Proof Let i and j be adjacent nodes in Q_n expressed as binary strings and let $i^{'}$ and $j^{'}$ be the bitwise complements of i and j so that

$$i + i^{'} = j + j^{'} = 2^n - 1$$

As i and j are adjacent, the complementary nodes $i^{'}$ and $j^{'}$ are adjacent. Thus, for every edge whose weight is z there exists another edge whose weight is $2(2^n - 1) - z$.

References

[1] F. Harary, *Graph Theory*. Addison-Wesley, Reading, 1969.

[2] F. Harary, J.P. Hayes and J-H. Wu, *A survey of the theory of hypercube graphs*, Comput. Math. Appl., to appear.

Department of Computer Science
New Mexico State University
Box 3CU Las Cruces
New Mexico 88003-0099
USA

Algebraic Techniques Of System Specification

Mike Holcombe

1 Introduction

The design of complex software systems is a relatively new occupation and is still in its infancy. With the rapid growth in the applications of microprocessor technology more and more areas of life are being affected and in some of this activity there is serious cause for concern. Many manufacturers are using microcomputers to control safety-critical systems. Such systems are usually defined to be systems, the malfunctioning of which could lead directly to injury or death on a small (local) or large (global) scale. Examples of recent products and systems that have caused injury or death through inadequate software design are :

- Chemical processing plants,
- Washing machines,
- Car cruise controls,
- Intensive care systems,
- Industrial robots, etc.

In many of these systems there is a serious problem in the formal specification of the total system and its environmental interaction. Most interactive systems involve three important components :

- the system,
- the environment,
- the user.

Each component interacts and 'communicates' with all the others and it is therefore crucially important that we take this into account at all stages of the design process, from specification, design, validation, maintenance and disaster analysis.

Much activity currently centres around the development of rigorous techniques, often based on formal logics, for the verification of systems. For this to be practical it is essential that the system specification is based on a complete model of the environment and the user's behaviour. Any verification of the system can only be valid subject to a correct model of these two important aspects of the total situation.

The modelling of complex environments has been an important research activity for many years, involving, perhaps, thermodynamics, hydrodynamics, electromagnetics, materials theory etc. and many sophisticated models have been produced. However even these models are far from being complete but they are all that we have available. In the design of a safety-critical control system for some industrial process, a chemical plant or a nuclear power station, it is only possible to validate the software subject to the model of the environment being realistic.

The Bide Report has examined some of the problems facing the Information Technology industry following the UK Government's Alvey Initiative and has stressed how important the user interface is in any computer system. The report makes the point that, no matter how reliable and well designed a system is, if the user interface is not well designed and sympathetic to the user's needs then the success of the system as a whole is in serious doubt. This is especially true in the case of interactive, safety-critical systems. When we turn to the problem of modelling the actions of a user, which could be fundamental to the safety of the system, we have a serious problem. Although much experimental evidence has been amassed about user behaviour much of it is contradictory and there is no body of formal theory which could act as a basis for reasoning about such important matters. Several attempts are being made to develop rigorous design methodologies to take account of these problems. These methodologies require the development, as does any method which is trying to design the user interface, of a sensible series of models of user behaviour and belief. The construction of formal user conceptual models is an area of importance and these models must be based on some sort of foundational logic that is rich enough for the expression of possibly irrational and ill-defined beliefs about the system. The recent work on 'belief' logics looks very promising in this respect [4].

One basic problem with current specification and analysis methodologies is that they tend to be rather specialised and cannot always deal with different aspects of a system and its environment. We have developed a method, based on the theory of X-machines (see [8]), which enables the formal description and analysis of most aspects of a system and its environment in a unified way. Systems may involve concurrent or real-time processing and yet the X-machine model is sufficiently robust that it can be used to specify such systems. Analogue aspects of a real-time control system can be described using X-machines with a topological basis. At the heart of such machines are suitable models of data types and operations which can be expressed either in model-theoretic form, such as is used in VDM or Z, or in functional or algebraic paradigms (some elementary ideas from these approaches are discussed in §2).

2 The specification of data types

One very promising approach to the design of more reliable software systems is the formal specification of data types and operations. There are several approaches; the two most popular are algebraic specification and model-based specification.

The algebraic approach to data type specification involves the definition of data types in terms of universal algebras. Let us suppose, as before, that our system involves a collection of sets and functions or operators. There may be some sets that are constructed of products of other sets and so on. We specify first a collection of basic sets and operators. In our examples we will consider the specification of data types needed in the design of a simple word processor, since that is a system that many people may now be familiar with. The most important set is the set of finite sequences of letters and numerals which we call $seq[Char]$. Although this is a basic set it does involve some interesting mathematical problems. We will construct a specification of this data type from the more primitive type $Char$.

Let $Char$ denote the set of all possible symbols to be used for the construction of documents, so $Char = \{A, a, B, b, \ldots, Z, z, 0, 1, \ldots, 9, \square\}$, where $\square$ represents a blank space.

The main operations that we wish to carry out with sequences are

1. construct sequences,

2. combine sequences,

3. test to see if a sequence is the null sequence,

4. extract the leftmost symbol of a string,

5. delete the leftmost symbol from a string.

These operations will be defined using functions. We first identify the sets *Char*, *seq*[*Char*] and *Bool*, the 2-valued truth set. There are then some function declarations:

$$\begin{aligned} null &: \rightarrow seq[Char] \\ | &: Char \times seq[Char] \rightarrow seq[Char] \\ * &: seq[Char] \times seq[Char] \rightarrow seq[Char] \\ isnull &: seq[Char] \rightarrow Bool \\ head &: seq[Char] \rightarrow Char \\ tail &: seq[Char] \rightarrow Char \end{aligned}$$

Here we are postulating that a null sequence, denoted by $\hat{}$, exists and this is defined by the first function declaration. The next thing we can do is to generate sequences of length 1 using the second function and perhaps write $a|\hat{}$ as $< a >$ etc. Then $b| < a >$ would represent $< ba >$ and so on. Further applications of the functions described above could be

$$head(c|(a|\hat{})) = c, \qquad tail(c|(a|(b|\hat{}))) = a|(b|\hat{})$$

and so on.

However we have not given a precise semantics for these functions and this is done using equations like the following :

$$\begin{aligned} head(x)|tail(x) &= x \\ x*(y*z) &= (x*y)*z \\ x*\hat{} &= \hat{}*x = x \quad \text{etc.} \\ isnull(\hat{}) &= T \end{aligned}$$

where $x, y, z \in seq[Char]$. The precise choice of the equations to describe the semantics of the data type is not uniqely determined as long as the algebraic model that these equations represent is consistent with the original system requirement.

To take a more abstract view we can postulate the existence of a set of 'sorts' that will, eventually, be replaced by explicit sets like *Char* and *seq*[*Char*]. Let us call these sorts $s_1, s_2, \ldots, s_n$. Then we define various operators $w_1 : \rightarrow s_2$, $w_2 : s_1 \times s_2 \rightarrow s_2$, $w_3 : s_2 \times s_2 \rightarrow s_2$, $w_4 : s_2 \rightarrow s_3$ etc. to represent null, $|$, $*$, *isnull* and so on.

We can now define the abstract concept of a *data algebra*. A *signature* is a pair $\Sigma = (S, \Omega)$ where S is a set of sorts, and $\Omega = \{\Omega_{x,s}\}$ is a set of operators indexed by pairs of the form (x, s) where $x \in S^*$, $s \in S$,(S^* is the free semigroup generated by S, x is called the 'arity' of the operators in $\Omega_{x,s}$). Thus $w_2 \in \Omega_{s_1 s_2, s_2}$ etc. A Σ-*algebra* is a pair $A = (S_A, \Omega_A)$ containing a family S_A of carrier sets s_A for each sort $s \in S$, and a family of operations $w_A : s_{1,A} \times \ldots \times s_{n,a} \rightarrow s_A$ for each operator $w \in \Omega_{s_1 \ldots s_n, s}$.

Now a *specification* consists of a pair $D = (\Sigma, E)$ where Σ is a signature and E is a set of Σ-equations. A D-algebra is any Σ-algebra which satisfies the set of equations E. A central result of the theory is that there exists an initial D-algebra (in the categorical sense). This initial D-algebra can then serve as a model for the specification of the system.

Such algebraic specifications can be 'implemented' using a language such as OBJ which is available for many mainframe computers. The standard reference for this work is now [9].

In model-based specifications the data types are defined in terms of sets and functions or operations defined on these sets with a semantics prescribed by a collection of predicate sentences or an explicit (possibly recursive) construction.

Example. Consider the possible fundamental data type associated with a simple word processor. We form the set, *seq*[*Char*], of all finite sequences or words from *Char*, including the empty word $\hat{}$, constructed above. Usually we will write a sequence in the form $< abcdefg >$.

The set *DOC* is defined to be the product

$$seq[Char] \times seq[Char]$$

and this represents the state of a simple document with a document of the form (α, β) representing the situation

$$\alpha \blacksquare \beta$$

that is, the string of symbols corresponding to α and the string corresponding to β with the cursor over the first symbol of β. It is possible to introduce a

more realistic representation of a document broken up into lines, paragraphs pages, windows etc. at a later stage.

We will use the notation Z, see [5], and declare each function with its semantics given below it.

$$
\begin{array}{rcl}
move & : & DOC \nrightarrow DOC \\
delete & : & DOC \nrightarrow DOC \\
insert & : & DOC \nrightarrow Char \nrightarrow DOC \\
print & : & DOC \rightarrow seq[Char]
\end{array}
$$

$$
\begin{array}{l}
\mathrm{dom}\, move = \mathrm{dom}\, delete = \{ l, r \mid l \neq \hat{} \} \\
(\forall (l, r) : DOC; a : Char) \\
\quad move(l * < a >, r) = (l, < a > * r)\,; \\
\quad delete(l * < a >, r) = (l, r)\,; \\
\quad insert(l, r)\, a = (l * < a >, r)\,; \\
\quad print(l, r) = l * r\ :
\end{array}
$$

Notes. (1) The notation $f : A \nrightarrow B$ means that the function is partial and not necessarily completely defined.
(2) The notation ":" is often used in place of $\in$ and $A \rightarrow B$ means the set of all functions from A to B.
(3) We use $f\ a$ to represent $f(a)$.
(4) dom means domain.

Using these definitions we can describe more complex data types and functions and consequently build up a more detailed and realistic specification of the data types and operations associated with the system. For example we need to be able to move in the opposite direction to the way the function *move* works. This can be done by constructing a simple function that 'reverses' a string of characters and then apply this in composition with the existing *move* function (before and after) suitably adapted for the type *Doc*. We can then define higher level functions which include direction parameters '*right*' and '*left*'.

The approach taken in [5] takes this view. A less constructive approach which just describes the properties that a function must satisfy without actually describing how this function can be constructed is also used in practice. The book [6] describes some simple examples of this approach. VDM is another, similar, approach with a more structured implementation environment which is discussed in [7].

3 Dynamic system specification

Although data type specification is of great importance there are several aspects of a system that are better specified by a more 'dynamic' model. The use of various types of machine is becoming more widely used for the formal specification of systems.

We discuss the concept of an X-machine, which is a general model of computation with the intention of using this model in the specification of computer systems.

The main mathematical model of computation is the Turing Machine. Athough this has received much study in a variety of theoretical areas it is not used by software engineers for the specification of systems, the principle reason being that the model is based at a very low level of abstraction and is not very amenable to analysis and system development. Less general models, such as finite state machines, machines with stacks and/or registers and Petri nets, however, are the basis of many system specification and development methodologies.

The use of graphical elements in a specification methodology is attractive from the point of view of user understanding, conveying dynamic information, and system refinement. Since the Turing Machine model is impractical and the finite state machine model is too restrictive, it would seem that the graphical advantages possessed by these models are not going to be available for general system specification. However, there is a much more appropriate model of computation that can, when combined with suitable data type methods, provide us with an appropriate environment for the description and analysis of arbitrary systems. Since this model also has very promising capabilities for use in discussing concurrent systems, it seems worthy of further investigation.

We start with the definition of the *X-machine* and show how this definition relates to previously studied concepts such as Turing Machines, push down machines and finite state machines. Then we examine some elementary aspects

of the theory of X-machines and conclude with a few examples. It should be remarked that although these machines were introduced in 1974 [9] they have not received much attention.

Let X be any non-empty set, henceforth referred to as the fundamental data type, and Φ a finite set of relations defined on X. Thus Φ consists of relations of the form $\phi: X \to X$. If one prefers we can regard each ϕ as a function, which is possibly incompletely specified, from the set X into the set $\mathcal{P}(X)$, the set of all subsets of X (also known as the power set of X).

Intuitively X represents the set of data to be processed and ϕ are the set of functions or relations that carry out the processing. In some cases the data type X can represent internal architectural details, such as contents of registers etc. and it is in this way that the model can assume its full generality.

Clearly we need to specify some relationship between the input and output information of the overall system and the data type 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. In many cases, as in much processing, these sets are free semigroups or subsets of free semigroups (i.e., languages over some finite alphabet).

Two coding relations, $\alpha: Y \to X$ 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 a suitable output format. Some examples will demonstrate how this works in a few basic cases.

Finally we need to describe some suitable control structure that will actually determine how the processing is performed. This structure is very similar to the state transition graph of a finite state machine and will appear familiar. However, this appearance masks a model of considerable computational power since much of the similarity with finite state machines is concerned with the *control* of the processing and not with the *type* of processing that the machine performs. Nevertheless, the similarities with finite state machines are extremely useful since they allow us, at times, to apply techniques for the analysis of machines that have proved to be tremendously successful.

The final ingredient is the *state space* of the machine, which consists of a finite set, Q, of states and a function

$$F: Q \times \Phi \to \mathcal{P}(Q)$$

called the *state transition function.*

For many purposes this state space can be described using a graph which has the elements of Q at the nodes (vertices) and for each $q, q_1 \in Q$, $\phi \in \Phi$ there is a labelled arc

$$q \xrightarrow{\phi} q_1$$

precisely if $q_1 \in F(q, \phi)$.

It is also necessary to identify a subset $I \subseteq Q$ of initial states and a subset $T \subseteq Q$ of terminal states. An initial state will be indicated in the state space by being the target of an unlabelled and sourceless arrow, e.g.,

$$\to q$$

whereas a final state will be described by being the source of an unlabelled and targetless arrow, thus:

$$q \to$$

An example of a state space is given in Fig 1.

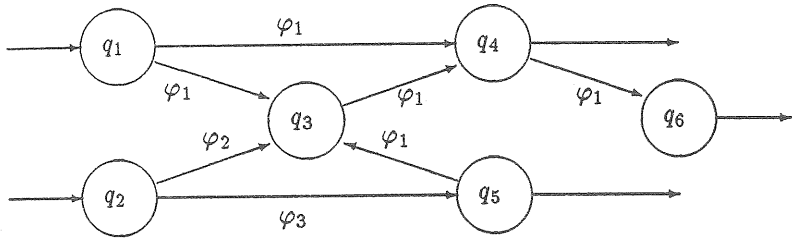

Fig.1 The state space of an X-machine.

In Fig. 1 states q_1 and q_2 are initial states and states q_4, q_5 and q_6 are terminal states. This example is of a *non-deterministic* machine; witness the two arrows leaving q_1 labelled with ϕ_1. It is also incomplete in the sense that no arrow labelled with ϕ_1 leaves state q_2.

The formal definition of an X-machine is presented in the following definition.

Definition. An X-machine is a 10-tuple:

$$M = (X, \Phi, Q, F, Y, Z, \alpha, \beta, I, T);$$

where

X, Y, Z are non-empty sets;
Φ is a set of relations on X;
Q is a *finite* non-empty set;
$F: Q \times \Phi \to \mathcal{P}(Q)$ is a, possibly partial, function;
$\alpha: Y \to X$ and $\beta: X \to Z$ are relations;
$I \subseteq Q$ and $T \subseteq Q$ are subsets.

Remark. The relations appearing in the definition are often functions or partial functions in many examples. The definition is presented here for the record in its most general setting. The set $\mathcal{P}(Q)$ denotes the power set (or set of subsets) of Q.

We call Y the *input type* and α the *input relation*. The set Z is the *output type* and β is the output relation.

The process of computation that this machine performs can be described by choosing an element $y \in Y$ from the input type and studying how this element is processed.

First the input relation is applied to the element y to produce an element or set of elements $\alpha(y)$ of X.

Next a path in the state space of the machine is selected that starts from a state in I and ends in a state from T. There may, in a non-deterministic or incomplete machine, be many or none. If a path is selected it will determine a sequence from Φ^* using the labels of the arcs of the path in order. If the labels of the arcs are $\phi_1, \phi_2, \ldots, \phi_n$ then the word

$$\phi_1 \circ \phi_2 \circ \ldots \phi_n$$

defines a composite relation (or function) on the set (or type) X. (In this notation we apply the relation ϕ_1 then ϕ_2 and so on, which is a common practice in algebra but may seem unusual elsewhere!)

When this composite relation is applied to $\alpha(y)$ we obtain an element or subset of X and this yields an element or subset of the output type Z on applying β.

The result of the computation is thus

$$\beta((\phi_1 \circ \phi_2 \circ \ldots \phi_n)(\alpha(y)))$$

If at any stage we find that the result of a partial computation

$$(\phi_1 \circ \phi_2 \circ \ldots \phi_k)(\alpha(y))$$

is the empty set for some $k \leq n$ then we will regard that computation as halting and the output, if any, is obtained by applying β as before. Fig. 2 gives a diagrammatical interpretation of the process.

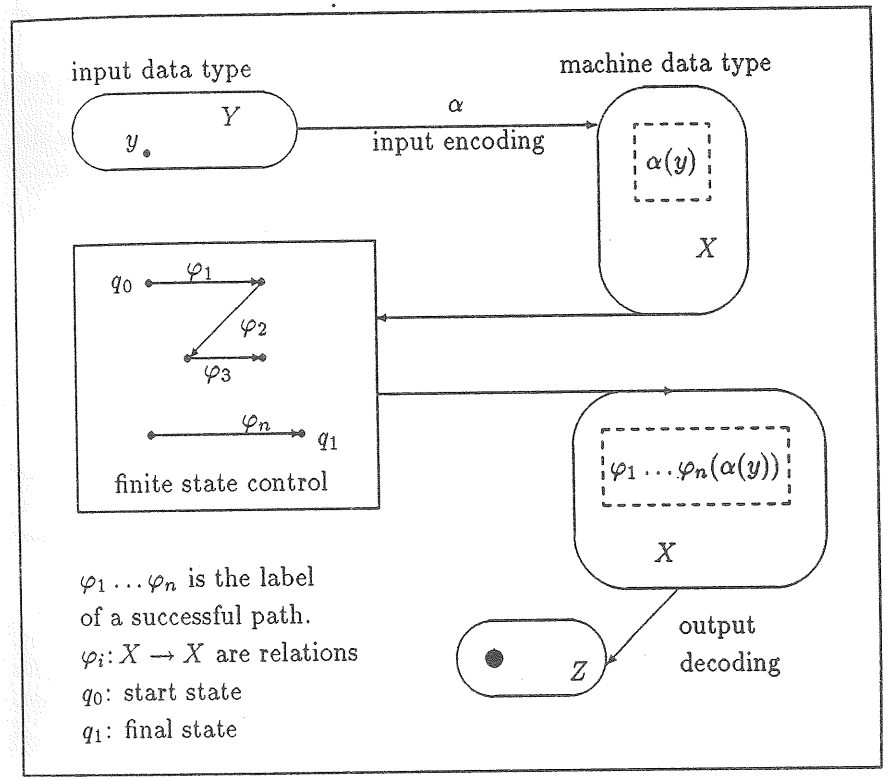

Fig. 2 An X-machine computation.

4 Some examples of X-machines

The most general model of computation so far investigated in any detail is the Turing machine model and its equivalent theories. There is, however, a newcomer to the scene that is claimed to be more general, namely the Quantum computer of Deutsch [10]. We do not intend to enter the controversy surrounding this new model and its relevance to computer science at this stage, merely note its existence. We will, however, demonstrate that the Turing machine is just a special case of the X-machine defined above.

Before we examine the connections between X-machines and other machines we need to introduce some terminology.

Let Σ be any non-empty set. Some relations will now be defined on the set Σ^* of all finite sequences or words in Σ. For any $\sigma \in \Sigma$ we define some fundamental relations:

$$
\begin{array}{ll}
L_\sigma : \Sigma^* \to \Sigma^* & (\forall x \in \Sigma^*)\ xL\sigma = \sigma x \\
L_\sigma^{-1} : \Sigma^* \to \Sigma^* & (\forall x \in \Sigma^*)\ xL_\sigma^{-1} = \{\, y \in \Sigma^* \mid \sigma y = x \,\} \\
R_\sigma : \Sigma^* \to \Sigma^* & (\forall x \in \Sigma^*)\ xR_\sigma = x\sigma \\
R_\sigma^{-1} : \Sigma^* \to \Sigma^* & (\forall x \in \Sigma^*)\ xR_\sigma^{-1} = \{\, y \in \Sigma^* \mid y\sigma = x \,\}
\end{array}
$$

$$
left : \Sigma^* \times \Sigma^* \to \Sigma^* \times \Sigma^*
$$

$$
(a,b)left = (reverse(tail(reverse(a))), head(reverse(a)) * b)
$$

(The purpose of the last string processing function will become clearer when we consider a later example, essentially it transfers the last symbol of the first word to the front of the second word. The standard functions reverse, head and tail are assumed to be defined already as is concatenation, $*$.)

The Turing machine model. The essential features of a Turing machine consist of an alphabet d, a finite set of states Q and a finite set of n-tuples ($n = 4$ or 5) which describe the behaviour of the machine under various circumstances. The set of 5-tuples that we will use here will be elements of the form

$$(q, q_1, \theta, \theta_1, d)$$

where $q, q_1 \in Q$; $\theta, \theta_1 \in \Sigma \cup \hat{}$ where $\hat{}$ denotes a blank; and either $d = L$ or $d = R$. The interpretation of such a tuple is that if the machine is in state q and the current symbol being scanned is θ then the next state is q_1, the symbol θ_1 is printed on the tape instead of θ and the read-write head is moved 'left' if $d = L$ and 'right' if $d = R$. Further details and examples of Turing machines will be found in many texts on the theory of computer science.

Added to this is a start state q_0 and a set $T \subseteq Q$ of terminal states. The initial tape contains a string of characters from the set Σ^* which is input to the machine in the state q_0. Processing consists of applying a sequence of appropriate tuples so that if at any stage the machine is in state q and is reading the tape symbol θ then any tuple of the form

$$(q, q', \theta, \theta', d)$$

where $q' \in Q$, $\theta' \in \Sigma \cup \{\hat{}\}$, $d \in \{L, R\}$ can be applied to yield the next state q', the symbol θ replaced by the symbol θ' and the tape head moved either left or right.

If the tape head moves left then the processing takes a tape of the form

$$[\sigma_1\sigma_2 \ldots \sigma_k, \sigma_{k+1} \ldots \sigma_n]$$

with the head reading the symbol σ_k and either produces a resultant tape of the form

$$[\sigma_1\sigma_2 \ldots \sigma_{k-1}, \sigma_k'\sigma_{k+1} \ldots \sigma_n]$$

where σ_k' is the new symbol printed on the tape after applying the tuple or

$$[\sigma_1\sigma_2 \ldots \sigma_{k-1}, \sigma_{k+1} \ldots \sigma_n]$$

For a right move the resultant tape is of the form

$$[\sigma_1\sigma_2 \ldots \sigma_k'\sigma_{k+1}, \sigma_{k+2} \ldots \sigma_n]$$

or

$$[\sigma_1\sigma_2 \ldots \sigma_{k+1}, \sigma_{k+2} \ldots \sigma_n]$$

In some cases the tuple may involve the replacing of a symbol on the tape by a blank.

In the context of an X-machine we first define the set X as

$$X = \Sigma^* \times \Sigma^*$$

The set of states is Q and the initial and terminal states as in the Turing machine case. For each tuple of the form

$$(q, \sigma, q', \sigma', L)$$

we insert an arrow from q to q' labelled by the relation

$$R_\sigma^{-1} \times L_{\sigma'}$$

on X. For each tuple of the form

$$(q, \sigma, q', \sigma', R)$$

we insert an arrow from q to q' labelled by the relation

$$\phi = (R_\sigma^{-1} \times 1) \circ (R_{\sigma'}^{-1} \times 1) \circ left$$

etc. The definition of the input and output relations for the X-machine are given next.

$$\alpha, \beta : \Sigma^* \to \Sigma^* \times \Sigma^*$$
$$(a)\alpha = (\hat{}, a)$$
$$(a, b)\beta = a$$

This interpretation is of a Turing machine that behaves as a function on Σ^*. If the machine halts during a computation this means that there is no arrow leaving the current state which has, as a label, an applicable relation. The result is then obtained by use of the decoding relation.

Finite state machines. The classical model of a finite state machine can be represented as an X-machine in the following way.

Let Q be a finite state set, Σ a finite input set and Ω a finite output set; then a finite state machine is a quintuple

$$A = (Q, \Sigma, \Omega, F, G)$$

where $F : Q \times \Sigma \nrightarrow Q$ and $G : Q \times \Sigma \nrightarrow \Omega$ are partial functions defining the next state and output functions.

The X-machine is defined as follows, The set $X = \Omega^* \times \Sigma^*$, the set of states is Q and the sets of final and initial states are also equal to Q. The set of relations Φ are defined as follows. If $q, q' \in Q$, $\sigma \in \Sigma$, $\theta \in \Omega$ are such that $F(q, \sigma) = q'$ and $G(q, \sigma) = \theta$ then we insert an arrow from state q to state q' labelled by the relation

$$\phi = R_\theta \times L_\sigma^{-1}$$

The input and output codes are given by

$$\alpha : \Sigma^* \to X \qquad \text{where } \alpha(a) = (\hat{}, a)$$

$\hat{}$ being the empty string, and

$$\beta : X \to \Omega^* \qquad \text{where } \beta(a, b) = a\,.$$

If it is necessary to only carry out computations starting from a given initial state we will define I to be the singleton set containing this state.

The X-machine computes exactly the same sequential function as does the original finite state machine.

In the previous section we gave the general definition of an X-machine and illustrated this with some examples to show that the concept is fully general. In this section we will briefly review some of the theory of X-machines, although at this time this theory is not as well developed as it might be. The definition of the *behaviour* of an X-machine can be made in terms of the function or relation that it computes or in terms of the language it recognizes.

Let $M = (X, \Phi, Q, F, Y, Z, \alpha, \beta, I, T)$ be any X-machine. If

$$c : q_0 \xrightarrow{\phi_1} q_2 \xrightarrow{\phi_2} q_2 \to \ldots \xrightarrow{\phi_n} q_n$$

represents a sucessful path in the state space of M, so that $q_0 \in I$ and $q_n \in T$, then the relation

$$|c| = \phi_1 \circ \phi_2 \circ \ldots \circ \phi_n : X \to X$$

will be called the *relation defined* by that *labelled path*. The *behaviour* of M is then

$$|M| = \bigcup |c| : X \to X$$

where the union is taken over all the successful paths in the state space.

The *relation computed* by the machine is then defined as

$$f_M = \alpha \circ |M| \circ \beta : Y \to Z$$

For the recognition of languages we define the output set to be $\hat{}$ and the output function $\beta : X \to Z$ yields a subset

$$A = \hat{} f_M$$

of Y.

The article [1] discusses some of the applications of this material. We can develop a methodology for the description of systems by a combination of the

data type methods of the first sections with the machine based methods of the latter ones. In situations when architectural features of the system are important, these can be incorporated into the X-machine by defining the set X suitably, perhaps including models of registers etc.

References

[1] M. Holcombe, *X-machines as a basis for dynamic system specification*, Software Engineering Journal (in press).

[2] M. Holcombe, *Formal methods in the specification of the human-machine interface*, Int. CIS Journal 1(1) (1987), 24–34.

[3] M. Holcombe, *Goal-directed task analysis and formal interface specifications*, Int. CIS Journal 1(4) (1987), 14–22.

[4] R. Fagin and J.Y. Halpern, *Belief, awareness and limited reasoning*, Art. Intel. 34 (1988), 39–76.

[5] B. Suffrin, *Formal specification of a display oriented text editor*, Science of Comp. Prog. 1 (1982), 157–202.

[6] I. Hayes, *Specification case studies*, Prentice-Hall, 1986.

[7] C.B. Jones, *Systematic software development using VDM*, Prentice-Hall, 1986.

[8] S. Eilenberg, *Automata, languages and machines*, Academic Press, 1974.

[9] H. Ehrig and B. Mahr, *Fundamentals of Algebraic specification*, vol.1 EATCS Monographs, Springer 1985.

[10] D.A. Deutsch, *The quantum computer and the Church-Turing thesis* Proc. Royal. Soc. A 400 (1985), 97–117.

Department of Computer Science
University of Sheffield

Crossed Modules

Graham Ellis

Crossed modules were invented almost 40 years ago by J.H.C.Whitehead in his work on combinatorial homotopy theory [W]. They have since found important roles in many areas of mathematics (including homotopy theory, homology and cohomology of groups, algebraic K-theory, cyclic homology, combinatorial group theory, and differential geometry). Possibly crossed modules should now be considered one of the fundamental algebraic structures. In this article we give an account of some of the main occurrences and uses of crossed modules and we describe some recent developments in their theory.

Before presenting the definition of a crossed module, we shall consider several motivating examples. Throughout G denotes an arbitrary group.

Example 1 Let N be a normal subgroup of G. The inclusion homomorphism $N \to G$ together with the action ${}^g n = gng^{-1}$ of G on N is a crossed module.

Example 2 If M is a $\mathbb{Z}G$-module then the trivial homomorphism $M \to G$ which maps everything to the identity is a crossed module.

Example 3 Let $\partial\colon H \to G$ be a surjective group homomorphism whose kernel lies in the centre of H. There is an action ${}^g h = \tilde{g}h\tilde{g}^{-1}$ of G on H where $\tilde{g}$ denotes any element in $\partial^{-1}(g)$. The homomorphism ∂ together with this action is a crossed module.

Example 4 Suppose that G is the group $Aut(K)$ of automorphisms of some group K. Then the homomorphism $K \to G$ which sends an element $x \in K$ to the inner automorphism $K \to K$, $k \mapsto xkx^{-1}$ is a crossed module.

Each of these examples consists of a group homomorphism with an action of the target group on the source group. Before stating the precise algebraic properties needed by such a homomorphism for it to be a crossed module, let us consider some more substantial examples.

Example 5 Let X be a topological space in which a point x_0 has been chosen. Recall that the fundamental group $\pi_1(X, x_0)$ consists of homotopy classes of