CORRECT SYSTEMS -

BUILDING BUSINESS PROCESS SOLUTIONS.

Mike Holcombe
(University of Sheffield)

Florentin Ipate
(Romanian-American University, Bucharest)

PREFACE.

"A *correct system* is one that works, one that I understand how to use, one that does not keep going wrong and one that solves the problems that *I* face."

A user.

"A *correct system* is one that has been formally proven to satisfy the mathematical formula that defines it."

An academic computer scientist.

Are these two statements talking about the same thing? The first statement is a perfectly reasonable desire. Does the second statement offer anything relevant to this desire? In order to try to bridge what seems to be an enormous gap, a chasm in fact, that exists between the two points of view we need to consider further the fundamental issues involved.

We will ask the following questions:

- What do we mean by a CORRECT system?
- What do we mean by a SYSTEM?
- How can we build correct systems?
- What has software engineering to offer in our endeavours to do this?

These and other questions will form the focus of this book. We will also try to look at some of the wider issues involved in building systems that work, examine a number of myths about the subject and look a little into the future. Some of the emphasis on the development of correct systems is oriented around the use of mathematical techniques, notations and proofs. Although mathematics is an important facet of the process of creating a correct systems we will not let ourselves become too obsessed with it! It will be necessary from time to time but we hope to emphasise the use of simple *user-friendly* mathematical methods rather than the usual battery of high powered techniques.

We begin by considering a number of fundamental questions.

1. What is a system?

Computers and software are affecting our lives in more and more ways. Society is becoming very dependent upon them and we expect this trend to continue. One might ask whether this is a desirable trend - the answer is *probably - yes and no* but the fact remains it is happening and so we must try to address the consequences for systems and software engineers. Without computers there are many things that improve the quality of life - even save and preserve life - which would not be available, for example, medical treatment systems, smart environmental protection systems, transport systems, and many, many more. There are also computer systems which might threaten our quality of life - either directly - such as systems for controlling information about people and observing their activities in totalitarian states or indirectly through the computer system being badly designed - such as faults in software that controls: nuclear power stations, airliners or radiation treatment.

In many situations the computer is used as part of a solution to some problem. This problem can manifest itself in many different contexts, in business, in engineering, in education, in the home. We need to examine the problem carefully, once this has been done we may be able to devise a solution involving the use of computers in some way.

“The first step in solving a problem is understanding it. However, we often propose solutions to problems that we do not understand and then are surprised when the solutions fail to have the anticipated effect.” [1]

This problem of understanding the problem is crucial. If we want to build computer systems to solve problems we need to be able to understand what the system has to do in order for it to be a solution.

This raises the task of being able to reason about a problem in a coherent, consistent, complete and meaningful way. This subject could be called Systems Modelling. We need to be able to model systems, so what is a system?

- It is not just the computer, the hardware, the software -
- it includes these but also a number of other aspects including:
- the operators, if any,
- the communication links between different components,
- the interfaces between the system and its environment,
- the documentation,
- the data upon which it depends,
- as well as its economic, legal, human and social context.

Without taking into consideration all of these things we may not understand the system and the way it operates sufficiently well to be able to build and operate it properly - and thus solve the problem.

In principle, we need to be able to model all these aspects of a system, its environment and its behaviour in some detail - before we can understand the problem and its potential solutions. How is this modelling achieved? A number of approaches to modelling are available.

2. Modelling systems.

Informal descriptions, usually in natural language, are used to describe some of the aspects of the model and informal "common sense" reasoning is then applied to explore the consequences of the model. This approach may be adequate if the situation is fairly simple and the consequences of getting the solution wrong are unimportant. Despite these qualifications this is the main method used over the years and we haven't done too badly, so far, have we? However, increasing demands for quality, safety, economy and the increasing complexity of both the problems and the solutions are creating requirements that this approach may not be able to satisfy.

Problems include:

- ambiguity inherent in the method and its languages;
- inadequate methods of analysis;
- difficulty in using computers to help with the analysis;
- the problems caused by the management of the analysis of these large scale problems.

Another approach is to use, what are often referred to as *structured analysis* techniques [2], or what we would call *semi-formal* approaches. These are a blend of diagrams and text, usually satisfying a standard format.

They can be very effective and the majority of systems modelling and development is carried out using them, but they still have problems with:

- ambiguity;
- incompleteness;
- lack of rigorous analysis - it is difficult to reason in a precise way with inherently imprecise concepts;

but this approach is commonly used in software engineering-oriented industries and is fairly well supported by tools.

A further approach is through the use of *formal notations* - these are essentially mathematical and often involving unfamiliar and abstract symbolism. They are:

• not usually very easy to use without extensive training;

• not well supported by tools;

• easily affected by problems of scale;

but -they have benefits:

• they can be used for rigorous analysis,

• they are unambiguous;

•they have a strong theoretical foundation (sometimes).

There has been much emphasis, particularly amongst the academic community, in the use of these abstract methods, formal notations and mathematical processes as the means for achieving the goals of creating robust, correct and usable systems in an efficient and cost-effective way. There has been little real evidence, however, either that the industrial take up of these ideas has occurred to any significant extent, even in highly specialist niche application areas or that they deliver what is claimed of them.

There are a number of possible reasons for this, these include:

- the unfriendly nature of many of the notations used - particularly those based on symbolic mathematical notations (VDM, CCS, temporal and other formal logics, etc.). Away from the rarefied atmosphere of the mathematically-oriented research laboratories these notations appear to be obscure, artificial and unusable. The use of the phrase "formal methods" is also misleading since few of the notations actually involve any sort of mature and explicit *methodological* framework - formal specifications can be understood with suitable training in the language but the construction of correct specifications is a problem which is several orders of magnitude greater.
- the basis of many of the notations is not adequate for the sorts of applications that need this approach most. For example, traditional formal methods are capable of describing simple information systems applications where time, dynamic event handling and concurrent processes are not major issues, process algebra approaches can handle concurrency but the modelling of time or of complex data structures as well is very difficult and rarely practical.
- the essence of constructing a "correct" system is to identify the client/user's requirements accurately and completely. Currently this is an area where formal notations have proved inadequate, numerous prototypes are built and evaluated in an incremental way and this is costly and error prone.
- the building of a working implementation is generally regarded as a separate process which should be isolated from the specification process. This philosophy seems to result in the erection of barriers between the specification and the implementation which reification and refinement have great difficulty in overcoming in a practical context.
- formal methods rarely consider the fact that the system will be tested and the testing of an implementation is usually relegated to the last phase of a project. This results in ineffective test management and poor quality products. Testing must be considered at all stages of the product life-cycle and in particular a "design for test" philosophy must pertain throughout - it is possible to take design decisions that make future testing of components and the complete system difficult or, indeed, impossible.

Another approach is the use of *simulation*. Essentially, this involves the construction of a software program which will behave like a particular part of the system being modelled. This is then investigated to study its properties. Simulation requires the repeated use of representative input values to animate the program and demonstrate what part of the system might do - or how several simulated parts might interact together. *Prototypes* are often used as simulations, the lessons learned help to guide the production of the final product. The approach suffers from many of the problems of the semi-formal approach.

In many problems there are opportunities to use all these approaches, using the more intensive and difficult methods for the more critical and sensitive areas of the problem. This may create difficulties, however, if parts of the system cannot be depended upon because of poor understanding of some aspects of the problem. We may not understand the problem sufficiently well to be able to identify the less critical parts. If we are to use this approach and to economise on using formal methods we must be aware of the dangers. There is a belief that we will eventually be able to use formal methods as part of a software engineering approach to model any problem and develop an effective solution. We will examine this belief later.

3. The empirical gap.

However, in Software Engineering there is little empirical evidence of the superiority of one method over another in large scale projects. There is a crisis of intellectual respectability in the subject. Not only is the evaluation of the methods used weak, the selection of the types of system and problem to focus on is very restrictive. In order to convince, in a scientific manner, that method A is better than method B in large design projects (and that is where the problems are) we must present rigorous evidence drawn from carefully controlled experiments of suitable complexity. This is, more or less, impossible in practical terms. Is there an alternative approach? The use of theoretical models of computing systems can provide some alternative approaches and this is the fundamental philosophy underlying this work.

We will develop our ides around the concept of a *computational model*. This is a class of mathematical theories which are thought to encapsulate the essence of the sort of systems that can be built with computer technology. There are a number of such theories, all of which have been shown to be equivalent. These include Turing machines [3], Markov algorithms [4], recursive functions [5] etc. Our approach is closest to the Turing model but it possesses a rather more practical flavour and, at the same time, the overt mathematical nature is modified and made more appealing by the judicious use of diagrams.

In the absence of a realistic empirical method for evaluating a design method by the use of large scale and repeatable scientific experiments we will focus on building a complete method for the identification, design, and testing of systems based on our theoretical foundation and framework. The full benefits of this will become more apparent later but it is possible, at this stage, to get some insight on what we hope to show.

Suppose that we have an easy to use modelling language that is based on a computational model paradigm. The first stage of solution building is to develop a model of the proposed system in its environment and to analyse the behaviour of this model. In order to discuss the model with the domain experts (the clients) we need to be able to represent it in a way that is understandable to them. The analysis should lead through a number of steps, we will call them *refinement steps*, to more complete models and, ultimately, to models that can be implemented in a more or less automatic form. Now we are in a position to examine what we have built and to try to establish if the behaviour of the executable/code/fabrication in the case of a hardware design match up with the detailed behavioural model we have built up during the requirements and specification stage. Because we are using computational models we know that both the specification and the implementation are just computational systems and thus expressible as computational models. So we can now use the theory of our computational models to develop highly powerful methods for comparing them, in other words, of testing the functionality of the implementation to see if it agrees with the specification. It is important to note that we must carry out some testing process since the implementation will have been generated by software tools, such as compilers, synthesisers, etc. which are always going to contain faults. We cannot assume that they always operate 100% correctly. Going one stage further, the operating environment, the operating system, the fabrication process etc. are also going to be subject to faults. Hence the final working system may be faulty because of all sorts of problems. We need to carry out functional tests to try to establish whether the system is working properly. Formal verification cannot guarantee this since it ill be based on a highly simplified model of the system generation process and its working environment. If we are going to have a powerful testing method we will need to integrate this into the design process, introduce concepts such as *design for test* to make best use of our testing methods and to look at the way in which test refinement can progress alongside design refinement in a practical way.

To be convincing the approach should address as many types of system as is possible, although allowing for the fact that certain types of system will require certain types of specialised design.

Consider a list of types of systems which we might like to use as representatives of the sort of challenges faced by computer scientists and software engineers.

- an automatic bank machine;
- a confidential database/management information system;
- an airline flight control system;
- a speech recogniser;
- an automatic reasoning device (theorem prover);
- a natural language processing system;

These systems/problem domains pose many different issues for the software engineer. We will try to explore some of them.

4. Correctness - what does this mean?

We now turn to the question of trying to understand what it means for a system to be correct. The simple answer is that the system is correct if it can be shown to satisfy the requirements.

Two issues arise -

- what is a requirement?
- how do we demonstrate that the system satisfies a set of requirements.

What is a *requirement*? This, and the process of *requirements engineering*, is the subject of considerable current research. Capturing the requirements from a client is difficult. It is also an ongoing process, since requirements can change with time. It requires that both client and software engineer can articulate clearly and communicate effectively. Both must have a consistent understanding of the problem, the environment and the solution constraints. Ideally they should also share some similar social, cultural and political attitudes, otherwise they may have rather different objectives when making judgments about unwritten aspects of the requirements.

Consider some alternative social paradigms:

Table 1:

	Technocratic paradigm	**Environmental paradigm**
core values	**economic growth, nature is a resource, controlling nature**	**self-sustaining, nature respected, harmony with nature**
economy	**market forces, risk + reward, individual perspective**	**public interest, safety, social perspective**
society	**centralised, large-scale, ordered**	**decentralised, small scale, flexible**
knowledge	**confidence in technology, rationality of means**	**concern about technol-ogy, rationality of ends**

Adapted from S. Cotgrove, “Risk, value conflict and political legitimacy.” [6].

So the underlying attitudes might be somewhat in tension - a client might think that safety means doing the minimum required, whereas a designer might regard safety as being much more than that - perhaps the best that current knowledge can provide (we have to accept that no safety-critical system is likely to be totally safe).

In a perfect world the entire set of requirements should be expressible in a precise, unambiguous language so that there can be no misunderstanding. However, the requirements must not only describe the functional behaviour that is required but many other issues: the cost of production, operation, change and maintenance, the needs of operators, customers, regulators and government authorities, and the general public.

Solutions must be *legal*. They must not expose users or developers to breaking the law, perhaps through the unauthorised use of someone else' property, ideas, designs, etc. or through some other aspect of the system.

Solutions must be *clearly definable*. For example, saying that the solution system has to be "easy to use" is not very helpful if the phrase is not defined in a way that enables potential solutions to be judged properly. Thus if the delivered product is found wanting under this category a messy court case is avoided. Ease of use can conflict with other issues such as safety - asking operators to confirm critical input values, by repeating them conflicts with ease of use as commonly perceived - but might be important to safeguard safety.

Thus, even if the requirements could be expressed very precisely, we have to resolve conflicts between many different aspects. In academic software engineering the common belief is that the requirements must be identified precisely. Then a formal specification can be built so as to be a technical reference point for subsequent design and implementation activity and quality control. This might be valid if the detailed requirements statements could always be used as a basis for system understanding and design.

This is not the case. In "straightforward" situations - perhaps well understood information systems, databases and in real time control systems - formal languages have been developed which allow for the construction of a formal specification of aspects of the required system. Much current thinking is based around the belief that a formal specification will prevent problems with the design and implementation of the system since it will act as a clear, unambiguous definition of the required solution. However, creating a formal specification is hard. In most cases only with the functional behaviour of the system is considered. The more qualitative attributes such as user friendliness etc. rarely feature. In many applications even the construction of a functional specification is far from straightforward. Even experts in formal methods rarely use their methods in the development of their tools. This is something of a sham but there are good reasons for it. For example, how can the functional specification of an automatic theorem prover to be made explicit? Here is an example where a very general, and thus vague, statement can be made about a system, but it is very difficult to turn it into a precise, formal specification. This, despite the fact that formal logic and the definition of what a theorem is have been extensively studied by logicians for centuries.

Artificial Intelligence (A.I.) systems are also often like this. A speech recogniser is very difficult to specify except in general terms; “a system that will recognise speech utterances in (British) English to an accuracy of 99% using a dictionary containing 10,000 words.” How can that be made more explicit in such a way that it could then be used as a basis for design refinement, implementation and correctness proving?

A similar problem exists for natural language processing systems where the sort of sentences that might need to be analysed or translated might never have been written before.

Software Engineering does not seem to be addressing these issues very successfully.

Thus, there are problems deciding what the requirements should be. Not only are the non-functional requirements difficult to express in precise terms but also the functional requirements for many types of systems are hard to describe. If an attempt is made we still face the problem of validating them with the client/user. The communication gap between client and engineer will pose a considerable obstacle here.

Many alternative approaches exist to try to deal with these problems. Gilb [7] describes ways in which metrics can be used to define and measure almost any type of non-functional requirement. Then the process of establishing whether a solution meets that requirement can be attempted. The problem of functional requirements for A.I. are addressed in many ways. Test corpora might be available which will be the basis for seeing if the system behaves in an appropriate way. The problem here, is that one might tune the system to fit the test sets but these may not be typical. Similar problems occur with neural net solutions. Requirements might be expressed in the form of examples - essentially representative sets of test sets which, hopefully, create enough cases to test a system effectively.

In all of these approaches the question of *validating* the system - establishing that it is correct - or in accordance with the requirements for a solution, will depend on the methods used to define the required system. The requirements may include the need for the system to be safe. Note that, ensuring that a system is safe is not the same as ensuring that the system is correct - unless safety is a specific issue identified in the requirements and being correct can guarantee safety.

Correctness, then, is concerned with the requirements. It should include more than functional correctness, things like - reliability, safety, usability etc. are all aspects that must be verified if they feature in the requirements. But we must also recognise that we may have a complete set of requirements but they may be for the wrong system, in other words *the requirements are not correct*. This will always be a risk and whatever techniques we use to represent our requirements must be understandable to the clients. This may then help to overcome the requirements capture problem to some extent.

5. Designing for correctness.

Transforming the requirements into a working system is the next issue. Consider a number of different contexts and possibilities. Firstly, there is the formally specified system, with a precise mathematical definition and a formal reasoning process that provides correctness by means of mathematical proofs that the implementation must satisfy the requirements. This is a dream of many in the formal methods community. It is probably unrealistic. A proof of correctness may be necessary but cannot be sufficient - we cannot model precisely the operating environment, including the users, the hardware and the communication channels which are all vital aspects of the system. Correctness proofs can offer some assurance but not TOTAL assurance.

No system will be released without extensive testing. Testing is a major activity of system development - up to 90% of project effort is involved in testing in some highly critical systems and even ordinary solutions may consume 50% of the project in testing. Techniques for designing testing strategies are mostly based on experience and “ad hoc” approaches. Myers [8] states that testing is much harder than design. It is also often less rewarding, no-one loves the test engineer (or the income tax inspector!).

Most approaches to testing systems depend on the existence of some detailed starting information - generally the code (in the absence of a formal specification) - from which test sets can be constructed systematically. A functional testing method requires some functional specification as a starting point and effective techniques for constructing efficient and revealing test sets are not available. Methods such as category-partition and boundary value testing, for example, are highly dependent on random effects.

Any methods must be well-founded and capable of robust analysis - it is particularly important to be quite clear about the foundations of any approach when the evaluation of any design method cannot sensibly be carried out on the basis of industrially sized case studies.

At the end of any testing process a decision has to be taken as to the likelihood of serious faults remaining in the implementation. This decision, up to now, has had to be taken on the basis of economic and marketing considerations, rather than on the basis of scientific analysis.

Little underlying theory exists; the evaluation of the effectiveness of testing methods is based on small case studies. Convincing empirical results on test effectiveness from large scale projects is lacking. Some attempts at utilising a formal specification as a basis for generating test sets have been made with limited success. A new technique for building test sets, based on a proper theory and which relates to an overall design method will be described in this book.

Turning to less formally defined requirements, how do we achieve correctness in such contexts? Here some comments by Wilks and Partridge [9] are relevant. They conclude that software engineering has little to offer A.I. Rarely are the methods of software engineering appropriate. This is because the problem domain is very often poorly defined, the systems are not closed and easily formalised. One needs to use an experimental approach in an attempt to understand the problem domain better, let alone the problem itself. Often a series of computational models (programs) are built which are then iterated until some reasonable approximation to the ill-defined system is available. This can then be tested at some considerable length and cost, if usually randomly. The RUN - UNDERSTAND - DEBUG - EDIT approach seems to be popular in this domain.

If a precise definition of the requirements is not available what does correctness mean? It is probably best approached from another direction. Wilks and Partridge talk of a system being "adequate" rather than correct. So perhaps a system is adequate if it is not inadequate. If we can say more about what we don't want rather than what we do want this might be a productive approach. This is not entirely crazy! Safety analysis in safety-critical systems is based on identifying unsafe states and removing them.

So

safe = not unsafe

and

correct = not incorrect.

Thus we enter a philosophy of verification that is fault based. We wish to demonstrate that undesirable behaviour is absent and to do this we will use formal proofs, where possible, but always rigorous testing. Looking at other things from this angle might be valuable. In other words the requirements document should also include what is NOT required.

There is a tendency to overkill in software design. To overspecify. Adding too many features. Offering more than is really required rather than what is essential. Kletz [10] said " We spend money on complexity but not on simplicity." How many of the features in an everyday word processor are used by the vast majority of people? Just because we can do it does not mean we should do it. We should concentrate on what is adequate not what is possible. However, there are compelling practical reasons for making systems as powerful as possible. We may need some of the extra features soon. It is very hard to adapt and extend existing systems when we need to. Part of the problem is that we rarely have enough detailed information about the existing system, developing new modules and bolting them on is a complex and error prone activity.

Possible solutions include a more modular or component based software industry. Reusing as much as possible. Reuse is one of the motivating pressures on object-oriented design. But it raises as many problems as it solves. Reusing objects and classes that have been developed successfully in other contexts is a plausible strategy. However there are problems. One is finding the right things to reuse, the problems of classifying, cataloging and retrieving suitable objects and classes is very hard. The context in which they are used might be very different. The modern air traffic control system is a case in point. With the greatly increased level of traffic expected should we adapt the present systems and reuse what we can or do the new demands on the system require us to completely redesign the system from scratch? Wiener considers this and other issues in[11].

Object oriented systems [12] and, in particular, object oriented programming languages such as C^{++} [13] and lately Java [14], have become extremely popular and widely used. Testing object-oriented systems, however, is not well understood [15]. Assuming that an object that has been well tested in another system/context can be used without repeating much of the testing in the new context is erroneous. A common place for programming errors to occur is at interfaces in software. Object-oriented systems have many more interfaces than traditional systems.

Therefore the testing problem and the demonstration of correctness or adequacy appears to be much harder. So we are coming to the conclusion that: correctness or adequacy will be dependent on testing and, where possible, formal proofs. Assuming that we can identify what the solution is supposed to be like, that is, the requirements can be identified.

In summary:

- defining systems and understanding their properties is difficult.
- engineering requirements is not well understood.
- identifying the functional requirements of A.I. systems may be difficult or even impossible.
- software engineering has focussed on a very restrictive class of problems.
- correctness has many facets - adequacy is another view of it.
- we may wish to have correct systems but this may not be achievable, adequate systems are a possible substitute.
- testing is and will remain a vital part of software engineering.
- more research on the relationships between testing and design is required.

6. An outline of the book.

Part 1. Building correct systems.

Chapter 1. Models of computer based systems.
The book begins with an introduction to the modelling of types of computer based systems. The most well known method is the state machine model, an integral component of most university computing courses and a widely used method in parts of industry, notably in hardware systems design and real time systems design. The model is very restricted, however, and we examine some more general approaches. Statecharts are an attempt to use finite state machine in the context of industrial reactive systems. It enables a complex state machine to be divided into manageable “chunks” and consequently more complex systems can be examined. We note some limitations with this model and introduce our main idea, a very general, powerful but intuitive model called the X-machine. We conclude with some simple examples.

Chapter 2. Business processes, problems and solutions.
What is a business process, what is an enterprise model? These issues are both key and universal. By constructing precise models of business processes and the way they interact with each other, the enterprise model, we can better understand the context and the problem. Identifying the requirements using a formal business process model is then possible. We look at this process and show how the X-machine can be used to think about the system from the user’s perspective. Examples of X-machine specifications of a simple business process are examined. A business process that involves time is also considered and this, together with a microprocessor example demonstrate the generality of the approach.

Chapter 3. Testing, testing, testing!
A review of the practice of testing begins a more detailed look at an important method of functional testing, namely state machine based testing. The limitations of the finite state machine model described in the previous chapter limit the applicability of this type of testing. The complete testing method derived from the stream X-machine theory is then described in the context of a simple example. An important aspect of the approach is the clear and unambiguous identification of design for test conditions that must be satisfied by a specification if the full power of the testing method is to be exploited. The design philosophy is one of defining a model and a system in terms of the integration of its lower lever components. If these are tried and trusted then the testing method will find *all* faults in the system. This then gives us a way of reasoning about the faults that remain in a system when testing has been completed.

Chapter 4. Integrating testing and design into the lifecycle.
Refinement is a key concept in our approach to software and systems engineering. It is unlikely that a complete model can be built in one go. The need to explore, understand and confirm the functional requirements of a system necessitate an incremental approach. The application of the refinement principle to the generation of a test set is a major advantage in terms of the time and cost of creating test sets and carrying out testing. The development of incremental models and test sets can be linked naturally with the idea of a component. We consider a precise definition of a component which can be related fully to the refinement design and testing strategy outlined.

Chapter 5. A case study.
This chapter is an account of applying the approach to a real problem and the client's reaction. We describe the client’s business processes and build a complete enterprise model. This leads to a formal specification of the proposed solution. Extracts from this model are given. The specification was implemented incrementally using a widely used 4GL (Visual Basic) and this enabled the client to comment on and confirm the requirements. Such changes must be addressed by any specification method and were dealt with comfortably in this case. The complete system was tested using the method explained above, it has since been in use for many months with no faults of any kind. The case study demonstrates that the methods outlined constituted a simple and effective procedure for building a correct system.

Part 2. Theoretical foundations.

Chapter 6. Theory of X-machines.
The basic theory of X-machines is introduced in this chapter. A particular class of X-machines, called stream X-machines, are investigated in depth. The theory of stream X-machine refinement, a key concept in the desire to model and analyse systems incrementally, is examined in detail.

Chapter 7. Complete functional testing.
This chapter begins with the background necessary to understand the testing method. Thus state machine morphisms, state machine minimality and state equivalence are discussed. The basic results of state machine testing theory, test set construction, transition cover, characterisation set and the complexity of this method are examined. The stream X-machine testing theory is now studied and this leads to a consideration of the design for test conditions, the fundamental test function and the fundamental theorem of complete functional testing. We conclude with a discussion of the complexity of this type of testing.

Chapter 8. Refinement testing.
The theory of refinement testing is introduced and described in detail. The testing of refinements is studied and is followed by the details of how the refinement testing method can be used in the context of a component based design process. This is illustrated by a simple example.

Bibliography.

Acknowledgements.

We would like to thank the members of the Verification and Testing Research Group in the Department of Computer Science, University of Sheffield. In particular Kirill Bogdanov has read much of the manuscript and contributed in many ways to the development of the testing method and its associated tools, Sarah Chambers, Tony Simons, Camilla Jordan and Matt Fairtlough who have all read parts of the work and provided many helpful comments. Also to Klaus Grimm, Heiko Doer and Harbarjhan Singh of Daimler-Benz Research Laboratories, Berlin, who have funded work on the method and supported the research in many ways. We would also like to thank Mohamedd Al-Qaed who took the ideas and applied them to a real development project, much of the detail in Chapter 5 is due to him. Both Gilbert Laycock and Andreas Grondoudis have made important contributions to the X-machine testing method. A number of our senior students have also contributed to the greater understanding of X-machines in the design and testing of specific systems and have acted as “guinea-pigs” in trying out these ideas in their projects and assignments.

CONTENTS