Chapter 6.

Conclusions.

6.1. X-machines - a computational model framework.

The Chomsky hierarchy of languages and associated computational models have been a cornerstone in the theory of computation for many years. The X-machine generalises these computational models and provides us with an unified framework for discussing the whole issue of classifying computable functions or languages. We can look to X-machines to provide us with a richer hierarchy of languages and functions then the traditional one. In particular, the stream X-machine model could be used for classifying context-sensitive (or possibly recursive) languages as suggested in Chapter 2. Obviously, this issue requires much further investigation, as does the use of the X-machine model for classifying non-Turing computable functions.

6.2. X-machines - a basis for a specification language.

A number of formal specification languages and methods have been proposed in recent years. They range from model based languages such as Z or VDM to executable algebraic approaches like OBJ and process algebra approaches, principally CCS and CSP.

X-machines combine the ability to model data structures, functions and relations of languages such as Z or VDM with the graphical advantages of finite state machines. The state diagram of the machine represents the control structure of the machine and this is separated from the data set and the definitions of processing functions. This control structure gives an overall view of the system and communicates its main features.

This view will be enriched by specifying the processing functions. These can be expressed using Z or a functional language such as ML or using traditional mathematical notation. Alternatively, these could be X-machines themselves, so the X-machine model can be used at more then one level in the specification.

This combination of state diagrams (the *control structure*), data structures and processing functions makes the X-machine a convenient and intuitive

specification method in which different features of the system will be communicated at the appropriate level in an intuitive way.

The notion of state exists in CCS. CCS (see Milner [44]) was devised to give a general theoretical account of concurrent, asynchronous, non-deterministic computation and is most useful for analysing the communication structure of a system. There is a highly automated tool (the Concurrency Workbench) for doing this for finite-state models. However, the emphasis is on modelling systems with many parallel components and data flow is difficult to represent in pure CCS. More complex data structures can be represented in value-passing CCS. However, tools which support the full value-passing version are only now being developed and still face theoretical problems. On the other hand, the potential of X-machines in areas such as concurrency and distributed systems has yet to be considered.

A comparison between the use of X-machines and OBJ as specification methods has been carried out on a case study (i.e. Gordon's Computer, see Howe [35]). The conclusions were that X-machines were easier to understand and use, require less prior knowledge and produce specifications closer to what our intuition is of what the system is supposed to do.

Recently, The X-machine model has been used in several case studies, ranging from modelling interactive systems (see Laycock [41]) and user interfaces (Holcombe & Duan [30]) to fairly complicated hardware devices (see Chiu [5] and Howe [35]). We believe that the simplicity and the power of the method can make it popular with industry.

6.3. Stream X-machines.

Stream X-machines are a natural class of X-machines in which the inputs and outputs are sequences of characters and an input/output pair is read/produced whenever a transition is performed. Clearly, they are very well suited for modelling interactive systems. However, several case studies have shown that the model can be used to model a much wider range of systems (in fact, all the case studies mentioned in the previous section use the stream X-machine model). Also, the stream X-machine appears to handle time-dependent systems quite easily, as illustrated by Fairtlough et al., [15], by a case study.

In Chapter 2 we showed that, by using an hierarchical approach (i.e. the processing functions are themselves modelled using stream X-machines), the model can cope with increasing computational problems.

6.4. Minimality and equivalence.

Minimality is not straightforward with (stream) X-machines. For an arbitrary X-machine $\mathcal{M}$, there is an X-machine $\mathcal{M}$' with only one state and one processing function that exhibits the same behaviour in terms of the overall input/output

function it performs. However, the single processing function is not (necessarily) of the same type (i.e. from the same set Φ) as the processing functions of $\mathcal{M}$ and it would almost certainly be a far more complicated function. Minimisation of this extreme kind is not of much practical use. The resulting model would be difficult to understand and it would be very unlikely to correspond to any intuitive model of the system it was intended to describe.

Instead, we introduce the idea of minimality within a particular type, i.e. finding a stream X-machine $\mathcal{M}'$ with the same type Φ as a certain machine $\mathcal{M}$ that behaves 'similarly' in terms of the input/output function it performs. The concepts of Φ-minimality and minimal covering w.r.t. Φ were introduced. The first indicates that $\mathcal{M}'$ behaves identically to $\mathcal{M}$. The second is used when $\mathcal{M}'$ does everything that $\mathcal{M}$ does and possesses, in addition, some extra functionality. These are concepts that could be useful in practice, since they correspond to the 'smallest' (i.e. as far as the state set is concerned) specifications that behave identically or cover a given specification.

However, even when the minimality problem is restricted to machines having the same type Φ, the problem of finding these minimal machines is difficult to address unless Φ satisfies some special properties. We solved this problem for machines whose Φ's are complete and output-distinguishable (these are exactly the 'design for testing conditions' required by our testing method). We have proved that in this case the Φ-minimal machine is unique up to an isomorphism of the associated automata and that the problem of finding this machine can be reduced to finite state machine minimisation. We have also given a procedure for constructing all the minimal coverings of a particular stream X-machine.

6.5. Animation of executable (stream) X-machines.

It is straightforward to represent a finite state machine in a functional language such as ML. Also, if we make the reasonable assumption that the processing functions are computable, then the processing functions can be represented as ML functions. So, ML would be a suitable choice of programming language for writing and animating (stream) X-machines. Such a tool would be able to check easily for properties involving the associated automaton of the machine (e.g. minimality) or to convert this into a minimal form.

A prototype generic stream X-machine animator has been constructed and tested on several cases and has worked satisfactorily. Work is in progress to construct a graphical design tool for the design and simulation of stream X-machines. However, a more general tool that allows the processing functions to be specified in terms of stream X-machines and also carries out refinements would be needed.

6.6. Testing and stream X-machines.

Stream X-machines are the basis for our theoretical testing method. Clearly, if the implementation is not restricted in some way (i.e. it can be *any* stream X-machine), then the problem of constructing a *finite* test set that finds *all* faults is unsolvable (this is because an *infinite* memory cannot be tested using a finite set of inputs). The way to get around this problem is to use a *reductionist* approach. The stream X-machine method is neither 'black box' nor 'white box'. The implementation is considered to be a 'black box' containing known elements. Therefore, we assume that the basic processing functions are implemented correctly and we test whether the *control structure* of the system (i.e. the state transition diagram) is correct or not. We also assume that some 'design for testing' conditions are met.

So, how do we address the problem of testing an infinite memory with a finite test set ?

· Firstly, the completeness of the type Φ ensures that each state that is accessible via a path in the associated automaton of the machine will be reached using an input sequence that follows that path. Therefore, each state in the implementation can be checked against the corresponding state in the specification.

· Secondly, the set $M \times \Sigma$ is divided into the domains of the ϕ's and at least one pair (m, σ) for each one of these domains is checked. This is in principle similar to the underlying idea of the category-partition method (although the domains of the ϕ's are not necessarily a partition of $M \times \Sigma$). Unlike the category-partition though, the criterion by which the values to be tested are chosen is clearly defined by the method. It is based on the system specification and does not depend on the ability or experience of the person that carries out the testing.

· Finally, since Φ is output-distinguishable and correctly implemented, it follows that, if the machines (one representing the specification, the other the implementation) give the same output on the same input sequence s, then their computations will follow the same sequence of ϕ's (i.e. s will cause the two machines to follow two paths identically labelled). Hence, the problem of testing a stream X-machine will be reduced to one of testing a finite state machine (i.e. its associated automaton).

The advantage of the method is that it guarantees that the system is *fault-free provided* that the basic components are *fault-free*.

The reductionist approach can be continued further and the ϕ's can be tested using our approach if they are represented as stream X-machines. At the bottom level of the reduction, we shall have simpler functions that the developer is confident are fault-free or can be tested using alternative methods (e.g. category-partition). This hierarchical approach suits the increasing modularisation we see in the development of software and provides a potential way of dealing with large scale systems. On the other hand, a hierarchical approach increases the computational capability of the stream X-machine model.

The test sets generated by the method are of manageable size as is the application process. If the processing functions are computable by some algorithms, then the process of generating the test set can be automated. Clearly, the method has to be supported by automated systems and suitable tools that need to be provided.

6.7. Refinement.

If (stream) X-machines are to be useful as a tool for specification, there needs to be some way of developing existing machines into more complex and more detailed versions without starting anew with each modification.

The refinement we have introduced allows (generalised) stream X-machine specifications to be developed gradually. An unrefined machine specification $\mathcal{M}$ is produced first. This will be the 'control' machine of the refinement. The refined machine $\mathcal{M}'$ will be obtained from $\mathcal{M}$ by mapping sequences of characters from the refined input alphabet Σ into inputs to the control machine $\mathcal{M}$. This mapping will depend on the current state and memory value of the control machine $\mathcal{M}$. This will also result in an expansion of the output alphabet. This mapping (in fact a set of mappings) will be specified using refinement modules and the link between these and the control machine is achieved via some transfer functions. Each state in $\mathcal{M}$ will be refined by a module and a refinement function will determine which module refines which state. This entire transformation will result in an expansion in the state set and the memory of the initial machine.

The method can be used for separating the user interface from the core functionality of the system. It can also be used when some components of the system are to some extent self-contained and it will be natural to attempt to specify them separately. One advantage of the method is that it fits the construction of a program in terms of a control program and several sub-programs. Therefore, the refinement modules can be implemented separately and then used in the main implementation.

Clearly, other refinements could be defined. In Fairtlough et al., [15], a case study illustrates the process of expanding the functionality to a stream X-machine via some transformations such as:

· Enlarging the state set and the input/output alphabets.

· Enlarging the state set, the memory and the input/output alphabets. This can be seen as a process of linking two machines via some extra inputs.

· Enlarging the memory set.

The whole refinement/expansion process appears to be intuitive and easy to manage.

Clearly, other types of refinement/expansion have to be considered and testing issues have to be addressed.

6.7.1. Refinement testing.

There are two ways of testing a machine specification using a refinement of the type presented in Chapter 5. The first is to construct the refined machine explicitly and use the stream X-machine testing method. This is feasible when the number of states of the refined machine is not too large (e.g. the word processor in section 5.4).

An alternative approach is to implement the refinement modules and the basic functions separately and to test the integrated system. This approach is suitable when the number of states of the refined machine is quite large or when the basic refinement modules can be implemented quite easily (i.e. they are standard procedures or objects from a library or can be obtained from these very easily, e.g. example 5.1.6). A prerequisite of this approach is that the implementations of refinement modules and the basic functions of the control machines must be fault-free. Therefore, these have to be tested separately before the integration testing is carried out. The stream X-machine testing method can be used to test the refinement modules and also the processing functions, if these are expressed in terms of some (lower level) machines. Again, some 'design for test' conditions have to be met.

The method requires careful testing management to deal with the augmentations required by the 'design for testing' conditions. However, the clear benefit of the method is that it guarantees that the system is fault-free provided that its basic components have been correctly implemented.

Very few existing functional testing methods attempt to deal with the integration issue. Many work on the assumption that if a system contains several components (modules, procedures, etc.) and each of these are correct, then somehow the whole system will also be correct. For example, let us see how a category-partition method can be used to test an implementation of a stream X-machine refinement. The method will require the following steps:

- Testing the basic functions of the control machine and the refinement modules.
- Testing the refinement modules.
- Testing the control machine.

However, there is no guarantee that, if all these are correct, the whole system will also be correct. Furthermore, the test set of the control machine will be a set of sequences of inputs for this machine (i.e. a set in I^*), without any indication being given of how these can be obtained from the real inputs (i.e. sequences from Σ^*) by the refinement modules. This is another important drawback of the method.

6.8. Formal verification.

The X-machine model can be used formally to verify properties of algorithms. This may be done by examining paths through the X-machine and comparing the corresponding input/output function with a higher level description of the algorithm.

However, proofs by hand are time consuming and error prone, so some methods and tools that permit an automatic verification of the system requirements would be desirable. From this point of view we consider that two directions are worth pursuing.

Firstly, if X-machines could be translated into models for which the theory required for such automatic verification exist (e.g. CCS, the value passing version), then these could be used to develop automatic verification tools based on the X-machine model.

Secondly, we could consider a combined verification and testing strategy based on X-machines in which the ϕ's are formally verified using some sort of automatic tool and the whole system is then tested using the stream X-machine testing method. In this case, the test set generated by this method could be used to test the machine specification against its implementation and also to verify that the specification satisfies the system requirements.

One could argue that our stream X-machine testing method has certain similarities to a verification method in the sense that, unlike traditional testing methods, we are able to *prove* that if the implementation passes all the tests in the test set, then it matches the specification. Also, some restrictions are imposed on the way in which the implementation is constructed. However, the difference is that we do not attempt to prove the equivalence of the two models (one representing the implementation and the other the specification) starting from the lowest level of the system. Instead, we assume that the ϕ's are implemented correctly and we prove the equivalence of these models at a higher level. So, it appears that a combination of formal verification and testing in which the lower level ϕ's are verified and then the integration of these ϕ's is tested will provide us with an effective way of building fault-free systems.

There is clearly a case for hoping that formal verification and testing could be used *together* to provide a scientifically based engineering approach to software development.

Obviously, this area requires much further investigation. The aim is to develop an integrated specification, verification and testing methodology based on the X-machine model. We hope to have made a start in this direction.