Chapter 1.

Introduction.

1.1. Software quality via formal methods and testing.

The subject of software quality, of delivering a *correct* implementation of the *correct* system, has occupied the attention of many software engineers and generated a substantial literature. Although much progress has been made there is still a long way before we can be satisfied that the methods for design, construction and management of software systems is at a level of quality that is required in the modern world.

1.1.1. Correct specifications via formal methods.

Central to the problem of software quality is that it is extremely difficult to clarify exactly what is required of a very complex system. A variety of techniques have been introduced to help deal with the difficulties of developing computer systems. Amongst these principles is the ideas that the appropriate mathematical techniques could be used to *specify* the requirements of a computer system.

The main advantages of using *formal methods* for specifying software systems is the clarity, conciseness and precision of mathematics. A *formal specification* will enable us to reason in a very rigorous manner about the system's behaviour, thus making sure that the system does exactly what it is supposed to do.

1.1.2. Correct implementation via testing.

However, a correct (formal) specification is not in itself sufficient for delivering software of high quality. The *implementation* of this specification is the product we are ultimately interested in. So, it is crucial that the implementation *is correct* with respect to the specification.

1.1.2.1. Correct implementation via proofs.

One way to ensure this is to use proofs. Attempts to prove that the implementation is correct w.r.t. the specification after the implementation is complete are rarely successful on large scale systems. Instead, a process of refinement can be used.

The specification represented in some suitable formal notation is converted into an implementation using a series of simple refinements, each of which is easy to prove. In this way, there should be no faults present in the implementation. However, if the proof is constructed "by hand", there is no guarantee that there will be no errors made in constructing the proof, and so guarantee that no faults are introduced in the implementation. This problem can be overcome by using automatic proof systems to aid a human in the construction of a proof, or ultimately to perform the entire proof construction. However, there is still a major problem, in that the automatic proof system and the systems of axioms used in it must be known to be correct: the tool must have been proven at some point. In addition there must be a formal description of the environment and the actual physical environment must be proven consistent with the formal model. All these problems reduce the practical use of proofs in the context of the complexity of today's applications.

1.1.2.2. Testing.

An alternative method of achieving correctness is *testing*. Indeed, in practice, even the systems that are proven to be correct are very seldom released without being tested. Testing attempts to detect *all* the faults that are present in the implementation so they can be removed.

A number of testing techniques exist. Many sophisticated (and expensive) tools are available on the market and many designers look to these to provide the answer to the problem of building fault-free systems.

However, most testing methods are practice-driven approaches rather than carefully constructed engineering methods based on a defensible well-founded strategy. In particular, very few of the existing methods allow us to make any statements about the type and the number of faults that remain in the implementation after testing is completed (this idea will be expanded in Chapter 4). Thus we cannot measure the effectiveness of our testing activities in any rigorous way. Hence, the goal of testing (i.e. to detect *all* the faults in the implementation) is not achieved.

1.1.2.3. Formal methods and testing.

On the other hand, although a great deal of research has been done in the area of formal methods and their practical use for the specification of software systems, testing issues are very seldom mentioned by those within the formal methods community. All too often formal methods and testing have been regarded as mutually exclusive approaches in the development of software systems.

However, since testing is concerned with ensuring the correctness of an implementation against a (possibly formal) specification, we believe that examining the common ground between formal methods and testing will bring benefits to both these fields and ultimately to the problem of software quality.

Indeed, by considering testing from the specification phase, we shall produce formal specifications that are more easily testable. Conversely, by developing a testing theory based on a particular formal model, we can place testing on a theoretical basis, thereby providing a more convincing approach to the problem of detecting *all* faults and allowing us to make sensible and theoretically defensible claims about the level and type of faults that remain in the implementation after testing is completed.

We shall be seeking to bridge the gap between these two fields and develop a testing theory based on a formal specification model, namely the *X-machines.*

1.2. X-machines.

In this thesis we shall introduce Eilenberg's *X-machines*, [12], as a formal model for the description of computer system specifications. We shall analyse from a theoretical point of view the use of X-machines both as a basis for a formal specification language and a theory of testing. We shall seek to address some theoretical issues regarding the use of X-machines as a specification method (i.e. minimality, refinement) and to develop a theory of testing based on this model.

1.2.1. Why X-machines ?

Recently, a major thrust of software system specification has centred around the use of models of data types, methods such that Z or VDM (Wulf et al. [60]). 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. However, one basic problem with these abstract specification methodologies, whether Z, VDM, or whatever, is that they lack the ability to express the dynamics of the system in a convenient and intuitive form. If the system has many functions, interacting in a complex way, they may offer little guidance as to how to integrate the individual functions into the system as a whole. Algebraic methods such as OBJ suffer from similar problems although, since they are executable, some scope for symbolic simulation exists and can provide some insight into how the system will operate.

On the other hand, the dynamics of a system are often better represented in graphical ways and the use of state machines diagrams is important in software (Wasserman et al. [56]) and hardware design (Clements [7]). But the finite state machine model is rather simple and has several drawbacks. Firstly, its computational power is inadequate, and therefore even some very common situations cannot be handled using this approach (see Chapter 2). Secondly, there is no way to model data structures independently of the control structure. In fact, it is difficult to model any non-trivial data structure using finite state machines.

More powerful computational models exist (i.e. Turing machines, pushdown automata, etc.), but they are not used by software engineers for the specification of real systems, the reason being that these models are based at a very low level of abstraction and are not amenable to analysis and system development.

However, there is a much more appropriate model (i.e. X-machines) that combines the graphical advantages of finite state machines with suitable data structures, thus providing an appropriate environment for a rigorous description and analysis of arbitrary systems.

1.2.2. X-machines - a blend of finite state machines and data structures.

Introduced by Eilenberg in 1974, [12], X-machines have received little further study. Holcombe, [27], 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 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. Each arrow in the finite state machine diagram is then labelled by a function from Φ, 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:

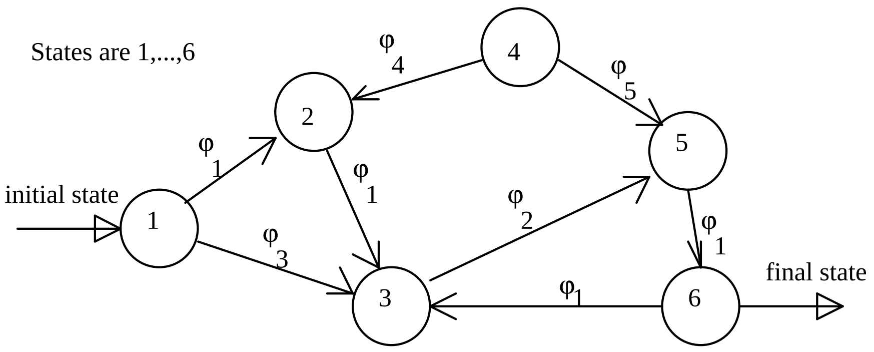

Figure 1.1.

We haven't specified X or the functions φ_i 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 underlying data type 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 $\varphi_1(\varphi_2(\varphi_1(\varphi_1(x)))) \in X$. So we are assuming that $x \in$ domain φ_1; $\varphi_1(x) \in$ domain φ_1; $\varphi_1(\varphi_1(x)) \in$ domain φ_2; and $\varphi_2(\varphi_1(\varphi_1(x))) \in$ 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.

The advantages of this model include:

It is intuitive and 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 diagram 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 Chapter 2).

Also, we have a further reason for proposing X-machines both as a basis of a specification method and a theory of testing. Some theoretically based testing methods aimed at finite state machines exist, due to Chow, [6], and Fujiwara et al., [16] (see also Chapter 4). Obviously, these are restricted to the finite state machine model. But they provide us with a basis for developing a testing theory for the more general X-machine.

1.2.3. Stream X-machines.

However, as we shall see later on in the following chapter, the generality of the X-machines can be an obstacle in the way of developing a testing method based on this model. The way in which we shall overcome this problem will be by restricting the basic processing functions that the model can use to those that satisfy certain conditions or have certain forms. In this way, we can obtain different subclasses of X-machines, depending on the particular conditions that their Φ satisfies.

However, we still have a problem, in the sense that by restricting the processing functions that an X-machine can use, we could also restrict the nature and the computational capability of the systems that the machine can specify. Therefore, what we shall be looking for will be a subclass of X-machines such that:

it is general enough to cope with a wide range of computational problems

it is restrictive enough to provide a reasonable basis for a testing theory.

The machines belonging to this subclass will be called *stream X-machines* and our testing theory will be aimed at those.

1.3. Thesis summary.

Chapter 2 will survey the main existing computational models and introduce the X-machines as a unified computational model framework of which the above mentioned models are particular cases. Two natural subclasses of X-machines (namely straight-move stream X-machines and stream X-machines) will be defined and discussed from two points of view:

how general they are as specification tools; this involves assessing the above subclasses according to the computational problems that they can cope with;

to what extent they can provide a reasonable basis for a theoretical testing method.

Chapter 3 will investigate further the stream X-machine. The class of (partial) functions that this model computes will be identified and two simple operations (i.e. sequential and parallel composition) that can be performed on stream X-machines will be defined. Also, some minimality problems will be defined and discussed.

Chapter 4 is concerned with testing and is divided into two main parts. The first one will survey the testing methods currently in use and discuss to what extent they can achieve the goal of testing (i.e. to find *all* faults). The second part will present our stream X-machine testing and will illustrate it with a case study.

Chapter 5 will define a stream X-machine refinement as a way of developing stream X-machine specifications gradually. This will be illustrated with a case study. A testing for refinement method will also be given.