appear along transitions, they can also appear associated with the entrance to or exit from a state. There are several models of a *step*, the process by which a computation in a statechart proceeds. They have widely differing consequences. One popular semantics is as follows: actions associated with the entrance to a state q are executed in the step in which q is entered as if they appear on the transitions leading into q. Similarly, actions associated with the exit from q are executed in the step in which q is exited, as if they appear on the transition exiting q. In addition, each state can be associated with *static reactions* (SRs) that are to be carried out as long as the system is in (and not exiting) the state in question.

Example 1.3.1.
An example of a print buffer between a network link and the actual printer is presented

in Figure 1.6. The initial specification is a two state model (shown on top), the *power* button selects between the two states. In turn *On* is an AND state that contain two sub-states - these are separated by a dashed line and run concurrently, the one on the right filling a buffer with data from the network and the other printing. These are both OR states and contain two BASIC states to represent normal operation and no operation when the buffer is either full or empty.

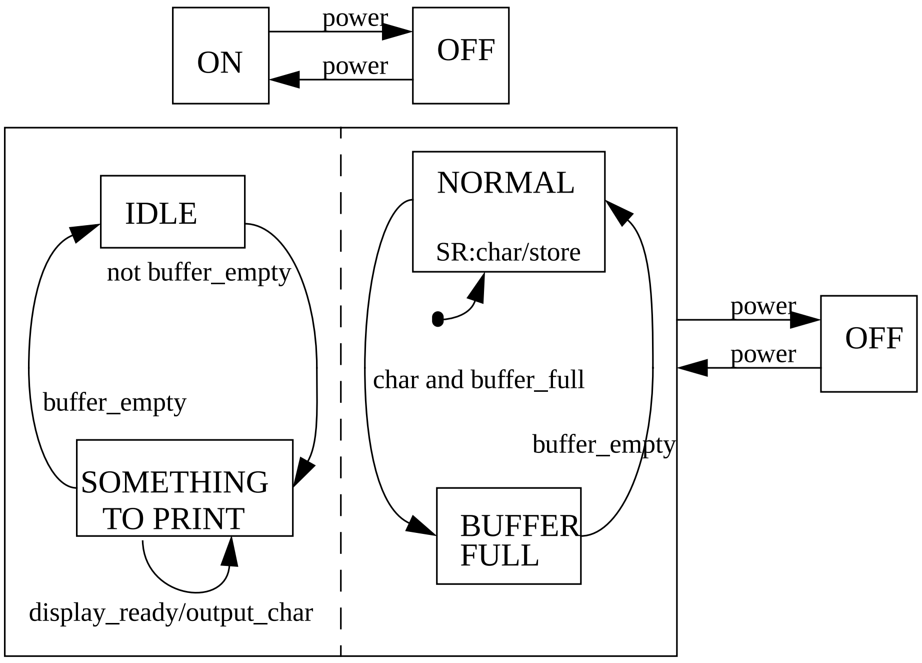

Figure 1.6 A simple statechart model of a buffer.

Thus the process of constructing a statecharts specification is gradual, an hierarchy of specifications can be built and this results in an increase in the clarity and efficiency of the specification and provides a means for avoiding the state explosion problem. As the use of statecharts has developed a number of special features and customs have evolved. This may have been convenient for design engineers since it enabled them to model certain specific aspects of systems more easily, the consequence of this, however, has been a loss of clarity and consensus concerning the semantics of a statechart. This has been compounded by the widespread use of tool, STATEMATE [32], which supports the use of statecharts. For example, it has been possible to design models where transitions traverse directly across many levels of superstate boundaries and to build models where it is unclear what the system should do. There could be several transitions enabled at the same time which could carry out contradictory actions. A number of attempts have been made to describe the semantics of statecharts and of the STATEMATE versions of statecharts. These have partially clarified the situation but there is still much confusion about the true meaning of the differing dialects of statecharts, [33].

1.4. A general computational model: X-machines.

Generally, finite state machines and their relatives are very good at expressing the dynamics of a system. The use of graphical elements in a specification methodology is attractive from the point of view of user understanding, conveying dynamic information concisely and intuitively. Thus control aspects of software programs are very often described as finite state machines. However, all these state transition models lack the ability to represent complex data structures. In fact, it is difficult to model any non-trivial data structure using finite state machines. Also, other, more complex, machine models such as the Pushdown Automata and Turing machines are too low level and hence of little use for specification and design of real systems.

However, there is a generalisation of all these machines that can, when combined with suitable data processing methods, can provide us with an appropriate environment for the description and analysis of arbitrary systems. The *X-machine* integrates the two aspects of a software system specification - the control and the data processing - while allowing them to be described separately. A finite state machine like diagram is used to model the system control while the data processing can be represented using methods such as Z or VDM or any appropriate formal notation. The method can combine the dynamic features of graphical methods with suitable data representations, thus sharing the benefits of both these worlds.

Introduced by Eilenberg in 1974, [28], *X-machines* have received little further study. Holcombe, [34], proposed the model as a basis for a possible specification language and since then a number of further investigations have demonstrated that this idea is of great potential value for software engineers. In its essence an X-machine is like a finite state machine with a set of states, Q, but with one important difference. A *basic data set*, X, is identified together with a set of *basic processing functions*, Φ, which operate on X; Φ is called the *type* of the machine.

So we identify a set of basic processing functions and a set X of data that these functions can process. The way these functions are controlled is by way of a state diagram like the finite state machine model.

Each arrow in the finite state machine diagram is then labelled by a function from Φ, thus the *next state (partial) function* will be of the form

$$\mathbf{F}: Q \times \Phi \to Q.$$

Then the sequences of state transitions in the machine determine the processing of the data set and thus the function computed. The data set X can contain information about the internal memory of a system as well as different sorts of output behaviour so it is possible to model very general systems in a transparent way.

This method allows the control state of the system to be easily separated from the data set, the set X is often an array consisting of fields that define internal structures such as registers, stacks, database filestores, input information from various devices, models of screen displays and other output mechanisms.

Consider a simple, abstract example, illustrated in Figure 1.7:

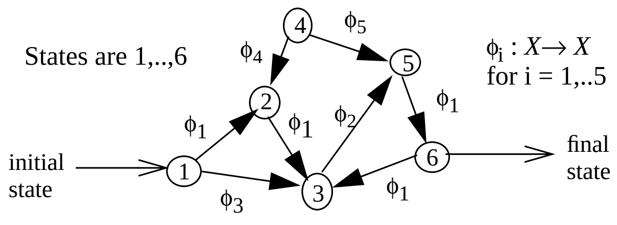

Figure 1.7 A simple X-machine.

We haven't specified X or the functions ϕ_i that belong to the set Φ but this is sufficient to provide a basic idea of the machine. It starts in a given initial state (control state) and a given state of the system's basic data set X (the data state); there are a number of paths that can be traced out from that initial state, these paths are labelled by functions ϕ_1, ϕ_2 etc. Sequences of functions from this space are thus derived from paths in the state space and these may be composed to produce a function that may be defined on the data state. This is then applied to the value x, providing that the composed function is defined on X. This then gives a new value, $x' \in X$ for the data state and a new control state. Usually, the machine is *deterministic* so that at any moment there is only one possible function defined (that is the domains of the functions emerging from a given state are mutually disjoint).

From the diagram we note that there is a possible sequence of functions from state 1 to state 6, here a specified terminal state, labelled by the functions ϕ_1, ϕ_1, ϕ_2, ϕ_1. Assuming that each value is defined this path then transforms an initial value $x \in X$ into the value $\phi_1(\phi_2(\phi_1(\phi_1(x)))) \in X$. So we are assuming that $x \in$ domain ϕ_1; $\phi_1(x) \in$ domain ϕ_1; $\phi_1(\phi_1(x)) \in$ domain ϕ_2; and $\phi_2(\phi_1(\phi_1(x))) \in$ domain ϕ_1. The computation carried out by this path is thus a transformation of the data space as well as a transformation of the control space.

Clearly we need to specify some relationship between the input and output information of the overall system and the data set X, especially when X contains information that is not directly involved with the system input and output. This is done by specifying two sets Y and Z, to represent the *input* and *output information*, respectively.

Two *coding functions*

$\alpha: Y \to X$

and

$\beta: X \to Z$

describe how the input is coded up prior to processing by the machine and how the subsequently processed data is then prepared (or decoded) into suitable output format.

In many cases the inputs and outputs of the machine will behave as orderly streams of symbols, i.e.

$$Y = Input^{*} \text{ and } Z = Output^{*},$$

where *Input* will be called the *input alphabet* and *Output* the *output alphabet*.
Then the set X will often have the form

$$X = Output^{*} \times Mem \times Input^{*},$$

where *Mem* is a (possibly infinite) set called *Memory.*

In this case the input code will define the *initial memory value* $m_0 \in Mem$, i.e.
$\forall\, in^{*} \in Input^{*}$ an input sequence

$$\alpha(in^{*}) = (<>, m_0, in^{*})$$

and the output code will extract the output sequence of any data value (out^{*}, m, in^{*}) if the machine has read the whole input sequence (i.e. $s = <>$), i.e. $\forall\, out^{*} \in Output^{*}$, $m \in Mem, in^{*} \in Input^{*}$

$$\beta(out^{*}, m, in^{*}) = \begin{cases} out^{*}, & \text{if } in^{*} = <> \\ \varnothing, & \text{otherwise} \end{cases}$$

The basic idea is that the machine has some internal memory, *Mem*, and the stream of inputs determine, depending on the current state of control and the current state of the memory, the next control state, the next memory state and any output value.

Of special interest for specifying user interfaces are those X-machines in which each processing (partial) function $\phi \in \Phi$,

$$\phi: Output^{*} \times Mem \times Input^{*} \rightarrow Output^{*} \times Mem \times Input^{*}$$

is of the form whereby, given a value of the memory and an input value, ϕ can change the memory value and produce an output value, the input value is then discarded.
More formally,

$\phi(out^{*}, m, <>)$ is undefined $\forall\, out^{*} \in Output^{*}, m \in Mem$,

and

$\forall\, m \in Mem, in \in Input$ either

$\phi(out^{*}, m, in :: in^{*})$ is undefined $\forall\, out^{*} \in Output^{*}, in^{*} \in Input^{*}$ or

$\exists\, m' \in Mem, out' \in Output$ (that depend on m and in) such that

$\phi(out^{*}, m, in :: in^{*}) = (out^{*} :: out, m', in^{*}), \forall\, out^{*} \in Output^{*}, in \in Input^{*}$

(where $a::b$ is the concatenation of two strings or sequences, a and b)

Machines satisfying these properties are called *stream X-machines* [35].

Clearly any processing function ϕ of a stream X-machine is completely determined by the values of *out* and *m'* as a function of *m* and *in*. Thus, we can simplify the notation by referring to such a processing function as $\phi: Mem \times Input \rightarrow Output \times Mem$. That is, instead of saying

$\phi(out^* , m, in :: in^*) = (out^* :: out, m', in^*)$,

with $m, m' \in Mem$, $out \in Output$, $in \in Input$, $\forall\ out^* \in Output^*$, $in^* \in Input^*$ we say

$\phi(m, in) = (out, m')$, the rest being understood implicitly.

Thus we will often describe the functions ϕ in the form:

$\phi : (m, in) \rightarrow (out, m')$

$\phi : (m_1, in_1) \rightarrow (out_1, m_1')$

$\phi : (m_2, in_2) \rightarrow (out_2, m_2')$

and so on.

This is the notation we shall be using in the rest of this book.

1.5. An X-machine example.

The word processor described in Example 1.2.2 can easily be specified as a stream X-machine. Here, the operations of the processor will become (partial) functions that operate on the Cartesian product of the input sequence, internal data structure (memory) and the output sequence - the set of all these partial functions will make up the type of the machine, Φ. Thus suitable data types must be defined and the input, output and memory set will consists of a Cartesian product or a disjoint union of these types.

Let *CHARACTERS*, *SYMBOLS* and *DRAWINGS* be the set of characters, mathematical symbols and drawings that the system can process (for now we will not make any assumptions about the shape and size of these drawings) and let

$ELEMENTS = CHARACTERS \cup SYMBOLS \cup DRAWINGS$

Let also *STRINGS* be the set of all sequences of *ELEMENTS*, i.e.

$STRINGS = ELEMENTS^* = \text{seq}(ELEMENTS)$,

Then the document the word processor operates on can be described as a tuple (x, z, y, p), with $x, z, y \in STRINGS$ and p takes two values, either $p = right$ or $p = left$. Here x is the first non-highlighted part of the document and the reverse of the sequence *y*, *rev*(*y*), is the other non-highlighted part of the document. If $p = right$, then the selected text is *z* and the cursor is on its right hand side. If $p = left$, then the highlighted text is *rev*(*z*) and the cursor is on its left hand side.

For example, if the document is

"ab**cde**f g"
$\quad\quad\uparrow$

(i.e. $\uparrow$ marks the position of the cursor and the highlighted section is written as boldface).

then x = "ab", y = "g f", z = "cde" and p = right.

If the document is

“*ab**cde**f g*”
$\uparrow$

then x = *“ab”*, y = *“g f”*, z = *“edc”* and p = *left*.

If no part of the document is selected (i.e. $z = \varepsilon$), then the value of p is not relevant.

Thus, the set of documents the word processor operates on is defined as

$DOCUMENTS = STRINGS \times STRINGS \times STRINGS \times POSITIONS,$

where

$POSITIONS = \{left, right\}.$

The internal memory of an X-machine specification of the word processor will consists of suitable representations for the document and for the text selected in clipboard - this is a sequence of *ELEMENTS*. Thus

$M = DOCUMENTS \times STRINGS.$

The current memory value will be denoted by $((x, y, z, p), c)$, where (x, y, z, p) represents the current document and c is the part of the document copied into the clipboard. The initial memory value is

$m_0 = ((\varepsilon, \varepsilon, \varepsilon, left), \varepsilon).$

The input alphabet *Input* will consists of the inputs required for the successful execution of all operations. These inputs are normally defined in some specification document but for now we will make some obvious assumptions about them. Also, the names we will be using are self-explanatory.

We shall assume that, with the exception of *type*, *insert_symbol* and *insert_drawing* all the commands of the processor are triggered by a single input each (e.g. *back_space* triggers *delete*, *left_arrow* triggers *move_left*, *cut_option* triggers *cut*, etc.). Thus if we denote by $Input_0$ the enumeration of these inputs, i.e.

$Input_0 = \{back\_space, left\_arrow, right\_arrow, cut\_option, ... \}$

then *Input* can be written as

$Input = Input_0 \cup ELEMENTS$

(obviously *type* is triggered by *CHARACTERS*, *insert_symbol* is triggered by *SYMBOLS* and *insert_drawing* is triggered by *DRAWINGS*).

For the sake of simplicity we will consider that a system output will consist of the current document, i.e.

$Output = \ DOCUMENTS$

Since our specification is a stream X-machine the definitions of all processing functions will be of the form

$\phi(((x, y, z, p), c), in) = (out, ((x', y', z', p'), c'))$, where $out \in Output$, $in \in Input$ and $((x, y, z, p), c), ((x', y', z', p'), c') \in M$

Some samples of processing functions are given next.

if $in \in CHARACTERS$ then **type**$(((x, z, y, p), c), in) = (out, ((x', z', y, p), c))$, where

$out = (x', z', y, p)$,
$x' = x :: in$,
$z' = \varepsilon$

i.e. **type** removes the selected part of the document - if any - and inserts a character on the left of the cursor.

if $in \in DRAWINGS$ then **insert_drawing**$(((x, z, y, p), c), in) = (out, ((x', z', y, p), c))$, where

$out = (x', z', y, p)$,
$x' = x :: in$,
$z' = \varepsilon$

i.e. this is similar to **type**.

delete$(((x, z, y, p), c), back\_space) = (out, ((x', z', y, p), c))$, where

$out = (x', z', y, p)$

$$x' = \begin{cases} x, \text{ if } z \in STRINGS - \{\varepsilon\} \\ \mathbf{front}(x), \text{ if } z = \varepsilon \end{cases}$$

$z' = \varepsilon$

i.e. it deletes the selected part of the document if this is not empty and removes the first character on the left of the cursor otherwise.

Note: For an alphabet A, **front**: seq(A)$\rightarrow$ seq(A) is a function that removes the last character of a non-empty sequence of elements in A and leaves the empty sequence unchanged, i.e.

front$(< >) = < >$ and
front$(x :: a) = a$ for all $x \in$ seq(A), $a \in A$.

Refinement is quite straightforward in the context of X-machines: a processing function can be itself described - maybe at a lower level - as an X-machine thus the refinement process is achieved by replacing certain arcs in the original machine with suitable X-machines. For example, we can describe in more detail the process of inserting a drawing by the above word processor. For simplicity sake, we shall assume that only lines and rectangles can be inserted and that the insertion process consists of the following sequence of steps:

- select shape (line or rectangle)
- select the first point (this is a corner in the case of rectangles)
- select the second point (this is the opposite corner in the case of rectangles).

Let $POINTS$ be the set of points that can be selected and

$SHAPES = \{line, rectangle\}$

and let

$DRAWINGS_R = SHAPES \times POINTS \times POINTS$

(i.e. a drawing is considered to de defined by shape and positions of two points)
Let also

$ELEMENTS_R = CHARACTERS \cup SYMBOLS \cup DRAWINGS_R$,

$STRINGS_R = ELEMENTS_R^*$,

$DOCUMENTS_R = STRINGS_R \times STRINGS_R \times STRINGS_R \times POSITIONS$

and

$M_R = DOCUMENTS_R \times STRINGS_R$

(more generally, for any of the sets, A above, A_R denotes the set obtained by substituting *DRAWINGS* with $DRAWINGS_R$ in the definition of A)

Then the memory, input and output sets of the refined X-machine are:

$M_{Ref} = M_R \times (SHAPES \times POINTS)$,

$Input_{\mathrm{Ref}} = (Input - DRAWINGS) \cup SHAPES \cup POINTS$,

where *Input* is the input alphabet of the original X-machine specification,

$Output_{\mathrm{Ref}} = DOCUMENTS_R$.

With the exception of *insert_drawing* no processing function of the original machine was triggered by inputs in the *DRAWINGS* set, so any such

ϕ: $M \times (Input - DRAWINGS) \rightarrow Output \times M$

can be transformed into a processing functions of the refined machine

ϕ_{Ref}: $M_{Ref} \times Input_{Ref} \rightarrow Output_{Ref} \times M_{Ref}$

in a straightforward manner. Indeed, let

gamma: $M \times (Input - DRAWINGS) \rightarrow Output$ and

memory: $M \times (Input - DRAWINGS) \rightarrow M$

be the projections of ϕ onto *Output* and *M* respectively, i.e. for any $m \in M$, $in \in Input$,

$\phi(m, in) = (gamma(m, in), memory(m, in))$.

Then for any $m_R \in M_R$, $m_1 \in SHAPES \times POINTS$ and $in \in Input - DRAWINGS$,

$\phi_{Ref}((m_R, m_1), in) = (gamma_R(m_R, in), ((memory_R(m_R, in), m_1)$, where

$gamma_R$: $M_R \times (Input - DRAWINGS) \rightarrow Output_R$ and

memory: $M_R \times (Input - DRAWINGS) \rightarrow M_R$

are obtained by substituting *DRAWINGS* with $DRAWINGS_R$ in the definitions of *gamma* and *memory* respectively.

For example **type** becomes $\mathbf{type_R}$ defined by

if $in \in CHARACTERS$ then

$\mathbf{type_R}((((x, z, y, p), c), (u, v)), in) = (out, (((x', z', y, p), c), (u, v)))$, where

$out = (x', z', y, p)$,

$x' = x :: in$,

$z' = \varepsilon$

Note: $((x, z, y, p), c) \in M$, $u \in SHAPES$, $v \in POINTS$, $out \in Output_{Ref}$.

The core of the refinement process is achieved by substituting the arc labelled

insert_drawing with a stream X-machine that decomposes the insertion of a line or rectangle in the 3 steps described above (see Figure 1.8).

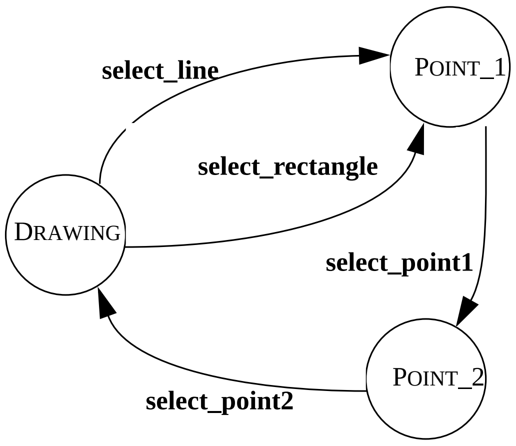

Figure 1.8 A simple stream X-machine example.

Some sample definitions of the new processing functions are given next.

select_line$(((x, z, y, p), c), (u, v)), line) = (out, (((x, z, y, p), c), (u', v)))$, where
$out = (x, z, y, p)$
$u' = line$

if $in \in POINTS$ then
select_point1$(((x, z, y, p), c), (u, v)), in) = (out, (((x, z, y, p), c), (u, v')))$, where
$out = (x, z, y, p)$
$u' = in$

if $in \in POINTS$ then
select_point2$(((x, z, y, p), c), (u, v)), in) = (out, (((x', z', y, p), c), (u, v)))$, where
$out = (x', z', y, p)$,
$x' = x (u, v, in)$
$z' = \varepsilon$

Obviously, any such refinement process has to be supported by a series of theorems that ensure that the transformation preserve, in some sense, the functionality of the original X-machine. This more theoretical point of view will be discussed in Part 2. In many practical situations though, as illustrated by the above example, the refinement techniques - such as the extension of the state space, the fundamental type X and the set of processing functions - can be appreciated easily from an intuitive point of view. This feature is one of the attractions of X-machines.

The advantages of the X-machine model include:
It is intuitive and a number of trial evaluations have indicated that it is easy to use.

It is built on current knowledge and does not involve any revolutionary concept. Indeed, the model is a blend of state diagrams and formal descriptions of data types and functions that can be expressed easily in a language such as Z or as functions in

ML or a similar functional language or using traditional mathematical notations.

It allows for the capture of the dynamic system information in a very intuitive manner. The use of state diagrams helps a great deal in this sense.

It allows a system to be specified at different levels. Indeed, a processing function from a high level X-machine specification can be expressed itself as a lower level X-machine.

It is sufficiently general to cater for all computational problems. Indeed, it is fairly clear that the Turing machine, which is accepted as the mathematical model of computation, can be easily described as an X-machine (see [28], [36]). Thus, unlike finite state machines, the X-machine can cope with very complex functionality - in fact the functionality of any hardware or software system can be described as an X-machine.

Like finite state machines, X-machines can be extended by adding new features to the original model. Similar to finite state machines, we can define a Traceable X-machine by adding a stack $S = Q^*$ that keeps a history of state changes and extending the state transition function $\mathbf{F}: Q \times \Phi \to Q$ to $\mathbf{F_t}: Q \times \Phi \to Q \cup \{trace\}$; obviously, the semantics of $\mathbf{F_t}$ is identical to that defined for traceable automata. For example, the *undo* operation can be added very easily to our X-machine specification of the word processor in this way. Traceable X-machines can be more suitable for describing certain types of user interfaces than standard X-machines. Unlike finite state machines though, this expansion does not extend the computation ability of the model (i.e. it can be shown easily that a traceable X-machine with basic data set X and type Φ can be simulated by an X-machine with basic type $X' = X \times S$ and type Φ' obtained by extending the processing functions in Φ with "push" and "pop" operations performed on the S component of X'; another proof of this fact is given in [37]). Thus the theoretical results developed for X-machines that will be presented in the next chapters can also be applied to traceable X-machines after suitable transformations.

One final issue worth discussing is the ability of the X-machine model to deal with real-time behaviour. The main formal specification methods (Z, VDM, OBJ, etc.) do not, in their original form, address the problem of modelling time in real-time systems; coping with time has usually required the augmentation of the language with special constructors and operations which have often caused a significant increase in the complexity of the notation and method and the consequent reduction in its attractiveness as an useful tool in industry to use. The same is also true of the process calculi such as CCS [38], which are good at modelling the communication in a concurrent system but need complex augmentation to handle data and time. Unlike these, X-machines allow the integration of time into the model in a very simple manner; for example the clock can be represented as the set of natural numbers, starting from some arbitrary point and advancing by one for a small enough time interval. The clock can be then easily incorporated into the machine memory and a special input (let us call this *tick*) will be used to increment the time by one unit; thus any "time" transition will be triggered by the last tick that brought the clock to the appropriate value (see also [39] for more detail).

Another approach is the extension of the X-machine model so that it can cope with hybrid or continuous behaviour. There are a number of formalisms that incorporates continuous transitions into finite state machines (e.g. phase transition systems [40], hybrid automata [41], hybrid state charts [42]), their common idea is to use arcs for representing discrete transitions and to associate each continuous transition with a state. This idea has also been used in the context of X-machines thus giving rise to hybrid machines or hybrid X-machines (see [43]). However, these hybrid X-machines require further study and we will not focus on them here.