Using Formal Specifications to Support Testing

R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan

Formal methods and testing are two important approaches that assist in the development of high quality software. While traditionally these approaches have been seen as rivals, in recent years a new consensus has developed in which they are seen as complementary. This article reviews the state of the art regarding ways in which the presence of a formal specification can be used to assist testing.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program, Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.2 [**Software Engineering**]: Design Tools and Techniques

1. INTRODUCTION

With the growing significance of computer systems within industry and wider society, techniques that assist in the production of reliable software are becoming increasingly important. The complexity of many computer systems requires the application of a battery of such techniques. Two of the most promising approaches are formal methods and software testing.

Traditionally formal methods and software testing have been seen as rivals. That is, they largely failed to inform one another and there was very little interaction between the two communities. In recent years, however, a new consensus has developed. Under this consensus, these approaches are seen as complementary [Bowen et al. 2002; Hoare 1996]. This has led to work that explores ways in which they complement each other.

The use of a formal specification or model eliminates ambiguity and thus reduces the chance of errors being introduced during software development. Naturally, there still remains the issue of obtaining a formal specification that matches the actual customer requirements and this is complicated by the tendency for stated requirements to change during development. Where a formal specification exists, both the source code and the specification may be seen as formal objects that can be analyzed and manipulated. A formal specification could be analyzed in order to explore the consequences of this specification and potentially find mistakes (see, for example, [Kemmerer 1985]). If this is done then we have greater confidence that we are testing the *system under test (SUT)* against the *actual* requirements. The use of a formal specification introduces the possibility of the formal and, potentially, automatic analysis of the relationship between the specification and the source code. This is often assumed to take the form of a proof, but such a proof cannot guarantee operational correctness. For this reason, even where such a proof is believed to exist, it is important to apply dynamic testing (see, for example, [DeMillo et al. 1979; Fetzer 1988]).

Software testing is an important and, traditionally, extremely expensive part of the software development process, with the importance and cost depending on the nature and criticality of the system. Studies suggest that testing often forms more than fifty percent of the total development cost, and hence dominates the overall production cost. Where formal

specifications and models exist, these may be used as the basis for automating parts of the testing process and this can lead to more efficient and effective testing. It may transpire that its support for test automation is one of the most significant benefits of formal model building. The links between testing and formal methods do, however, go well beyond generating tests from a formal specification.

For example, if we have a formal specification then we may be able to use this as the basis for generating a test oracle: a system that determines whether an observed input/output behaviour is consistent with the specification. Naturally, the datatypes used in the specification are often more abstract than those in the SUT, in which case it is necessary to have a user-supplied abstraction function. Note, however, that in general such an abstraction function can be uncomputable.

Many systems have a persistent internal state and testing then includes finding an input sequence that takes the system to a required state for a test and checking that the state is correct after the test. Again, formal modelling can help us here, and provide support for finding appropriate paths through a finite state structure. There is, as ever it seems, a fundamental limitation to what is achievable due to the feasible path problem: a chosen path might not be feasible and it is undecidable whether a path is feasible. While this problem affects almost all formalisms, there are general approaches such as constraint solving and theorem proving that can assist.

The presence of a formal specification or model makes it possible for the tester to be clearer about what it means for a system to pass a test. This may be achieved through the use of test hypotheses [Gaudel 1995] or design for test conditions [Ipate and Holcombe 1997; Holcombe and Ipate 1998]. Similar ideas can be found in the generation of tests from finite state machines in which some *fault model* is assumed [ITU-T 1997]. Using these approaches it is possible to generate tests that determine correctness with respect to a specification under certain conditions, circumventing Dijkstra's famous aphorism that testing can show the presence of bugs, but never their absence [Dijkstra 1972]. If program analysis could be used in order to either prove that these conditions hold or provide confidence in them holding, then we would have a very tight coupling between specification and testing; a combined formal analysis of specification and test could provide very strong guarantees of correctness.

On a slightly different tack, information gathered by testing may assist when using a formal specification. Testing can be used in order to provide initial confidence in a system before effort is expended in attempting to prove correctness. Where it is not cost effective to produce a proof of conformance, the developers may gain confidence in the SUT through systematic testing. This might be complemented by proofs that critical properties hold. A proof of correctness might also use information derived during testing. Finally, a proof of correctness relies upon a model of the underlying system and dynamic testing might be used to indirectly check that this model holds. An interesting challenge is to generate tests that are likely to be effective in detecting errors in the assumptions inherent in a proof.

In this article we explore the many ways in which the presence of a formal specification can support testing. We explore issues such as: what sorts of test cases do we wish to produce; what does the result of applying these test cases tell us; and, how can we generate them? We also discuss issues regarding testability — a property whose significance is recognized in hardware development but much less so in software development. The paper is structured as follows. In Section 2 we provide a brief review of formal methods

and Section 3 then discusses some general relationships between formal methods and testing. Sections 4–8 describe work on generating tests from a specification written using a model based notation such as Z, VDM, or B; a finite state machine; a process algebra; a hybrid specification language; or an algebraic specification language respectively. Section 9 describes the use of model checking and constraint satisfaction techniques in automating test generation. Section 10 discusses future research directions and Section 11 draws conclusions.

2. INTRODUCTION TO FORMAL METHODS

Formal specification languages are mathematically-based languages whose purpose is to aid the construction of systems and software. Often backed by tool support, they can be used to both describe a system and also then to analyze its behaviour, possibly verifying key properties of interest.

The engineering rationale behind formal methods is that time spent on specification and design will be repaid by a higher quality of product, this contributing to the commercial rationale of trying to reduce the cost of rework later on. Formal methods of course do not guarantee correctness, but their use aims to increase our understanding of a system by revealing errors or aspects of incompleteness that might be (very) expensive to rectify at a later date. Naturally, development is still likely to be iterative, since requirements usually change during a development project and the use of formal specification languages cannot be expected to entirely eliminate errors.

Over the last 25 years formal methods have reached a sufficient level of maturity so that they are routinely applied in hardware construction, and applied with a certain frequency in software construction, particularly for safety critical software. We now describe the main types of formal specification languages before discussing the problem of test generation in Sections 4–8.

2.1 Specification Languages

The primary idea behind a formal method is that there is benefit in writing a precise *specification* of a system, and formal methods use a formal or mathematical syntax to do so. This syntax is usually textual but can be graphical. A *semantics* is also provided, that is, a precise meaning is given for each description in the language.

A specification of a system might cover one or more of a number of aspects, including its functional behaviour, its structure or architecture, or even cover aspects of non-functional behaviour such as timing or performance criteria.

A precise specification of a system can be used in a number of ways. First, it can be used as the process by which a proper understanding of the system can be articulated, thereby revealing errors or aspects of incompleteness. The specification can also be analyzed or it can be verified correct against properties of interest.

A specification can also be used as a vehicle for driving the development process, either through refining the specification towards code or by direct code generation. Of course, a key aspect of the development process is testing, and a specification can also be used to support the testing process. Indeed, the purpose of this paper is to explore this issue in some depth.

A variety of different formal specification techniques exist, some are general purpose whilst others stress aspects relevant to particular application domains (e.g., concurrent systems). Most are backed-up by varying degrees of tool support. In what follows we survey

some of the most popular notations as a prelude to a discussion on how their use can be integrated into the testing process.

2.2 Model-Based Languages

There are a number of different ways to write a precise specification. One approach is to build a *model* of the intended behaviour, and languages such as Z [Spivey 1988; 1992], VDM [Jones 1991] and B [Abrial 1996] do so by describing the states the system could be in together with operations that change the state.

The states of the system are typically described using sets, sequences, relations and functions, and operations are described by predicates given in terms of pre- and post-conditions. There are a number of ways to structure such a specification. Z, for example, uses a language of *schemas* to do so, where each schema consists of a declaration together with a predicate which constrains the schema.

Consider, for example, the specification of a bounded stack. We describe the possible states of the stack via a schema. We would like to use the stack to keep things that come from a particular set, say *Object*. This can be specified in Z as follows:

$$[Object]$$

This defines a set, *Object*, and at this level of abstraction we are not concerned with the structure of the elements of this set. Further, in order to specify the bounded stack, we define a constant, say *maxSize*, beyond which the size of stack cannot grow.

$$maxSize == 20000$$

Now, we can describe the possible states of the stack using the following schema:

Stack

$$items : \operatorname{seq} Object$$

$$\#items \leq maxSize$$

This declares *items* to be a sequence of elements from the set *Object* such that the length of *items* does not exceed *maxSize*. The following initialization schema *StackInit*, gives the initial configuration of the system. Priming ($'$) of a variable denotes the after state of that component. Thus *items'* refers to the after state of variable *items*.

StackInit

$$Stack'$$

$$items' = \langle\rangle$$

The following operation, *Top*, returns the topmost value on the stack provided it is not empty. The *Top* operation does not modify the state of *Stack*.

Top

$$\Xi Stack$$
$$x! : Object$$

$$items \neq \langle\rangle$$
$$x! = head\, items$$

We can further define the usual *Pop* and *Push* operation on the stack with obvious functionality as follows:

Pop

$\Delta Stack$
$x! : Object$

$items \neq \langle\rangle$
$items' = tail\, items$
$x! = head\, items$

Push

$\Delta Stack$
$x? : Object$

$\#items < maxSize$
$items' = \langle x? \rangle \frown items$

Various conventions are used in Z, for example, names ending in ? denote input, and names ending in ! denote output ($x?$ and $x!$ are thus inputs and outputs of type *Object*, respectively). $\Delta Stack$ denotes a potential change of state for components declared in *Stack*, and $\Xi Stack$ denotes no change in state in an operation schema.

Further information about Z and applications is given in [Spivey 1988; 1992]. In Section 4 we discuss methods for testing on the basis of a model-based specification.

2.3 Finite State-Based Languages

Model-based languages such as Z, VDM and B can describe arbitrarily general systems, and have potentially infinite state. This generality has a drawback in that it makes reasoning less amenable to automation. This drawback is not, however, present in the case of *finite state-based* specification languages.

As their name suggests, finite state-based languages define their state from a finite set of values, which are often presented graphically with state transitions representing changes of state akin to operations in a notation such as Z. Examples of such languages include finite state machines (FSMs) [Lee and Yannakakis 1996], SDL [ITU-T 1999], Statecharts [Harel and Gery 1997] and X-machines [Holcombe and Ipate 1998]. FSM based test techniques have been considered when testing from such specifications. Much of the work on testing software from an FSM has been motivated by protocol conformance testing, since FSMs are suitable for specifying the control structure of a communications protocol. However, more recently, FSM based testing has been used within an approach called model-based testing in which a model is produced in order to drive testing (see, for example, [Farchi et al. 2002; Grieskamp et al. 2002]). Note that the term 'model-based' is used rather differently in model-based testing and model-based specification languages.

An *FSM* can be formally defined as $F = (S, X, Y, h, s_0, D_F)$ where

—S is a set of n states with s_0 as the initial state;

—X is a finite set of input symbols;

—Y is a finite set of output symbols;

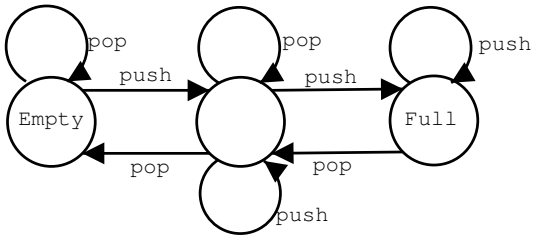

Fig. 1. An FSM for a bounded stack

—$D_F \subseteq S \times X$ is the specification domain; and

—$h : D_F \longrightarrow \mathcal{P}(S \times Y) \setminus \emptyset$ is a behaviour function, where $\mathcal{P}(S \times Y)$ is the powerset of $S \times Y$.

For $(s,a) \in D_F$ and $(t,b) \in h(s,a)$, $\pi = (s,a/b,t)$ will be called a *transition* from s to t with input a and output b. A machine F is said to be a *deterministic* (FSM) when $| h(s,a) | = 1$, for all $(s,a) \in D_F$. In a deterministic FSM (DFSM) instead of the function h we use the *next state function* δ and the *output function* λ ($\delta(s,a) = t$, $\lambda(s,a) = b$, for transition π). An FSM is said to be *completely specified* if $D_F = S \times X$.

Given an FSM F there is a corresponding language $L(F)$: the set of input/output sequences (traces) that can occur from the initial state of F. Given FSMs F_1 and F_2 it is normal to say that F_1 conforms to F_2 (or is a reduction of F_2) if and only if F_1 and F_2 have the same input alphabets, F_1 is defined whenever F_2 is defined, and $L(F_1) \subseteq L(F_2)$. In the case where F_1 and F_2 are DFSMs, this reduces to $L(F_1) = L(F_2)$.

An FSM representation of a bounded stack would be given as shown in Figure 1. Here there are three states representing the conditions: the stack is empty; the stack is full; and the stack is not empty and is not full. Note that transitions associated with *top* have not been included since these do not alter the state. Further, in this diagram the FSM is complete and thus the action of *pop* on the empty stack and *push* on a full stack are stated.

The FSM as shown is non-deterministic since, when the stack is not full and is not empty, the actions *pop* and *push* may leave the stack in this state or move it to a different state. For example, if the stack contains one element then *pop* moves it to the state *Empty* and otherwise *pop* does not change the state. If the size of the stack is known, it is possible to produce a deterministic FSM that represents the stack. Here, if the stack size is n, there would be one state for each $0 \leq i \leq n$, the state associated with i representing the condition in which the stack contains i elements.

Many specification languages such as Statecharts, SDL, and X-machines that have a finite state structure have additional internal data. Transitions represent operations that can access this data, change this data, and have guards that can refer to this data. Such a specification is an *extended finite state machine (EFSM)*. If the data consists of variables with finite types then an EFSM can be expanded out to form an FSM, although a combinatorial explosion can occur. Even if the types are not finite we describe such specifications as being finite state-based since they model the system using a finite set of states and an internal memory.

Whilst techniques such as Z, VDM and B are primarily targeted towards the description of sequential systems, notations such as SDL, Statecharts and X-machines allow explicit representation of concurrent activity. Specifications in these languages can often be seen

as one or more EFSMs that may communicate. In Section 5 we discuss methods for testing on the basis of a finite state-based specification.

2.4 Process Algebra State-Based Languages

Concurrency can be given a very elegant algebraic treatment, and *process algebras* describe a system as a number of communicating concurrent processes. Examples include CSP [Hoare 1985], CCS [Milner 1989] and LOTOS [ISO 1989a].

Finite-state based languages such as Statecharts and SDL can also be used to describe a system as a set of communicating concurrent processes. However, process algebras have a rich theory that provides alternative notions of conformance described in terms of implementation relations. The implementation relations capture several types of observations that can be made, in addition to traces (sequences of inputs and outputs), and different properties of the environment.

CSP, for example, describes a system as a collection of communicating *processes* running concurrently and synchronizing on *events*.

A CSP definition of a stack might be given as follows:

$$\begin{aligned} &Stack(\langle\rangle) = push?x : X \rightarrow Stack(\langle x\rangle) \\ &Stack(\langle y\rangle \frown s) = push?x : X \rightarrow Stack(\langle x\rangle \frown \langle y\rangle \frown s) \\ &\qquad\qquad \Box \\ &\qquad\qquad pop!y \rightarrow Stack(s) \end{aligned}$$

Here a parameterized process is described by guarded equations. Channels such as *push* and *pop* can have input, $?x : X$, or output, $!y$, associated with them. These *events* can prefix behaviour (denoted using $\rightarrow$), allowing the sequencing of events to be prescribed. Choice is modelled here by the operator $\Box$, and importantly there are operators to express concurrent and communicating behaviour. Note that this stack example is similar to one that can be written in a finite state-based language. An example given below brings out the differences between process algebras and finite state-based languages.

Interleaving is the concurrent execution of two processes where no synchronization is required between the components. It is described by the process

$$P_1 \,|||\, P_2$$

in which processes P_1 and P_2 execute completely independently of each other (and do not synchronize even on events in common).

CSP also has a number of operators to describe parallel composition which allow processes to selectively synchronize on events. For example, the *interface parallel* operator synchronizes on events in a given set A but all other events are interleaved. This allows P_1 and P_2 to evolve separately, whilst events in A are enabled only when they are enabled in *both* P_1 and P_2.

A *labelled transition system (LTS)* may be used to describe the behaviour of a specification written in a process algebra and thus work on testing from such specifications has focused on testing from LTSs. Intuitively an LTS can be thought of as a representation of the behaviour of a system defined by the occurrence of events which change the state of the system. Informally [1] they are best represented as trees or graphs, where nodes represent states and edges labelled with events represent transitions. See, for example, Figure 2 that

[1] A formal definition of an LTS is given in, for example, [Tretmans 1996].

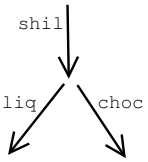

Fig. 2. A simple LTS

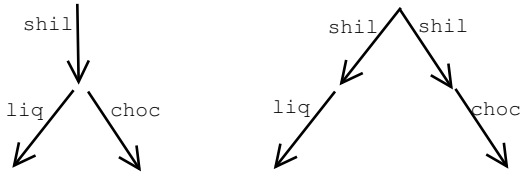

Fig. 3. Non-equivalent agents p_1 (left) and p_2 (right)

describes an example from [Tretmans 1996]. Here, there is one initial action, *shil*, which represents the input of a shilling. Having received the input of a shilling, the system may now interact with its environment through the action *liq* or the action *choc* that represent the output of a bar of liquorice or a bar of chocolate respectively. Having done this, the system is not capable of any further action; it is said to deadlock.

The events occurring as labels are observable, and denote the set of actions the system may perform. A special event τ is used to denote an internal (silent) action. Internal actions are not observable.

Recall that the semantics of an FSM F is the regular language $L(F)$ and that in order for one FSM F_1 to conform to another FSM F_2 we must have that $L(F_1) \subseteq L(F_2)$. This corresponds to observing elements from the set of *traces* that the SUT may produce.

DEFINITION 1. *A trace is a sequence of observable actions (or events). The traces of a labelled transition system specification S, traces*(S)*, are all sequences of visible actions that S can perform.*

For example, the set of traces of the LTS in Figure 2 is $\{\varepsilon, \langle shil\rangle, \langle shil, liq\rangle, \langle shil, choc\rangle\}$, where ε denotes the empty sequence.

LTSs have a richer notion of conformance than FSMs, reflecting the possibility of making a wider range of observations. By contrast to DFSMs, trace equivalence is almost never used. The conformance relations used with LTSs leads to agents that would be equivalent in an FSM, being distinguishable. Consider the example given in Figure 3, described in [Tretmans 1996]. Consider the first agent, p_1. Having performed *shil*, p_1 is capable of performing either *liq* or *choc*. In contrast p_2, having performed *shil*, either is capable of performing *liq* but not *choc* or is capable of performing *choc* but not *liq*. The two agents in Figure 3 are distinguishable where it is possible to observe the *refusal* of an event. However, since the set of sequences of labels are the same, the corresponding FSMs are equivalent.

An important observation is deadlock: an agent being in a state in which it cannot act. It is often useful to reason about the sequences of actions that may take an agent to a deadlock

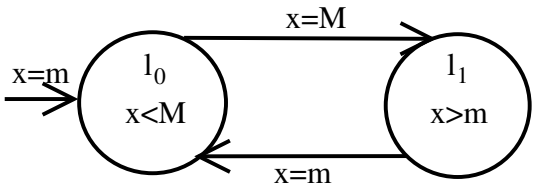

Fig. 4. Hybrid Automaton for Temperature controller

state.

There are a number of implementation relations that may be used with LTSs and some of these are described in Section 6. They reflect different forms of communication and varying properties of the environment. These implementation relations also vary in how they treat input and output: some do not distinguish input and output in their treatment while other (more recently developed) implementation relations reflect the nature of testing by treating input and output differently, insisting that the SUT cannot block input and the environment cannot block output. In Section 6 we discuss implementation relations and methods for testing on the basis of an LTS.

2.5 Hybrid Languages

Many systems are built with a combination of analog and digital components. In order to specify and verify such systems it is necessary to use a specification language that encompasses both discrete and continuous mathematics. There has been recent interest in these *hybrid languages*, such as CHARON [Alur et al. 2000; Hur et al. 2003].

A simple example of a nonlinear hybrid system is that of a *Temperature Controller*. The temperature of a room is controlled through a thermostat which continuously senses the temperature and turns the heater on and off. The temperature is governed by a set of differential equations. When the heater is off, the temperature x decreases according to

$$x(t) = \theta e^{-Kt}$$

where t is the time, θ denotes the initial temperature and K is a constant determined by the room. When the heater is on, the temperature is governed by

$$x(t) = \theta e^{-Kt} + h(1 - e^{-Kt})$$

where h is a constant that depends on the power of the heater. Initially, we assume that the heater is on and the temperature is m degrees. We also wish that the temperature is kept between m and M degrees. A hybrid automaton describing the system is given in Figure 4.

As can be seen the automaton has two states l_0 and l_1 with an initial condition $x = m$. Each state is annotated by an invariant and has an associated physical law that governs its rate of change (these laws are not shown in Figure 4). For example while in state l_1 (in which the heater is off) the room temperature is always kept greater than the minimum temperature and the temperature rate of change is governed by $x(t) = \theta e^{-Kt}$. As the room temperature drops to m degrees the heater is turned on (state l_0).

In Section 7 we discuss methods for testing on the basis of a hybrid specification.

2.6 Algebraic Languages

Process algebras are amenable to algebraic manipulation; however, there are also languages which describe a system solely in terms of its algebraic properties. These algebraic specification languages describe the behaviour of a system in terms of *axioms* that characterize its desired properties. Examples of algebraic specification languages include OBJ [Goguen and Tardo 1979; Goguen and Malcolm 2000] and the Common Algebraic Specification Language (CASL) [Mosses 2004; Bidoit and Mosses 2003]. In this paper we use CASL in order to illustrate the approach.

In mathematical terms an *algebra* (or an *algebraic system*) consists of: (1) a set of symbols denoting values of some type, referred to as the *carrier set* of the algebra; and (2) a set of operations on the carrier set. To describe the rules that govern the behaviour of the operations, it is necessary to specify:

—The *syntax* of the operations. This is done via a signature for each operation giving the domain and range (or co-domain), corresponding, in effect, to the input parameters and the output of the operation, respectively.

—The *semantics* of operations. This is done via equations (or axioms) that implicitly describe the required properties. The axioms are usually formulated as equations each of which may be qualified by a condition.

For example, an algebraic specification of a much-simplified bank system with an `account` type might have operations:

—`empty` to create a new empty account;

—`credit` to credit money to an account;

—`debit` to withdraw money from an account;

—`balance` to enquire about the balance remaining in an account.

Assuming the existence of a specification of `Nat`, the natural number type, which is used to represent money just for simplicity, the complete specification with operation signatures and appropriate equations might be of the form:

```
spec Bank =
     Nat
then
     sort account
     ops
          empty:  account
          credit :  Nat * account -> account;
          debit :  Nat * account -> account;
          balance :  account -> Nat
     vars
          n:  Nat; acc:  account
     axioms
          balance( empty ) = 0;
          balance( credit(n,acc)) = balance(acc) + n;
          balance(acc) ≥ n => balance( debit(n,acc) ) = balance(acc) - n;
          balance(acc) < n => balance( debit(n,acc) ) = 0 end
```

The first axiom says that the balance of an empty account is zero. The second axiom says that the balance after crediting `n` units of money to an account `acc` is the balance before the transaction with `n` added. The third axiom handles a debit in a similar way, but subtracting rather than adding and only provided the withdrawal would not make the balance negative. The last axiom says that an attempt to debit more than is in the account empties the account of the money that is present, leaving a balance of zero, i.e. no overdraft is allowed.

Sometimes it is useful to classify operations into the following distinct categories: constants, constructors, transformers and observers. A *constant* operation returns an initial object of the abstract type under consideration. *Constructor* and *transformer* operations alter (or transform) a value of the type in some way. The difference between constructors and transformers is that the former, together with constant operations, form a minimal set of operations for generating any value of the type, i.e. the carrier set. *Observer* operations return values of some type other than the one under consideration. In the example, `empty` is a constant operation, `credit` and `debit` are constructor operations and `balance` is an observer operation. The specification, as written, possesses no transformer operations.

One important argument in favour of the algebraic approach to specification is that the equations can sometimes be used to provide a mechanism for evaluation of syntactically valid, but otherwise arbitrary combinations of operations. For example, the sequence of taking an empty account, crediting 100 units of money to it, and then enquiring about the resultant balance in the account, would be phrased as the expression `E`, where:

```
E = balance( credit(100,empty) )
```

Using axiom 2 with `n=100` and `acc=empty`, this can be written as:

```
E = balance( empty )+ 100
```

Then, using axiom 1, this becomes:

```
E = 0 + 100
```

The process just illustrated, that uses the axioms as rewrite rules, is known as *term rewriting* and if every sequence of rewrites converges to a unique normal form in a finite number of steps, then the specification is said to be *canonical*. `E` is an example of a *ground term* since, unlike the axioms, it does not involve any free variables. Term rewriting is particularly useful, as far as testing is concerned, since algebraic specifications can then, in effect, be executed with chosen test situations. However, in general, theorem proving has greater potential, since it enables interesting properties of such specifications to be proved and we will see in Section 8 that this is sometimes required in testing.

It should be noted that there are many different algebraic specification notations, each with their own concrete syntax. Indeed, even within the same notation family there can be numerous differences in dialect. Nonetheless, the ideas described above, whereby the syntax of operations is specified by means of their signatures and the behaviour of operations in combination with each other is specified by means of equations, are generic. In Section 8 we discuss methods for testing on the basis of an algebraic specification.

3. FORMAL METHODS AND TESTING

3.1 Introduction

Software testing is mostly about *empirically* checking correctness. Formal methods, on the other hand, have traditionally been about the *formally* verifying the correctness of software. A major thrust of formal methods is the introduction of system models early in the lifecycle, against which the software can be proven through the use of appropriate mathe-

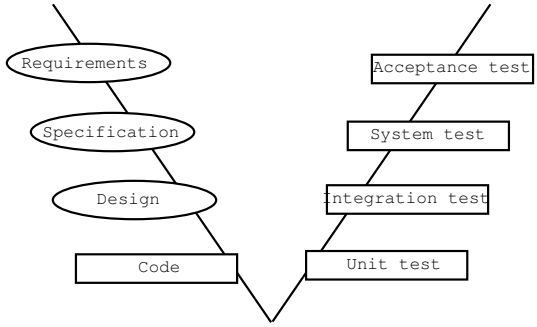

Fig. 5. Code-based testing

matics.

A starting point for examining the relationship between these two disciplines is to realize that both are model-based. The question naturally arises, what benefit can be accrued from using testing techniques in relation to the models used in formal methods? And what benefits might there be in using the mathematical basis of formal methods in the domain of testing?

By using formal methods and testing together, we can reduce the cost of development by applying testing techniques much earlier in the lifecycle while defects are relatively inexpensive to correct. We might also be able to automate more of the testing process by:

—Generating functional test cases from the specification of a system.

—Deriving provably correct oracles for checking the results of tests.

Figure 5 portrays, in the form of the V-model, the limited benefits that can be accrued to testing when the only formal model available is the code itself. Down the left-hand side each deliverable can be checked but since only the code is a formal object this checking is manual and depends upon interpreting potentially vague and ambiguous documents. Static analysis of the code can assist testing in deriving adequacy criteria, and dynamic analysis of the code can be used to assess whether the criteria have been met.

Figure 6 portrays the far greater possibilities that exist when formality is introduced into higher layers, the specification and design. Now, in addition to code-based testing, the following model-based test processes become possible:

—Properties of the specification can be proved. The specification can be "tested" using model checking or theorem proving.

—The specification can be validated by using testing techniques based on "executing" the abstract model through animation.

—Coverage criteria can be applied to the abstract model represented by the specification, e.g. coverage of all logical conditions, or coverage of all paths through a statechart. This allows us to determine whether we have covered the specification as well as the code in testing. Naturally, as with code-based testing, automated test generation is complicated by the feasibility problem: the problem of deciding the feasibility of a path is undecidable.

—System level functional test cases can be generated from the specification.

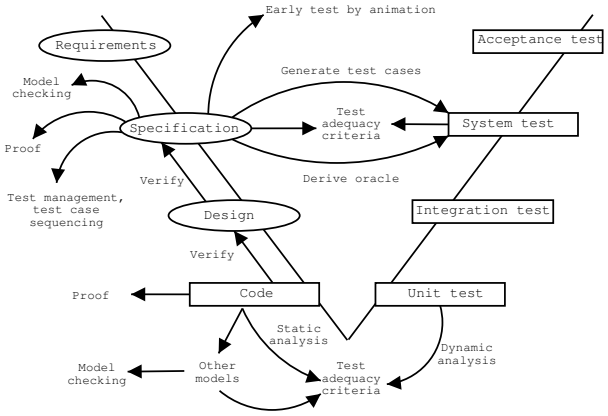

Fig. 6. Early model-based testing

—An oracle for system tests can be derived for checking the results of actual test executions. As discussed earlier, this is complicated if the types used in the specification and implementation are different, since we then need a user defined abstraction function.

—The design, and in turn the code, can be verified against the specification.

—Formal methods can suggest new kinds of execution models of code, with new kinds of test adequacy criteria.

—Properties of the code can be proved.

—Execution models can be "tested" using model checking.

—Test management can be enhanced by reasoning about, for instance, the sequencing of test cases.

The above processes that apply to the formal specification apply equally to other formalized system models, such as the design, giving a model-based approach to testing at every level of development.

We now describe a theoretical framework for understanding the relationship between testing and proof, based largely on a key paper by Gaudel [Gaudel 1995]. In addition to addressing the testing/proof relationship, the framework helps explain some important things about the adequacy of finite test sets, and how their selection can be justified.

3.2 Foundations for Combining Formal Methods and Testing

While it may seem that all testing can tell us is whether an implementation conforms to the specification on a given test set, it has been argued that it can tell us more than this. Goodenough and Gerhart formalized this intuition through the notions of a test technique being *valid* and *reliable* [Goodenough and Gerhart 1975]. A test technique is valid if for every faulty program the test technique is capable of producing a test set that shows that the program is faulty. A test technique is reliable if every test set that can be produced using the technique leads to the same verdict (an implementation either passes all of the test sets or fails all of the test sets). If a test technique is both valid and reliable then any test set produced using this technique will determine whether an implementation is correct.

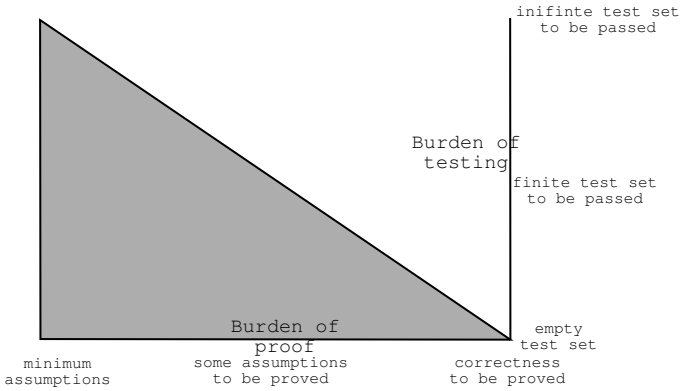

Fig. 7. Framework for the relationship between proof and testing

However, it has been observed that these properties are not independent: if a test technique fails to have one of these properties it must have the other [Weyuker and Ostrand 1980].

Weyuker and Ostrand introduced an alternative notion, that of a revealing subdomain [Weyuker and Ostrand 1980]. A subdomain D_i (of the input domain) is revealing if either the implementation conforms to the specification on all values from D_i or the implementation conforms to the specification on none of the values in D_i. Given a revealing subdomain it is sufficient to choose only one test case from this subdomain. If one can divide the input domain into a finite number of revealing subdomains then by taking one test case from each subdomain one gets a finite test set that is guaranteed to determine correctness: the SUT is correct if and only if it passes all of the test cases in the test set.

Young and Taylor [Young and Taylor 1989] present a taxonomy of both static and dynamic approaches. They classify techniques as either sampling or folding, where sampling techniques check a sample of values while folding techniques use abstraction to reduce the set of values to be considered (they fold together states). Thus, for example, choosing a test case from each of a finite set of subdomains is a sampling technique while symbolic evaluation techniques use folding but also use sampling when not all paths through a program are considered [Young and Taylor 1989].

A key paper by Marie-Claude Gaudel [Gaudel 1995], which draws on the previous work of Goodenough and Gerhart, and Weyuker and Ostrand has laid down a framework in which the relationship between testing and proof can be understood. As illustrated in Figure 7, the framework admits a spectrum with two extremes: one at the top-left of the diagram where no proof is carried out, but an infinite amount of testing is necessary; the other at the bottom-right, in which no testing is performed, but a complete proof of the system and its context is necessary.

Movement between the two extremes is possible by making assumptions about the artefact under test that support the selection of smaller, eventually finite, test sets. Most test selection techniques are based on assumptions such as

> if the program is correct for one set of input values in this range, then it is correct for all such input values

or

> if the program is correct for one set of values that execute a particular path, then if is correct for all input values that execute that path.

Such assumptions are referred to as test selection hypotheses. Although there are probably several other types of test selection hypothesis, Gaudel [Gaudel 1995] identifies two types. The first example hypothesis above, for instance, is typified as a *uniformity hypotheses*, being about the uniformity of program behaviour on ranges of data. It is this kind of assumption that is made every time test points are selected from an input data subdomain.

The members of another class of hypotheses identified by Gaudel are known as *regularity hypotheses*. These are about the assumed regularity of program behaviour as the size of data increases. For example, it might be assumed that if the program works for buffer sizes 0, 1 and 2, then it will work for buffers of all sizes. This would be the underlying assumption being made when the system is tested only on buffer sizes less than 3.

By stating test selection hypotheses and the resulting set of tests, the burden of validation is shifted from testing to proof, i.e. complete validation would involve demonstrating successful execution of the tests, and proving the test hypotheses. It is thought to be Professor Tony Hoare who first pointed out that a test is like proving the base case of an inductive proof; establishing the test hypothesis is akin to proving the inductive step. Unfortunately, the base case of an inductive proof is usually the easy bit; proving the inductive step, the test hypothesis, is hard.

There is a sense in which the amount of work represented through the spectrum is invariant: at the extremes, infinite testing and complete proof of correctness are usually both infeasible; in the middle, there may be a finite amount of testing to be done, but it seems likely that complete proof of the assumptions that support the selection of a finite test set remains infeasible. In the light of this, both proof and testing can only hope to increase our confidence in correctness, but never demonstrate it absolutely. Note that the idea of a test hypothesis is closely related to that of a fault domain [ITU-T 1997]: a set $\mathcal{F}$ of models such that it is believed that the SUT is functionally equivalent to some (unknown) element of $\mathcal{F}$.

3.2.1 *Description of Framework.* This section describes the framework under the simplifying assumption that programs are deterministic; it is straightforward to generalize this to a nondeterministic SUT. Programs can only be proven correct, or demonstrated to be correct by testing, with respect to a specification. We start with a characterization of programs and specifications. For simplicity, it is assumed that a program, P, is a partial function from concrete input values, I, to concrete output values, O, and that P has no state:

$$P \in I \nrightarrow O$$

It is assumed that P terminates on all input values in its domain, $dom(P)$. Write $P(t)$ to represent the result of executing P on input value $t \in dom(P)$. The value t can be viewed as a test of program P.

A specification, S, is a relationship between abstract input values, D, and abstract output values, R:

$$S \in D \leftrightarrow R$$

Because it is a relation rather than a function, a specification may define several possible output values for a given input value. For a given input, $d \in dom(S)$, the set of permissible results is the image of the set $\{d\}$ in S, written $S(|\ \{d\}\ |)$.

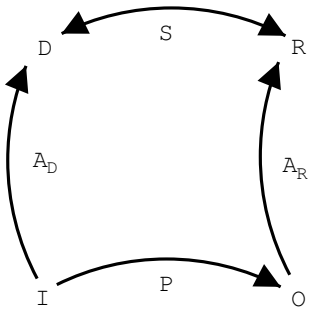

Fig. 8. Program correctness with respect to a specification

To compare the results of a program, $P \in I \nrightarrow O$, against its specification, $S \in D \leftrightarrow R$, a relation between abstract and concrete values is needed. For this, the following abstraction functions are used:

$$A_D \in I \twoheadrightarrow D$$

$$A_R \in O \twoheadrightarrow R$$

Every abstract value is required to have at least one concrete representation, making A_D and A_R surjections, i.e. every element of D is in the range of A_D, and likewise for R and A_R. The existence of such functions, of course, is not guaranteed in general, but it is assumed that such a specification context does exist, and this assumption is referred to as H_{MIN}, the minimum hypothesis.

The essentials are now in place to talk about program correctness. First, the program is required to be defined where the specification is defined. So all abstract input values in $dom(S)$, must have corresponding concrete representations for which the program executes. In other words, the inverse image of $dom(S)$ must be contained in $dom(P)$.

$$A_D^{-1}(\!|\, dom(S) \,|\!) \subseteq dom(P)$$

Secondly, the result of running the program on any test $t \in dom(P)$ must be correct with respect to the specification. Of course, some behaviours of P may be outside the scope of the specification, in which case any behaviour is acceptable:

$$\forall t \in dom(P) \bullet A_D(t) \in dom(S) \Rightarrow A_R(P(t)) \in S(\!|\, \{A_D(t)\} \,|\!)$$

For shorthand, if $A_D(t) \in dom(S) \Rightarrow A_R(P(t)) \in S(\!|\, \{A_D(t)\} \,|\!)$ holds for a given P, S and t, then $CORRECT(P,S,t)$ holds, or in the case of a set of tests T,

$$CORRECT(P,S,T) \Leftrightarrow \forall t \in T \bullet CORRECT(P,S,t)$$

Solving $CORRECT(P,S,t)$ is known as the "Oracle" problem (which in general is uncomputable).

This structure amounts to the commuting diagram shown in Figure 8. The sense in which this diagram commutes is that the function resulting from the composition of P and A_R is contained in the relations resulting from the composition of A_D and S. This can be more formally expressed in the following way.

P; $A_R \subseteq A_D$; S

Complete proof of program correctness amounts to proving this relationship. The other extreme is exhaustive testing: running the program on every possible input, usually an infinite set, and checking each result. This exhaustive test set is called T_{MAX} and must contain at least $A_D^{-1}(|\ dom(S)\ |)$.

Now the framework in Figure 7 can be slightly adapted. At the top, the program P is correct with respect to its specification S if the minimum hypothesis H_{MIN} can be proven, and if every test in the infinite test set T_{MAX} gives a correct result. At the bottom, P is correct with respect to S with no testing necessary if: $H_{MIN} \wedge P$; $A_R \subseteq A_D$; S can be proven. In the middle of the spectrum, there are other choices of H and T, which correspond to selections of tests and the test selection hypotheses that accompany them.

3.2.2 *Test Selection.* Designing a test strategy for a program involves selecting a finite test set that is considered to be adequate, in some sense. Usually, the sense of adequacy is expressed in terms of covering certain paths in the program, or testing every subdomain from a partition of the input data. There is an implicit belief in the uniformity of the data, or in the regularity of the program structure, or both, which leads to the selection of a single test case as a representative of a whole class of test cases.

The framework serves to emphasize the importance of defining, for every selected test set T, the set of hypotheses H that express the adequacy of the set T. This is one dimension of the relationship between testing and proof: it is necessary both to show the success of the tests in T, and to prove the selection hypotheses in H.

Gaudel defines two kinds of selection hypothesis:

—Uniformity, which plays on the homogeny of a range of data. Here, if one test t is passed in a uniform set T, then the hypothesis is that every test in T will be passed:
For a given P, S and T,

$$\forall t \in T \bullet (CORRECT(P,S,t) \Rightarrow CORRECT(P,S,T))$$

—Regularity, which plays on the structure, or size, of data. In this case, if all tests on data up to a certain size limit l are correct, then the hypothesis is that every test on larger data will be correct:
For a given P, S and T, and some notion, *size*, of size let $T' = \{t \in T \bullet size(t) < l\}$ in $CORRECT(P,S,T') \Rightarrow CORRECT(P,S,T)$

3.2.3 *Examples of Test Selection Hypotheses.* Consider a program `sqrt` to calculate the positive integer square root of an integer. Its specification S requires a single integer parameter, and specifies the result as the highest integer whose square is less than or equal to the input. Functional testing of this program will typically focus on boundary analysis of the single input. Data may be partitioned into three equivalence class:

$$\{\{0\}, 1..maxint-1, \{maxint\}\}$$

The belief that it is sufficient to test a single point of data from the class $1..maxint-1$ gives rise to the following uniformity hypothesis:

$$(t_1 \in \{0\} \wedge t_2 \in 1..maxint-1 \wedge t_3 \in \{maxint\} \wedge$$
$$CORRECT(sqrt,S,\{t_1,t_2,t_3\})) \Rightarrow CORRECT(sqrt,S,0..maxint)$$

Consider a bounded stack, instantiated with a size n and the type of data to be processed. Operations on the stack change its state, treated here by considering the state as an input and output to each operation.

A uniformity hypothesis may be used to select only a finite number of data values to feed into the stack, and a regularity hypothesis may be used to limit the size of stack to be tested, e.g. if it works for stacks of size less than 3, then it will work for all sizes of data.

Some other kind of hypothesis may be used to reflect that the program is expected to behave the same regardless of the type of data it is instantiated to contain: if it works for integers, then it will work for all types.

3.2.4 *Test Selection Refinement.* Gaudel also introduces the idea of moving up and down the test selection spectrum by means of a form of refinement. Consider two hypothesis/test set pairs (H_1, T_1) and (H_2, T_2). Intuitively, (H_1, T_1) refines (H_2, T_2) if

—the hypotheses get stronger, i.e. $H_1 \Rightarrow H_2$, and

—the ability of T_1 to detect defects is at least as strong as T_2 where H_1 is known to hold.

The simplest way of viewing this is that the refined test set is smaller, $T_1 \subseteq T_2$, but this is not necessarily the case; it might be a different set altogether, based on stronger hypotheses. More formally, (H_1, T_1) refines (H_2, T_2) if and only if $H_1 \Rightarrow H_2 \wedge (H_1 \wedge CORRECT(P, S, T_1) \Rightarrow CORRECT(P, S, T_2))$

Using this idea, the framework suggests a systematic step-wise approach to test set development, in which the infinite set of tests is gradually refined into a workable set, while the corresponding set of test selection hypotheses is collected.

3.3 Summary

Gaudel's approach to the formalization of testing has been presented as a framework for understanding the relationship between testing and proof. In essence, every test is the base case of an inductive proof, whose inductive step forms a test selection hypothesis remaining to be proved.

It turns out that these test selection hypotheses are difficult to discharge. However, a consequence of this framework for testers is that test selection hypotheses should at least be stated for every test set. Merely making an explicit statement of these selection hypotheses will engender greater understanding of the testing process.

Further investigation of this framework could lead to an understanding of how carrying out partial proofs of correctness could mitigate the need for some testing: "If I have proved part of my code to be correct, what remains for me to test?" or "Having carried some testing, what remains for me to prove?".

4. TESTING FROM MODEL-BASED FORMAL SPECIFICATIONS

4.1 Introduction

Model-based formal specification languages, such as the Z notation, the B-method, and VDM, allow the description of a computer-based digital artefact in terms of an abstract state and operations on that state. The states of the system are typically described using sets, sequences, relations and functions. The operations are described by predicates given in terms of pre and post conditions. This approach was described in Section 2 and here we will use the specification of a bounded stack given there.

4.2 Test Selection Strategies

Test selection criteria based on model-based specifications have been considered in the literature especially in the context of the Z notation (for example, [Ammann and Offutt 1994; Carrington and Stocks 1994; Hall 1988; Stocks and Carrington 1996]); however, the use of the B-Method [Legeard et al. 2002b], VDM [Dick and Faivre 1993] and combinations of methods [Hierons et al. 2001] have also been considered.

4.2.1 *Partitioning based methods.* Partition testing is a standard strategy in which the input space is partitioned into sub-domains, and test cases are drawn from each sub-domain. Most test generation techniques for model-based specifications are based on the principle of partitioning the input domain into equivalence classes on the basis of the specification [Amla and Ammann 1992; Dick and Faivre 1993; Hall 1988; Hierons 1997b]. These equivalence classes are subdomains of the input space upon which it is assumed that the behaviour of the SUT is uniform and thus the techniques relate to Gaudel's *uniformity hypothesis*. A small sample of data is then selected from each class: if the uniformity hypothesis holds then it is sufficient to choose one datum in each class.

Hall [Hall 1988] gives an approach for testing from a Z specification by classifying the test domains. The idea is based on considering various combinations after partitioning of the input and output sets (domains) and states into subsets (subdomains), which typically appear in the declaration part of a specification, and conditions contained in the predicates of the specification. This approach is structured, but less rigorous as simple partitions of the input space are constructed by examining the obvious divisions of the inputs.

Dick and Faivre [Dick and Faivre 1993] demonstrate that, by rewriting the precondition and postcondition of a VDM specification to Disjunctive Normal Form (DNF), much of the partitioning process can be automated. They also describe a technique for extracting a finite automaton (FA) from the specification; the FA can be used to drive test execution. The four steps in partition analysis are:

—Extraction of definitions by collecting all parts (precondition, postcondition, invariants);
—Unfolding of all definitions (in case of recursive definitions the unfolding is limited to some predefined limited number);
—Transformation of the definition to DNF to get the disjoint sub-domains; and
—Further simplification of each sub-domain.

Consider for example the *Pop* operation on the *Stack* (Section 2). Applying the steps of partition analysis would yield the following predicate which describes a single test domain:

$$\exists\, items, items' : \operatorname{seq} Object;\ x! : Object \bullet$$
$$\#items \leq maxSize \land items \neq \langle\rangle \land x! = head\, items \land items' = tail\, items \tag{1}$$

The approach of Hörcher [Hörcher and Peleska 1995] applies to the selection of test cases and the evaluation of test results. For test case selection, each operation is decomposed into a collection of sub-cases defining a set of equivalence classes. With this end in mind, the operation's predicate part is transformed into DNF. Each Z schema describes an individual test class. This approach is similar to that described by Dick and Faivre [Dick and Faivre 1993] for VDM specifications.

Amla and Ammann [Amla and Ammann 1992] applied the category-partition method [Ostrand and Balcer 1988] to Z specifications. The formal test specifications, in [Amla and

Ammann 1992], are written in a language called TSL (Test Specification Language). A TSL specification is divided into parameter, environment variable, and result sections. The TSL generator produces test cases based on the combinations of input and environment conditions specified in the result sections. Each specified combination of choices results in a test frame. Ammann and Offutt [Ammann and Offutt 1994] provide a method for specifying the combinations of choices to be tested: the tester can then produce test cases by (potentially automatically) generating actual data values for each choice.

The combination of category partition testing with the DNF approach [Dick and Faivre 1993] has been used for generating test cases in [Singh et al. 1997]. This is done in two steps. In the first step, a classification-tree describing high-level test cases is constructed from the formal specification of the test object. The second step involves further refinement of high-level test cases by generating a DNF for them.

Domain propagation [Carrington and Stocks 1994] is a partition analysis strategy devised in conjunction with the development of the Test Template Framework (cf. Section 4.2.2). The domain propagation strategy uses a library of input domains for each operator. For example, for binary set operations like $A \cup B$ or $A \cap B$, the suggested input domains include the combinations of possibilities for sets A and B where either of them is empty, one is a subset of the other, they are joint and disjoint. In the *Push* operation schema, for the sequence concatenation operator, the two cases, $items = \langle\rangle$ and $items \neq \langle\rangle$, will be considered.

To apply domain propagation to an operator, the disjunction of its input domains is conjoined to the predicate containing the operator. Then the whole schema is transformed to DNF, and the false branches are removed. The effect is to propagate the interesting sub-domains of an operator up to the top level of the operation.

In addition to partitioning of the input domain, other methods to derive tests from model-based specifications have been studied. These are described in the following section.

4.2.2 *Other methods and formal test frameworks.* In practice, the abstract specifications are often refined in multiple steps to obtain the final implementation (executable code). Every refinement step provides more concrete information about the system by reducing the non-determinism and possibly enlarging the domain. The relationship between testing and refinement has been explored by Derrick and Boiten [Derrick and Boiten 1999] in the framework of Z specifications. It describes how the FA used to drive test execution changes upon refinement by providing a method to calculate an FA for a refinement from an abstract FA.

Another strategy given in [Carrington and Stocks 1994] is specification mutation. This idea is inspired by mutation testing of programs. The specification is mutated, and for each mutation a test is derived that distinguishes the behaviours of the mutated and original specifications. The effect is to ensure that the SUT does not implement any of the incorrect specifications. The mutations are chosen in order to simulate real faults and thus the belief is that a test set that shows that the mutants are faulty will not be passed by a faulty program. It is natural to capture this belief by a test hypothesis, but it should be noted that this test hypothesis is not one of those considered by Gaudel. The approach can also suffer from one of the standard problems with mutation testing: there can be mutants that are equivalent to the specification and equivalence is undecidable.

The *Test Template Framework (TTF)* [Carrington and Stocks 1994; Stocks 1993; Stocks and Carrington 1996], uses the Z notation to conduct black box testing. A test template

is a Z schema that describes a set of test cases. A test template represents a set of values that are equivalent according to the test techniques used to generate the template, and can be refined by incorporating different test techniques. Test cases are generated from the final test templates, which partition the input space. The test template framework therefore gives a structure within which test case generation algorithms can be applied.

Authors have considered the problem of testing from an object-oriented model-based language such as ZEST [Stepney 1995] and Object-X [Cusack and Wezeman 1993; Fletcher and Sajeev 1996]. There has also been work on testing from SOFL (Structured Object-based Formal Language) specifications [Liu 1999; Liu et al. 2000; Offutt and Liu 1999].

Another direction for the use of formal methods to aid testing is the formalization of software testing criteria. A unification of two categories of test criteria, using program-based and specification-based approaches, has been presented by Behnia and Waeselynck [Behnia and Waeselynck 1999] for B models. Vilkomir and Bowen have formalized control-flow testing criteria using the Z notation [Vilkomir and Bowen 2001; 2002]. In [Vilkomir and Bowen 2001], Z schemas are presented for definitions of some of the main control-flow criteria. In [Vilkomir and Bowen 2002], a new Reinforced Condition/Decision Coverage criterion is proposed and formalized using Z. These formal definitions help to eliminate the possibility of ambiguities.

Rice and Seidman [Rice and Seidman 1998] consider software analysis and testing at the architectural level. They suggest using the architectural style description language ASDL for this aim. An ASDL description is made up of three elements: templates, settings, and units. A generic Z schema that is invariant across all styles describes each element. The authors consider the extension of a prototype ASDL toolset to include facilities for analysis and testing.

Proposed approaches for specific tasks during testing include also testing the non-existence of initial state in Z specifications [Miao et al. 1999], comparisons of the Z proof with various types of testing [King et al. 2000], using dynamic specification slicing in validating and debugging the Z specification [Chang and Richardson 1994], and testing documents during B method based software development [Taouil-Traverson and Vignes 1996].

4.3 Test Automation

There is the potential to automate many of the test generation strategies described in the previous sections. However, automation has its own problems and challenges. Automatic generation of tests from formal specifications has been considered in [Ammann and Offutt 1994; Burton 2002; Hayes 1986; Jackson and Vaziri 2000; Meudec 1997]. Different heuristics are typically used to partition the input domain into equivalence classes and to select test data from these classes.

4.3.1 *Automation using partitioning and finite state automaton.* An algorithm that generates a partition of the input domain from a Z specification was introduced by Hierons [Hierons 1997b]. It is shown that such a partition can be used both for generation of test cases and for the production of a FA model, which can be used to control the testing process. The specifications are rewritten to a form in which both a partition of the input domain and the states of a FA can be derived. The described rewrite rules are shown to be confluent and terminating.

In [Legeard et al. 2002a], an automatic test generation method for boundary testing from Z or B specifications is presented. The method avoids the construction of the complete FA

thereby avoiding the state explosion and non-discovery (determining all states and transitions) problems. Firstly, subsets of the state space (boundary goals) are computed from the DNF. The boundary states are obtained by traversal of the states specified by the boundary goals. The method is supported by a tool-set: the BZ-Testing-Tool environment [Ambert et al. 2002].

Burton [Burton 2002] describes a technique for automatically generating tests from Z specifications based on user defined testing criteria. The method is applicable to both partitioning and fault-based testing criteria. Automation is made possible by formally specifying heuristics for generating tests using general-purpose theorem proving tools. The condition under which a fault is detected is also formally specified in [Burton 2002]. The techniques have been implemented in the CADi$\mathbb{Z}$ general purpose theorem prover. This approach has also be applied to Statechart specifications by first generating Z specifications from them [Burton 2002].

A similar approach for generating test cases from B specifications, based on user-defined criteria, is presented in [Aertryck et al. 1997]. The authors describe a general framework, CASTING, that has been developed to support test generation.

Meudec [Meudec 1997] applied the idea of partitioning and TTF domain propagation strategy to VDM-SL expressions, and extended it to work better with quantifiers. His thesis discusses possible automation, and concludes that the complete simplification and VDM automatic proving support that is needed may be beyond the state of the art, but that constraint logic programming looks promising. Atterer [Atterer 2000] presents automatic test data generation, based on Meudec's technique, from VDM-SL specifications.

In order to test an operation with a value from a subdomain we may need to set up the internal state since each element of a subdomain defines a value for both the state and the input to the operation. This introduces a sequencing issue: we may need to apply a sequence of operations in order to set up the state for a test. The typical solution to this problem is to generate a FA from the specification and use this as the basis for sequencing [Dick and Faivre 1993; Hierons et al. 2001; Murray et al. 1998; Watanabe and Sakamura 1996]. Naturally, if we choose a path through this FA then there is no guarantee that this path is feasible.

4.3.2 *The oracle problem.* A specification can form the basis for generating an oracle [Aichernig 1999; Hayes 1986; Richardson et al. 1992; Waeselynck and Behnia 1998]. Where it is possible to animate a specification this animation can be used as an oracle [Mikk 1995; Ciancarini et al. 1996; Al-Amayreh and Zin 1999; Aichernig 2001a; 2001b]. Often the types used in the specification and implementation are different, in which case we need abstraction functions but if development has proceeded through formal refinement then such functions will have been produced.

Treharne et. al. [Treharne et al. 1998] consider an alternative approach in which a prototype implementation is used. In their approach the results of test cases are evaluated by executing the prototype specified in B and these test cases are further refined in order to make them suitable for the actual SUT.

4.3.3 *Other methods.* Jackson [Jackson and Vaziri 2000] describes a method for finding bugs in the code based on specified properties. The method generates counterexamples for the executions that violate the specification. The specification language, Alloy, is used for specifying the properties of code. Alloy is influenced by Z and is strictly first order,

making automatic analysis possible. There are two steps involved in checking properties. First, the program statements are encoded into relational constraints over before and after states. Second, conjunction of encoded formulae along with the negation of a specification is constructed. The satisfying assignments for this formula give executions that violate the specifications. SAT solvers are used for checking these formulae.

4.4 Case Studies and Tools

Approaches and techniques that use model-based formal specifications to assist software testing have been applied for testing diverse safety-critical computer systems.

4.4.1 *MATIS.* The Multi-modal Airline Travel Information System MATIS is an interactive system, which allows a user to query a database about flight information. Various search templates allow input from modalities such as speech, keyboard, and mouse. Considering MATIS as an example, MacColl and Carrington [MacColl and Carrington 1998] demonstrate how testing information for interactive systems can be derived from formal specifications. The specification of MATIS uses Z to define the functionality and presentation, and CSP to define interaction.

For the functionality and presentation, the Test Template Framework (TTF) [Stocks and Carrington 1996] was used to derive a hierarchy of test information. As particular testing strategies, type-based selection, partition analysis and domain propagation were identified. The type-based approach was used for sets and here two cases were used: the set is empty or it is non-empty. Naturally, all of these can be seen as applying some uniformity hypothesis.

A test template hierarchy was produced for the following main operations of MATIS: creating queries, providing a value for a named field and searching the database. It was found that the functionality and presentation views led to different but overlapping sets of test templates. The tests produced from the Z specifications were complemented by test cases produced from the CSP, essentially by devising an equivalent finite state automaton M and producing a set of paths of M that included all transitions of M. One of the main conclusions was that the different viewpoints were useful for test generation and led to different specification based test cases. However, while this piece of work showed how specification based tests can be produced the test effectiveness was not evaluated.

4.4.2 *VCS.* The Voice Communication system VCS 3020S [Aichernig 2000; Hörl and Aichernig 2000] supports voice communication between pilots and air traffic control personnel via radio. VCS is a switched system for real-time transmission of continuous voice data. VDM++ was used for formalizing the main part (the safety-critical core) of the functional requirements. This formalization was carried out in parallel with the design process and aimed to validate the functional requirements and the existing acceptance tests.

The formalization process was based on 140 natural language requirements. The process of producing a formal specification led to 108 questions being raised and this resulted in 33 changes to the natural language requirement [Aichernig 2000]. It was stated that this phase resembled a validation of the requirements.

A set of 65 test cases already existed, with a combined length of 200 steps. A total of 60% of these were formalized using VDM++. Each action by the test personnel and the corresponding system's reaction were translated into calls to methods of VDM++'s test specification. This formalization process led to 16 errors being found in the test cases and thus being corrected. The specification was animated using the test cases and the IFAD

VDM++ Toolbox (version 6.0). Animation led to seven minor errors being found in the formal specification and thus assisted in validating the specification.

The IFAD VDM++ Toolbox was used to obtain test coverage information. It was reported that the test cases gave quite a good coverage of the formal specification. It was argued that this was both because the informal test generation process had been very thorough and also because the coverage criterion used was 'rather weak' and thus a good coverage might mean relatively little [Aichernig 2000]. However, it was also observed that the thorough manual test generation process, for a safety-critical system, failed to produce a test set that covered the entire formal specification [Hörl and Aichernig 2000]. It was also reported that by considering the specification it was possible to produce 'more economic' test cases each of which covered more system functionality.

4.4.3 *FlowBus.* FlowBus is an application integration product, or middleware, for communication in a number of ways via a single application programming interface [Bicarregui et al. 1997]. Its main function is to provide a distributed, multi-platform, inter-application message handling service. One part of this, the queue administration tool (QAT) was developed formally using VDM and B. VDM and B were chosen because of the tool support that was available. The development process included three main steps:

—producing the initial VDM specification using VDM through Pictures (VtP);

—translate VDM specification by hand into B; and

—automatic generation of C source code from the B.

Test cases were generated from the VDM specification, which assisted the validation of the manual translation of the VDM into B. The VDM Analysis Tool (VAT) was used for partition analysis of operations and the system state using the DNF method. About 150 test domains were devised and about 500 test cases were produced for 34 operations. Three faults in the original VDM were discovered. The test cases were also used to check the B designs through animation using the B-Toolkit. This process was applied to 15 of the 34 operations and led to 5 faults being found, including one that was said to be potentially serious.

4.4.4 *GSM 11.11.* GSM is a widely used standard for digital mobile phone systems. In [Legeard et al. 2002b], the authors consider the B specifications of a simplified version of the GSM subset. For test generation, the TTF method was used. For this, the B specification was translated into Z, and then the TTF method was applied to generate tests from the Z specification. The main approach used was the DNF method. Although some tools were used (mainly, Z/EVES), TTF generation is a manual process, requiring extensive expertise. The Z schemas as well as the B specification of a fragment of the GSM 11-11 Standard are provided in [Legeard et al. 2002b].

Once the TTF leaves have been produced it is necessary to produce test sequences that 'cover' the leaves. The authors make the observation that it is often undesirable to instantiate the values of variables in the TTF leaves before test sequence generation since this makes it more likely that an infeasible test case is defined: there is no feasible path that includes the test case/leaf.

In parallel, the BTT method was used to generate tests from the B specification and the two test sets were compared. BTT generated over 200 tests and 14 tests were generated via TTF. However, it was reported that the BTT test set contained a lot of repetition and

potentially could be made more efficient. Based on the GSM case study, Legeard, Peureux and Utting noted the following distinctions between BTT and TTF:

—Typically a lot more tests are generated by BTT than TTF;
—The BTT method maintains a clearer distinction between the state variables and input parameters than the TTF;
—BTT generates more accurate oracles than TTF;
—TTF gives a more detailed analysis of the interesting partitions of the complex operations or predicates.

The main conclusion of [Legeard et al. 2002b] is that BTT is better designed for automation than TTF. It is also stated that some human interaction is probably desirable since full automation can lead to an excessively large test set.

4.5 Summary

There has been much work on testing from a Z, B, or VDM specification. The test generation techniques are typically based on the uniformity hypothesis: they partition the input domain into a set of subdomains on which the behaviour of the specification is uniform. Test cases are then generated from each subdomain, and potentially around the boundaries of the subdomains. At present it appears that no other test hypotheses have been applied. However, a mutation based approach [Burton 2002; Carrington and Stocks 1994] has been used in which test data generation is driven by the search for test data that distinguishes between the specification and mutants that represent potential faults. This process can be seen in terms of one of two test hypotheses. One test hypothesis is that if the SUT is faulty then its behaviour is equivalent to that of one of the mutants. However, this does not correspond to the usual motivation for mutation. An alternative test hypothesis is that if a test set distinguishes between the specification and mutants then it distinguishes between the specification and any faulty SUT.

There have been several promising case studies in this area. The presence of a model-based specification clearly helps test automation by assisting in the development of an oracle. However, the expressive nature of model-based languages does raise issues in automated test generation since powerful analysis tools are required (typically theorem provers or constraint solvers) and many of the problems are generally undecidable.

Model-based languages are often used to specify state based systems. When applying a test case it may be necessary to find a sequence of operations to set up the internal state of the SUT before the test case can be applied. Most reported approaches involve deriving a finite automaton from the specification and generating the sequences from this but naturally a chosen path through the finite automaton may not be feasible. Interestingly, the literature appears to say very little about a related problem, that of checking that the state of the SUT is correct after a test. It may be possible to adapt approaches from testing from finite state specifications in which this is a major issue.

5. FINITE STATE-BASED TESTING

5.1 Introduction

Many systems have a finite state structure and FSMs are a natural way to describe such a structure. The problem of testing from an FSM has thus received much attention. Moore proposed the conceptual framework for FSM based testing in his seminal paper [Moore

1956] referring to a concept largely used by physicists, called *gedanken experiments*, and Hennie enunciated the fundamental principles of transition checking [Hennie 1964]. The work was initially mainly motivated by automata theory and sequential circuit testing but later was found to be relevant to the conformance testing of communication protocols [Lee and Yannakakis 1996]. More recently FSM based techniques have been applied in model-based testing [Grieskamp et al. 2002; Farchi et al. 2002].

FSMs define a relatively poor language in terms of expressiveness and abstraction techniques. However, the lack of expressiveness introduces some significant advantages when analyzing FSMs. Many problems that are uncomputable for more general languages are computable for FSMs and are often computable in low-order polynomial time. This facilitates automating test sequence generation.

Many approaches for testing from an FSM rely on either a test hypothesis [Gaudel 1995] or a fault model [ITU-T 1997]. In both cases a test set might have the property that, under the assumptions made, the test set is guaranteed to determine correctness. Usually the test hypothesis, or fault model, places a bound on the number of states of the SUT.

When testing based on an FSM F we typically assume that the SUT is equivalent to an unknown FSM I (the minimum hypothesis) and that a test set is supplied in order to check whether I conforms to F. If F is deterministic then I conforms to F if it is equivalent to F. If F is nondeterministic then there are alternative notions of I conforming to F including F and I being equivalent, I being a reduction of F, or F being quasi-equivalent[2] to I. FSM based testing methods are discussed in a number of textbooks [Gill 1962; Kohavi 1978] and summarized in some survey papers [Lee and Yannakakis 1996; von Bochmann and Petrenko 1994; Petrenko 2001; Sidhu and Leung 1989].

FSM based techniques are based on definitions of conformance that are described in terms of the language defined by the corresponding FA. They thus only consider the traces observed; more general conformance relations exist when testing from a Process Algebra specification and these are discussed in Section 6.

A simple example from [Fujiwara et al. 1991] is shown in Figure 9. Here $F = (S, X, Y, h, s_0)$ is an FSM specification in which $S = \{s_0, s_1, s_2\}$ is the set of states, $X = \{a, b, c\}$ is the set of input symbols, $Y = \{e, f\}$ is the set of output symbols, and the behaviour function is $h(s_0, a) = \{(s_1, e)\}$, $h(s_0, b) = \{(s_1, f)\}$, $h(s_0, c) = \{(s_2, e)\}$, $h(s_1, a) = \{(s_0, f)\}$, $h(s_1, b) = \{(s_2, f)\}$, $h(s_1, c) = \{(s_1, f)\}$, $h(s_2, a) = \{(s_2, f)\}$, $h(s_2, b) = \{(s_0, e)\}$, $h(s_2, c) = \{(s_0, e)\}$. It is clear that F is completely specified ($D_F = S \times X$) and deterministic.

5.2 Test criteria

Typical testing strategies aim to detect *output faults*, i.e. transitions producing the wrong output, and *state transfer faults*, i.e. transitions going to incorrect states of the SUT I. The test set, usually called a *test suite*, is built from the specification model F.

In testing we often wish to determine whether actions corrupt the internal state. In order to achieve this we need to apply further operations on the system that distinguish the expected internal state from (incorrect) alternatives. Two states $s, t \in S$ are *distinguished* by the input sequence x if $h_2(s, x) \cap h_2(t, x) = \emptyset$. In the example, b distinguishes s_2 from both s_0 and s_1. The notion of an input sequence distinguishing two states has been extended to the use of a set of input sequences or an adaptive experiment [Petrenko et al. 1996].

[2]Under quasi-equivalence, if a specification F is partially defined then we need only consider the behaviour of SUT I for input sequences on which the behaviour of F is defined [Petrenko 1991].

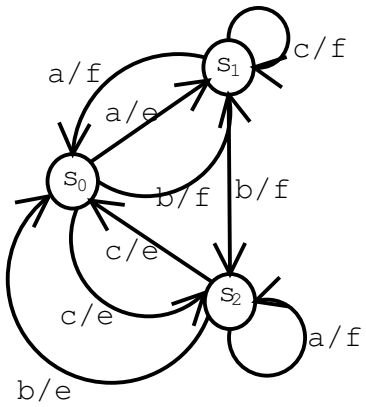

Fig. 9. The FSM F

In testing from a deterministic FSM (DFSM) we often use one of the following approaches in order to check the internal state.

(1) A set W of input sequences is called a *characterization set* if for every pair of distinct states $s,t \in S$ there is an input sequence $w \in W$ such that $\lambda(s,w) \neq \lambda(t,w)$. It is straightforward to show that $\{\langle a\rangle, \langle b\rangle\}$ is a characterization set for F.

(2) A *distinguishing* (D-) sequence of F is an input sequence D for which $\lambda(s,D) \neq \lambda(t,D)$, for every pair $s,t \in S$, $s \neq t$. The sequence $\langle a,a\rangle$ forms a D-sequence for F.

(3) A *Unique Input Output* (UIO-) sequence for a state $s \in S$ of FSM M is an input/output sequence x/y such that x/y can be observed from s and from no other state of M: $y = \lambda(s,x) \neq \lambda(t,x)$, for any $t \in S \setminus \{s\}$. Since $\langle a,a\rangle$ is a D-sequence for F the corresponding input/output sequences are UIOs for the states of F. For example, $\langle a/e, a/f\rangle$ is a UIO for s_0. However, there are shorter UIOs, such as the UIO $\langle b/e\rangle$ for s_2.

An FSM F is *minimal* if no FSM with fewer states is equivalent to F. Any minimal FSM has a characterization set. Not every minimal DFSM has a D-sequence or UIO-sequences. FSM F is *strongly connected* if for each ordered pair of states (s,t) there exists an input sequence that takes F from s to t.

If F is a DFSM then a set Q of input sequences is a *state cover* of F if Q contains the empty sequence and for each state $s \in S$ there is $x_s \in Q$ such that $s = \delta(s_0, x_s)$. A set P of input sequences is a *transition cover* of F if for each transition $\pi = (s, a/b, t)$ there are sequences $p, pa \in P$ such that $s = \delta(s_0, p)$. These notions may be extended to nondeterministic FSMs: rather than insist that every state is reached by an element of Q we instead insist that every deterministically-reachable state[3] is reached by a sequence in Q.

The *set of sequences of transitions* emerging from $s \in S$ is denoted by $Tr(s)$ and the input sequences by $Tr(s)^{in}$. A partial nondeterministic FSM is *reduced* if for any distinct states $s_i, s_j \in S$, $i \neq j$, it follows that $Tr(s_i) \neq Tr(s_j)$. For more details concerning these concepts see, for example, [von Bochmann and Petrenko 1994; Fujiwara et al. 1991; Gill 1962; Lee and Yannakakis 1996; Luo et al. 1994; Sabnani and Dahbura 1988].

[3] A state s is deterministically reachable if and only if there is some input sequence x such that the input of x when the FSM is in its initial state must lead to s being reached.

5.3 Test generation

5.3.1 *Completely specified, deterministic FSMs.* First we consider the case of *deterministic, minimal and completely specified* FSMs. Here the conformance relation between F and I is *equivalence* (F is equivalent to I if and only if they produce the same output sequence for every input sequence).

The *transition tour* (TT) method [Naito and Tsunoyama 1981] produces a test suite that executes all transitions of the specification. From the example in Figure 9 we might produce the following test sequence: $\langle a/e, c/f, a/f, b/f, b/f, a/f, c/e, c/e, b/e\rangle$. This method only attempts to find output faults and can be seen as being based on the test hypothesis that each transition of I has the expected final state. A variant of TT with even lower fault detection, called *modified T method* [Sato et al. 1989] covers all states, but not necessarily all transitions. Both methods make no attempt to find state transfer faults.

The *D-method* [Gonenc 1970; Hennie 1964; Kohavi 1978] relies on the existence of a distinguishing sequence D and is based on the test hypothesis that I contains no more states that F. The method checks whether all the states of the specification $s_i, 0 \leqslant i \leqslant n-1$, are correctly implemented by the SUT using a sequence $D\tau(t_0, s_1)D \ldots \tau(t_{n-1}, s_0)D$, where $t_i = \delta(s_i, D)$, $0 \leqslant i \leqslant n-1$ and for $s, t \in S$, $\tau(s,t)$ denotes an input sequence that takes the machine from state s to t. In order to check a transition $\pi = (s, a/b, t)$ when the machine is in state s' a sequence $\tau(s', s'')D\tau(t'', s)aD$ is used that transfers the machine to the intermediary state s'' that is verified and then $\tau(t'', s)$ takes the machine from t'' to s and finally with the input a the transition π is tested and its final state verified. The D-method checks all the transitions for both output and transfer faults. Efficient ways of minimizing the test suite have been proposed [Hierons and Ural 2006; Ural et al. 1997].

The transition checking approach introduced by Hennie [Hennie 1964] has been adapted using the *simple I/O sequences* [Hsieh 1971] which then has become known as *UIO sequences* [Sabnani and Dahbura 1988] and the method called the *UIO-method*. These methods assume that I has no more states than F and there is a reset that brings I from every state to the initial state. For every transition from s to s' accepting input a we use a test sequence $\tau(s_0, s)aUIO_{s'}$, where $UIO_{s'}$ is the UIO sequence for s'. Consider, for example, a test for the transition $t = (s_1, b/f, s_2)$ from the example. Since t has final state s_2, and this state has UIO $\langle b/e\rangle$, the test sequence $\langle a/e, b/f, b/e\rangle$ is produced. A test suite is obtained by concatenating, for each transition, the transition's test sequence prefixed with the reset transition [Sabnani and Dahbura 1988].

There are a number of problems associated with the use of a reset, including the fact that the reset may be difficult to realize [Broekman and Notenboom 2003; Hierons 2004a; Yao et al. 1993]. Another approach, which does not rely on reset transitions but assumes that I has no more states than F, builds the test suite as an optimal path that contains test segments for all transitions [Aho et al. 1988]. A test segment from a transition t leaving state s is a test sequence for t without the prefix that reaches state s. These test segments may be combined, possibly by including additional transitions, to produce a single test sequence. Further optimizations are obtained in the case of *overlapping test sequences* [Yang and Ural 1990; Miller and Paul 1993; Hierons 1997a].

The above versions of the UIO-method do not check that the UIOs identify the states of I. They can thus be seen as being based on the test hypotheses that I has no more states than F and that the UIOs used identify the states of I. If we instead use the test hypothesis that I has no more states than F then the resultant test does not guarantee to reveal all faults

[Chan et al. 1989]. Similar to the D-method an additional step can be included in order to verify the states when reset transitions are present [Chan et al. 1989](UIOv-method) or not [Yao et al. 1993] (UIOG-method). In each case the resultant test sequence is guaranteed to determine correctness as long as the test hypothesis holds. Even though both D-sequence and UIO-sequences do not have a polynomial upper bound [Lee and Yannakakis 1994], for many applications the FSMs used do have short UIO-sequences for all states [Sabnani and Dahbura 1988] and therefore the class of FSMs having UIO sequences has received much attention for test generation.

The D-method and UIO-method assume that the SUT has no more states than the specification FSM. Where this assumption might not hold, the *W-method* can be used. The W-method relies on a test suite Z provided by $Z = Q(\{\varepsilon\} \cup X \cup \ldots X^{m-n+1})W$, where Q is a state cover, ε is the empty string, m represents an upper bound on the number of states of the SUT, and W is a characterization set [Chow 1978; Vasilevskii 1973]. This method, that assumes that a reset transition is present and correctly implemented, has full fault coverage relative to the assumption that the number of states of the SUT does not exceed m. Consider the example, state cover $Q = \{\varepsilon, \langle a\rangle, \langle c\rangle\}$ and characterization set $W = \{\langle a\rangle, \langle b\rangle\}$. Assuming that the implementation has no more than 4 states, we get the following test suite: $\{\varepsilon, \langle a\rangle, \langle c\rangle\}(\{\varepsilon\} \cup \{\langle a\rangle, \langle b\rangle, \langle c\rangle\} \cup \{\langle a\rangle, \langle b\rangle, \langle c\rangle\}\{\langle a\rangle, \langle b\rangle, \langle c\rangle\})\{\langle a\rangle, \langle b\rangle\}$. Shorter test suites can often be obtained by using sets of prefixes of sequences from W for the identification of the final states of the transitions that do not appear in the state cover set (*Wp-method*) [Fujiwara et al. 1991].

The literature on testing from a DFSM discusses fault models rather than test hypotheses. However, these are similar concepts and almost all of the techniques for generating a test suite from a DFSM use a combination of some of the following test hypotheses.

(1) There are no state transfer faults.
(2) The sequences generated for checking the states of F are also effective in checking the states of I.
(3) I has no more states than F.
(4) I has no more than m states for a given integer m.
(5) There is a reset operation that is known to take I back to its initial state irrespective of the current state.

5.3.2 *Partial FSMs.* Many real specifications are partial: they do not cover all possible state-input combinations in the FSM specification [Sidhu and Leung 1989]. Consequently a *partially defined* FSM is used instead. Due to various interpretations of the missing transitions, called *undefined transitions*, *non-core*, or *don't care*, different semantics have been considered [von Bochmann and Petrenko 1994]. These transitions may be considered *implicitly defined* which means that either they are substituted by looping transitions with null outputs or with transitions going to an *error* state and yielding an *error* output. In this way a complete FSM specification is obtained and any of the testing strategies presented above may be applied. Another view on these transitions considers them to be *forbidden transitions*; these are never executed by the implementation. In this case the test suite must either avoid these transitions or check that the transitions cannot be executed.

5.3.3 *Nondeterministic FSMs.* An NFSM can be nondeterministic for one or more of the following reasons [Cavalli et al. 1996]. First, for a state s and input x there may be two or more transitions from s with input x where all of these transitions have the same output.

Second, for a state s and input x there may be two or more transitions from s with input x where these transitions have different output. If the only nondeterminism is of the first type then it can be transformed out. However, if the second type is present then the NFSM is not equivalent to any DFSM [Cavalli et al. 1996]. A third source of nondeterminism is internal transitions that have no input; if all the internal actions from a state have the same output then it is equivalent to the first case and otherwise it is equivalent to the second case [Cavalli et al. 1996].

Nondeterminism can arise in the specification for several reasons. First, it can represent nondeterminism that is required in the SUT I, in which case I conforms to F if it is equivalent to F. Alternatively, nondeterminism can represent options. In this case it is sufficient that all of the behaviours in I are contained in F; I is a reduction of F. More general notions of conformance have been considered for process algebra specifications and these are discussed in Section 6.

Nondeterminism introduces some additional issues: (a) observability of every possible response of the SUT associated with a particular input sequence and (b) the existence of non-equivalent states where for every input sequence there is some common output sequence that can be observed from both states with this input sequence [Luo et al. 1994]. In order to address the first problem a fairness assumption (test hypothesis) is often made, called the *complete testing assumption* [Luo et al. 1994], which says that there is some integer α such that if an input sequence x has been applied α times from some state $s \in S$ then it is guaranteed that every output sequence associated with s and x has been observed. Naturally the fairness assumption holds if the SUT is deterministic but it is not clear how it can otherwise be justified. The second problem led to the concept of an observable NFSM (ONFSM). An NFSM is *observable* if for any state $s \in S$, input $a \in X$, $(s,a) \in D_A$, and output $b \in Y$ there is at most one state $t \in S$ such that $(t,b) \in h(s,a)$. Every NFSM can be rewritten to an equivalent ONFSM but this rewriting can lead to a combinatorial explosion.

Various testing strategies have been devised for different types of FSMs representing the SUT I and utilizing different notions of conformance [Petrenko 2001]. When the specification is a minimal ONFSM, the implementation is an FSM with at most a given number of states and we are testing for equivalence, a test suite based on the Wp-method may be produced [Luo et al. 1994]. When the implementation is an FSM with a known upper bound on the number of its states, and the *reduction* relationship is used then a test suite can be built [Yevtushenko et al. 1991; Petrenko et al. 1994; Hierons 2004b]. This test suite is guaranteed to determine correctness under these test hypotheses. Note that where the SUT is known to be deterministic, the test suite may be further reduced [Petrenko et al. 1996].

5.3.4 *Communicating FSMs.* Many systems may be more naturally and simply modelled by a set of FSMs rather than a single FSM. The FSMs that may communicate and cooperate in this way are called *communicating FSMs (CFSMs)*. The communication is supported by queues and channels. A model consisting of CFSMs contains a number of CFSMs $F_1, \ldots, F_p$ [Luo et al. 1994] such that:

(1) each F_i consists of an FSM plus an input first-in and first-out (FIFO) queue where input symbols coming through the channels or from the environment are stored; F_i consumes inputs only from that queue;

(2) between each pair F_i, F_j there are two FIFO channels, each for one direction; and

(3) if F_i and F_j can communicate through a channel between them, then a symbol sent through this channel enters the input queue in the other machine.

A system of CFSMs can sometimes be transformed into an equivalent FSM by using exhaustive reachability analysis [Luo et al. 1994]. This is possible when all queues and channels have finite lengths. When inputs from the environment can be sent to the system only if all the queues and channels are empty (the *slow environment* assumption) [Luo et al. 1994], then the resulting FSM has up to $n_1 \ldots n_p$ states, where n_i is the number of states of F_i. In order to avoid the combinatorial state explosion some methods aim to test the transitions of the individual components [Hierons 2001]. The CFSM model is also used as a framework for testing an FSM in context [Petrenko 2001].

An alternative approach, which does not require the CFSMs to be expanded, is called the Hit-or-Jump algorithm [Cavalli et al. 1999]. This algorithm aims to produce a test sequence that executes every transition of an SUT A that is operating in a context defined by another machine C. This algorithm is iterative, terminating when the sequence contains every transition of A. At each stage the algorithm applies a search, with bounded depth, for a sequence that executes a transition of A that has not yet been covered. If such a sequence is found then this is a 'hit', the sequence is added to the current test sequence and the search repeats. If a sequence is not found then a randomly generated sequence, a 'jump' is added to the current sequence and the process is repeated.

5.3.5 *Extended FSMs.* Many finite state based specifications are actually *extended FSMs (EFSMs).* An EFSM is an FSM extended with input and output parameters, context variables, operations and predicates defined over context variables and input parameters [Petrenko et al. 1999]. It is possible to apply a uniformity hypothesis to the data and generate a set of test sequences from the FSM produced by abstracting out the data from the EFSM. This can be seen as testing the control structure of the SUT. However, the paths chosen need not be feasible in the EFSM. Recent work has shown how the feasible path problem can be overcome for EFSMs in which all operations and guards are linear [Uyar and Duale 1999; Duale and Uyar 2004].

If we do not apply this uniformity hypothesis then testing an EFSM means testing both components of it: the control portion and the data portion [Petrenko et al. 2004; Ural et al. 2000]. These two parts may be seen as being distinct elements that are independently tested: the FSM-based methods may be applied for the control part and data flow testing methods are used for the data part [Ural et al. 2000]. Alternatively, sometimes an EFSM can be transformed into an equivalent FSM [Petrenko et al. 1999]. These methods often run into the well known state explosion problem [Lee and Yannakakis 1996]; in some cases there are more effective methods [Petrenko et al. 1999]. In the context of X-machines, which are a way of describing EFSMs [Eilenberg 1974], a set of test hypotheses have been proposed. These test hypotheses, also called design for test conditions, significantly simplify the test generation problem and eliminate the feasible path and state explosion problems [Holcombe and Ipate 1998; Ipate and Holcombe 1997; Hierons and Harman 2000; 2004; Ipate and Holcombe 2000; Balanescu et al. 1999]. They eliminate the feasible path problem by requiring that all paths are feasible, potentially achieving this by adding unique inputs to trigger the transitions.

The techniques described above all aim to produce a test sequence that in some sense covers the specification EFSM. However, they can suffer from the state explosion problem and the feasible path problem unless strong restrictions are applied. An alternative

approach has been proposed in which a set of *test purposes* is produced and for each test purpose we generate a test case that satisfies the test purpose. A test purpose is a description of the requirement for a test, such as executing a particular transition or some sequence of transitions. A test purpose can be represented by a finite state automaton and a test case then automatically generated by applying reachability analysis to the product of the test purpose and the specification. While this reachability analysis can suffer from the state explosion problem, it has been found that tools such as TVEDA that apply an on-the-fly approach are often effective in practice [Kerbrat et al. 1999; Jard and Jéron 2005]. Typically the test purposes are provided by the tester but they can be generated automatically from an EFSM specification for a given test criterion such as executing all transitions [Kerbrat et al. 1999]. An alternative, more restricted, type of test purpose is a required sequence of interactions which can be described as a message sequence chart (MSC). There are tools such as AUTOLINK that apply reachability analysis to the product of the specification and an MSC in order to produce a test sequence [Schmitt et al. 1998].

Some attempts have been made to convert specifications made in the context of other formal models into FSM equivalent descriptions in order to use testing methods based on FSMs. This includes methods for converting variants of Z [Dick and Faivre 1993; Derrick and Boiten 1999; Hierons 1997b], Statecharts [Bogdanov 2000; Hierons et al. 2001], and SDL [von Bochmann et al. 1997] specifications into equivalent FSMs models.

5.4 Case studies and Tools

5.4.1 *Applying graph based methods to protocols.* Sidhu and Leung [Sidhu and Leung 1989] reported on experiments in which they applied four standard test techniques: the TT-method, the D-method, the UIO-method, and the W-method. They performed experiments using two FSMs, one of which was a small FSM they created while the second was a transport protocol. For each FSM they randomly generated a large number of faulty FSMs and evaluated the techniques by applying the corresponding test sequences to these faulty machines. They found that the test sequence generated using the TT-method missed some faulty machines while the test sequences generated using the other methods found all faulty machines. However, these experiments only considered two small FSMs: one with 5 states, the other with 15 states. The strength of this study is the large number of faulty machines generated: a total of 10 million for the FSM with 5 states and $100,000$ for the FSM with 15 states.

Test sequence generation, using UIOs, has been represented as a digraph optimization problem [Aho et al. 1988]. The authors report on their experience of using this approach in AT&T Bell Laboratories. They state that practical experience showed a reduction, of test sequence length, of at least a factor of three. However, they do not report details regarding the systems tested in this way or state what impact this had on test effectiveness.

5.4.2 *A protocol for military mobile radios.* Uyar et al. [Uyar et al. 2003] describe a recent application of UIOs and digraph optimization algorithms to a protocol for military mobile radios [(MIL-STD 188-220B) 1998]. The protocol was specified using in the order of 20,000 lines of the formal specification language Estelle [ISO 1989b] and was effectively in the form of a set of communicating EFSMs with timers.

Test sequences were automatically generated in the following manner. The EFSMs were rewritten (by a tool) into a form in which every path is feasible. While the problem of deciding whether a path is feasible is generally uncomputable, the authors restricted their

approach to cases where operations and guards are linear. The rewriting avoided the state explosion problem that occurs when EFSMs are rewritten to form FSMs. The digraph optimization algorithm of Aho et al. [Aho et al. 1988] was applied to the expanded EFSMs, using the rcpt tool, in the knowledge that the resultant paths were guaranteed to be feasible. This process resulted in in the order of 10,000 tests. The authors found that these tests gave in excess of a 200% increase in the number of transitions covered in testing. They also report that the tests are being applied in practice and have uncovered real faults.

5.4.3 *An Intelligent Network service.* The Hit-or-Jump method was evaluated on an Intelligent Network service [Cavalli et al. 1999]. This system was specified in about 3,000 lines of SDL. The authors report that the Hit-or-Jump approach implemented in the TESTGEN-SDL tool covered all of the transitions of the component of interest using a test sequence of length 150 while a random walk required a test sequence of length 1,402. The approach was not directly compared with those that produce the entire state space using reachability analysis. Instead, an attempt was made to generate this state space but this was terminated at a point when there were over two million transitions and over half a million states. The authors observed that by terminating at this point they had covered less that 50% of the transitions of the individual components.

5.4.4 *Pocket PC applications.* El-Far et al. [El-Far et al. 2001] describe the application of a model-based testing approach, based on FSMs, to several Pocket PC application supplied by Microsoft. In order to produce FSMs they had to apply some restrictions, such as putting an upper bound on the number of messages in an Inbox application. They report that when they applied this automated testing approach to a system near release they found some real faults. However, they do not say which test sequence generation techniques were applied, stating that this was confidential. They also made no attempt to compare the results with other techniques.

5.4.5 *AGEDIS case studies.* AGEDIS (Automated Generation and Execution of Test Suites in Distributed Component-based Software) was a three-year project that investigated the use of model-based testing. This used the specification language of the GOTCHA tool [Benjamin et al. 1999] which is based on finite automata (using an imperative programming language). GOTCHA generates test sequences to provide coverage of a specification's states and transitions. A related approach has been developed by researchers at Microsoft who produced a tool that generates an FSM from an Abstract State Machine Language (AsmL) specification and generates test sequences on the basis of this [Barnett et al. 2003].

Craggs et al. [Craggs et al. 2003] report on five case studies performed within the AGEDIS project; all used real industrial systems provided by partners. Two of the case studies were carried out in the early stages of AGEDIS and before any tools had been developed. The three other projects used AGEDIS tools. The AGEDIS tools use the AGEDIS Modelling Language (AML) which is similar to UML. Test generation used reachability analysis based on a model and a test purpose.

A number of interesting observations were made and we describe two of these here. When testing part of a Transit Computerization Project, it was found that the size of the SDL requirements specification made automated test sequence generation difficult. A model produced in the GOTCHA tool for the purpose of testing was much simpler: it has 57 states and 137 transitions in contrast to the $500,000$ states and $800,000$ transitions produced from the SDL specification. Thus, a model produced for the purpose of supporting

testing was found (in testing) to be more useful than the requirements specification. However, no results were published regarding the effectiveness of the test sequences. When considering a non-deterministic model, for a publish-subscribe system, it was found that it was helpful if the test suites were not preset: they were implemented using a tool that determines which input to apply next on the basis of the previous input/output values.

5.4.6 *A network controller: Media Oriented Systems Transport (MOST).* Pretschner et al. [Pretschner et al. 2005] investigated the effectiveness of model-based testing. They applied manual and automated test generation, in each case with or without a model. The system considered was a network controller, Media Oriented Systems Transport (MOST), for an infotainment network used within automotive industries [MOST Cooperation 2002]. This was modelled as a set of twelve communicating EFSMs. The authors state that when implemented this model corresponded to about 12,300 lines of C (not counting comments).

For model-based testing, the AUTOFOUS tool [Huber et al. 1997] was used in order to produce the model and constraint logic programming was used to generate test cases from the model and test specifications. The authors found that the process of producing a formal model identified problems with the requirements. The tests produced on the basis of a model were found to be more effective than those produced without a model: they found more errors. Interestingly, the key difference found was in the number of requirement errors found: the tests produced with or without a model found a similar number of faults caused by programming errors but the tests produced from a model found more faults caused by errors in the requirements. This was in addition to issues discovered in building the model. The tests produced from a model using automated and manual approached were equally effective when the test size was fixed. More errors were found when automated test generation was used to produce a larger test set (an extra 11% with a test set of six times the size). The results thus suggest that separate value is provided by both the use of a model as the basis for test generation and by larger test sets but not by automated test generation on its own.

5.4.7 *Generating tests from test purposes.* There are tools such as TVEDA that aim to produce test cases that satisfy given test purposes (see, for example, [Kerbrat et al. 1999]). Typically the test purposes are provided by the tester but it has been observed that these can be generated automatically [Kerbrat et al. 1999]). An alternative, more restricted, type of test purpose is a required sequence of interactions which can be described as a message sequence chart (MSC). There are tools such as AUTOLINK that automatically produce test cases from an EFSM model on the basis of an MSC (see, for example, [Schmitt et al. 1998]).

5.5 Summary

This section has reviewed the literature on testing software on the basis of an FSM model or specification. This is an area that has received much attention and that contains many relevant results. The interest has been motivated by the widespread use of state-based specifications and in particular their use in specifying communications protocols. More recently, FSMs have been used in model-based testing.

Most work in the area of testing from finite state-based models does not discuss test hypotheses but instead considers fault models. However, fault models are an extremely similar concept: like test hypotheses they postulate restrictions on the behaviour of the SUT and are used to drive test generation. In fact, some of the standard assumptions

made in FSM based testing are similar to the test hypotheses of Gaudel. For example, the assumption when testing from an FSM with n states that the SUT behaves like an unknown FSM with at most m states can be seen as a regularity hypothesis: if this holds and the SUT is faulty then there exists an input sequence of length at most $m+n-1$ that leads to a failure. When testing from an EFSM it is common to apply a uniformity hypothesis on the data.

The lack of expressiveness of FSMs but their suitability for automated test sequence generation has led to situations in which developers and testers write specification or models in more expressive languages and a tool converts these into FSMs that then act as the basis for test sequence generation. While this seems to provide the benefits of both expressive languages and automated test sequence generation, a number of challenges remain. For many systems we cannot simply produce an FSM by taking all combinations of values for internal variables and all states: this may lead to an infinite state space and even if the state space is finite it is likely to be extremely large. If we apply an abstraction then we have two issues to consider. First, this may abstract out information that could help us in testing. Second, the abstraction could lead to a model in which there are paths that are feasible and so could be chosen in test sequence generation, but do not correspond to feasible paths in the original model. There has been some work on rewriting an EFSM in order to eliminate this feasible path problem but this work either assumes that all operations and predicates are linear [Uyar and Duale 1999; Duale and Uyar 2004] or it places restrictions on the structure of the EFSM and potentially requires theorem proving or constraint satisfaction techniques [Hierons et al. 2004]. Naturally, the general problem is not computable.

New challenges are likely to result from current trends, towards systems that are highly distributed and possibly include mobile components. If we consider the state space of a system composed of a set of distributed components we get a combinatorial explosion. Thus, for such systems, our analysis is likely to be at least exponential in the worst case. If we wish to test such systems we need ways of overcoming this problem. One possibility might be to identify common classes of system for which the required analysis is tractable. Potentially we might also take advantage of approaches such as partial order reduction that are used in model checking to overcome such problems [Clarke et al. 1999]. Of course, in testing we have the additional problem that want these properties to hold in the SUT as well as our model.

6. TESTING FROM PROCESS ALGEBRA SPECIFICATIONS

6.1 Introduction

Process Algebras, such as LOTOS [ISO 1989a], CCS [Milner 1989] and CSP [Hoare 1985], provide an elegant formalism that focuses on the communication between entities. Since labelled transition systems (LTSs) are often used to describe the semantics of process algebras, this section discusses the problem of testing on the basis of labelled transition systems. Indeed, LTSs themselves form a suitable basis for modelling the behaviour of processes and components, particularly where issues of concurrency arise.

6.2 Test cases with state-based verdicts

In an LTS context, testing is seen as the interaction between an agent representing the SUT and an agent representing the test case. A test case is thus an LTS in which different sequences of interactions lead to different verdicts. There are two standard verdicts for

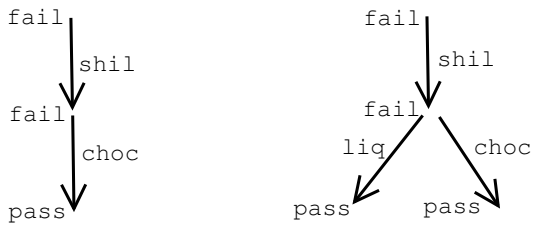

Fig. 10. Test cases t_1 and t_2

a test run: pass (the SUT has passed the test run) and fail (an erroneous behaviour has been observed and so the SUT is known to be faulty). The verdict inconclusive has also been considered [Brinksma 1988]: while the behaviour observed is consistent with the specification, the SUT is not deemed to have passed the test. Typically this corresponds to the test objective not being achieved due to non-determinism in the specification and possibly the SUT.

A test case is a labelled transition system with a mapping v from states to verdicts. Suppose test case t and the SUT p are executed together and they deadlock with t in state s. Then the result of this test run is the verdict $v(s)$. The SUT *fails* a test case if one or more test runs lead to the verdict fail. It *passes* the test if and only if *all* possible test runs lead to the verdict pass.

Consider the test cases t_1 and t_2 given in Figure 10. t_1 involves the action *shil* followed by the action *choc*. Here, the implementation passes the execution with t_1 if it does not deadlock until both actions have occurred; otherwise it fails. In contrast, the implementation only fails t_2 if it either fails to allow action *shil* or, having allowed *shil*, it does not allow either *liq* or *choc*. Thus, the agent p_2 shown in Figure 11 may fail t_1 but passes t_2.

Given a test case, there may be many possible test runs. For example, p_2 may either pass or fail a single use of t_1 since, after the action *shil*, p_2 either moves to a state that only allows *liq* (and then fails the test) or moves to a state that only allows *choc* (and then passes the test run). Thus, even if a test run returns the verdict pass, there is no guarantee that the test case will produce pass for all test runs. Thus, a test case may have to be executed many times in order to guarantee that the lack of an observed failure means that the SUT passes the test; possibly an infinite number of times. If some fairness assumption is made then we may feel that it is sufficient to apply a test case some predetermined number of times and this can be seen as a test hypothesis.

A set of test cases forms a test suite. The following are clearly desirable properties of a test suite.

DEFINITION 2. (*1*) *A test suite is* sound *if from an implementation failing some test case from this suite it is possible to deduce that the implementation is faulty.*

(*2*) *A test suite is* complete *if from an implementation passing every test case from this suite it is possible to deduce that the implementation is correct.*

There are a number of algorithms for defining sound and complete test suites [Brinksma et al. 1998; Leduc 1991; Phalippou 1994; Tretmans 1996; Wezeman 1989]. Typically these algorithms show how one can generate sound test cases and have the property that if the SUT passes all possible resultant test cases then it must be correct. Unfortunately, since

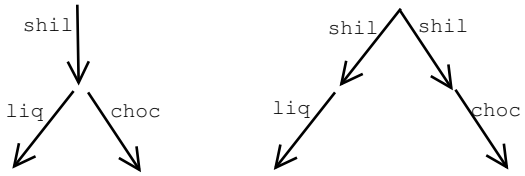

Fig. 11. Non-equivalent agents p_1 and p_2

fault models and test hypotheses are not used in order to restrict the possible behaviours of the SUT, the set of all possible test cases need not be finite. Naturally, different algorithms are required with different implementation relations. Some alternative implementation relations are given in Section 6.3 and some test generation algorithms are described in Section 6.4.

6.3 Implementation Relations

Implementation relations, conformance and testing are closely intertwined, since a test suite is used to determine whether an implementation conforms to a specification. Different implementation relations arise due to different testing scenarios, and in particular, the observations a test case can make of the SUT.

Different testing architectures gives rise to different interactions between the SUT p and the test case t. Initially we consider the simplest case of p and t communicating by synchronous interaction, i.e., p can perform an event a if and only if t performs the same event a. We denote this synchronous communication by $||$, thus we consider testing scenarios based upon the behaviour of $p \,||\, t$.

The observations made of p by t are related to the notion of implementation relations in the following manner: the observations of the implementation must be consistent with those of the specification. Thus, for an implementation I, specification S, if $runs(t,I)$ denotes the observations made by testing I with test case t, we can define an implementation relation $\leq_R$ by

$$I \leq_R S \text{ iff } \forall t \in LTS : runs(t,I) \subseteq runs(t,S)$$

Varying the power of the observations gives rise to different implementation relations. We now describe some of the implementation relations, though it should be noted that we do not cover all implementation relations discussed in the literature including some recent extensions with probabilities (see, for example, [López and Núñez 2004]).

6.3.1 *Trace preorder.* The simplest implementation relation is the trace preorder $\leq_{tr}$ that is similar to the notion of conformance used with FSMs.

DEFINITION 3. *If p and s are labelled transition systems with the same set of labels then $p \leq_{tr} s$ if and only if $traces(p) \subseteq traces(s)$.*

For example, it is straightforward to see that the agents in Figure 11 (a copy of Figure 3) are related by: $p_2 \leq_{tr} p_1$.

6.3.2 *Testing preorder.* A more discriminating implementation relation (and hence set of tests) is given by the testing (or failure) preorder, denoted $\leq_{te}$. This arises when we consider observations given by the following definition.

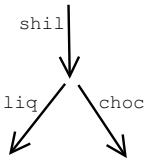

Fig. 12. A simple LTS

DEFINITION 4. *Suppose the test case t and the system p have the same label set L (i.e., same set of observable events). When testing p with t, the set $obs(t,p)$ denotes the traces that are capable of leading to deadlocks. That is, a trace σ is in $obs(t,p)$ precisely when $p \,||\, t$ can perform σ but after it has done this no action can be observed (i.e., it has deadlocked).*

In a similar fashion, the set of observations, $obs'(t,p)$, that t can make with respect to p denotes the traces that are capable of being performed. That is, a trace σ is in $obs'(t,p)$ precisely when $p \,||\, t$ can perform σ.

Thus the set $obs(t,p)$ denotes the traces that are capable of leading to deadlocks, while $obs'(t,p)$ denotes the traces that may be observed.

For example, consider the process p given in Figure 12 (a copy of Figure 2) and the test p_2 given in Figure 11. It is possible for $p \,||\, p_2$ to deadlock after either $\langle shil, liq\rangle$ or $\langle shil, choc\rangle$ and so $obs(p,p_2) = \{\langle shil, liq\rangle, \langle shil, choc\rangle\}$. Further, it is clear that $obs'(p,p_2) = \{\varepsilon, \langle shil\rangle, \langle shil, liq\rangle, \langle shil, choc\rangle\}$.

The testing preorder is then defined via these notions of observation.

DEFINITION 5. *Given systems p and s with label set L, $p \leq_{te} s$ if and only if for every test case t with label set L we have that $obs(t,p) \subseteq obs(t,s)$ and $obs'(t,p) \subseteq obs'(t,s)$.*

Although the testing preorder is in some senses a natural notion of testing and implementation for a concurrent system, it suffers from the disadvantage that it requires quantification over all possible traces in checking the test cases, as opposed to all traces from the specification.

6.3.3 *The conf relation.* The implementation relation *conf* was introduced by Brinksma to overcome this problem. The observations it relies on are similar to those made by the testing preorder. However, it only requires one to check for consistency of observations made with respect to what was in the original specification.

DEFINITION 6. *Given systems p and s with label set L, p conf s if and only if for every test case t with label set L we have that: $obs(t,p) \cap traces(s) \subseteq obs(t,s)$ and $obs'(t,p) \cap traces(s) \subseteq obs'(t,s)$.*

Both requirements consider traces from the specification only. The first states that whenever a test case t may deadlock, when interacting with the SUT p after trace $\sigma \in traces(s)$ then t may deadlock after trace σ when interacting with the specification s.

Testing using the *conf* relation concerns checking whether the SUT does not have unspecified deadlocks for traces in the specification. However, it does not check whether the SUT has extra traces, thus it can allow additional functionality in the SUT. This is not the case with the testing preorder: the testing preorder can thus be seen as the combination of conf with the trace preorder, i.e., $\leq_{te} = \leq_{tr} \cap$conf.

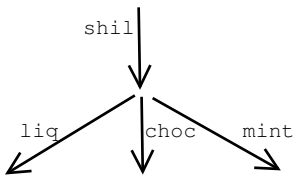

Fig. 13. Process p_3

To illustrate the *conf* relation, consider a test case that first applies *shil* and then applies *choc*. When combined with p_2 this may deadlock after one action since p_2 is capable of moving to a state from which it cannot perform *choc*. However, when combined with p_1, this test case cannot deadlock until both actions have occurred. Thus $\neg(p_2 \; conf \; p_1)$.

Now consider the process p_3 given in Figure 13. It is clear that p_3 conforms to p under *conf* since p_3 may be produced from p by simply adding a new action that is not mentioned in the trace of p. In contrast, we have that $p_3 \not\leq_{te} p$ since p_3 may perform a sequences of actions that p cannot: the trace $\langle shil, mint \rangle$.

6.3.4 *The ioco relation.* The implementation relations described so far all see communication as processes synchronizing on actions. However, processes may communicate through message exchange: a process sends a message to other processes. Importantly, testing usually involves communicating through messages being passed between a tester and the SUT. Communication between the tester and the environment is asymmetric: the SUT cannot block input and the environment cannot block output.

It is normal to call an LTS in which we differentiate between input and output an *Input Output Transition System (IOTS)*: this is an LTS in which the set L of labels is partitioned into the set L_I of input and the set L_O of outputs. It is usual for the name of an input to start with ? and the name of an output to start with ! and thus a process p sending the value a to process p' is represented through a transition in p with label $!a \in L_O$ and a transition in p' with label $?a \in L_I$. By definition an IOTS is input-enabled: for every state s and input $?a$ there must be a transition from s with label $?a$. The inability of a process to produce either an output or perform an internal action in state q is represented by adding a (*quiescence*) transition from q to q with label δ [Tretmans 1996]. Testing requires the ability to observe the outputs and thus it is necessary to be able to observe quiescence. While this is normally implemented using a timeout, in general we cannot observe quiescence and so the ability to observe this can be seen as a test hypothesis. An additional test hypothesis is that the specification and implementation are input-enabled, since it must be possible to describe then as IOTSs.

Given a process p and trace σ, p *after* σ represents the set of processes that can occur after σ. Given a set P of processes, $out(P)$ denotes the set of output events that processes in P can produce. Note that this can include quiescence. The *ioco* relation adapts *conf* in a manner that reflects this asymmetry between input and output [Brinksma et al. 1997].

DEFINITION 7. *Given systems p and s with label set L that has been partitioned into a set L_I of inputs and a set L_O of outputs, p ioco s if and only if for every trace $\sigma \in traces(s)$ of s we have that the following holds*

$$out(p \; after \; \sigma) \subseteq out(s \; after \; \sigma)$$

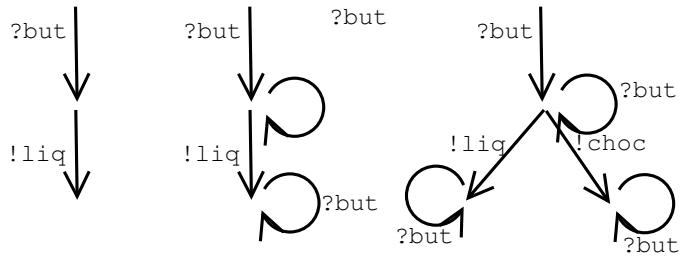

Fig. 14. Processes s, q_1, and q_2

The intuition behind this has two elements: if p can produce output $!x$ after $\sigma \in traces(s)$ then s must also be able to produce $!x$ after σ and if after σ the process p can be in a state from which it cannot produce output (quiescence) then so can s. Implementation choice is allowed where the response to an input in a state is not given in the specification IOTS.

Consider the processes s, q_1 and q_2 from Tretmans [Tretmans 1996] given in Figure 14 (quiescence is not shown). Then q_1 *ioco* s but not q_2 *ioco* s since after the button is pressed, $?but$, q_2 can produce output $!choc$ but s cannot.

6.4 Test Generation

The literature contains a range of test generation algorithms for the differing implementation relations. Here we sketch an overview of two such algorithms for generating a sound test suite: one for *conf* and one for *ioco*.

6.4.1 *Test generation for conf*. We now describe a recursive test generation algorithm for *conf* [Tretmans 1996] that at each stage is parameterized by a set A of actions. If s cannot perform any actions from A, except possibly the unobservable event τ, the test suite is the process *stop* that represents deadlock. Otherwise, for each action a from A, a test case t_a is produced. Let $A = \{a_1, \ldots, a_n\}$. Then the test suite generated is:

$$(a_1 \rightarrow t_{a_1}) \Box (a_2 \rightarrow t_{a_2}) \Box \ldots \Box (a_n \rightarrow t_{a_n})$$

Suppose that after a, s may be any one of $s_1, \ldots, s_m$. Then t_a is the test case for $(\tau \rightarrow s_1) \Box (\tau \rightarrow s_2) \Box \ldots \Box (\tau \rightarrow s_n)$.

It remains to say how the set A of actions for s may be determined. One possible choice is to have A containing at least one element from the set of actions of each process s' such that s may become s' through internal actions only. If all possible tests that may result from different choices for A are produced then the overall test suite is guaranteed to be complete [Tretmans 1996].

Now consider the process p_1 given in Figure 11 and the choice $A = \{shil\}$. We thus get a test $t_1 = shil \rightarrow t_1'$. After *shil* the process p_1 is a process that can either perform *choc* and then deadlock or perform *liq* and then deadlock: this is the process $choc \rightarrow stop \Box liq \rightarrow stop$. Thus t_1' is the test produced for the process $\tau \rightarrow (choc \rightarrow stop \Box liq \rightarrow stop)$. Thus t_1' is $choc \rightarrow t_1'' \Box liq \rightarrow t_1''$ where t_1'' is the test case for $\tau \rightarrow stop$. Thus $t_1' = choc \rightarrow stop \Box liq \rightarrow stop$. The overall test for p_1 is thus $shil \rightarrow (choc \rightarrow stop \Box liq \rightarrow stop)$.

If we apply the same process to p_2 the key difference is that there are two possible processes after *shil*. The final test t_2' produced from p_2 is $shil \rightarrow (\tau \rightarrow choc \rightarrow stop \Box \tau \rightarrow liq \rightarrow stop)$. Test cases t_1' and t_2', without verdicts, are shown in Figure 15.

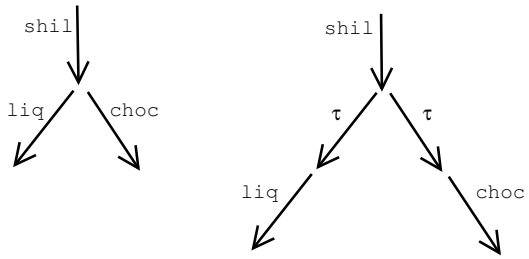

Fig. 15. Test cases t'_1 and t'_2

6.4.2 *Test generation for ioco.* We now describe a test generation algorithm given in [Brinksma et al. 1997] that takes as input a subset $\mathcal{F}$ of the set $traces(s)$ of s and produces a test case for the implementation relation *ioco*. The aim is to test the output of the SUT after traces in $\mathcal{F}$. The test generation algorithm is nondeterministic but each application of it produces a single test case which is deterministic: whenever an input is to be applied in this test case there is only one input that can be applied. Thus, each state of the test case is either a pass state, a fail state, has one transition leaving it with a label from L_I, or has one transition leaving it for each element of L_U plus a transition with label θ representing observing quiescence.

The recursive algorithm works in the following way. At each stage the algorithm nondeterministically chooses to either terminate, apply an input, or observe an output. If the choice made is to terminate then the verdict is pass and no more inputs or observations are required. The choice to apply an input can only be made if the set $\mathcal{F}$ contains a trace in which the next action after the current trace is an input, in which case such an input is applied. If the choice is to observe an output then the test case encapsulates three situations.

(1) If an output that is not specified is provided and the trace observed until this point is in $\mathcal{F}$ then return fail since this represents an erroneous behaviour after an element of $\mathcal{F}$.
(2) If an output that is not specified is provided and the trace observed until this point is not in $\mathcal{F}$ then return pass since we cannot observe an output after a trace in $\mathcal{F}$ by continuing to test from this point.
(3) If the output is consistent with the specification then testing continues.

The test case produced in the above manner only returns fail if it leads to an unspecified output after some trace in $\mathcal{F}$. If for some $\mathcal{F}$ the SUT passes all test cases that can be produced using this algorithm then the SUT must conform to s under the restriction $ioco_{\mathcal{F}}$ of *ioco* to $\mathcal{F}$ [Brinksma et al. 1997]. Naturally, we can set $\mathcal{F}$ to be the set of traces of s.

6.5 Case studies and tools

Some of the main case studies in this area were produced in the Cote de Resyste project in which the TorX tool was developed [Tretmans and Brinksma 2003]. TorX is a prototype test tool for the *ioco* implementation relation that operates in an on-the-fly manner. There are two modes: random selection in the automatic mode and interactively by the user in the manual mode. In automatic mode, at each step TorX determines the set of events (inputs and outputs) that should be possible and randomly selects one of these: it thus randomly chooses to either apply an input or observe an output. The TorX architecture includes a

module (explorer) that analyzes the specification and thus by introducing a new module it is possible to use TorX with a different specification language. Explorers have been developed for several languages including LOTOS and Promela, the language used by the SPIN model checker [Tretmans and Brinksma 2003].

The applicability of TorX was investigated through a small 'chatbox protocol', the Conference Protocol, that sits on top of UDP. The stated intention was that this would be relatively simple and yet also quite realistic. An implementation was developed and 27 mutants of this produced. Two of these mutants were intended to be equivalent under the implementation relation used (ioco) while the others were intended to be faulty. Testing using TorX found all faulty implementations using test runs of length at most 500. The tool was also applied to the mutants that were intended to be correct and test runs of length in excess of 450,000 were produced.

The project investigated the testing of part of a Payment Box that was intended to work within an automated Highway Toll system. This system was designed to bill the owners of vehicles that passed a toll gate and thus would have to deal with many transactions in parallel. Additional complications were provided through the use of encryption which meant that the tool could not interact directly with the SUT. The implementation studied had been tested in a traditional manner. The case study found two faults. One of these faults was found when using a model checker to validate the formal specification and the other was found through testing. The test runs applied were of length up to 50,000, again showing that long test runs can be automatically generated and applied. In addition, it was found that parallelism was not problematic. However, while two faults were discovered several issues were raised by the case study. In particular, the real-time nature of the SUT was found to be a problem since the test tool was not always able to provide an input in the required time, leading to time-outs.

A third case study was the Philips EasyLink Protocol for communications between a television set and a video recorder. Here TorX was used to test an implementation of the television side of the protocol. It was reported that some faults were detected even though the implementation tested had passed through conventional testing, but these faults were described as 'non-fatal'.

The model checker FDR [Formal Systems (Europe) Ltd 1997] for CSP offers support for implementation relations such as trace, failures and failures-divergences refinement. The Concurrency Workbench for the New Century (CWB-NC) [Cleaveland et al. 2000], allows the user to describe a concurrent system in a design language such as CCS and analyze the behaviour of the system with different methods such as equivalence checking, preorder checking, model checking and random simulation. Different input languages can be used in the CWB-NC via use of the Process Algebra Compiler (PAC) [Sims 1999].

In a similar vein is CADP ("Construction and Analysis of Distributed Processes", formerly known as "CAESAR/ ALDEBARAN Development Package"), which is a toolbox for protocol engineering [Garavel and Hermanns 2002]. It is dedicated to the efficient compilation, simulation, formal verification, and testing of descriptions written in LOTOS. It accepts three different input formalisms: LOTOS; low-level protocol descriptions specified as LTSs; and networks of communicating automata. In addition to support for equivalence checking, model checking, simulation checking etc, it has support for the testing process via TGV [Fernandez et al. 1996], a tool for the generation of conformance test suites based on verification technology. TGV takes as entries a description of a protocol's behaviour

and a test purpose, which selects the subset of the protocol's behaviour to be tested. It produces test suites, which are used to assess the conformance of a protocol implementation with respect to the formal specification of the protocol. The generation of test sequences in TGV is achieved using on-the-fly state-space exploration. Traditionally, a test purpose is defined using an automaton forming a tree-like structure of states and transitions. However, loops are now allowed (see, for example, [Jard and Jéron 2005]).

A number of tools use a common test derivation engine with support for input in different formats. For example, TestComposer [Kerbrat et al. 1999] uses TGV and allows, amongst others, specifications written in MSC.

6.6 Summary

This section briefly reviewed material on process algebras, their underlying semantic model (labelled transition systems) and methods for generating tests from a labelled transition system (LTS) specification. The literature on testing from an LTS is quite different from that found for many other formalisms in that it focusses on the different types of communication and thus observation that might occur. This has resulted in a range of implementation relations and corresponding test suite generation algorithms.

Process Algebras and FSMs are similar in that they model or specify a system through a set of states and transitions between these states. Process Algebras are the more expressive, especially when dealing with nondeterminism, and the implementation relations relate to the ability of the tester to make observations. An interesting research question is whether such implementation relations can be effectively translated to the area of testing from an FSM with a fault model.

Test hypotheses and fault models have played a relatively small role in the literature on testing from a Process Algebra. There is the usual minimum hypothesis, that the SUT can be modelled using the same formalism. In addition, when using the *ioco* implementation relation it is necessary to be able to observe the failure of the SUT to produce an output (quiescence) and the implementation must be input-enabled and these can be seen as a test hypotheses. In contrast to the work on testing from FSMs there do not exist standard fault models and the development of such models could be a topic for future research. The lack of such fault models has led to most test generation algorithms being non-deterministic.

The research on testing from a Process Algebra specification has largely focussed on testing from an LTS. An LTS does not have data and the process of expanding out a description that contains data to form an LTS can lead to a combinatorial explosion and is not feasible if the datatypes are infinite. Recent work has investigated ways of overcoming this problem. One possibility is to use uniformity and regularity hypotheses: uniformity hypotheses are used to partition each datatype into a finite number of subdomains and regularity hypotheses are used to restrict the lengths of the traces considered. If we also include a fairness assumption then we can produce a finite test suite that is guaranteed to determine correctness as long as these test hypotheses hold [Gaudel and James 1998; Lestiennes and Gaudel 2002]. An alternative is to use Symbolic Transition Systems; conformance relations and test generation algorithms have recently been developed for such models [Frantzen et al. 2004].

It is sometimes possible to apply an alternative approach to testing from an LTS which involves converting the LTS specification into an FSM. A test suite is then generated from this FSM (see, for example, [Petrenko et al. 1994]). The advantage of this approach is the ability to use the fault models and test generation algorithms developed for FSMs. How-

ever, there is the disadvantage that either we are restricted to the implementation relations that are used in testing from an FSM or we have to produce a different transformation for each implementation relation.

7. TESTING HYBRID SYSTEMS

7.1 Introduction

Hybrid systems are dynamical systems consisting of interacting discrete and continuous components. Typically they are control applications and are often used in (safety/time) critical systems. Examples include air and sea traffic management systems [Godhavn et al. 1996], automated highway systems [Varaiya 1993], industrial automation [Ravn et al. 1995; Varaiya 1993], and automotive control systems [Antsaklis et al. 1995; Antsaklis 1997; Antsaklis et al. 1999; Stauner et al. 1997; Girard et al. 2005].

The complexity of these systems has recently risen sharply as they have to cope with massive sensory and actuator uncertainty and must act in *real time* within complex environments which are spatially extended, dynamic, stochastic, and largely unknown. Traditional approaches that dealt *separately* with their continuous and discrete aspects are inadequate.

Individual feedback control scenarios are naturally modelled as interconnections of modules characterized by their input/output behaviour. Modal control, by contrast, naturally suggests a state-based view, with states representing control modes. Modern, software-based control systems usually involve both styles of control. The distinct modelling techniques need to be reconciled in order to support a systematic methodology for the design, validation, and implementation of control software.

To meet these challenges, mathematical theories that combine, in a coordinated fashion, continuous mathematics (e.g., the differential and integral calculi) and discrete mathematics (e.g., first order logics) are used to provide adequate formal support for the modelling and analysis of hybrid systems, and the development of software for these systems.

7.2 Models and Specification Formalisms

The dichotomy between the *input/output* (feedback) view and the *state* (multi-modal) view is often presented in a restricted setting, as a dichotomy between continuous and discrete control. Continuous feedback control focuses on the analog interaction of the controller with a physical entity, through sensors and actuators. Continuous control models and design techniques have been developed, used, and validated extensively.

Commonly used models for hybrid systems, such as hybrid *automata*, combine *state-transition* diagrams for discrete behaviour with *differential equations* or inclusions for continuous behaviour. For example Manna and Pnueli [Manna and Pnueli 1992] modelled a hybrid system as a *phase transition system*[4], Φ. The semantics is based on *sampling computations* that sample the continuous behaviour of a hybrid system at important, countably many observation points. Φ is defined as $\Phi = \langle V, \Theta, \mathcal{T}, \mathcal{A}, I \rangle$ where

—V is a finite set of state variables that is partitioned into a set of discrete variables, V_d , and a set of continuous variables V_c: $V = V_d \cup V_c$.

—Θ is the initial condition characterizing the initial states.

—The finite set of transitions, $\mathcal{T}$.

[4]The time domain is taken to be the non-negative reals, $\mathcal{R}^+$ and that $\mathcal{R}^\infty = \mathcal{R}^+ \cup \{\infty\}$

—$\mathcal{A}$ is a finite set of *activities*. Each activity is a conditional differential equation of the form: $p \rightarrow \dot{x} = e$, where p is a predicate over V_d (often called *activation condition*), e is an expression over V and $x \in V_c$.

—The set I is a finite set of *important events* that should never be missed when sampling.

Within such a model, properties are specified using a temporal logic with the extension that state formulae (assertions) may refer to the variable T representing a real-time clock.

Because of the continuous-vs.-discrete focus, however, the mechanisms for composition and abstraction of hybrid automata are rather primitive. These models were developed for studying decision problems about the analysis of hybrid systems, yielding for example conditions under which reachability is decidable. However, they have a number of shortcomings when used for design. For example, they have limited expressive power (a simple delay element cannot be modelled as a hybrid automaton).

7.3 Design Synthesis and Verification

In theory, the approaches to hybrid controller design guarantee that the requirements specification is satisfied by design. In practice, system complexity often limits the applicability of automatic controller synthesis methods, and currently, a common approach to the design of hybrid controllers involves independently coming up with a reasonable design for both the discrete and continuous parts. The combined hybrid controller is then put together by means of interfaces, and verification/testing is carried out to ensure that it satisfies certain properties. Typically, some design errors are found, and the verification/testing attempt supplies the design engineer with diagnostic information that is helpful for reformulating the design. The process of simulation, verification attempts, and redesign is iterated until a successful design is obtained. This approach has been motivated by the success of verification and testing techniques for finite-state systems.

The push towards stronger verification and testing techniques has been in the direction of extending the standard finite state machine results to incorporate progressively more complicated continuous dynamics. The first extension was for systems with clocks [Alur and Dill 1990]. Support Tools for finite state machines with clocks have been implemented (KRONOS [Daws et al. 1995], UPPAAL [Bengtsson et al. 1995], and Timed COSPAN [Alur and Kurshan 1996]) and used successfully for the automatic analysis of real-time hardware and software. Since the possible values of clocks range over the real numbers the state space of real-time systems is infinite and so formal verification proceeds symbolically, by representing sets of states using symbolic constraints. Symbolic state space analysis techniques have been extended from real-valued clock variables to all real-valued variables whose trajectories can be characterized using piecewise-linear envelopes [Alur et al. 1993]. If the guarded assignments and invariants of hybrid automata are linear constraints on continuous states, and the differential equations are replaced by linear constraints on first derivatives, then we obtain the class of linear hybrid automata, which can be analyzed fully automatically. A typical linear constraint on first derivatives is a rectangular differential inclusion, which can be used to model, for example, time measured by clocks with bounded drift, or distance covered by vehicles with bounded speed.

Recall the example given in Figure 4 of Section 2. This is a simple nonlinear hybrid system that describes a Temperature Controller. The temperature of a room is controlled through a thermostat that continuously senses the temperature and turns the heater on and off. The temperature is governed by a set of differential equations.

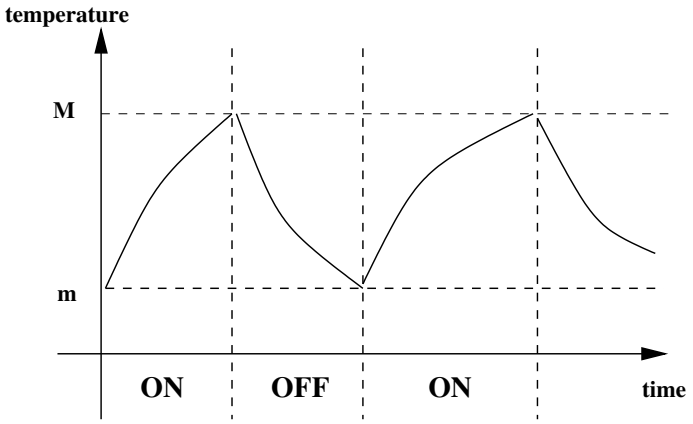

Fig. 16. Test run of automata

As can be seen the automaton has two states l_0 and l_1 with an initial condition set to $x = m$ (naturally, we get a different test run if we have a different initial condition). A test run of such a system is depicted in Fig.16. The test run shows the hybrid behaviour of the controller: the discrete behaviour of the heater as it changes from ON to OFF and vice versa. It also shows its continuous behaviours as time progresses within a state. Such a test run can be obtained using UPPAAL (which provides a graphical interface) and HYTECH.

7.4 Testing Techniques

As a result of the combination of a discrete state-space and continuous variables, the automated analysis of hybrid automata is challenging. Naturally, the problem of automating test data generation is also challenging. One testing approach is to simulate the hybrid automaton; the simulator can then drive test execution.

Tan et al. [Tan et al. 2004] describe an approach for testing from a hybrid automaton. They use the language CHARON that is a visual language for describing hybrid automata and for which there is tool support that will implement a CHARON model in C++. The approach described is to develop two additional hybrid automata for testing: one is deterministic and describes the test case while the other encapsulates the required (linear temporal logic) properties. These two automata play two roles. First, the model is validated through being composed with these two automata. Then, the SUT is tested by being composed with the code generated from these automata.

The simulation of hybrid automata can be computationally expensive and this can have an impact on testing. A variant on the hybrid language HCC [Gupta et al. 1998], called DCML, has been designed in an attempt to overcome this within the context of modelling hardware [Kondo and Yoshida 2005]. One of the key differences between DCML and HCC is that DCML does not allow the use of inequalities and this is said to simplify solving algebraic equations [Kondo and Yoshida 2005].

There has been significant interest in the use of timed automata and timed input output automata (TIOA) that extend finite automata and FSMs respectively with either time or a set of clocks. Such automata can be seen as a form of hybrid system in which there is one continuous variable that represents time. The importance of real-time properties for many

systems, and the use of specification languages such as SDL that include clocks, has led to much interest in testing from timed automata and TIOA and here we briefly review some of the work in this area.

The addition of time to a finite automaton leads to an infinite state machine, assuming time is represented by a variable with an infinite domain. Naturally, finite state structures are easier to analyze and thus easier to test from. In order to produce a finite model it is possible to define an equivalence relation on time, with a finite number of equivalence classes, and on the basis of this expand out the automaton to form a *region graph* (see, for example [Alur and Dill 1994; En-Nouaary et al. 2002]). En-Nouaary *et al.* show that when the trace equivalence conformance relation is used, test sequences can be generated from a grid automaton derived from the region graph of a TIOA by adapting standard FSM based test techniques [En-Nouaary et al. 1998; En-Nouaary et al. 2002]. The grid automaton is produced by using a coarse grained model of time in which the granularity depends on the number of clocks. The authors show that the resultant test suite is guaranteed to detect certain classes of faults, including timing faults such as constraint widening or restriction, assuming that a well-defined set of test hypotheses hold. These test hypotheses are similar to those used in testing from an FSM and include the assumption that the number of vertices of the region graph of the SUT is no greater than that of the specification, rather than placing a bound on the number of states of the automaton representing the SUT.

Cardell-Oliver [Cardell-Oliver 2000] investigated the problem of testing against a set of communicating TIAO with data specified in UPPAAL. Naturally the addition of either concurrency or data can lead to a state explosion problem, although since the datatypes are finite the feasible path problem is decidable. The approach taken is to define test views, which are similar to test purposes, and for each test view to produce a testable timed transition system (TTTS). Test sequences are then generated from each TTTS in order to check trace equivalence. The resultant TTTS gives complete fault coverage of the implementation's TTTS as long as the implementation's TTTS has no additional states, a fairness assumption holds, and it is possible to reset the implementation. A similar approach is described by Koné [Koné 2001] who test from a TIOA using a test requirement by taking the synchronous product of the automaton and the test requirement. The product is generated on-the-fly so that testing need not consider all states in the product.

In FSM based testing there are test generation techniques that are based on test hypotheses that place a limit m on the number of states of the SUT. The size of the resultant test suite increases exponentially as the difference between m and the number n of states of the specification increases and thus in practice m is chosen so that $m-n$ is small. Similar approaches are possible when testing from a TIOA. Interestingly, however, Cardell-Oliver [Cardell-Oliver 2000] notes that small errors in timing constraints in the SUT can lead to a significant increase in the number of states of the automaton from which tests are generated (in this case the TTTS). It is thus argued that a choice of m such that $m-n$ is small makes little sense here and thus the test generation algorithms use the stronger hypothesis $m=n$.

The approaches, for testing from TIOA, described above all considered the same notion of conformance: trace equivalence. However, the literature on testing from an LTS has shown that more general implementation relations exist and it is thus natural to ask whether other notions of conformance are relevant when testing real-time systems. Recently the ioco relation, described in Section 6, has been extended to *timed ioco (tioco)* [Krichen and Tripakis 2004; 2005]. Test trees are produced using symbolic reachability techniques.

Two cases are considered: a digital clock and an analogue clock. For a digital clock, both preset and on-the-fly test generation are described while only an on-the-fly technique is developed for an analogue clock. Naturally, a digital clock is more relevant to testing since the tester's clock will have a finite precision. An interesting additional extension to previous work allows the user to provide a model of the clock that could, for example, represent the degree of precision available.

Nielsen and Skou [Nielsen and Skou 2003] apply symbolic reachability analysis in order to generate test cases. They consider a restricted form of a time automaton, called an *Event Recording Automaton (ERA)*, and produce coarse equivalence classes of time based on the guards of the transitions. ERAs are timed automata in which the only way a value can be assigned to a clock is when an event a occurs, in which case the clock associated with a is reset. In addition, there are no internal actions. The stated advantage of using ERAs is that they can be determinized. The use of coarse equivalence classes has the potential to reduce the impact of the state explosion problem. Tests are generated based on a conformance relation similar to conf which thus does not distinguish between input and output; communication is through synchronization.

Recent work has considered the problem of testing against a model in which the delay introduced by an action is specified using a distribution [Núñez and Rodríguez 2003]. There is thus the problem of checking whether the true distributions are equivalent to the specified distributions. Naturally, this cannot be decided by testing but testing can be used to sample from the distributions. Combined with the use of statistical techniques, it is thus possible to estimate the true distribution up to a given level of confidence [Núñez and Rodríguez 2003].

7.5 Case studies and tools

Current work on testing hybrid systems has focussed on the use of simulation. For example, Tan et al. [Tan et al. 2004] describe a Model-based Monitor Instrumenting and Synthesizing Toolkit (M^2IST) that will produce a hybrid automaton from a property written in a linear temporal logic. The automaton is composed with the (instrumented) SUT in order to check that the property is preserved in testing. Tan et al. [Tan et al. 2004] applied their approach to a Sony AIBO robotic dog. The model contained analogue input (the visibility of the ball and the angle between the head of the dog and the ball) and a discrete control structure. The intention of the system was that the dog chases the ball. The property encoded in linear temporal logic was that an alarm would be raised if the dog loses track of the ball 50 seconds or more after the ball becomes visible, at which point it starts to bark. This property was successfully converted into CHARON code and composed with the system model and a test automaton for simulation. It was reported that the behaviour of the code for the actual robotic dog was consistent with the simulation.

Tools have also been developed for testing from timed automata. Nielsen and Skou describe a prototype test tool, for testing from an Event Automaton, called RTCAT [Nielsen and Skou 2003]. This applies symbolic reachability techniques and partitions the states based on the guards of transitions. The RTCAT tool was applied to the Philips Audio Protocol that had been designed for exchanging control information between audio/visual components [Nielsen and Skou 2003]. The protocol has a 5% tolerance to timing errors in order to allow some drift in clocks and should also detect message collision. The sender and receiver were both modelled as ERAs with 16 and 8 states respectively. Test generation was found to be feasible and it was reported that in each case state partitioning led to fewer

than 100 states and the total test length was less than 1,000 for breadth-first generation and less than 2,000 for depth-first. However, the test effectiveness was not assessed.

Cardell-Oliver described the application of her approach to a train-gate system using UPPAAL for test generation [Cardell-Oliver 2000]. The train-gate model consisted of three automata: a train, a controller, and a gate. The approach was found to be feasible for this example. The technique allows inputs and outputs to be hidden and includes a notion of environment variables being visible that leads to an equivalence relation on states in which equivalence classes represent equality of the values of visible variables. As one would expect, these factors were found to have an impact on the size of the resultant test graph (TTTS), as did the choice of time interval used by the clock. For example, moving from a time interval of 1 to a time interval of 30 led to a TTTS represented by a graph with 658 edges being replaced by a TTTS represented by a graph with 90 edges.

Krichen and Tripakis have produced a tool TTG (Timed Test Generator) that implements their test technique for generating tests from a TIOA using the implementation relation tioco [Krichen and Tripakis 2005]. The tool takes as input the specification and a Tick automaton that represents a clock and outputs an executable that performs test generation. There are four options regarding test generation: interactive, random, coverage based, or exhaustive to a given depth. Several forms of coverage have been implemented, such as all actions, all locations, and all states (combinations of locations, variable values, and symbolic clock valuations). Test generation is achieved through reachability analysis.

Krichen and Tripakis applied their approach to the Bounded Retransmission Protocol for transmitting files over a lossy medium [Krichen and Tripakis 2005]. In this protocol acknowledgment messages are used to confirm that a packet has been received; if the sender does not receive an acknowledgment within a specified time then a timeout occurs and the packet is resent. There is a given limit on the number of times a packet is sent. Experiments were performed with a clock automaton representing a precise clock and a clock period of 1. The authors performed test generation using a range of coverage criteria including all configurations, where a configuration represents a symbolic state which includes the location, values for variables, and a symbolic representation of the values of the clocks. The potential combinatorial explosion was demonstrated; while there were only 4 locations there were 14,687 configurations. However, it was found that it was possible to cover all configurations using 24 test trees of depth 6–53. Depth-first exhaustive test generation was found to be significantly less effective with, for example, only 18% of configurations being covered with 317 tests.

7.6 Summary

There has been much recent interest in hybrid systems that combine state structures with discrete and continuous variables. The inclusion of continuous variables adds significant challenges and makes the application of testing approaches based on the state structure problematic. It appears that the main focus has been on the development, analysis and simulation of hybrid automata; testing is achieved through simulation. By contrast, there has been little work on test hypotheses and notions of test coverage.

The situation is rather different for a special type of hybrid automaton: timed automata. Here there is only one continuous variable, time, possibly represented by clocks. By defining an equivalence relation on time it is possible to partition the states of the timed automaton and form a finite automaton from which test sequences can be produced. Classic FSM or LTS based test techniques, and their corresponding test hypotheses, can then be used.

While FSM test hypotheses and techniques can be used with timed automata, it is unclear how useful these are in practice. First, there is the possible state explosion when partitioning the states of the timed automaton. By using a coarser equivalence relation we can get fewer states but naturally the corresponding test sequences 'cover' less of the model. Currently there appears to be a lack of evidence regarding the trade-off between the cost of testing and test effectiveness as influenced by the choice of equivalence relation used. In addition, the test hypotheses used concern the automaton produced by state partitioning but it may be more meaningful, for the tester, to phrase test hypotheses in terms of the timed automaton model. For example, we might believe that the SUT is 'similar' to the timed automaton model, for some notion of similarity, but this need not relate to the same notion of similarity between the two automata produced using state partitioning.

8. ALGEBRAIC SPECIFICATIONS AND TESTING

8.1 Introduction

This section considers the relationship between algebraic specifications and testing. A number of authors have exploited the parallel that exists between algebraic specifications of *abstract data types* (ADTs) and classes in object-oriented (OO) programs. An ADT is an entity that encapsulates data together with operations that manipulate that data. In OO programs, classes are constructed by providing an interface that lists the operations (or *methods*) of the class and a *body* that actually implements the operations. A class may also contain instance variables (or *attributes*) that define the *state* of an object in the class. This similarity between ADT specifications and class interfaces suggests that there is a huge potential role for formal algebraic specifications in testing OO software.

8.2 Test Criteria

When algebraic specifications are used in testing, the problem of a potentially infinite number of test cases naturally still arises. As with more traditional forms of testing, test criteria can help in deciding on the adequacy of the selected test cases. In fact there are few criteria associated with testing from algebraic specifications but these will now be considered.

The earliest work that employed algebraic specifications in testing was the development of the DAISTS system [Gannon et al. 1981] and it is perhaps not surprising that it utilized versions of traditional structural testing criteria. For example, in order to increase confidence in the thoroughness of the user-defined test cases, DAISTS insisted that each part of both the axioms and the implementation code be executed at least once. The system also incorporated a rather more sophisticated expression-coverage criterion. This involved "isolation of each subexpression in both axioms and code and reporting any expression that was never evaluated or failed to vary on all evaluations" [Gannon et al. 1981]. In other words, this criterion ensured that all subexpressions evaluated to two different results.

One of the most significant developments in specification-based testing is the generic theory of Gaudel and colleagues [Gaudel 2001; Gaudel and James 1998] that was described in Section 3. Interestingly, initial instantiations of this work have focused on algebraic specifications. With regard to algebraic specifications, it is claimed that an exhaustive test set composed of all ground values of the axioms will guarantee program correctness if the exhaustive test set exposes no faults. Of course, this is not really practicable due to the infinite nature of the exhaustive test set. Hence, regularity and uniformity hypotheses, as

described previously in Section 3, may be introduced to limit the size of the test set.

A *regularity hypothesis* requires identification of some complexity measure, which typically, in the algebraic specification context, is the size of a test expression, where size can be measured as the number of constant and constructor operations occurring in the term. Then the regularity hypothesis says that if the implementation works correctly for all tests up to a certain size k say, it will work for all tests with size greater than k. A *uniformity hypothesis* says that if a program works correctly for some input from a subdomain, then it works correctly for all inputs from the subdomain, i.e. the program behaves uniformly in the subdomain. Interestingly, it has been noted that symbolic evaluation techniques could be used to check that a uniformity hypothesis holds in the SUT [Richardson and Clarke 1985]. Typically, in an algebraic specification, the domain of some abstract type may involve fairly obvious partitioning. Then, representative values of the subdomains may be chosen under the hypothesis that if the implementation works for tests involving those chosen values, it will work for all tests with other values from the subdomains. Clearly the use of regularity and uniformity hypotheses is not restricted to the context of algebraic specifications. However, it is true to say that they were first developed and used as criteria for helping restrict the potentially very large, if not infinite, number of test cases that might be generated for the exhaustive test set of an algebraic specification.

In practice, one of the perceived values of partitioning an input domain into subdomains on which the behaviour of the SUT should be uniform is the generation of test data around the boundaries in Boundary Value Analysis (BVA) (see, for example, [White and Cohen 1980; Clarke et al. 1982; Legeard et al. 2002a]). The essential idea is that the boundaries in the SUT may not be those in the specification (*domain errors*) and that by taking test cases either side of the expected boundary and close to this boundary we are likely to find such domain errors. This does not fit easily with the uniformity hypothesis that implicitly assumes that the boundaries in the SUT are correct. Naturally, we might produce alternative hypotheses that reflect the possibility of domain errors. One possibility is to assume that the actual boundaries in the SUT and those in the specification are related in a particular way. For example, if an expected boundary is a straight line or plane then we could assume that the actual boundary is either correct or is also of the same form (e.g. a line/plane) and is at least a certain 'distance' from the expected boundary, for some measure of distance such as the volume that lies between the two boundaries.

8.3 Test Generation

Test cases can be generated from algebraic specifications in two distinct, but related, ways: by using the syntax of operations; and by using the axioms. The first method was originally presented by Jalote [Jalote 1983] and later used by Jalote and Caballero [Jalote and Caballero 1988] in a controlled experiment to assess the effectiveness of the generated test cases at detecting faults in implementations. The same approach has also been used by Woodward [Woodward 1993] in the OBJTEST system, which was also the vehicle for a similar controlled experiment [Allen and Woodward 1996]. The second method was originally presented by Gannon et al. [Gannon et al. 1981] as part of the DAISTS system. It has also been used by Doong and Frankl [Doong and Frankl 1991; 1994] in their ASTOOT tool, by Gaudel and colleagues [Bernot et al. 1991; Bougé et al. 1986; Choquet 1986] with an approach based on logic programming, and more recently by Tse and colleagues [Chen et al. 1998; Chen et al. 2000; Chen et al. 2001].

As an illustration of the first method of test case generation by purely using operation

```
—balance( empty )
—balance( credit( 100, empty ) )
—balance( credit( 100, credit( 100, empty ) ) )
—balance( credit( 100, debit( 100, empty ) ) )
—balance( credit( balance( empty ), empty ) )
—balance( credit( balance( empty ), credit( 100, empty ) ) )
—balance( credit( balance( empty ), debit( 100, empty ) ) )
—balance( debit( 100, empty ) )
—balance( debit( 100, credit( 100, empty ) ) )
—balance( debit( 100, debit( 100, empty ) ) )
—balance( debit( balance( empty ), empty ) )
—balance( debit( balance( empty ), credit( 100, empty ) ) )
—balance( debit( balance( empty ), debit( 100, empty ) ) )
```

Fig. 17. Example test expressions generated from the syntax of the operations in an algebraic specification

syntax, consider the OBJTEST system [Woodward 1993] that automatically generates test expressions from algebraic specifications in the OBJ notation. Each operation in an OBJ algebraic specification is taken in turn and, for each of the arguments in its domain, further operations are substituted whose ranges are of the appropriate type. This substitution continues up to a user-defined depth of operation nesting (i.e. in effect, this is a regularity hypothesis). In order that a finite number of ground terms are generated, specific values have to be selected for certain data types such as the natural numbers (i.e. in effect, this is a uniformity hypothesis). Various other user controls exist to prune what is essentially an exhaustive enumeration. For the example Bank specification from Section 2.6, an automatically generated set of test cases with operation nesting depth at most four, `balance` as the observer (i.e. last) operation in the sequence and the monetary amount fixed solely at 100, would be as listed in Figure 17. If the test expressions can be reduced to normal form this gives rise to an oracle for the testing process. Of course, such abstract test cases need appropriate modification to be used as part of a test script for testing the actual implementation. In addition, if a sort is not observable then in testing we cannot compare values and so such terms cannot be directly used as oracles (see, for example, [Machado 2000]).

As an illustration of the second method of test case generation by using the axioms, consider the DAISTS system [Gannon et al. 1981] that required the user to supply terms that were substituted for the free variables of the axioms. For example, using the Bank specification, for axioms 3 and 4 one might choose the following two substitutions:

```
—n = 50, acc = debit( 25, credit( 100, empty ) )
—n = 75, acc = debit( 100, credit( 100, empty ) )
```

DAISTS then invoked the implementation functions corresponding to the sequence of operations on the left-hand side and on the right-hand side of the axiom and compared the results. A discrepancy would have indicated a fault.

The tool LOFT (LOgic for Functions and Testing) produced tests cases based on axioms but also explicitly used test hypotheses [Marre 1995]. The tool implemented the uniformity and regularity hypotheses and was capable of automatically generating such hypotheses. In order to apply regularity hypotheses, LOFT contained functions that determined the number of non-constant generators. Unfolding was used in order to suggest possible uniformity hypotheses, in effect subdomains defined by cases in one axiom were introduced

into other axioms and this process could be repeated. This process was automated, with the user specifying which operations should not be unfolded and defining a limit on the depth of unfolding.

The ASTOOT system of Doong and Frankl [Doong and Frankl 1991; 1994] was in many ways rather similar, except that a tool was used to generate random sequences of syntactically-valid operations having particular properties. For each sequence `S1`, the axioms were used as rewrite rules to generate an equivalent sequence `S2`. ASTOOT also had the facility for generating a non-equivalent sequence, so that, in general, test cases were of the form `(S1, S2, tag)`, where `tag` is either `equivalent` or `non-equivalent`. The operation sequences were written in a manner more akin to a program trace rather than a functional form. For example, the expression `E` of Section 2.6 might appear as: `E = empty.credit(100).balance`.

Chen et al. [Chen et al. 1998] improved on the ASTOOT approach of using pairs of equivalent ground terms as the basis for test case generation by introducing the notion of a fundamental pair. A fundamental pair is formed by replacing all the variables on both sides of an axiom by normal forms rather than just any syntactically-valid term. Chen et al. proved that a complete implementation of a canonical specification is consistent with respect to all equivalent terms, if and only if it is consistent with respect to all fundamental pairs. Thus, the use of fundamental pairs as test cases is sufficient to cover all equivalent terms. For the Bank example of Section 2.6, the two equivalent ground terms:

```
—balance( credit( 50, credit( 100, empty ) ) )
—balance( credit( 100, empty ) ) + 50
```

can be obtained by replacing the variables `n` and `acc` in axiom 2 by the normal forms `50` and `credit( 100, empty )` respectively; hence they form a fundamental pair. However, the following pair of equivalent ground terms is not fundamental:

```
—balance(credit( balance(credit(50,empty)), credit(100,empty) ))
—150
```

Although the set of fundamental pairs is a proper subset of the set of equivalent ground terms, it may still be an infinite set. Chen et al. suggested an algorithm for generating a finite number of test cases that relies on constructing all patterns of normal forms from the constants and constructors of the abstract type under test, such that the pattern length does not exceed some positive integer *k*, determined from white-box analysis of the implementation or, alternatively, supplied by the tester. This is not unlike a regularity hypothesis. Chen et al. also recommend testing using non-equivalent ground terms, like Doong and Frankl.

8.4 Case studies and tools

8.4.1 *A Text Editor.* DAISTS [Gannon et al. 1981] stands for Data-Abstraction Implementation, Specification and Testing System. It was a compiler-based tool developed by Gannon, McMullin and Hamlet that allowed ADTs implemented in SIMPL-D to be tested for consistency with specification axioms, given user-supplied test cases and an equality function for each new type in the specification. McMullin and Gannon [McMullin and Gannon 1983] used the DAISTS system to specify, implement and test a record-oriented text editor similar to the one described by Kernighan and Plauger [Kernighan and Plauger

1981]. The editor required a list of editing commands in a file, which it then applied to the file of text that was to be edited. The editor maintained a pointer to the current position in the file being edited and the editor commands operated relative to that position. Commands included text insertion, deletion, and pattern search with substitution.

Although the text editor was a well-known and carefully documented situation, several problems were found in the requirements when the algebraic specification was developed. The full text editor specification consisted of four modules with a total of 211 axioms. Unit testing ensured that the axioms and the implementation were consistent for the chosen test data. As mentioned previously, the built-in monitoring of the DAISTS system required test data to be supplied to execute every axiom branch and every implementation statement. With regard to the DAISTS requirement of expression coverage, namely that each expression in the axioms and the code had at least two different values during testing, a value of 95% was easy to achieve, but higher than that required considerable effort. Integration testing exposed two faults. One fault was manifested as a function that extracted a subrecord with one character too many and originated from an incorrect $>=$ in place of a $>$. The other fault was manifested as a routine that inserted duplicate characters and, in fact, both the specification and the implementation were wrong in the same way.

McMullin and Gannon concluded that, although the use of algebraic axioms as a formal specification language was unwieldy in parts, the overall experience was a successful one with requirements deficiencies being highlighted and very few faults persisting until the integration testing stage.

8.4.2 *A Unix-like Directory Display Tool.* Gerrard et al. [Gerrard et al. 1990] advocated the use of algebraic specifications to perform 'design time testing'. As a case study they used a specification of a tool for producing a flattened representation of a hierarchical Unix-like directory structure. Given as input a directory structure in the form of a tree and a directory name within the tree, the tool was required to output in a list-like manner the underlying subtree that had the given name as its parent.

Gerrard et al. distinguished between two kinds of formal specification: a requirements specification which states the nature of a problem non-algorithmically and a constructive specification to 'blueprint the design for a solution'. For the latter they used the OBJ algebraic specification language with its equationally-defined ADTs and for the former they used an enrichment of OBJ with so-called 'theory' modules that accommodate first-order logic.

In essence, the requirements specification constitutes theorems to be proven about the constructive specification. Since verification was not practically feasible however, they relied upon testing of the constructive OBJ specification using its term rewriting feature and regularity and uniformity hypotheses. The number of names in the directory structure was used as the complexity measure for the regularity hypothesis and the value 5 was chosen giving rise to 22 different tree structures. Even with this limitation the number of test cases was infinite, because there were infinitely many names from which to choose the required five. Since no semantic value was attached to the names, a uniformity hypothesis was used to justify an arbitrary set of four names for populating a single representative from each of the 22 trees. Two observer operations were developed in OBJ to help check that the original tree structure and its flattened form were identical. One of these observer operations required enumeration of the paths from the root node for each of the tree structures. In total, this approach gave rise to $2,024$ separate test expressions for execution by the term

rewriting engine, every one of which gave the expected result. The test expressions were generated manually, but Gerrard et al. remark that some degree of automation would have been beneficial.

It should be emphasized that this case study differs from the others described here in that the algebraic specification itself is being validated against a more abstract specification. Such validation can have significant value since fixing faults in a specification is much more expensive if we do not find these faults until later development phases. Gerrard et al. consider the development of the algebraic specification as a step in the design process and its testing is seen as a way of ensuring a secure design. Of course, we can never entirely get away from the issue of whether our specification or requirements statement is correct since ultimately we are aiming to satisfy a set of unstated user requirements rather than a formal statement of these requirements.

8.4.3 *An Automated Subway System.* Dauchy et al. [Dauchy et al. 1993] used an algebraic specification of a safety-critical piece of software to derive test cases. The specification was for the safety control part of the onboard software of an unmanned subway system; it was written in the PLUSS algebraic specification language. The LOFT tool was used to perform automatic test data selection on two modules. LOFT [Bernot et al. 1991; Bougé et al. 1986; Dauchy et al. 1993] stands for LOgic for Functions and Testing and was developed by Gaudel and colleagues. It transformed axioms specified in PLUSS into Horn clauses. Then a logic programming system in Prolog is utilized to generate test data automatically. The process is controlled by regularity and uniformity hypotheses chosen by the user.

One module, called `doors`, was concerned with the control mechanism for opening the doors. Its function was to raise an alarm and invoke emergency braking if a dangerous situation occurs. In general, the doors should normally only open on the platform-side while the train is not moving. However, to increase passenger flow the doors are allowed to open when the speed is less than some small value. Track-side doors are permitted to open if the train is stopped on a side track.

The other module, called `overspeed`, was responsible for firing an alarm if the train's speed was too high. In normal operation the limiting speed is variable and is the smallest of four separate limit speeds, depending on: (1) the position of the first train ahead; (2) the point before which the train must stop; (3) some authorizations from ground-based equipment; and (4) the geometry of the track.

The full specification consisted of ten other, more primitive, modules in addition to the two that were the focus of concern. One key module used by both the `doors` and the `overspeed` modules was called `state`; it defined the data structure to store the current values of the train parameters.

Modules were tested bottom-up, i.e. every used module in a given module under test was assumed to be already tested. Uniformity hypotheses were applied to types defined in the used modules, i.e. it was assumed good enough to have just one value of a subdomain in the test set. Regularity hypotheses were applied to types defined in the module under test. Few details were supplied of the number of tests generated and there is no report of any faults discovered. However, Dauchy et al. remark that application of their approach to the `doors` module was 'surprisingly easy' and led to some integration testing scenarios that the manufacturer had not planned to test. Application to the `overspeed` module was apparently more difficult, largely because the decomposition of the domain of the limiting

speed was the product of the decompositions for the four separate limiting speeds in an axiom for the minimum of four parameters. This led to some proposed refinements in the test selection tool in order to keep the number of test cases tractable.

8.4.4 *Two Abstract Data Types.* ASTOOT [Doong and Frankl 1991; 1994] stands for A Set of Tools for Object-Oriented Testing and can be considered as a generalization of DAISTS. It was developed by Doong and Frankl and was aimed at class testing of OO programs. It consisted of three components: a test driver generator, a compiler and a simplifier. The driver generator used the interface of the class under test to build a test driver that, when executed, read test cases, executed them and then checked the results. The compiler and simplifier worked together forming a tool for automatic generation of test cases from algebraic specifications in the LOBAS notation.

Doong and Frankl [Doong and Frankl 1991; 1994] describe two case studies that involved using ASTOOT to test 'buggy' ADT implementations of a priority queue and a sorted list. Although described as case studies, the investigation had some features more indicative of a controlled experiment. The priority queue was implemented using a 'heap' that is a complete binary tree in which each node is greater than or equal to its children. A single off-by-one fault was introduced in the delete operation. The sorted list was implemented using a 2-3 tree (a special case of a B-tree) and consisted of approximately $1,000$ lines of Eiffel code. The buggy version was produced by deleting one particular line of the implementation.

For each ADT, correct algebraic specifications were produced in the authors' LOBAS specification language. Using the specification axioms, several thousand test cases were randomly generated for each ADT with various sequence lengths of operations, various ranges in which parameter values could lie and various frequencies of occurrence of the different operators involved. For each test case the equivalent simplified sequence after term rewriting was generated. The test cases were executed on the buggy implementations with the help of an automatically-produced test driver. By using an approximate method for determining observational equivalence of the classes under test, the percentages of test cases were determined that exposed the fault in each implementation. For both ADTs, results showed that long original sequences do better than shorter ones, provided that the range of parameters is large enough to take advantage of the lengths. Doong and Frankl offered some tentative guidelines for test case generation that reflected this finding.

8.4.5 *Other tools.* Daistish [Hughes and Stotts 1996], as the name implies, was a system similar to DAISTS but was designed to be more directly applicable to OO programs that operate, not by function application, but rather by means of methods that may alter the state of an object as a side effect. It was developed by Hughes and Stotts as a Perl script that processed an ADT specification, along with the code for implementing the ADT, to produce a test driver. Prototype versions were developed for programs written in Eiffel and C++.

ROCS [Chen et al. 2000] stands for Relevant Observable ContextS which is the technique implemented in this system by Chen, Tse and Deng. It first utilizes the fundamental pair approach of Chen et al. [Chen et al. 1998], combined with regularity and uniformity hypotheses, to generate a finite number of test cases. To determine observational equivalence of objects resulting from the set of fundamental pairs, a series of methods called the 'relevant observable context' is constructed from the implementation of a given algebraic

specification. The ROCS system has been embedded into a C++ interpreter.

8.5 Summary

Testing using algebraic specifications has been a continuing and well-established area of research since the early 1980s, when Gannon et al. [Gannon et al. 1981] developed the DAISTS system. Algebraic specification based testing has a number of advantages in that it can lead to a high degree of automation in terms of generating test cases, generating test drivers and checking results. The influential work of Gaudel and colleagues [Bernot et al. 1991; Bougé et al. 1986; Gaudel 2001; Gaudel and James 1998] on generic specification-based testing was initially applied to algebraic specifications. The notion of a testing context incorporating test hypotheses, such as the regularity and uniformity kinds, is particularly well suited to algebraic specifications and gives the approach a solid theoretical foundation.

With the rise in popularity of OO ideas, a number of researchers have made use of the connection between algebraic specifications of ADTs and classes in OO programs. The Daistish system of Hughes and Stotts [Hughes and Stotts 1996] was an explicit attempt to adapt the DAISTS approach to an OO setting. Another notable example was the work of Doong and Frankl [Doong and Frankl 1991; 1994] on testing classes in OO programs. It focused on the interaction of operations and was directly inspired by the theory of algebraic specifications for ADTs. Their ASTOOT tool had the interesting feature of automatically generating, not only pairs of equivalent operation sequences, but also pairs of non-equivalent sequences, i.e. negative test cases. Chen et al. [Chen et al. 1998; Chen et al. 2000] improved on Doong and Frankl's work by proposing an approach for test case selection from algebraic specifications based on the notion of fundamental pairs. They have proved the sufficiency of the fundamental pair approach for test generation [Chen et al. 1998].

One of the difficulties with respect to algebraic specification based testing has been the lack of any well-defined incremental strategy for integration testing. Most work is geared to testing at the class or unit level. If the approach is to be useful at the system testing level, strategies for testing from hierarchical modular specifications need to be developed. Zhu [Zhu 2003] has highlighted this issue and started to address the problem. In addition, Machado has shown how one can define oracles and generate test suites from a structured specification [Machado 2000]. Chen et al. [Chen et al. 2001] have also considered testing interactions among objects of different classes in a given cluster and decided that algebraic specifications were not ideal. As part of the TACCLE methodology, they have promoted the use of 'contract' specifications for such purposes, in recognition of the current trend "to integrate different specification languages, each able to handle different aspects of a system" [Clarke and Wing 1996].

9. FORMAL VERIFICATION AND ITS ROLE IN TESTING

Formal verification offers a rich toolbox of mathematical techniques that can both support and supplement the testing of computer systems. The toolbox contains varied techniques such as *temporal-logic model checking* [Clarke et al. 1986; Queille and Sifakis 1982], *constraint logic programming* [Marriott and Stuckey 1998], *propositional satisfiability* [Lynce and Marques-Silva 2002] and *theorem proving* [Robinson and Voronkov 2001], with modern automated tools for verifying software often combining several of them.

Of most relevance regarding its relation to testing is model checking, for two reasons.

First, it is a fully automated verification technique that is today incorporated in many commercial systems design tools and has proved useful in a wide range of case studies [Clarke and Wing 1996]. Secondly, model checkers [Cimatti et al. 1999; Holzmann 2003] provide witnesses and counterexamples for the truth or violation of desired temporal properties, respectively, which can not only be fed into simulators for animation but can also be used for generating test cases.

9.1 Automated Reasoning

Automated reasoning, and in particular model checking, plays an ever increasing role in testing. Model checking involves the use of decision procedures to determine whether a model of a discrete state system satisfies temporal properties formalized in a *temporal logic*. These decision procedures conduct a systematic generation and exploration of the underlying system's state space [Clarke et al. 1999; Manna and Pnueli 1995]. If the system model is finite state, this exploration may be conducted automatically using *model checking algorithms* [Clarke and Emerson 1981; Clarke et al. 1986; Lichtenstein and Pnueli 1985; Queille and Sifakis 1982; Vardi and Wolper 1986].

Temporal logics [Bradfield and Stirling 2001; Emerson 1990; Pnueli 1977; Stirling 1992] make it possible to express statements about a system's behaviour as it evolves over time. Typically, assertions include *safety properties*, defining what should always be true of a system, and a set of *liveness properties*, reflecting conditions that a system must eventually satisfy. The most widely used temporal logics are LTL [Manna and Pnueli 1995; Pnueli 1977] and CTL [Clarke et al. 1986]. LTL is a *linear-time* temporal logic that interprets formulae over system runs, which makes it suitable for specifying test sequences. In contrast, CTL is a *branching-time* logic that interprets formulae over computation trees, which enables one to reason about structural properties of the underlying system and to consider various coverage criteria employed in testing.

Model checkers either work on system models that are provided by a user, or automatically extract system models from software source code. Examples of model checkers following the former approach include *NuSMV* [Cimatti et al. 1999] whose modelling language targets hardware systems, and *Spin* [Holzmann 2003] whose modelling language *Promela* is aimed at modelling distributed algorithms and communications protocols. Examples of the latter approach include the model checker *Java PathFinder* [Havelund and Pressburger 1998] which interfaces with Java, *Verisoft* [Godefroid 1997], operating on C/C++ programs and C programs-based *SLAM* [Ball and Rajamani 2001] and *BLAST* [Henzinger et al. 2003].

The main challenge in model checking arises from the complexity of today's systems, since model checking algorithms are linear in the size of the studied system's state space. Thus, implementations of model checkers are based on clever data structures and techniques for storing and manipulating large sets of states. *Binary Decision Diagrams* (BDDs) [Bryant 1986; McMillan 1993], as employed in *NuSMV*, is a prime example of such a data structure. Advanced model checking techniques include *partial-order reduction* [Godefroid 1996; Peled 1998; Valmari 1990], such as that implemented in the *Spin* model checker, which exploits semantic symmetries in models; and *on-the-fly* algorithms [Henzinger et al. 1996; Holzmann 1996] which construct only those states of a model that are relevant for checking the temporal properties of interest.

Questions regarding the semantics of software are often undecidable and software often gives rise to models with either infinite or very large state spaces. This means that a model-

checker may either run out of memory or fail to complete a search in a reasonable amount of time. In order to extend the scope of model-checking, one either has to improve model-checking techniques or use *abstraction* in order to simplify the problem. For example, some integer variables might be turned into booleans by only discriminating between the values of those variable being zero or non-zero.

An example of the former is [Eisner 2005] who reports success applying symbolic model-checking to a C program implementing a cache. Reference [King et al. 2000] reports that proving conformance of a large safety-critical system written in Spark ADA to its Z specification proved more effective in finding faults than testing, because people involved in proof construction had to understand what is being proven. This is consistent with the observation mentioned in Section 5.4.

Model-checking using abstraction is described in [Corbett et al. 2000; Holzmann and Smith 2001]. More recent approaches, such as the *SLAM* and *BLAST* tools, automatically and incrementally construct models from source code by discovering and tracking those predicates over program variables that are relevant to verifying a temporal property at hand. If a path violating the property is discovered, it is necessary to determine whether this path is only an artifact of the model, due to an overly aggressive abstraction, or a genuine counterexample. Checking this involves collecting constraints on program variables along the counterexample path and using decision procedures (such as those employed in theorem proving) to check if they are collectively satisfiable. If the counterexample path turns out to be infeasible, a more precise model can be constructed by the addition of a few constraints derived from those collected, so as to make the considered path infeasible without overburdening the model with all those constraints. Subsequently, another attempt to find a counterexample can be attempted and so on.

9.2 Formal Verification and Testing

At the first sight, formal verification and testing seem to be quite different things. Automated verification is a static activity that involves analyzing system models, with the analysis completely covering all paths in a model. In contrast, testing is a dynamic activity that studies the real-world system itself, i.e., its implementation or source code, but often covers only a limited number of system paths.

Nevertheless, this distinction between model checking and testing has increasingly become blurred, as more and more model checkers work directly on the source code of software implementations, rather than on user-provided models. We have already mentioned *BLAST* and *SLAM* [Eisner 2005; Gunter and Peled 2005] which operate on C source code. Model checkers for Java code include *Bandera* [Corbett et al. 2000], *Java PathFinder* [Havelund and Pressburger 1998] and *SAL* [Park et al. 2000], which combine model checking with abstraction and theorem proving techniques, too. Another example for source code verification is the *VeriSoft* model checker [Godefroid 1997] which systematically searches state spaces of concurrent programs written in C or C++ by means of a state-less search heuristic that borrows ideas from partial-order reduction. When executing source code in this manner, send and receive primitives as well as control structures are extracted and checked on-the-fly. Facilities for extracting models from source code have recently also been included in *Spin* [Holzmann 2003]. However, the trend of checking temporal properties directly on software implementations is not an activity restricted to *compile-time*, but may also be conducted at *run-time* which corresponds to what is called *run-time monitoring* [Artho et al. 2005; Havelund and Rosu 2004].

The most important role for formal verification in testing is in the automated generation of test cases. In this context, model checking is the formal verification technology of choice; this is due to the ability of model checkers to produce counterexamples in case a temporal property does not hold for a system model. The question of interest is how best to derive input sequences in order to test some implementation against its specification. In the context of *conformance testing* [Lee and Yannakakis 1994], for example, one may assume that a specification is given as a state machine and has already been successfully model-checked against temporal properties ϕ. To generate test sequences, one can then simply model-check the specification again, but this time against the negated properties $\neg\phi$. The model checker will prove $\neg\phi$ to be false and produce counterexamples, in the form of system paths highlighting the reason for the violation. These counterexamples are essentially the desired test sequences [Callahan et al. 1996]. Naturally, a number of issues remain since this approach is not always feasible and there is the problem of mapping test sequences expressed at the level of the model used to concrete test cases.

This basic idea of using temporal formulæ as 'test purposes' has been adapted to generating test sequences for many design languages, including *Statecharts* [Hong et al. 2001], *SCR* [Ammann and Black 2000; Gargantini and Heitmeyer 1999], *SDL* [Engels et al. 1997] and *Promela* [de Vries and Tretmans 2000]. In these approaches, the temporal properties ϕ mentioned above are either derived from user requirements, such as usage scenarios [Engels et al. 1997], defined by a tester in the form of a test purpose LTS [Jard and Jéron 2005] (Section 5.4) or generated according to a chosen coverage criterion [Hong et al. 2001]. Indeed, many coverage criteria based on control-flow or data-flow properties can be specified as sets of temporal logic formulae [Hong et al. 2003; Hong et al. 2002], including *state* and *transition coverage* as well as criteria based on *definition-use pairs*. Given a specification, reference [Ammann et al. 1998] proposes to mutate it and use a model-checker to derive sequences distinguishing between the two; given a way to generate all likely faulty implementations, it is possible to produce tests to detect such faults.

Recently, novel approaches to combining model checking and testing have been proposed, which involve *learning strategies* [Peled 2003]. *Black-box checking* [Peled et al. 2002] is intended for acceptance tests where one neither has access to the design nor the internal structure of the system-under-test. This kind of checking iteratively combines Angluin's algorithm for learning the black-box system, Vasilevskii-Chow's algorithm for black-box testing the learned model against the system (Section 5), as well as automata-based model checking [Vardi and Wolper 1986] for verifying various properties of the learned model. *Adaptive model checking* [Groce et al. 2002] may be seen as a variant of black-box checking where a system model does exist but may not be accurate. In this case, learning strategies can be guided by the partial information provided by the system model. However, counterexamples produced via model checking must then be examined for whether they are genuine or the result of an inaccuracy in the model.

A different learning approach is described in [Ernst 2001] which uses the Daikon tool to prove non-trivial properties of software. Daikon monitors a program which is run on a number of tests, collecting information on relations between variables. By their nature, such relations do not necessarily hold for all possible runs and the likelihood of this depends on the quality of a test set used. It turns out that many such relations can be automatically proven by a theorem-prover and subsequently used as lemmas to automatically demonstrate properties of the program considered [Win and Ernst 2002].

Another interesting line of research involves model checking of programs where code fragments, such as procedures, are missing. In *Unit checking* [Gunter and Peled 2005], the behaviour of the missing procedure is provided by specifications of drivers and stubs. These specifications employ logical assertions in order to relate program variables before and after a missing procedure's execution. Given a specification of program paths suspected of containing a bug, a program under investigation is searched for possible executions that satisfy the specification. Theorem-proving technologies are used to calculate path conditions symbolically, so as to report only bugs within paths that can indeed be executed during actual program runs.

The model checker *BLAST* has been extended to automatically generate test vectors for driving a given program into locations exhibiting a desired predicate [Beyer et al. 2004]; similar results are reported in [Gunter and Peled 2005; Artho et al. 2005]. As the underlying technology relies on symbolic execution for handling arithmetic operators and alias relationships between program variables, paths to such locations are checked for feasibility as in unit checking.

Theorem proving is also used by Burton [Burton 2002] for checking that category-partition tests (Section 4.2.1) cover the whole of the input space of a specification. Partitions are checked for emptiness in [Helke et al. 1997].

Formal techniques can be used to verify that an output from a system under test is consistent with a specification. This is done using theorem-proving in [Burton 2002] and model-checking in [Callahan et al. 1996].

Model-checking and theorem-proving are not the only two techniques for test generation, papers [Gotlieb et al. 1998; Pretschner et al. 2005] employ *constraint logic programming*; *propositional satisfiability* techniques are used in [Wimmel et al. 2000; Marinov and Khurshid 2001].

We have described above how verification techniques can be applied to test generation from a specification and how they can be used to check program code. Testing can be used to check specifications: (1) before proving properties, one might wish to have confidence that those properties actually hold, which can be accomplished by using approaches such as evolutionary computation to find a counter-example [Tracey et al. 2000]; (2) [Liu 1999] describes how tests can be generated for different parts of a specification in order to detect inconsistencies in it.

9.3 Summary

Formal verification, and in particular model checking, complements testing in various ways. First, both formal verification and testing may be carried out on a system model even before a single line of code has been written. Second, while the strength of traditional testing technologies lies largely in analyzing sequential code, model checking excels when investigating the communication behaviour of concurrent and multi-threaded systems. Third, formal verification techniques can be employed to generate test suites and results from testing can be used to construct better models of software for verification and test generation [Harder 2002]. When combined with theorem proving, constraint logic programming and propositional satisfiability techniques, model checking thus becomes a powerful tool for testing software.

10. FUTURE RESEARCH DIRECTIONS

This paper has reviewed the state of the art regarding ways in which formal specifications can be used to assist in software testing. In doing so, it has raised a number of issues and we have seen that there have been differing degrees of success with these issues across the formalisms. In this section we briefly review these issues, grouping them into the following types of research question.

(1) What sorts of tests do we need and what can these tell us?

(2) How can we generate the desired tests?

(3) How can we evaluate our techniques?

(4) What do we mean by 'testability' and how can we enhance this?

We now briefly consider each type of question.

10.1 What sorts of tests do we need and what can these tell us?

We test for a purpose — typically to detect faults and/or provide some degree of confidence in the quality of our system. So, we wish to use a test suite that achieves this purpose efficiently and effectively. However, we do not know a priori whether our system is faulty and, if it is, which test cases will have the highest likelihood of revealing faults. This is one of the fundamental issues in testing.

Fault domains, that capture the (believed) class of possible implementations, have been used in some areas primarily testing from finite state machines and stream X-machines. In these cases fault domains are often based on the notions of output faults and state transfer faults. Moreover, to make complete strategies possible, there is an upper bound on the number of states. Alternatively, a fault domain can simply restrict the input and output domains of the implementation to those of the specification and place an upper bound on the number of states of the implementation — under these conditions complete strategies still exist. These approaches originated in the area of testing hardware for manufacturing faults, as opposed to design/logical faults and here there is a good understanding of the types of faults that can occur and their potential impact. In contrast, it is unclear how we can place a useful upper bound on the number of states of software: one that leads to a test suite of reasonable size. This is because the test suite size grows exponentially in the possible number of extra states[5].

The use of test hypotheses has been proposed in testing from algebraic specifications (see, for example, [Gaudel 1995]) and these test hypotheses are similar to fault domains. It has also been shown that test hypotheses and fault domains can be used when comparing test criteria and test suites [Hierons 2002]. However, we have the problem of choosing appropriate test hypotheses — how can we know that these hold in our system? For both fault domains and test hypotheses we could link the choice with risk but it is unclear how to do this in a systematic manner.

Interestingly, the appealing ideas of fault domains and test hypotheses have largely been restricted to a few specification paradigms. For example, such ideas have only recently been applied to testing from label transition systems [Lestiennes and Gaudel 2002] and

[5] Adaptive techniques [Hierons 2004b] can help here since when using these approaches the combinatorial explosion is not guaranteed to occur. However, in the worst case they suffer from the same combinatorial explosion.

there has been very little work applying such approaches to testing from model based specifications[6]. This seems a little strange, and the existence of appropriate fault domains/test hypotheses for these formalisms is very much an open problem.

If we are to use a fault domain or test hypothesis we want to use one that is likely to hold for our system. Ideally, we also want to verify that it holds - either through static analysis/proof or testing. Thus, the issues here relate to testability, an issue that we discuss in Section 10.4.

It seems likely that the types of faults that we can expect to be present in a system depends upon several factors, such as system and problem properties (e.g. perceived risk), the development methodology used, the experience of the staff, and any tools used. For example, are we likely to see different types of faults if we use formal refinement or autocoding? Thus, if we want to develop useful fault domains or test hypotheses then we may well benefit from focussing on particular classes of systems and environments[7]. This suggests the following research question.

Are there well-defined classes of system for which we have a fault domain or test hypothesis that is:

(1) likely to hold;

(2) useful in test generation; and

(3) relatively easy to verify?

If we can find such classes of systems, it may be possible to devise targeted techniques for verifying the corresponding test hypotheses/fault domains.

10.2 How can we generate the desired tests?

Having decided what types of test cases we need, possibly on the basis of some fault domain or test hypothesis, we have the problem of generating a suitable test suite. One of the great benefits of having a formal model is the potential to automate the test generation process and we have seen that much attention has been paid to this problem.

It is arguable that the area in which there has been most progress in automated test generation is testing from a Finite State Machine (FSM) or Labelled Transition System (LTS). The presence of an FSM or LTS specification simplifies test generation since most of the relevant problems (such as equivalence) are decidable; this facilitates automation. Automation is further assisted by developments in model-checking and the fact that it is often possible to express test optimization in terms of graph traversal problems for which there are known algorithms. In contrast, when considering a specification in another formalism, often such problems are undecidable and this complicates test generation. As we have seen, there has been some success in test automation for other formalisms and developments in model-checking may well build on these results. However, in order to facilitate automated test generation it may be necessary to insist on testability properties. While we cannot expect a single notion of testability to be appropriate across all application domains, the development of domain/language specific testability properties and corresponding automated test generation algorithms may well prove to be a significant area for future research.

[6]The work on testing from boolean specifications has, however, considered classes of faults (see, for example, [Kuhn 1999; Tsuchiya and Kikuno 2002; Lau and Yu 2005]) and Burton showed how such an approach can be used when testing from a Z specification [Burton 2002].

[7]By a class of system we include factors such as the development methodology used and staff experience.

Even when dealing with FSM and LTS specifications, there are a number of research questions.

(1) **Testing concurrent/distributed systems**. Many systems are specified as a set of components that cooperate in order to produce the required functionality. Many approaches that analyze such specifications can encounter the state explosion problem — the number of states of the overall system can grow exponentially with the number of components we have. The question then is: when can we overcome the state explosion problem and how can we do this?

(2) **Testing from partial/incomplete specifications**. In practice many specifications are incomplete, sometimes because of the pressures of development schedules (only the 'core' elements are specified) and sometimes because a component's context means that it should not receive certain input values or sequences. However, many current test generation techniques assume that the specification is complete — the challenge is to extend these to incomplete specifications.

(3) **Systematic adaptive testing**. Adaptivity is useful, and sometimes essential, when testing from a non-deterministic specification. Consider, for example, the testing of some function involving two subsystems interacting where an initial connection must be established over an unreliable medium. If the test controls the process of setting up the connection then it must be adaptive since the number of attempts required cannot be known in advance. However, many test generation techniques assume that the specification is deterministic and produce preset test cases. Thus, there is the challenge of generating efficient systematic[8] test suites consisting of adaptive test cases. Note that some current test generation tools do apply adaptive/on-the-fly techniques [Kerbrat et al. 1999].

10.3 How can we evaluate our techniques?

There are many alternative test criteria and test generation techniques. It is thus natural for a tester to ask 'What criterion/technique is best for my system?'. Here 'best' refers to expected effectiveness and cost and thus relates to risk. In order to answer such questions we want to be able to reason about the test criteria and techniques and how these relate.

Hamlet [Hamlet 1989] noted that ideally we want a relation $\leq$ such that for criteria C_1 and C_2, $C_1 \leq C_2$ if and only if whenever there is some test suite satisfying C_1 that finds a fault then every test suite that satisfies C_2 will find this fault. If this is the case then we know that we lose nothing in terms of fault detecting power if we replace a test suite produced to satisfy C_1 with a test suite produced to satisfy C_2. However, Hamlet also observed that no real test criteria are related under $\leq$ and this has led to the use of weaker comparators such as subsumes.

It has recently been observed that when testing a system we can use information about the system, in the form of a fault domain or test hypotheses, when comparing test criteria or techniques [Hierons 2002]. For example, one criterion might be stronger than another, along the lines suggested by Hamlet, for systems with a given property but not for all systems. However, for this observation to be useful we need to find classes of system and real test criteria for which this is the case. Another open question is whether information

[8] By systematic we mean that they satisfy some given test criterion. In turn, the test criterion could be based on a fault domain or test hypothesis.

about the system can be used to refine other approaches to comparing test criteria and techniques (see [Weyuker 2002] for a review of approaches to comparing test criteria).

In addition to developments in theory, it would be extremely useful to have further empirical studies regarding the efficiency and effectiveness of alternative test techniques and criteria across a range of systems. This would be facilitated by the presence of a set of benchmark systems and specifications drawn from several application domains. Tools that allowed test techniques to be rapidly prototyped would also encourage empirical evaluation.

10.4 What do we mean by 'testability' and how can we enhance this?

There are several rival definitions of testability. However, there appears to be at least two issues that relate to testability.

(1) How easy is it to generate a test suite that satisfies the given test criterion?
(2) How good are test suites, that satisfies the test criterion, at finding any faults that are present in the SUT?

It is thus clear that testability depends on system and specification properties and which test criterion or technique we are using. For example, there has been work on improving the testability of code when applying Genetic Algorithms in test data generation [Harman et al. 2004]. Most work on testability has considered the code rather than the specification. However, there are some exceptions.

(1) Some approaches to testing from an extended finite state machine (EFSM) involve abstracting away the data and guards/conditions from the EFSM to form a finite state machine (FSM) and then generating sequences/paths from the FSM. For each path produced in this way we then have to find an input sequence that leads to the corresponding path in the EFSM being traversed. Unfortunately the presence of infeasible paths in the EFSM can cause problems here. There has thus been interest in transforming an EFSM into one that either contains no infeasible paths or one in which the test criterion can be satisfied using paths that are known to be feasible. While this problem is generally uncomputable, there are classes of systems for which it can be solved [Uyar and Duale 1999; Duale and Uyar 2004].
(2) When testing a state-based system we need to set up the internal state for a test and check it after a test. This problem is simplified if we are allowed to augment our specification with functions that are designed to achieve this.
(3) Sometimes a specification of a state-based system may be rewritten in order to aid testing [Burton 2002]. Specifically, some specifications can be rewritten to simplify the problem of distinguishing states of the system.
(4) Testability can relate to properties of the environment, an example being the 'reasonable environment' assumption often made when testing from communicating FSMs. Here it is assumed that the SUT processes an input before the next input is received.

11. CONCLUSIONS

This survey has described formal methods, software testing, and a number of ways in which a formal specification can be used in order to assist testing. We have seen that software testing benefits from the presence of a formal specification in a number of important ways.

In particular, the presence of a formal specification aids test automation and allows the tester to reason about test effectiveness.

Interestingly, a number of related ideas have appeared in different domains. For example, the notion of a test hypothesis was introduced for testing against an algebraic specification. This is similar to the use of a fault model in testing from a finite state machine.

There has been much work on using formal specifications to improve the testing process. In contrast, there seems to have been little work on the use of testing to aid in the application of formal methods. Further, there has not been a signification amount of work on combining static analysis and testing. This seems to be an area that holds much promise, since static and dynamic analysis provide such different types of information: typically, static analysis provides general information about a model of the implementation while dynamic analysis provides specific information about the actual implementation under test.

A number of major challenges remain. We have seen that a test set may be guaranteed to determine correctness under certain well defined conditions. Where we can verify that the implementation under test satisfies these conditions, the test set determines correctness. This observation leads to the question of whether there exist such conditions or assumptions that are likely to hold for a wide range of systems and are relatively simple to verify. It seems likely that such assumptions will be domain specific. A further issue arises where we have refined the specification down to code. Where such a process has occurred, what sorts of tests are likely to be useful? Can we direct our testing so that it is likely to find problems in the refinement or to check the axioms that underpin the proof of correctness of the refinement? If the refinement has not been proved to be correct, what impact does the potential for refinement errors have on the processes of converting abstract test cases into concrete test cases and checking that the observed output is consistent with the specification?

There is a very tangible benefit to answering the above research questions. In his invited talk at ICSE '05, Littlewood [Littlewood 2005] reaffirmed the inadequacy of test data to provide statistical evidence for the dependability of high-integrity software systems. He supports the use of Probabilistic Networks (see, for example, [Fenton et al. 2002]) to fuse disparate sources of evidence to substantiate a probabilistic claim about the reliability or safety of a system. We believe that a sound formal understanding of the precise division of responsibilities between formal proof (applied to programs, specifications and test hypotheses) and testing (as empirical evidence) is a necessary precondition to the development of such assessment models.

Acknowledgements

This work was supported by EPSRC Formal Methods and Testing (FORTEST) grant GR/R43150. We would like to thank the members of the FORTEST network for the many useful discussions and presentations that have informed the ideas presented in this paper. We would also like to that the anonymous referees whose valuable comments have significantly strengthened this paper.

REFERENCES

ABRIAL, J.-R. 1996. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press.

AERTRYCK, L. V., BENVENISTE, M., AND LE METAYER, D. 1997. CASTING: A formally based software test generation method. In *ICFEM'97: 1st International Conference on Formal Engineering Methods, IEEE, Hiroshima, Japan*.

AHO, A. V., DAHBURA, A. T., LEE, D., AND UYAR, M. U. 1988. An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours. In *Protocol Specification, Testing, and Verification VIII*. Elsevier (North-Holland), Atlantic City, 75–86.

AICHERNIG, B. K. 1999. Automated black-box testing with abstract VDM oracles. In *Computer Safety, Reliability and Security: the 18th International Conference, SAFECOMP'99, Toulouse, France, September 1999*, M. Felici, K. Kanoun, and A. Pasquini, Eds. Lecture Notes in Computer Science, vol. 1698. Springer-Verlag, 250–259.

AICHERNIG, B. K. 2000. Formal specification techniques as a catalyst in validation. In *5th IEEE High Assurance systems engineering symposium (HASE 2000), Albuquerque, New Mexico*. 203–207.

AICHERNIG, B. K. 2001a. Systematic black-box testing of computer based systems through formal abstraction techniques. Ph.D. thesis, Graz University of Technology, Austria.

AICHERNIG, B. K. 2001b. Test-case calculation through abstraction. In *Formal Methods Europe (FME 2001)*, J. N. Oliveira and P. Zave, Eds. Lecture Notes in Computer Science, vol. 2021. Springer-Verlag, 571–589.

AL-AMAYREH, A. AND ZIN, A. M. 1999. PROBE: A formal specification-based testing system. In *20th International conference on Information Systems*. 400–404.

ALLEN, S. P. AND WOODWARD, M. R. 1996. Assessing the quality of specification-based testing. In *Proceedings of the 3rd International Conference on Achieving Quality in Software*. Chapman and Hall, Florence, Italy, 341–354.

ALUR, R., COURCOUBETIS, C., HENZINGER, T., AND HO, P. 1993. Hybrid automata: An algorithmic approach to the specification and analysis of hybrid systems. *Lecture Notes in Computer Science 736*, 209–229.

ALUR, R. AND DILL, D. 1990. Automata for modelling real-time systems. In *Automata, Languages and Programming, 17th International Colloquium, ICALP90*. Lecture Notes in Computer Science, vol. 443. Springer-Verlag, 322–335.

ALUR, R. AND DILL, D. L. 1994. A theory of timed automata. *Theoretical Computer Science 126,* 2, 183–235.

ALUR, R., GROSU, R., HUR, Y., KUMAR, V., AND LEE, I. 2000. Modular specification of hybrid systems in CHARON. In *Hybrid Systems: Computation and Control, Third International Workshop, HSCC 2000*. 6–19.

ALUR, R. AND KURSHAN, R. 1996. Timing analysis in COSPAN. *Lecture Notes in Computer Science 1066*, 220–231.

AMBERT, F., BOUQUET, F., CHEMIN, S., GUENAUD, S., LEGEARD, B., PEUREUX, F., UTTING, M., AND VACELET, N. 2002. BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In *Formal Approaches to Testing Software (FATES 2002)*. INRIA, Brno, Czech Republic, 105–120.

AMLA, N. AND AMMANN, P. 1992. Using Z specifications in category partition testing. In *COMPASS '92, 7th Annual Conference on Computer Assurance*. Gaithersburg, MD, USA, 15–18.

AMMANN, P., BLACK, P. E., AND MAJURSKI, W. 1998. Using model checking to generate tests from specifications. In *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods (ICFEM '98)*. Brisbane, Australia, 46–54.

AMMANN, P. AND OFFUTT, J. 1994. Using formal methods to derive test frames in category-partition testing. In *9th Annual Conference on Computer Assurance (COMPASS 94), Gaithersburg, Maryland*. 69–80.

AMMANN, P. E. AND BLACK, P. E. 2000. Test generation and recognition with formal methods. In *International Workshop on Automated Program Analysis, Testing and Verification*. Limerick, Ireland, 64–67.

ANTSAKLIS, P., KOHN, W., NERODE, A., AND SASTRY, S., Eds. 1995. *Hybrid Systems II*. Lecture Notes in Computer Science, vol. 999. Springer-Verlag.

ANTSAKLIS, P. J., Ed. 1997. *Hybrid Systems IV*. Lecture Notes in Computer Science, vol. 1237. Springer-Verlag.

ANTSAKLIS, P. J., KOHN, W., LEMMON, M., NERODE, A., AND SASTRY, S. 1999. *Hybrid Systems*. Lecture Notes in Computer Science, vol. 1567. Springer-Verlag.

ARTHO, C., BARRINGER, H., GOLDBERG, A., HAVELUND, K., KHURSHID, S., LOWRY, M., PASAREANU, C., ROSU, G., SEN, K., VISSER, W., AND WASHINGTON, R. 2005. Combining test case generation and runtime verification. *Theoretical Computer Science 336,* 2–3, 209–234.

ATTERER, R. 2000. Automatic test data generation from VDM-SL specifications. M.S. thesis, Queen's University of Belfast, Northern Ireland, UK.

BALANESCU, T., COWLING, T., GEORGESCU, H., GHEORGHE, M., HOLCOMBE, M., AND VERTAN, C. 1999. Communicating stream X-machines systems are no more than X-machines. *Journal of Universal Computer Science 5*, 494–507.

BALL, T. AND RAJAMANI, S. 2001. Automatically validating temporal safety properties of interfaces. In *SPIN 2001*. Lecture Notes in Computer Science, vol. 2057. Springer-Verlag, 103–122.

BARNETT, M., GRIESKAMP, W., NACHMANSON, L., SCHULTE, W., TILLMANN, N., AND VEANES, M. 2003. Towards a tool environment for model–based testing with AsmL. In *Formal Approaches to Testing, LNCS volume 2931*. Springer–Verlag, Montreal, Canada, 252–266.

BEHNIA, S. AND WAESELYNCK, H. 1999. Test criteria definition for B models. In *World Congress on Formal Methods*. 509–529.

BENGTSSON, J., LARSEN, K. G., LARSON, F., PETTERSSON, P., AND YI, W. 1995. UppAal: A tool suite for automatic verification of real-time systems. *Lecture Notes in Computer Science 1066*, 232–243.

BENJAMIN, M., GEIST, D., HARTMAN, A., MAS, G., SMEETS, R., AND WOLFSTHAL, Y. 1999. A study in coverage-driven test generation. In *36th Design Automation Conference (DAC'99)*. 970–975.

BERNOT, G., GAUDEL, M.-C., AND MARRE, B. 1991. Software testing based on formal specifications: A theory and a tool. *IEE/BCS Software Engineering Journal 6,* 6, 387–405.

BEYER, D., CHLIPALA, A., HENZINGER, T., JHALA, R., AND MAJUMDAR, R. 2004. Generating tests from counterexamples. In *ICSE 2004*. IEEE Computer Society, 326–335.

BICARREGUI, J., DICK, J., MATTHEWS, B., AND WOODS, E. 1997. Making the most of formal specification through animation, testing and proof. *Science of Computer Programming 29/1–2*, 55–80.

BIDOIT, M. AND MOSSES, P. 2003. *CASL User Manual: Introduction to Using the Common Algebraic Specification Language*. Lecture Notes in Computer Science, vol. 2900. Springer-Verlag.

BOGDANOV, K. 2000. Automated testing of Harel's Statecharts. Ph.D. thesis, The University of Sheffield.

BOUGÉ, L., CHOQUET, N., FRIBOURG, L., AND GAUDEL, M.-C. 1986. Test sets generation from algebraic specifications using logic programming. *The Journal of Systems and Software 6,* 4, 343–360.

BOWEN, J. P., BOGDANOV, K., CLARK, J., HARMAN, M., HIERONS, R. M., AND KRAUSE, P. 2002. FORTEST: Formal methods and testing. In *26th IEEE Computer Software and Applications (COMPSAC 2002)*. 91–101.

BRADFIELD, J. AND STIRLING, C. 2001. Modal logics and mu-calculi: An introduction. In *Handbook of Process Algebra*. Elsevier Science, 293–330.

BRINKSMA, E. 1988. A theory for the derivation of tests. In *Protocol Specification, Testing, and Verification VIII*. North-Holland, Atlantic City, 63–74.

BRINKSMA, E., HEERINK, L., AND TRETMANS, J. 1997. Developments in testing transition systems. In *IFIP TC6 10th International Workshop on Testing of Communicating Systems*. Kluwer, Cheju Island, Korea, 143–166.

BRINKSMA, E., HEERINK, L., AND TRETMANS, J. 1998. Factorized test generation for multi-input/output transition systems. In *IFIP TC6 11th International Workshop on Testing of Communicating Systems*. Kluwer, Tomsk, Russia, 67–82.

BROEKMAN, B. AND NOTENBOOM, E. 2003. *Testing Embedded Software*. Addison–Wesley, London.

BRYANT, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers 35,* 8, 677–691.

BURTON, S. 2002. Automated testing from specifications. Ph.D. thesis, The University of York.

CALLAHAN, J., SCHNEIDER, F., AND EASTERBROOK, S. 1996. Automated software testing using model-checking. In *SPIN '96*. Rutgers University, 118–127.

CARDELL-OLIVER, R. 2000. Conformance tests for real-time systems with timed automata specifications. *Formal Aspects of Computing 12,* 5, 350–371.

CARRINGTON, D. A. AND STOCKS, P. A. 1994. A tale of two paradigms: Formal methods and software testing. In *Z User Workshop, Cambridge 1994*, J. P. Bowen and J. A. Hall, Eds. Workshops in Computing. Springer-Verlag, 51–68.

CAVALLI, A. R., CHIN, B.-M., AND CHON, K. 1996. Testing methods for SDL systems. *Computer Networks and ISDN Systems 28,* 12, 1669–1683.

CAVALLI, A. R., LEE, D., RINDERKNECHT, C., AND ZAÏDI, F. 1999. Hit-or-jump: An algorithm for embedded testing with applications to in services. In *Formal Methods for Protocol Engineering and Distributed Systems (FORTE XII)*. 41–56.

CHAN, W. Y. L., VUONG, S. T., AND ITO, M. R. 1989. On test sequence generation for protocols. In *IFIP WG6.1 Ninth International Symposium on Protocol Specification, Testing and Verification*. 119–130.

CHANG, J. AND RICHARDSON, D. 1994. Static and dynamic specification slicing. In *4th Irvine Software Symposium, Irvine, Calfornia, USA*.

CHEN, H. Y., TSE, T. H., CHAN, F. T., AND CHEN, T. Y. 1998. In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology 7,* 3, 250–295.

CHEN, H. Y., TSE, T. H., AND CHEN, T. Y. 2001. TACCLE: A methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology 10,* 4, 56–109.

CHEN, H. Y., TSE, T. H., AND DENG, Y. T. 2000. ROCS: an object-oriented class-level testing system based on the relevant observable contexts technique. *Information and Software Technology 42,* 10, 677–686.

CHOQUET, N. 1986. Test data generation using a prolog with constraints. In *The Workshop on Software Testing*. IEEE Computer Society Press, Los Alamitos, CA, 132–141.

CHOW, T. S. 1978. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering 4*, 178–187.

CIANCARINI, P., CIMATO, S., AND MASCOLO, C. 1996. Engineering formal requirements: Analysis and testing. In *8th International Conference on Software Engineering and Knowledge Engineering (SEKE), Lake Tahoe*. 385–392.

CIMATTI, A., CLARKE, E. M., GIUNCHIGLIA, F., AND ROVERI, M. 1999. NuSMV: A new symbolic model verifier. In *CAV '99*. Lecture Notes in Computer Science, vol. 1464. Springer-Verlag, 495–499.

CLARKE, E. M. AND EMERSON, E. A. 1981. Synthesis of synchronization skeletons for branching time temporal logic. In *Workshop on Logics of Programs*. Lecture Notes in Computer Science, vol. 131. Springer-Verlag, 52–71.

CLARKE, E. M., EMERSON, E. A., AND SISTALA, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems 8,* 2, 244–263.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. 1999. *Model Checking*. MIT Press.

CLARKE, E. M. AND WING, J. M. 1996. Formal methods: State of the art and future directions. *ACM Computing Surveys 28,* 4, 626–643.

CLARKE, L. A., HASSELL, J., AND RICHARDSON, D. J. 1982. A close look at domain testing. *IEEE Transactions on Software Engineering 8,* 4, 380–390.

CLEAVELAND, R., LI, T., AND SIMS, S. 2000. *The Concurrency workbench of the new century*. SUNY, Stony Brook.

CORBETT, J. C., DWYER, M. B., HATCLIFF, J., LAUBACH, S., PASAREANU, C. S., ROBBY, AND ZHENG, H. 2000. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*. IEEE Computer Society Press, 439–448.

CRAGGS, I., SARDIS, M., AND HEUILLARD, T. 2003. AGEDIS case studies: Model-based testing in industry.

CUSACK, E. AND WEZEMAN, C. 1993. Deriving tests for objects specified in z. In *Z User Workshop, London 1992*, J. P. Bowen and J. E. Nicholls, Eds. Workshops in Computing. Springer-Verlag, 180–195.

DAUCHY, P., GAUDEL, M.-C., AND MARRE, B. 1993. Using algebraic specifications in software testing: A case study on the software of an automatic subway. *The Journal of Systems and Software 21,* 3, 229–244.

DAWS, C., OLIVERO, A., TRIPAKIS, S., AND YOVINE, S. 1995. The tool kronos. In *Hybrid Systems*. Lecture Notes in Computer Science, vol. 1066. Springer, 208–219.

DE VRIES, R. G. AND TRETMANS, J. 2000. On-the-fly conformance testing using SPIN. *Software Tools for Technology Transfer 2,* 4 (Mar.), 382–393.

DEMILLO, R., LIPTON, R. J., AND PERLIS, A. J. 1979. Social processes and proofs of theorems and programs. *Communications of the ACM 22,* 5, 271–280.

DERRICK, J. AND BOITEN, E. 1999. Testing refinements of state-based formal specifications. *Software Testing, Verification and Reliability 9*, 27–50.

DICK, J. AND FAIVRE, A. 1993. Automating the generation and sequencing of test cases from model based specifications. In *FME '93: Industrial Strength Formal Methods*, J. C. P. Woodcock and P. G. Larsen, Eds. Lecture Notes in Computer Science, vol. 670. Springer-Verlag, 268–284.

DIJKSTRA, E. W. 1972. Notes of structured programming. In *Structured Programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. Academic Press.

DOONG, R.-K. AND FRANKL, P. G. 1991. Case studies on testing object-oriented programs. In *The Symposium on Testing, Analysis and Verification (TAV4)*. ACM Press, New York, 165–177.

DOONG, R.-K. AND FRANKL, P. G. 1994. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology 3,* 2, 101–130.

DUALE, A. Y. AND UYAR, M. U. 2004. A method enabling feasible conformance test sequence generation for EFSM models. *IEEE Transactions on Computers 53,* 5, 614–627.

EILENBERG, S. 1974. *Automata, languages and machines*. Vol. A. Academic Press.

EISNER, C. 2005. Formal verification of software source code through semi-automatic modeling. *Software and System Modeling 4,* 1, 14–31.

EL-FAR, I. K., THOMPSON, H. H., AND MOTTAY, F. E. 2001. Experiences in testing pocket PS applications. In *International Internet & Software Quality Week Europe Conference*.

EMERSON, E. A. 1990. Temporal and modal logic. In *Handbook of Theoretical Computer Science*. Vol. B. North-Holland, 995–1072.

EN-NOUAARY, A., DSSOULI, R., AND KHENDEK, F. 2002. Timed Wp-method: Testing real-time systems. *IEEE Transactions on Software Engineering 28,* 11, 1023–1038.

EN-NOUAARY, A., DSSOULI, R., KHENDEK, F., AND ELQORTOBI, A. 1998. Timed test cases generation based on state characterization technique. In *IEEE Real-Time Systems Symposium*. 220–229.

ENGELS, A., FEIJS, L. M. G., AND MAUW, S. 1997. Test generation for intelligent networks using model checking. In *Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS '97*. Lecture Notes in Computer Science, vol. 1217. Springer-Verlag, 384–398.

ERNST, M. D. 2001. Summary of dynamically discovering likely program invariants. In *International Conference on Software Maintenance (ICSM 2001)*. 540–544.

FARCHI, E., HARTMAN, A., AND PINTER, S. 2002. Using a model-based test generator to test for standard conformance. *IBM systems journal 41,* 1, 89–110.

FENTON, N., KRAUSE, P., AND NEIL, M. 2002. Software Measurement: Uncertainty and Causal Modeling. *IEEE Software 10,* 4, 116–122.

FERNANDEZ, J.-C., JARD, C., JERON, T., AND VIHO, G. 1996. Using on-the-fly verification techniques for the generation of test suites. *Lecture Notes in Computer Science 1102*, 348–359.

FETZER, J. H. 1988. Program verification: The very idea. *Communications of The ACM 31,* 9, 1048–1063.

FLETCHER, R. AND SAJEEV, A. S. M. 1996. A framework for testing object-oriented software using formal specifications. In *Ada-Europe'96, International conference on reliable software technologies, Montreux, Switzerland*. 159–170.

Formal Systems (Europe) Ltd 1997. *Failures-Divergence Refinement: FDR2 User Manual*. Formal Systems (Europe) Ltd.

FRANTZEN, L., TRETMANS, J., AND WILLEMSE, T. A. C. 2004. Test generation based on symbolic specifications. In *Formal Approaches to Software Testing, 4th International Workshop (FATES 2004)*. Linz, Austria, 1–15.

FUJIWARA, S., VON BOCHMANN, G., KHENDEK, F., AMALOU, M., AND GHEDAMSI, A. 1991. Test selection based on finite state models. *IEEE Transactions on Software Engineering 17,* 6, 591–603.

GANNON, J., MCMULLIN, P., AND HAMLET, R. 1981. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems 3,* 2, 211–223.

GARAVEL, H. AND HERMANNS, H. 2002. On combining functional verification and performance evaluation using CADP. In *International Symposium of Formal Methods Europe (FME)*. 410–429.

GARGANTINI, A. AND HEITMEYER, C. 1999. Using model checking to generate tests from requirements specifications. In *ESEC '99*. Lecture Notes in Computer Science, vol. 1687. Springer-Verlag, 146–162.

GAUDEL, M.-C. 1995. Testing can be formal too. In *TAPSOFT '95*. Springer-Verlag, 82–96.

GAUDEL, M.-C. 2001. Testing from formal specifications, a generic approach. In *Ada–Europe*. Lecture Notes in Computer Science, vol. 2043. Springer-Verlag, 35–48.

GAUDEL, M.-C. AND JAMES, P. R. 1998. Testing algebraic data types and processes: a unifying theory. *Formal Aspects of Computing 10,* 5–6, 436–451.

GERRARD, C. P., COLEMAN, D., AND GALLIMORE, R. M. 1990. Formal specification and design time testing. *IEEE Transactions on Software Engineering 16,* 1, 1–12.

GILL, A. 1962. *Introduction to the Theory of Finite-State Machines*. McGraw-Hill.

GIRARD, A. R., SPRY, S., AND HEDRICK, J. K. 2005. Real-time embedded hybrid control software for intelligent cruise control applications. *IEEE Robotics and Automation Magazine 12,* 1, 22–28.

GODEFROID, P. 1996. *Partial-order Methods for the Verification of Concurrent Systems — An Approach to the State-explosion Problem*. Lecture Notes in Computer Science, vol. 1032. Springer-Verlag.

GODEFROID, P. 1997. Model checking for programming languages using VeriSoft. In *Principles of Programming Languages*. ACM Press, 174–186.

GODHAVN, J.-M., LAUVDAL, T., AND EGELAND, O. 1996. Hybrid control in sea traffic management systems. In Hybrid Systems III. *Lecture Notes in Computer Science 1066*, 149–160.

GOGUEN, J. A. AND MALCOLM, G. 2000. *Software Engineering with OBJ: Algebraic Specifications in Action*. Kluwer Academic Publishers.

GOGUEN, J. A. AND TARDO, J. J. 1979. An introduction to OBJ: a language for writing and testing formal algebraic specifications. In *The IEEE Conference on Specifications of Reliable Software*. IEEE Computer Society Press, 170–189.

GONENC, G. 1970. A method for the design of fault detection experiments. *IEEE Transactions on Computers 19*, 551–558.

GOODENOUGH, J. B. AND GERHART, S. L. 1975. Toward a theory of test data selection. *IEEE Transactions on Software Engineering 1,* 2, 156–173.

GOTLIEB, A., BOTELLA, B., AND RUEHER, M. 1998. Automatic test data generation using constraint solving techniques. In *ISSTA '98*. ACM Press, 53–62.

GRIESKAMP, W., GUREVICH, Y., SCHULTE, W., AND VEANES, M. 2002. Generating finite state machines from abstract state machines. In *Proceedings of the ACM SIGSOFT Symposium on Software Testing and Analysis*. 112–122.

GROCE, A., PELED, D., AND YANNAKAKIS, M. 2002. Adaptive model checking. In *TACAS 2002*. Lecture Notes in Computer Science, vol. 2280. Springer-Verlag, 357–370.

GUNTER, E. AND PELED, D. 2005. Model checking, testing and verification working together. *Formal Aspects of Computing 17*, 201–221.

GUPTA, V., JAGADEESAN, R., AND SARASWAT, V. A. 1998. Computing with continuous change. *Science of Computer Programming 30,* 1–2, 3–49.

HALL, P. A. V. 1988. Towards testing with respect to formal specification. In *2nd IEE/BCS Conference on Software Engineering '88*. 159–163.

HAMLET, R. 1989. Theoretical comparison of testing methods. In *Third Symposium on Testing, Analysis and Verification*. ACM, Key West, Florida, USA, 28–37.

HARDER, M. 2002. Improving test suites via generated specifications. Tech. Rep. 848, MIT Dept. of EECS. 4 June.

HAREL, D. AND GERY, E. 1997. Executable object modelling with Statecharts. *IEEE Computer*, 31–42.

HARMAN, M., HU, L., HIERONS, R. M., WEGENER, J., STHAMER, H., BARESEL, A., AND ROPER, M. 2004. Testability transformation. *IEEE Transactions on Software Engineering 30,* 1, 3–16.

HAVELUND, K. AND PRESSBURGER, T. 1998. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer 2,* 4, 366–381.

HAVELUND, K. AND ROSU, G. 2004. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design 24,* 2, 189–215.

HAYES, I. J. 1986. Specification directed module testing. *IEEE Transactions on Software Engineering 12,* 1, 124–133.

HELKE, S., NEUSTUPNY, T., AND SANTEN, T. 1997. Automating test case generation from Z specifications with Isabelle. In *ZUM '97: The Z Formal Specification Notation*, J. P. Bowen, M. G. Hinchey, and D. Till, Eds. Lecture Notes in Computer Science, vol. 1212. Springer-Verlag, 52–71.

HENNIE, F. C. 1964. Fault-detecting experiments for sequential circuits. In *5th Annual Symposium on Switching Circuit Theory and Logical Design*. 95–110.

HENZINGER, T., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2003. Software verification with Blast. In *SPIN 2003*. Lecture Notes in Computer Science, vol. 2648. Springer-Verlag, 235–239.

HENZINGER, T. A., KUPFERMAN, O., AND VARDI, M. Y. 1996. A space-efficient on-the-fly algorithm for real-time model checking. In *CONCUR '96*. Lecture Notes in Computer Science, vol. 1119. Springer-Verlag, 514–529.

HIERONS, R. M. 1997a. Testing from a finite state machine: Extending invertibility to sequences. *The Computer Journal 40,* 4, 220–230.

HIERONS, R. M. 1997b. Testing from a Z specification. *Software Testing, Verification and Reliability 7,* 1, 19–33.

HIERONS, R. M. 2001. Checking states and transitions of a set of communicating finite state machines. *Microprocessors and Microsystems, Special Issue on Testing and testing techniques for real-time embedded software systems 24,* 9, 443–452.

HIERONS, R. M. 2002. Comparing tests in the presence of hypotheses. *ACM Transactions on Software Engineering and Methodology 11,* 4, 427–448.

HIERONS, R. M. 2004a. Minimizing the number of resets when testing from a finite state machine. *Information Processing Letters 90,* 6, 287–292.

HIERONS, R. M. 2004b. Testing from a non-deterministic finite state machine using adaptive state counting. *IEEE Transactions on Computers 53,* 10, 1330–1342.

HIERONS, R. M. AND HARMAN, M. 2000. Testing comformance to a quasi-non-determinstic stream X-machine. *Formal Aspects of Computing 12,* 6, 423–442.

HIERONS, R. M. AND HARMAN, M. 2004. Testing conformance of a deterministic implementation to a non–deterministic stream X–machine. *Theoretical Computer Science 323,* 1–3, 191–233.

HIERONS, R. M., KIM, T.-H., AND URAL, H. 2004. On the testability of SDL specifications. *Computer Networks 44,* 5, 681–700.

HIERONS, R. M., SADEGHIPOUR, S., AND SINGH, H. 2001. Testing a system specified using Statecharts and Z. *Information and Software Technology 43,* 2, 137–149.

HIERONS, R. M. AND URAL, H. 2006. Optimizing the length of checking sequences. *IEEE Transactions on Computers 55,* 5, 618–629.

HOARE, C. A. R. 1985. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science.

HOARE, C. A. R. 1996. How did software get so reliable without proof? *Lecture Notes in Computer Science 1051*, 1–17.

HOLCOMBE, M. AND IPATE, F. 1998. *Correct Systems: Building a Business Process Solution*. Springer-Verlag.

HOLZMANN, G. 2003. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley.

HOLZMANN, G. J. 1996. On-the-fly model checking. *ACM Computing Surveys 28,* 4, 120–120.

HOLZMANN, G. J. AND SMITH, M. H. 2001. Software model checking: extracting verification models from source code. *Software Testing, Verification and Reliability 11,* 2, 65–79.

HONG, H., CHA, S., LEE, I., SOKOLSKY, O., AND URAL, H. 2003. Data flow testing as model checking. In *ICSE 2003*. IEEE Computer Society, 232–243.

HONG, H., LEE, I., SOKOLSKY, O., AND URAL, H. 2002. A temporal logic based theory of test coverage and generation. In *TACAS 2002*. Lecture Notes in Computer Science, vol. 2280. Springer-Verlag, 327–341.

HONG, H. S., LEE, I., SOKOLSKY, O., AND CHA, S. D. 2001. Automatic test generation from Statecharts using model checking. In *FATES '01*. BRICS Notes Series, vol. NS-01-4. BRICS, 15–30.

HÖRCHER, H.-M. AND PELESKA, J. 1995. Using formal specifications to support software testing. *The Software Quality Journal 4,* 4, 309–327.

HÖRL, J. AND AICHERNIG, B. K. 2000. Validating voice communication requirements using lightweight formal methods. *IEEE Software 17,* 3, 21–27.

HSIEH, E. P. 1971. Checking experiments for sequential machines. *IEEE Transactions on Computers 20*, 1152–1166.

HUBER, F., SCHÄTZ, B., AND EINERT, G. 1997. Consistent graphical specification of distributed systems. In *4th International Symposium of Formal Methods Europe (FME '97)*. 122–141.

HUGHES, M. AND STOTTS, D. 1996. Daistish: systematic algebraic testing for OO programs in the presence of side-effects. In *International Symposium on Software Testing and Analysis*. ACM Press, San Diego, 53–61.

HUR, Y., FIERRO, R. B., AND LEE, I. 2003. Modeling distributed autonomous robots using CHARON: Formation control case study. In *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003)*. 93–98.

IPATE, F. AND HOLCOMBE, M. 1997. An integration testing method that is proved to find all faults. *International Journal of Computer Mathematics 63,* 3&4, 159–178.

IPATE, F. AND HOLCOMBE, M. 2000. Generating test sets from non-deterministic stream X-machines. *Formal Aspects of Computing 12,* 6, 443–458.

ISO. 1989a. *ISO 8807:1989 Information processing systems, Open Systems Interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behaviour*. ISO.

ISO. 1989b. *ISO 9074; Estelle – A formal description technique based on an extended state transition model*. ISO.

ITU-T. 1997. *Recommendation Z.500 Framework on Formal Methods in Conformance Testing*. International Telecommunications Union, Geneva, Switzerland.

ITU-T. 1999. *Recommendation Z.100 Specification and Description Language (SDL)*. International Telecommunications Union, Geneva, Switzerland.

JACKSON, D. AND VAZIRI, M. 2000. Finding bugs with a constraint solver. In *ISSTA'00: International symposium on software testing and analysis*. ACM Press, 14–25.

JALOTE, P. 1983. Specification and testing of abstract data types. In *7th International Computer Software and Applications Conference (COMPSAC '83)*. IEEE Computer Society Press, Chicago, 508–511.

JALOTE, P. AND CABALLERO, M. G. 1988. Automated testcase generation for data abstraction. In *12th International Computer Software and Applications Conference (COMPSAC '88)*. IEEE Computer Society Press, Chicago, 205–210.

JARD, C. AND JÉRON, T. 2005. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer 7,* 4, 297–315.

JONES, C. B. 1991. *Systematic Software Development using VDM*, 2nd ed. Prentice Hall International Series in Computer Science.

KEMMERER, R. A. 1985. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering 11,* 1, 32–43.

KERBRAT, A., JÉRON, T., AND GROZ, R. 1999. Automated test generation from SDL specifications. In *SDL Forum*. 135–152.

KERNIGHAN, B. W. AND PLAUGER, P. J. 1981. *Software Tools in Pascal*. Addison-Wesley, Reading, MA.

KING, S., HAMMOND, J., CHAPMAN, R., AND PRYOR, A. 2000. Is proof more cost-effective than testing? *IEEE Transactions on Software Engineering 26,* 8 (Aug.), 675–686.

KOHAVI, Z. 1978. *Switching and Finite State Automata Theory*. McGraw-Hill, New York.

KONDO, K. AND YOSHIDA, M. 2005. Use of hybrid models for testing and debugging control software for electromechanical systems. *IEEE/ASME Transactions on Mechatronics 10,* 3, 275–284.

KONÉ, O. 2001. A local approach to the testing of real-time systems. *Computer Journal 44,* 5, 435–447.

KRICHEN, M. AND TRIPAKIS, S. 2004. Black-box conformance testing for real-time systems. In *11th International SPIN Workshop*. 109–126.

KRICHEN, M. AND TRIPAKIS, S. 2005. An expressive and implementable formal framework for testing real-time systems. In *17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems (TestCom 05)*. 209–225.

KUHN, D. R. 1999. Fault classes and error detection capability of specification–based testing. *ACM Transactions on Software Engineering and Methodology 8,* 4, 411–424.

LAU, M. F. AND YU, Y.-T. 2005. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering and Methodology 14,* 3, 247–276.

LEDUC, G. 1991. Conformance relation, associated equivalence, and a new canonical tester in LOTOS. In *Protocol Specification, Testing, and Verification XI*. Elsevier (North-Holland), 249–264.

LEE, D. AND YANNAKAKIS, M. 1994. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers 43,* 3, 306–320.

LEE, D. AND YANNAKAKIS, M. 1996. Principles and methods of testing finite-state machines. *Proceedings of the IEEE 84,* 8, 1089–1123.

LEGEARD, B., PEUREUX, F., AND UTTING, M. 2002a. Automated boundary testing from Z and B. In *Formal Methods Europe (FME 02)*. Lecture Notes in Computer Science, vol. 2391. Springer, 21–40.

LEGEARD, B., PEUREUX, F., AND UTTING, M. 2002b. A comparison of the BTT and TTF test-generation methods. In *Formal Specification and Development in Z and B (ZB 2002)*. 309–329.

LESTIENNES, G. AND GAUDEL, M.-C. 2002. Testing processes from formal specifications with inputs, outputs and data types. In *13th International Symposium on Software Reliability Engineering (ISSRE 2002)*. 3–14.

LICHTENSTEIN, O. AND PNUELI, A. 1985. Checking that finite state concurrent programs satisfy their linear specification. In *Principles of Programming Languages*. IEEE Computer Society Press, 97–107.

LITTLEWOOD, B. 2005. CASTING: Dependability assessment of software-based systems: State of the art. In *ICSE'05: 27th International Conference on Software Engineering, Missouri, USA*. 6–7.

LIU, S. 1999. Verifying consistency and validity of formal specifications by testing. In *World Congress on Formal Methods*, J. Wing, J. Woodcock, and J. Davies, Eds. Lecture Notes in Computer Science, vol. 1708. Springer-Verlag, 896–914.

LIU, S., FUKUZAKI, T., AND MIYAMOTO, K. 2000. A GUI and testing tool for SOFL. In *APSEC 2000: 7th Asia-Pacific Software Engineering Conference*. IEEE, 421–425.

LÓPEZ, N. AND NÚÑEZ, M. 2004. An overview of probabilistic process algebras and their equivalences. In *Validation of Stochastic Systems*. Lecture Notes in Computer Science, vol. 2925. Springer, 89–123.

LUO, G. L., PETRENKO, A., AND VON BOCHMANN, G. 1994. Selecting test sequences for partially-specified nondeterministic finite state machines. In *7th IFIP Workshop on Protocol Test Systems*. Chapman and Hall, Tokyo, Japan, 95–110.

LUO, G. L., VON BOCHMANN, G., AND PETRENKO, A. 1994. Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. *IEEE Transactions on Software Engineering 20,* 2, 149–161.

LYNCE, I. AND MARQUES-SILVA, J. 2002. Building state-of-the-art sat solvers. In *15th European Conference on Artificial Intelligence (ECAI'02)*. 166–170.

MACCOLL, I. AND CARRINGTON, D. 1998. Testing MATIS: A case study on specification-based testing of interactive systems. In *FAHCI98: Formal Aspects of Human Computer Interaction Workshop*. British Computer Society, 57–69.

MACHADO, P. D. L. 2000. Testing from structured algebraic specifications. In *Algebraic Methodology and Software Technology (AMAST 2000)*. 529–544.

MANNA, Z. AND PNUELI, A. 1992. Verifying hybrid systems. In *Hybrid Systems*. Lecture Notes in Computer Science, vol. 736. Springer-Verlag, 4–35.

MANNA, Z. AND PNUELI, A. 1995. *Temporal Verification of Reactive Systems*. Springer-Verlag.

MARINOV, D. AND KHURSHID, S. 2001. TestEra: A novel framework for automated testing of Java programs. In *Proceedings of the 16th IEEE Conference on Automated Software Engineering (ASE 2001)*. IEEE, San Diego, CA.

MARRE, B. 1995. LOFT: A tool for assisting selection of test data sets from algebraic specifications. In *TAPSOFT'95: Theory and Practice of Software Development*. Lecture Notes in Computer Science, vol. 915. Springer, 799–800.

MARRIOTT, K. AND STUCKEY, P. 1998. *Programming with Constraints: An Introduction*. The MIT Press, Cambridge, MA, USA.

MCMILLAN, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers.

MCMULLIN, P. R. AND GANNON, J. D. 1983. Combining testing with formal specifications: A case study. *IEEE Transactions on Software Engineering 9,* 3, 328–335.

MEUDEC, C. 1997. Automatic generation of software tests from formal specifications. Ph.D. thesis, Queen's University of Belfast, Northern Ireland, UK.

MIAO, H., GAO, X., AND LIU, L. 1999. An approach to testing the nonexistence of initial state in Z specifications. In *Test Symposium, 1999. (ATS'99) Proceedings*. 289–294.

MIKK, E. 1995. Compilation of Z specifications into C for automatic test result evaluation. In *ZUM '95: The Z Formal Specification Notation, 9th International Conference of Z Users*. Springer-Verlag, 167–180.

(MIL-STD 188-220B). 1998. *Military standard-interoperability standard for digital message device subsystems*. Department of Defense.

MILLER, R. E. AND PAUL, S. 1993. On the generation of minimal length conformance tests for communications protocols. *IEEE/ACM Transactions on Networking 1,* 1, 116–129.

MILNER, R. 1989. *Communication and Concurrency*. Prentice Hall International Series in Computer Science.

MOORE, E. P. 1956. Gedanken-experiments. In *Automata Studies*, C. Shannon and J. McCarthy, Eds. Princeton University Press.

MOSSES, P. D. 2004. *CASL Reference Manual: The Complete Documentation of the Common Algebraic Specification Language*. Lecture Notes in Computer Science, vol. 2960. Springer-Verlag.

MOST COOPERATION. 2002. MOST media oriented system transport — multimedia and control networking technology.

MURRAY, L., CARRINGTON, D., MACCOLL, I., MCDONALD, J., AND STROOPER, P. 1998. Formal derivation of finite state machines for class testing. In *11th International Conference of Z Users*. Lecture Notes in Computer Science, vol. 1493. Springer, 42–59.

NAITO, S. AND TSUNOYAMA, M. 1981. Fault detection for sequential machines. In *IEEE Fault Tolerant Computer Systems*. 238–243.

NIELSEN, B. AND SKOU, A. 2003. Automated test generation from timed automata. *Software Tools for Technology Transfer 5,* 1, 59–77.

NÚÑEZ, M. AND RODRÍGUEZ, I. 2003. Towards testing stochastic timed systems. In *Formal Techniques for Networked and Distributed Systems (FORTE 2003)*. Lecture Notes in Computer Science, vol. 2767. Springer, 335–350.

OFFUTT, J. AND LIU, S. 1999. Generating test data from SOFL specifications. *The Journal of Systems and Software 49,* 1, 49–62.

OSTRAND, T. J. AND BALCER, M. J. 1988. The category-partition method for specifying and generating functional tests. *Communications of the ACM 31,* 6, 676–686.

PARK, D., STERN, U., SKAKKEBAEK, J. U., AND DILL, D. L. 2000. Java model checking. In *International Workshop on Automated Program Analysis, Testing and Verification*. Limerick, Ireland, 74–82.

PELED, D. 1998. Ten years of partial order reduction. In *Computer Aided Verification, 10th International Conference (CAV '98)*. Lecture Notes in Computer Science, vol. 1427. Springer-Verlag, 17–28.

PELED, D. 2003. Model checking and testing combined. In *Automata, Languages and Programming, 30th International Colloquium (ICALP 2003)*. Lecture Notes in Computer Science, vol. 2719. Springer-Verlag, 47–63.

PELED, D., VARDI, M., AND YANNAKAKIS, M. 2002. Black box checking. *Journal of Automata, Languages and Combinatorics 7,* 2, 225–246.

PETRENKO, A. 1991. Checking experiments with protocol machines. In *Protocol Test Systems*. IFIP Transactions. North-Holland, 83–94.

PETRENKO, A. 2001. Fault model-driven test derivation from finite state models: Annotated bibliography. *Lecture Notes in Computer Science 2067*, 196–205.

PETRENKO, A., BORODAY, S., AND GROZ, R. 1999. Configurations in EFSM. In *IFIP Joint International Conference on Formal Description Techniques for Distributed Systems (FORTE XII) and Communication Protocols, and Protocol Specification, Testing, and Verification (PSTV XIX)*. China, 5–24.

PETRENKO, A., BORODAY, S., AND GROZ, R. 2004. Confirming configurations in EFSM testing. *IEEE Transactions on Software Engineering 30,* 1, 29–42.

PETRENKO, A., YEVTUSHENKO, N., LEBEDEV, A., AND DAS, A. 1994. Nondeterministic state machines in protocol conformance testing. In *Protocol Test Systems, VI (C-19)*. Elsevier Science (North-Holland), Pau, France, 363–378.

PETRENKO, A., YEVTUSHENKO, N., AND VON BOCHMANN, G. 1996. Testing deterministic implementations from nondeterministic FSM specifications. In *9th International Workshop on Testing of Communicating Systems (IWTCS'96)*. 125–140.

PHALIPPOU, M. 1994. Executable testers. In *Protocol Test Systems VI*. North-Holland, 35–50.

PNUELI, A. 1977. The temporal logic of programs. In *FOCS '77*. IEEE Computer Society Press, 46–57.

PRETSCHNER, A., PRENNINGER, W., WAGNER, S., KÜHNEL, C., BAUMGARTNER, M., SOSTAWA, B., ZÖLCH, R., AND STAUNER, T. 2005. One evaluation of model-based testing and its automation. In *27th ACM/IEEE International Conference on Software Engineering (ICSE 2005))*. St. Louis, Missouri, USA, 392–401.

QUEILLE, J. P. AND SIFAKIS, J. 1982. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*. Lecture Notes in Computer Science, vol. 137. Springer-Verlag, 337–351.

RAVN, A. P., RISCHEL, H., CONRAD, F., AND ANDERSEN, T. O. 1995. Hybrid control of a robot – a case study. In Hybrid Systems II. *Lecture Notes in Computer Science 999*, 391–404.

RICE, M. D. AND SEIDMAN, S. B. 1998. An approach to architectural analysis and testing. In *3rd International workshop on Software architecture*. ACM Press, New York, 121–123.

RICHARDSON, D., AHA, S. L., AND O'MALLEY, T. O. 1992. Specification-based test oracles for reactive systems. In *14th IEEE International Conference on Software Engineering*. IEEE, 105–118.

RICHARDSON, D. J. AND CLARKE, L. A. 1985. Partition analysis: A method combining testing and verification. *IEEE Transactions on Software Engineering 14,* 12, 1477–1490.

ROBINSON, A. AND VORONKOV, A. 2001. *Handbook of Automated Reasoning*. MIT Press.

SABNANI, K. AND DAHBURA, A. 1988. A protocol test generation procedure. *Computer Networks 15,* 4, 285–297.

SATO, F., MUNEMORI, J., IDEGUCHI, T., AND MIZUNO, T. 1989. Sequence generation tool for communication system. In *2nd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, (FORTE'89)*.

SCHMITT, M., EK, A., GRABOWSKI, J., HOGREFE, D., AND KOCH, B. 1998. Autolink – putting SDL-based test generation into practice. In *Testing of Communicating Systems, IFIP TC6 11th International Workshop on Testing Communicating Systems (IWTCS)*. Kluwer Academic Publishers, 227–244.

SIDHU, D. P. AND LEUNG, T.-K. 1989. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering 15,* 4, 413–426.

SIMS, S. 1999. *The Process Algebra Compiler User's Manual*. Reactive Systems, Inc.

SINGH, H., CONRAD, M., AND SADEGHIPOUR, S. 1997. Test case design based on Z and the classification-tree method. In *1st IEEE Conference on Formal Engineering Methods*. IEEE Computer Society, Hiroshima, Japan, 81–90.

SPIVEY, J. M. 1988. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press.

SPIVEY, J. M. 1992. *The Z Notation: A Reference Manual*, 2nd ed. Prentice Hall International Science in Computer Science.

STAUNER, T., MULLER, O., AND FUCHS, M. 1997. Using HYTECH to verify an automotive control system. *Lecture Notes in Computer Science 1201*, 139–153.

STEPNEY, S. 1995. Testing as abstraction. In *The Z Formal Specification Notation (ZUM 95)*. Lecture Notes in Computer Science, vol. 967. Springer, 137–151.

STIRLING, C. 1992. Modal and temporal logics. In *Handbook of Logic in Computer Science*. Vol. 2. Oxford University Press, 477–563.

STOCKS, P. A. 1993. Applying formal methods to software testing. Ph.D. thesis, University of Queensland, Australia.

STOCKS, P. A. AND CARRINGTON, D. A. 1996. A framework for specification-based testing. *IEEE Transactions on Software Engineering 22,* 11 (Nov.), 777–793.

TAN, L., KIM, J., SOKOLSKY, O., AND LEE, I. 2004. Model-based testing and monitoring for hybrid embedded systems. In *IEEE International Conference on Information Reuse and Integration (IRI - 2004)*. 487–492.

TAOUIL-TRAVERSON, S. AND VIGNES, S. 1996. Preliminary analysis cycle for B-method software development. In *EUROMICRO 96: 22nd EUROMICRO Conference, Beyond 2000: Hardware and Software Design Strategies*. IEEE, 319–325.

TRACEY, N., CLARK, J., MANDER, K., AND MCDERMID, J. 2000. Automated test-data generation for exception conditions. *Software Practice and Experience 30,* 1, 61–79.

TREHARNE, H., DRAPER, J., AND SCHNEIDER, S. 1998. Test case preparation using a prototype. In *B'98: 2nd International B Conference, Recent advances in the development and use of the B Method*, D. Bert, Ed. Lecture Notes in Computer Science, vol. 1393. Springer-Verlag, 293–311.

TRETMANS, J. 1996. Conformance testing with labelled transitions systems: Implementation relations and test generation. *Computer Networks and ISDN Systems 29,* 1, 49–79.

TRETMANS, J. AND BRINKSMA, E. 2003. TORX : Automated Model Based Testing. In *First European Conference on Model-Driven Software Engineering*, A. Hartman and K. Dussa-Zieger, Eds. Imbuss, Möhrendorf, Germany. 13 pages.

TSUCHIYA, T. AND KIKUNO, T. 2002. On fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology 11,* 1, 58–62.

URAL, H., SALEH, K., AND WILLIAMS, A. 2000. Test generation based on control and data dependencies within system specifications in SDL. *Computer Communications 23*, 609–627.

URAL, H., WU, X., AND ZHANG, F. 1997. On minimizing the lengths of checking sequences. *IEEE Transactions on Computers 46,* 1, 93–99.

UYAR, M. U. AND DUALE, A. Y. 1999. Resolving inconsistencies in EFSM modeled specifications. In *IEEE Military Communications Conference (MILCOM)*. Atlantic City, NJ.

UYAR, M. Ü., FECKO, M. A., DUALE, A. Y., AMER, P. D., AND SETHI, A. S. 2003. Experience in developing and testing network protocol software using FDTs. *Information and Software Technology 45,* 12, 815–835.

VALMARI, A. 1990. A stubborn attack on the state explosion problem. In *Computer Aided Verification, 2nd International Workshop (CAV '90)*. DIMACS, vol. 3. AMS, 25–42.

VARAIYA, P. 1993. Smart cars on smart roads. *IEEE Transactions on Automatic Control 38,* 2, 195–207.

VARDI, M. AND WOLPER, P. 1986. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS '86)*. IEEE Computer Society Press, 332–344.

VASILEVSKII, M. P. 1973. *Failure Diagnosis of Automata. Cybernetics*. Plenum Publishing Corporation.

VILKOMIR, S. A. AND BOWEN, J. P. 2001. Formalization of software testing criteria using the Z notation. In *25th Annual International Computer Software and Applications Conference (COMPSAC 2001), Chicago, Illinois*. IEEE Computer Society, 351–356.

VILKOMIR, S. A. AND BOWEN, J. P. 2002. Reinforced condition/decision coverage (RC/DC): A new criterion for software testing. In *2nd International Conference of B and Z Users (ZB 2002)*. 291–308.

VON BOCHMANN, G. AND PETRENKO, A. 1994. Protocol testing: Review of methods and relevance for software testing. In *ACM International Symposium on Software Testing and Analysis*. Seattle USA, 109–123.

VON BOCHMANN, G., PETRENKO, A., BELLAL, O., AND MARGUIRAGA, S. 1997. Automating the process of test derivation from SDL specifications. In *8th SDL forum, INT*. Evry, 261–276.

WAESELYNCK, H. AND BEHNIA, S. 1998. B model animation for external verification. In *2nd International Conference on Formal Engineering Methods*. 36–45.

WATANABE, A. AND SAKAMURA, K. 1996. A specification-based adaptive test case generation strategy for open operating system standards. In *18th International Conference on Software Engineering*. ACM, 81–89.

WEYUKER, E. J. 2002. Thinking formally about testing without a formal specification. In *Formal Approaches to Testing of Software*. Brno, Czech Republic, 1–10.

WEYUKER, E. J. AND OSTRAND, T. J. 1980. Theories of program testing and the the application of revealing subdomains. *IEEE Transactions on Software Engineering 6,* 3, 236–246.

WEZEMAN, C. D. 1989. The CO-OP method for compositional derivation of conformance testers. In *Protocol Specification, Testing, and Verification IX*. Elsevier (North-Holland), Atlantic City, 145–158.

WHITE, L. J. AND COHEN, E. I. 1980. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering 6,* 3 (May), 247–257.

WIMMEL, G., LÖTZBEYER, H., PRETSCHNER, A., AND SLOTOSCH, O. 2000. Specification based test sequence generation with propositional logic. *Software Testing, Verification and Reliability 10*, 229–248.

WIN, T. AND ERNST, M. 2002. Verifying distributed algorithms via dynamic analysis and theorem proving. Tech. Rep. 841, MIT Laboratory for Computer Science. 25 May.

WOODWARD, M. R. 1993. Errors in algebraic specifications and an experimental mutation testing tool. *IEE/BCS Software Engineering Journal 8,* 4, 211–224.

YANG, B. AND URAL, H. 1990. Protocol conformance test generation using multiple UIO sequences with overlapping. In *ACM SIGCOMM 90: Communications, Architectures, and Protocols*. Twente, The Netherlands, 118–125.

YAO, M., PETRENKO, A., AND VON BOCHMANN, G. 1993. Conformance testing of protocol machines without reset. In *Protocol Specification, Testing and Verification, XIII (C-16)*. Elsevier (North-Holland), 241–256.

YEVTUSHENKO, N. V., LEBEDEV, A. V., AND PETRENKO, A. F. 1991. On checking experiments with nondeterministic automata. *Automatic Control and Computer Sciences 6*, 81–85.

YOUNG, M. AND TAYLOR, R. N. 1989. Rethinking the taxonomy of fault detection techniques. In *IEEE/ACM International Conference on Sofware Engineering*. 53–62.

ZHU, H. 2003. A note on test oracles and semantics of algebraic specifications. In *the Third International Conference on Quality Software (QSIC 2003)*. Dallas, Texas, 91–98.