Contracts for Scenario-Based Testing of Object-Oriented Programs

Anna Mikhailova, Marielle Doche, Michael Butler

Department of Electronics & Computer Science, University of Southampton
Highfield, Southampton, SO17 1BJ, UK
aam@ecs.soton.ac.uk

1 Introduction

Black-box testing, extensively used for verification of procedural or modular programs, is one of the main steps of a software development process used to validate the behavior of a system. It is based on some form of contracts between the system under test and its environment, be that a human user or another hardware or software system. For object-oriented programs, the notion of contract is easily retained between an object (or a cluster of objects) as system under test, and a client of this object – as its environment. However, new complications arise due to the specifics of such programs: the notions of object and class, encapsulation, inheritance and polymorphism [8, 31]. Previous works on object-oriented testing agree that a class is the basic unit for black-box testing, and that its methods cannot be tested individually. Moreover, encapsulation obliges to test the behavior of a class under test via sequences of method calls.

For state-based testing, formal methods have been succesfully applied to procedural or modular programs [11, 17, ?] because these methods provide precise and unambiguous specifications. However, only a few works involve formal methods for object-oriented programs [31, 15, 6, 21, 20]. These works aim to build test scenarios as sequences of method calls and they are based on analysis of various formal models of the class under test.

In this paper, we propose an original approach to generating test scenarios as contract models. In case of complex systems, it is inconcievable in practice to test exhaustivelly these systems, and experienced testers focus rather on critical problems. In an object-oriented development such problems are modelled by scenarios of usage, which form a base for defining testing scenarios. These contracts can be precisely and unamigously specified by formal methods [22, 4, 5, 3]. Taking into account the specifics of testing object-oriented programs and having the synthesis of scenarios as an objective, we can identify three main steps in our approach:

1. *specifying the elementary test cases* of the system under test – each test case is a call of a method for a given context of the state space of the system under test. Traditional testing methods like partition analysis can be applied here on the input domain and state space.

2. *creating a test driver* – the engine capable of running all test cases in any order, any number of times.
3. *synthesising test scenarios* – generating sequences of test cases to test a particular method or a combination of methods, according to a chosen testing strategy.

In this work, we provide a formal foundation for this scenario-based method of testing based on contracts. This formal foundation allows us to reason about correctness issues.

The next section describes the problems of testing object-oriented programs and discusses related work. Section 3 introduces the formal context of the contract and test notions. We detail the three steps of our approach in the following sections and summarise in the concluding section.

2 Testing of Object-Oriented Programs

2.1 Challenges in Testing Object-Oriented programs

The importance of black-box testing for object-oriented and component-based programs is becoming ever more apparent with the wider acceptance and increasing popularity of COTS - Components Of The Shelf. For such components, that are usually distributed in a binary format, the existence of a black-box testing method is of paramount importance. Not only would the component producers be able to test the component for the absence (or presence) of bugs, but also the customers intending to acquire the component may use this testing method to convince themselves in the reliability and robustness of the component (or otherwise).

Traditional formal testing methods cannot be applied direclty to object-oriented programs because of their specificities: the notions of object and class, encapsulation, inheritance and polymorphism [8, 31]. For object-oriented approaches, the basic unit of organization is the class construct. Although class methods can be tested individually, it is impossible to reduce the testing of a class to testing of its methods, because tests cannot be designed, coded, and executed as if the methods were independent subroutines. The methods are always invoked on an object and can only be tested through an instance of the corresponding class. Every object carries a state and the context in which a method is invoked is determined not only by possible method parameters, but also, and mainly, by its state. Therefore, the behaviour of a method cannot be considered and tested independently from the object. This identifies class instances as a basic implementation for testing and their corresponding class as the basis for generating test cases.

Unlike procedural programs, object-oriented programs emphasise encapsulation of internal object state and interaction between objects through method calls. The encapsulation largely complicates the process of testing because it disallows direct access to the internal object state. One cannot observe the internal object state to evaluate results of a particular test case, and has to invoke the corresponding query methods on the object, if such methods are at all available. The current and natural trend for black-box testing of object-oriented programs is to execute the implementation under test to a sequence of input and output method calls.

Another complication arises because the input and output of methods are not only simple values, like Integer or Boolean, but also objects. Establishing the initial state of the system, to test a particular input-output pair, can therefore require a sequence of method calls on the object under test and other objects. Other aspects of the object-oriented programming paradigm, such as inheritance and subtyping polymorphism, introduce problems that were not previously found in traditionally structured software and require adapting of the testing methods used.

2.2 Current Formal Approaches to Testing Object-Oriented Programs

Formal methods have proved to be effective in providing sound methodology of black-box testing for procedural or modular programs [16, 10], using either state-based [11, 17, ?] or model-based [30, 14, 13, 18, 26] methods. However, the current formal approaches to testing are not adapted to dealing with object-oriented programs. For example, the approaches [27, 15] inspired from the state-based methods [11, 17, 29] are limited to class testing without inheritance mechanisms (a flattening mechanism is used to remove inheritance). The other state-based approaches, which are based on a formal model of the internal behavior of the class under test [31, 21], do not permit defining an oracle easily.

Techniques presented in [6, 20] focus on the external behavior of the class under test by generating tests from the interface of its methods. Test sequences are generated by analysis of the formal specification, described in logic or algebraic formalisms. In [19], the authors are also interested in the external behavior of the class under test, by defining some test sequences from use cases or interaction diagrams. It is the only work where test sequences are explicitly viewed as models of contracts for object-oriented programs, but they are not based on a formal description of the class under test.

In practice, all these methods define new classes or new objects to test the class under test: one to explicitly create and test an implementation of the class under test and another one to define the oracle.

2.3 Contracts for Scenario-Based Testing of Object-Oriented Programs

Separating interface from implementation has long been recognised as a good programming practice, and modern object-oriented programming languages, such as Java, even provide language mechanisms supporting this separation. Intrerfaces serve as abstract specifications of object behaviour and the classes implementing these intrefaces are supposed to conform to these specifications. Interfaces are usually syntactic and the specifications associated with them are usually informal. Adding formal specifications to interfaces has been and remains the focus of active research; for a survey we refer to [22].

Clearly, the existence of a logical framework within which one can specify and reason about the behaviour of objects and classes is a pre-requisite for developing a formally-based method of testing object-oriented programs. Here we use an object-oriented extension of the refinement calculus as developed in [22]. Our approach to reasoning about object-oriented programs suggests adding formal specifications of object behaviour to interfaces and ensuring that classes declared to implement these interfaces conform to these specifications (refine them). However, as the approach focuses on behavioural substitutability of subtype objects for their supertype objects, it also supports reasoning about object behaviour in programs where interfaces and implementations are unified, e.g. in C++ programs. This is possible because the refinement calculus permits reasoning about specifications and implementations in a uniform manner, treating specifications as abstract programs and implementations as concrete specifications.

The testing approach that we propose in this paper defines a new class and a new object for a given class under test (CUT). First of all, we define a new class, called test case class (TCC), which contains a constructor to create a new implementation of the CUT and test cases as method calls on this implementation. The test cases are obtained from each method of the CUT by partitionning its input domain and the state space of the object. For this purpose, partitionning techniques used in traditional software testing can be applyied [28, 11, 17, 29]. Thus test cases generation is based on the external behavior of the CUT, but it also takes into account its internal state space. This partitionning approach is described in more detail in the following section. A client of a TCC is then defined as a test driver. This test driver (described in Sec. 5) allows us to run various test cases.

The objective of our work is to generate test sequences depending on different coverage criteria (for example, calling at least once each test case). As in [19], we want to generate test scenarios as contracts between the CUT and its client. These scenarios can be deducted from use cases or interaction diagrams or, since we use a formal background, they can be defined via provers or constraint analysis. In this paper, we only give an example of scenario generation in section 6.

Finally, as in [6, 20], we propose to check an oracle by using *observer* methods i.e. those methods that do not modify the internal state of the CUT but provide by their output an image of the state. Such methods must be used at the end of a test scenario to check it. Expected output values can be calculated in advance from the formal specification.

3 Contract-Based Testing

3.1 Contracts for Black-box Testing

Formal methods have proved to be effective in providing sound methodology of black-box testing for procedural or modular programs [16, 10], expecially using state-based methods [11, 17, ?]. In all these approaches, black-box testing is usually based on some sort of contracts [7] between the implementation under test (IUT) and the user, be that a human user or another hardware or software system. The contract essentially stipulates how IUT is supposed to behave in response to a user input. The tester uses this contract to uncover errors by trying to encourage the IUT to breach the contract or by revealing inconsistencies, ambiguities or underspecifications in the contract itself. Formal mathematical methods, such as e.g. Z, VDM, and B, provide languages to define such a contract unambigously, soundly, and succinctly. We use the formal contracts language of the refinement calculus [5] extended in [22] to support the object-oriented style of program construction. The language is a generalisation of the specification languages based on pre- and post-conditions, such as Z, VDM, and B, and has the formal semantics in Higher Order Logic. Below, we briefly introduce the necessary background on contracts in the refinement calculus. A more detailed description as well as operational and weakest precondition semantics of contracts can be found in [4, 5].

3.2 Contracts and Refinement

A computation can generally be seen as involving a number of agents (programs, modules, systems, users, etc.) who carry out actions according to a document (specification, program) that has been laid out in advance. When reasoning about a computation, we can view this document as a contract between the agents involved. In this section we present a notation for *contract statements.*

Contract Statements A system is modelled by a global *state* σ of type (or *state space*) Σ. The state has a number of *program variables* $x_1, \ldots, x_n$, each of which can be observed and changed independently of the others. A *state predicate* $p : \Sigma \to Bool$ is a boolean function on the state. Since a predicate corresponds to a set of states, we use set notation ($\cup$, $\subseteq$, etc.) for predicates.

A change in the state of a system can be described by a functional *state transformer* f or by a *state relation* R. A state transformer $f : \Sigma \to \Gamma$ is a function mapping a state in Σ to a new state in Γ (where the new state can be the same as the old one). A state relation $R : \Sigma \to \Sigma \to Bool$ relates a state σ to a state σ' whenever $R\ \sigma\ \sigma'$ holds. A special notation $x := x' \mid b$ is used to generalize assignement to non-deterministic *relational assignment.* It sets variable x to a new value x' such that b holds.

Contracts are built from state changing functions, predicates and relations. The language of contracts distinguishes between responsibilities of communicating agents in a contract. In general, there can be a number of agents that are acting together to change the state of the system and whose behavior is bound by contracts. A group of agents that are acting together and coordinating their choices to achieve a certain goal can be collectively considered as one agent. The conflicting agents (or groups of agents) who are trying to prevent each other from achieving their goals are usually referred to as the *angel* and the *demon* and their choices as, respectively, *angelic* and *demonic*. For our purposes, the angel, whose viewpoint we are taking, is identified with the tester (which can include a test driver and an oracle checker), and the demon, whose choices are beyond our control and are nontransparent to us, with the implementation under test itself.

The *assignment statement* changes the state according to a state transformer. For example, the assignment statement $x := x + y$ requires the agent to set the value of program variable x to the sum of the values of x and y. The *assertion* $\{p\}$ of a state predicate p is a requirement that the agent must satisfy in a given state. The *assumption* $[p]$ of a state predicate p is a condition on which the agent can rely; if the condition does not hold, then the agent is released from any obligation to carry out his part of the contract. The *angelic relational update* $\{R\}$ is a contract statement that permits angelic (free) choice between all final states related by the state relation R to the initial state. The *demonic relational update* $[R]$ is non-deterministic from the angel's (tester's) point of view, in the sense that the choice of the next state is uncontrollable.

Two (or more) contracts can be combined by sequential composition or choice operators. In a *sequential contract* $S_1; S_2$ the contract S_1 is carried out first, followed by S_2. The *angelic choice* $S_1 \sqcup S_2$ allows the angel (tester) to choose between carrying out S_1 or S_2; the *demonic choice* $S_1 \sqcap S_2$ puts the choice between S_1 and S_2 beyond control of the angel.

The notation for contracts allows us to express all standard programming language constructs, like sequential composition, assignments, empty statements, conditional statements, loops, and blocks with local variables. For example, a while-loop can be expressed as the least fixpoint of function ($\lambda X \bullet \mathsf{if}\ g\ \mathsf{then}\ S; X\ \mathsf{else\ skip\ fi}$):

$$\mathsf{while}\ g\ \mathsf{do}\ S\ \mathsf{od}\ \ \widehat{=}\ \ (\mu X \bullet \mathsf{if}\ g\ \mathsf{then}\ S; X \mathsf{else\ skip\ fi})$$

A variant of iteration, the *iterative choice* contract statement, was introduced in [5] and extensively studied in [3]:

$$\begin{aligned}&\mathsf{do}\ \ g_1 :: S_1 \between \ldots \between g_n :: S_n\ \ \mathsf{od}\ \ \widehat{=}\\ &\qquad (\mu X \bullet \{g_1\}; S_1; X \sqcup \ldots \sqcup \{g_n\}; S_n; X \sqcup \mathsf{skip})\end{aligned}$$

This statement models an event loop in which the user can choose repeatedly among the alternatives that are enabled and execute the chosen alternative. The alternatives among which the choice is made at each iteration step are separated by $\between$; each alternative S_i is preceded by an enabledness condition g_i. The user can continue choosing among the alternatives until he decides to stop. We will abbreviate $g_1 :: S_1 \between \ldots \between g_n :: S_n$ by $\between_{i=1}^{n} g_i :: S_i$.

Apart from the standard programming language constructs, contracts allow us to express (non-executable) specification statements, such as $\{p\}; [R]$ which corresponds to an abstract pre-postcondition specification. The refinement calculus provides a synthesis method for transforming more abstract program structures into more concrete ones based on the notion of refinement of contracts.

Weakest Preconditions and Strongest Postconditions Every contract statement in our language of contracts has a weakest precondition predicate transformer semantics. The weakest precondition approach to total correctness of programs was pioneered by Dijkstra [12] and later was substantially develpped and extended by Back, Morgan, Morris, von Wright, and others [2, 24, 25, 5]. The strongest postcondition ...

For the purposes of this exposition, we only briefly explain the notions of weakest precondition and strongest postcondition, leaving out their definitions for the contract statements. These definitions may be found in [4, 5, **?**].

For any contract S and any condition q to be established, weakest precondition $\mathsf{wp}.S.q$ is defined to be the set of all initial states from which S is guaranteed to establish q.

For any contract S and any condition q, strongest postcondition $\mathsf{sp}.S.q$ is defined to be ...

Refinement of Contracts A contract S is *refined by* a contract S', written $S \sqsubseteq S'$, if any condition that we can establish with the first contract can also be established with the second contract. Formally, refinement is defined by pointwise extension of the subset ordering on predicates. For a state predicate $q : \Sigma \to \mathit{Bool}$ and predicate transformers S, S' of type $(\Sigma \to \mathit{Bool}) \to (\Gamma \to \mathit{Bool})$, we have

$$S \sqsubseteq S' \;\; \widehat{=} \;\; \forall q \bullet \mathsf{wp}.S.q \subseteq \mathsf{wp}.S'.q$$

Refinement is reflexive and transitive. In addition, the contract constructors are monotonic, so a contract can be refined by refining a subcomponent. The refinement ordering defines a lattice of predicate transformers (contracts), with the lattice operators meet $\sqcap$ and join $\sqcup$. The top element in this lattice, known as magic, can miraculously establish any postcondition and is non-implementable. The bottom element, abort, defines the notion of abortion. Large collections of refinement rules are given, for instance, in [5, 24].

3.3 Test Cases are Abstractions

Abstraction is dual to refinement. If $S \sqsubseteq S'$ then S is an abstraction of S'. To emphasize the search for abstractions rather than refinments, we will write $S' \sqsupseteq S$ to express that S is an abstraction of S'.

Since abort is trivially refined by any statement, it is the trivial abstraction of any statement. For obvious reasons, we don't want to consider aborting abstractions. The notion of a testing abstraction [relation] $\sqsupseteq_T$ was defined in [1] to separate non-aborting abstractions:

$$S \sqsupseteq_T T \;\; \widehat{=} \;\; S \sqsupseteq T \sqsupset \mathsf{abort}$$

In [1], an input-output test-case TC was defined as an abstract contract between the user and the unit under test:

$$\mathsf{TC}\ i\ o\ \ \widehat{=}\ \ \{x = i\}; [y := y' \mid y' = o]$$

where x is an input variable and y is an output variable. Intuitively, the contract $\mathsf{TC}\ i\ o$ states that if the user provides as input i, the state will be updated so that the output is equal to o. Such a TC is a formal pre-postcondition specification which is defined exclusively for an input i and which declares the expected output o.

We call a possible sequence of test cases a *test scenario*:

$$\mathsf{TS}\ \ \widehat{=}\ \ \mathsf{TC}\ i_1\ o_1; \mathsf{TC}\ i_2\ o_2 \cdot \mathsf{TC}\ i_n\ o_n.$$

In this work we want to produce test scenarios that conform to a specification. This means that each test scenario, and consequently each test case (as a sequence with only one element), is an abstraction of the specification. Moreover, we are only interested in defining those test scenarios that can detect errors. Obviously, an aborting abstraction must be rejected, as for example a test scenario where the postcondition of a test case is in contradiction with the precondition of the following test case. Formally, we have:

$$\{p\}; [Q]\ \sqsupseteq_T\ \ \mathsf{TS}$$

$$\{p\}; [Q]\ \sqsupseteq_T\ \ \mathsf{TC}\ i\ o$$

A collection TCs of test scenarios is known as a *test set*:

$$\mathsf{TCs}\ \ \widehat{=}\ \ \sqcup_j\ \mid j \in \mathsf{Nat} \cdot \mathsf{TS}\ i_j\ o_j$$

The tester makes a choice among the alternatives when selecting test scenarios from the test set. In practice, the test suites and the number of test scenarios in a test set must always be finite:

$$\mathsf{TCs}\ =\ \mathsf{TS}_1 \sqcup \ldots \sqcup \mathsf{TS}_n$$

Some coverage criteria according the specification can then be defined to decide when the tester can stop. For example, if the test cases describe all possible inputs of the specification, we can choose the test set so that each test case is covered at least once. We give some examples of coverage criteria in the last part of this paper.

Considering test cases as abstract contracts is an original approach with two significant benefits: on the one hand they are abstract and thus easy to understand, and on the other hand they are precise and thus umambiguos.

4 Test Case Generation for Object-Oriented Programs

4.1 The Proposed Approach to Object-Oriented Testing

In the approach that we advocate in this work the implementation under test is a class, called Class Under Test (CUT). The formal Specification of Class

Under Test (SCUT) is integrated with the interface that CUT implements. This has an obvious advantage of testing several classes implementing the same interface using the same test suite, because the latter is developped based on the specification associated with the interface. Note, however, that the same general principles are applicable when testing programs with unified interfaces and implementations. Implementations, i.e., classes under test, are treated in this case as their own (concrete) specifications.

In the context of object-oriented programs, a *test case* is a call to a method of CUT with specific input parameters. We propose to partition each method of the corresponding specification class SCUT according to its input parameters, if any, and its internal state space. Partition analysis is a traditionnal approach in specification-based testing [28, 11, 17, 29, ?]and it has also been applied in object-oriented specification-based testing [27, 15]. The more classical approach used in testing of structured programs is to transform a formula, which represents the precondition of a method, to a Disjunctive Normal Form (DNF), which can be done automatically [17, ?]. Each sub-formula of DNF then corresponds to a test case. To lead partition analysis, the tester can define some *uniformty hypotheses*, which define subdomains of values for which test case has always the same behavior. Then it is sufficient to invoke a test case with one value of each subdomain.

Naturally, it is meaningless to talk about methods in the absense of the class to which they belong, and we define a *test case class* (TCC) specifying the test cases for the specification of the class under test. An instance of SCUT is aggregated into a testing class. The latter incorporates all the test cases, i.e. the partitioned methods that are essentially the methods of SCUT split into (possibly several) new methods corresponding to the partitions. Each of these test cases asserts the corresponding condition on its inputs and the state space of the instance and redirects the call to the corresponding method of the aggregated instance of SCUT. The originality of this approach is that each test case contains simply a method call on the class instance under test and not an explicit description of a statement or the space state evolution.

Defining the test cases in a separate class TCC is beneficial for at least two reasons. First of all, this allows us to avoid modifying the class under test and its specification. Secondly, TCC can be easily reused in a hierarchical approach: whenever a class under test is extended, its TCC can be extended appropriately by adding test cases for the additional methods and by inheriting test cases for the original methods.

4.2 Formalisation of the concepts involved

In this subsection we present a formal foundation for our test cases. First we present a formal description of SCUT, then the formal basis for partitioning test cases, and then the resulting TCC. Finally, we show how such a partition approach satisfies correctness properties of a test set.

Specifying Classes An object is an abstract data type with internal state and methods that can be invoked on the object to carry out certain functionality and (possibly) change the object's state. We view a class as a template for creating objects.

$$
\begin{array}{l}
C \ = \ \mathsf{class} \\
\quad x : \Sigma \\
\quad C\,(\mathsf{val}\ x_0 : \Gamma_0) \ = \ K, \\
\quad m_1\,(\mathsf{val}\ x_1 : \Gamma_1, \mathsf{res}\ y_1 : \Delta_1) \ = \ M_1, \\
\quad \ldots \\
\quad m_n\,(\mathsf{val}\ x_n : \Gamma_n, \mathsf{res}\ y_n : \Delta_n) \ = \ M_n \\
\mathsf{end}
\end{array}
$$

Here $x : \Sigma$ are the instance variables which carry the object state. The *class constructor* C is used to instantiate objects and, semantically, is equivalent to a stand-alone global procedure which is associated with the class for encapsulation reasons. The statement K, representing the body of the constructor, introduces the instance variables into the state space and initializes them using the value parameter(s) $x_0 : \Gamma_0$. Methods named $m_1, \ldots, m_n$ are specified by statements $M_1, \ldots, M_n$ respectively. Invocation of a method on an object has a standard procedure call semantics, with the only difference that the value of the object's state is implicitly passed as a value-result argument. We will denote invocation of m_i on c with value and result arguments $v : \Gamma_i$ and $r : \Delta_i$ by $c.m_i(v, r)$.

Partitioning Methods into Test Cases Our approach to testing is based on a specification, therefore the method body definition M_i of a method m_i can be viewed as a specification of the form $\{p\}; S$, where $\{p\}$ is an assertion of the precondition of the method, and S is the body of the method describing the functionality that will be carried out if the precondition is met. The body S can be of the form $[Q]$, making the method specification into an abstract statement corresponding to a pre-postcondition specification, or it can be a sequential composition of executable and specification statements intermixed with method calls.

Using standard partitioning techniques (for example, the DNF-based approach) we can split the precondition p into a set of $p_1, \ldots, p_k$, such that $p_1 \cup \ldots \cup p_k = p$ and also such that for any pair of preconditions p_i and p_j, we have $p_i \cap p_j = \emptyset$. Using the definitions of refinement and of angelic choice (see section 2.2), we obtain that:

$$\{p\}; S \ = \ \{p_1\}; S \sqcup \ldots \sqcup \{p_k\}; S$$

and also

$$\{p\}; S \ = \ \{p_1\}; \{p\}; S \sqcup \ldots \sqcup \{p_k\}; \{p\}; S$$

because for any p_i and p, such that $p_i \subseteq p$, it is true that $\{p_i\}; \{p\} \ = \ \{p_i\}$.

When used in the context of method calls, this gives us for a method

$$m_i\,(\mathsf{val}\ x_i : \mathit{\Gamma}_i, \mathsf{res}\ y_i : \mathit{\Delta}_i) \;=\; \{p_i\}; S_i$$

the following formula:

$$c.m_i(v,r) \;=\; \{p'_1\}; c.m_i(v,r) \sqcup \ldots \sqcup \{p'_k\}; c.m_i(v,r)$$

where $p'_1, \ldots p'_k$ are the partitions of p_i adjusted to state the conditions on the method parameters of m_i, if any, and on the attributes of the object c.

Uniformity hypotheses which divide the domain $\mathit{\Gamma}_i$ into n subdomains $\mathit{\Gamma}_i^k$, lead to n test cases with

$$\{p_k\} \;=\; (x_i : \mathit{\Gamma}_i^k)$$

.

We can now define the test cases class, which aggregates an object c and contains as methods $t_1, \ldots, t_p$ all the test cases:

$$
\begin{array}{l}
\mathit{CTest} \;=\; \mathsf{class} \\
\quad c : C \\
\quad \mathit{CTest}\,(\mathsf{val}\ x_0 : \mathit{\Gamma}_0) \;=\; K', \\
\quad t_1\,(\mathsf{val}\ x_1 : \mathit{\Gamma}_1, \mathsf{res}\ y_1 : \mathit{\Delta}_1) \;=\; \{p'_1\}; c.m_1(x_1, y_1), \\
\quad \ldots \\
\quad t_k\,(\mathsf{val}\ x_1 : \mathit{\Gamma}_1, \mathsf{res}\ y_1 : \mathit{\Delta}_1) \;=\; \{p'_k\}; c.m_1(x_1, y_1), \\
\quad \ldots \\
\quad t_m\,(\mathsf{val}\ x_n : \mathit{\Gamma}_n, \mathsf{res}\ y_n : \mathit{\Delta}_n) \;=\; \{p'_m\}; c.m_n(x_n, y_n), \\
\quad \ldots \\
\quad t_p\,(\mathsf{val}\ x_n : \mathit{\Gamma}_n, \mathsf{res}\ y_n : \mathit{\Delta}_n) \;=\; \{p'_p\}; c.m_n(x_n, y_n) \\
\mathsf{end}
\end{array}
$$

Partition analysis ensures that any invocation of a method is covered by one of the subparts. Thus if we partition each method, our TCC covers any invocation specified in the SCUT.

4.3 Example: Collections

To illustrate the application of our approach in practice, we use a specification of the Collection interface from the Java Collections Framework (JCF) which is a part of the standard Java Development Kit 2.0.

Specifying Java Collections Framework As was stated in the description of JCF [9], "A collections framework is a unified architecture for representing and manipulating collections." This particular framework contains three parts: interfaces, implementations, and algorithms. The interfaces at the core of JCF form a hierarchy, whose root, the *Collection* interface, represents a group of objects, known as its elements. *Collection* is used to pass collections around and manipulate them when maximum generality is desired. Some *Collection* specializations allow duplicate elements and others do not; some are ordered and others are not.

To specify the behavior of *Collection* methods we model the underlying data structure the methods operate on by a bag (multiset) of *Object*[1] elements, as we want the collection to contain polymorphic elements, possibly duplicated or unordered. Furthermore, to specify the history of structural modifications, we will use an integer attribute *modified* which will be increased whenever elements are added to the original collection or removed from it. For the purposes of this exposition, we only present here the specifications of the data attributes, the constructor, and the basic operations of *Collection*. The complete specification of *Collection* can be found in [22, 23].

```
public interface Collection
   bag of Object elems;
   int modified;

   Collection() {
      elems, modified = ⟦⟧, 0;
   }

   boolean isEmpty() {
      res (#elems == 0);
   }

   boolean contains(Object o) {
      res (o ∈ elems);
   }

   boolean add(Object o) {
      boolean r | r == false;
      if (o ∈ elems){
         choose {skip; }
         or {elems, modified, r =
            elems + ⟦o⟧, modified + 1, true; };
      }
      else {elems, modified, r =
         elems + ⟦o⟧, modified + 1, true; };
      res r;
   }

   boolean remove(Object o) {
      boolean r | r == false;
      if (o ∈ elems){
         elems, modified =
            elems \ o, modified + 1;
         r = true;
      }
      res r;
   }

   Iterator iterator() {
      Iterator i = new Iterator(this);
      res i;
   }

   boolean containsAll(Collection c) {
      assert c != null;
      res c.elems ⊆ elems;
   }

end
```

In this specification highlighted in bold is the original *Collection* interface and the rest is the precise description of the intended behavior. The behavior of the constructor and the methods is specified in terms of operations on bags and integers, with # returning the number of elements in a bag, $\in, +$, and $\setminus$ representing containment of an element in a bag, bag summation, and element removal respectively.

[1] In Java the standard class *Object* is a superclass of all other classes, and a variable of type *Object* can hold a reference to an object of any other type.

The statement assert g, where g is a state predicate, is syntactic sugar for the assertion $\{g\}$. We use this notation in specifications of Java programs to avoid confusion, since $\{\}$ are used in Java for other purposes. Finally, the statement choose S_1 or $\ldots$ or S_n, used in the specification of the method *add*, represents a demonic nondeterministic choice $S_1 \sqcap \ldots \sqcap S_n$. In specifications of Java programs we use the syntax choose S_1 or $\ldots$ or S_n because we believe that it improves readability.

Creating a Testing Class As mentioned above, there are numerous standard techniques for partitioning the input state space of methods. Our partitioning strategy is based on case distinctions in the local state of *Collection* instances and method input parameters, if any. We create a testing class *TestCollection*, aggregate within it an instance *col* of *Collection*, and add to this class the partitioned methods, which redirect the requests for a method invocation to the corresponding method of *col*:

```
class TestCollection {
  Collection col;
  TestCollection() {
    col = new Collection();
  }
  boolean isEmpty1() {
    assert #col.elems = 0; res col.isEmpty();
  }
  boolean isEmpty2() {
    assert #col.elems > 0; res col.isEmpty();
  }
  boolean contains1(Object o) {
    assert #col.elems = 0;
    res col.contains(o);
  }
  boolean contains2(Object o) {
    assert #col.elems > 0 ∧ o ∈ col.elems;
    res col.contains(o);
  }
  boolean contains3(Object o) {
    assert #col.elems > 0 ∧ o ∉ col.elems;
    res col.contains(o);
  }
  boolean add1(Object o) {
    assert #col.elems = 0;
    res col.add(o);
  }
  boolean add2(Object o) {
    assert #col.elems > 0 ∧ o ∈ col.elems;
    res col.add(o);
  }
  boolean add3(Object o) {
    assert #col.elems > 0 ∧ o ∉ col.elems;
    res col.add(o);
  }
  boolean remove1(Object o) {
    assert #col.elems = 0;
    res col.remove(o);
  }
  boolean remove2(Object o) {
    assert #col.elems > 0 ∧ o ∈ col.elems;
    res col.remove(o);
  }
  boolean remove3(Object o) {
    assert #col.elems > 0 ∧ o ∉ col.elems;
    res col.remove(o);
  }
  Iterator iterator() {
    assert true;
    res col.iterator();
  }
```

```
    boolean containsAll_1(Collection c) {        boolean containsAll_2(Collection c) {
      assert c != null ∧ #col.elems = 0;           assert c != null ∧ #col.elems > 0;
      res col.containsAll(c);                      res col.containsAll(c);
    }                                            }
}
```

As the number of elements is a bag is always non-negative, we consider two distinct cases on $col.elems$, namely $\#col.elems = 0$ and $\#col.elems > 0$. When the bag is non-empty, we also consider two distinct cases, with the input parameter o beloning to the bag ($o \in col.elems$) and not belonging to it ($o \notin col.elems$).

Note that in the assertion clauses of the methods we directly access the instance variable *elems* of the aggregated *Collection* instance. We choose to include this direct access in the method specifications mostly for illustration and documentation purposes, to indicate the precondition of a particular method. Strictly speaking, such a direct access, in general, not only breaks encapsulation, but also simply cannot occur in practice, as the internal attributes of a class implementing a certian interface can and will probably be completely different from those that we would use in the specification of this interface.

In practice, such an assertion clause can either be commented away or (in languages supporting assertions) replaced with a call to a corresponding instance variable accessing method of *Collection* or of the class implementing it.

5 Generic Object Client as a Test Driver

In the previous section, we have defined a class that contains the test cases of our CUT. Now we have to describe how the tester can manipulate these test cases. For this purpose, we define a *test driver*. As was mentioned above, the behaviour of an individual method in a class cannot be considered and tested independently of the object associated with this method. Therefore, a test driver, whatever form it might take, has to create and intialize an object and take it to a corresponding state in order to test a method.

We define a test driver as a client (program) of the testing class TCC, or more precisely as a client of its objects. We are interested in a generalised client which performs not a specific sequence of method calls on an object, but is able to perform *all possible combinations* of method calls. Such a *generic object client* should be able to choose which methods to invoke, and how many times. Any concrete client invoking a particular method or a sequence of methods is then an instance (a special case) of the generic object client. Let us now explain how a test driver, which in our case is represented by the generic object client, can be formalised. The following description is adopted from [3], where the generic client object was originally presented.

5.1 Formalizing a generic object client

A client program using an object c instantiated by class C does so by invoking its methods. Every time the client has a choice of which method to choose for execution. In general, each option is preceded with an assertion which determines whether the option is enabled in a particular state. While at least one of the assertions holds, the client may repeatedly choose a particular option which is enabled and have it executed. The client decides on its own when it is willing to stop choosing options. Such an iterative choice of method invocations, followed by arbitrary statements not affecting the object state directly, describes all the actions a client program might undertake and is the *generic object client*:

$$\mathsf{begin\ var}\ l : \Lambda \bullet p; \mathsf{do}\ q_1 :: c.m_1(g_1, d_1); L_1 \between \ldots \between q_n :: c.m_n(g_n, d_n); L_n\ \mathsf{od\ end}$$

Here the construct inside the keywords do .. od is the iterative choice statement. Variables $l : \Lambda$ are some local variables initialized according to p, predicates $q_1 \ldots q_n$ are the asserted conditions on the state, and statements L_1 through L_n are arbitrary. The whole program statement is a contract between the object c and any client program using c. The method enabledness condition q_i corresponds to the assumptions made by the corresponding method m_i, as stated in its subcontract (the method body definition).

5.2 Example: Creating a Test Driver

A test driver is made to be a client of *TestCollection*. As presented above, *TestCollection* is a testing class for *Collection*, aggregating its instance and with methods corresponding to the methods of *Collection* partitioned on their input state spaces. We call the test driver *TestCollectionDriver* and specify it as follows:

$$
\begin{array}{l}
\mathit{TestCollectionDriver}\ =\ \{\\
\quad \mathit{tcol} = \mathsf{new}\ \mathit{TestCollection}();\\
\quad \mathit{int}\ s; \mathit{boolean}\ b; \mathit{Object}\ o_1, o_2; \mathit{Iterator}\ i; \mathit{Collection}\ c \mid o_1\ != o_2 \wedge c\ != \mathsf{nil};\\
\quad\ \between\ \mathit{tcol}.\#\mathit{elems} = 0 :: b = \mathit{tcol}.\mathit{isEmpty}_1()\\
\quad\ \between\ \mathit{tcol}.\#\mathit{elems} > 0 :: b = \mathit{tcol}.\mathit{isEmpty}_2()\\
\quad\ \between\ \mathit{tcol}.\#\mathit{elems} = 0 :: b = \mathit{tcol}.\mathit{contains}_1(o_1)\\
\quad\ \between\ \mathit{tcol}.\#\mathit{elems} > 0 \wedge o \in \mathit{elems} :: b = \mathit{tcol}.\mathit{contains}_2(o_1)\\
\quad\ \between\ \mathit{tcol}.\#\mathit{elems} > 0 \wedge o \notin \mathit{elems} :: b = \mathit{tcol}.\mathit{contains}_3(o_1)\\
\quad\ \between\ \mathit{tcol}.\#\mathit{elems} = 0 :: b = \mathit{tcol}.\mathit{add}_1(o_1)\\
\quad\ \between\ \mathit{tcol}.\#\mathit{elems} > 0 \wedge o \in \mathit{elems} :: b = \mathit{tcol}.\mathit{add}_2(o_1)\\
\quad\ \between\ \mathit{tcol}.\#\mathit{elems} > 0 \wedge o \notin \mathit{elems} :: b = \mathit{tcol}.\mathit{add}_3(o_1)\\
\quad\ \between\ \mathit{tcol}.\#\mathit{elems} = 0 :: b = \mathit{tcol}.\mathit{remove}_1(o_1)\\
\quad\ \between\ \mathit{tcol}.\#\mathit{elems} > 0 \wedge o \in \mathit{elems} :: b = \mathit{tcol}.\mathit{remove}_2(o_1)\\
\quad\ \between\ \mathit{tcol}.\#\mathit{elems} > 0 \wedge o \notin \mathit{elems} :: b = \mathit{tcol}.\mathit{remove}_3(o_1)\\
\quad\ \between\ \mathit{true} :: i = \mathit{tcol}.\mathit{Iterator}()\\
\quad\ \between\ c\ != \mathsf{nil} \wedge \#\mathit{elems} = 0 :: b = \mathit{tcol}.\mathit{containsAll}_1(c)\\
\quad\ \between\ c\ != \mathsf{nil} \wedge \#\mathit{elems} > 0 :: b = \mathit{tcol}.\mathit{containsAll}_2(c)\\
\quad \mathsf{od}\\
\}
\end{array}
$$

Note that the test driver is capable of accessing the internal instance variables of the object *tcol*. Thus the guards preceeding invocations of methods on *tcol* refer to *elems*, stating the conditions with respect to which the methods of *TestCollection* were originally partitioned. This encapsulation breaking is needed only at this abstract level and is not carried over to scenarios.

6 Test Scenarios and Oracles

6.1 From scenarios of usage to test scenarios

In the previous section we have defined test cases as method calls, and then we have described how to built a test driver from these test cases to run them. However it is impossible to run a test case individually: a scenario must be defined to create a new implementation to test and to obtain a state space where the test case can be invoked. Thus it appears to be beneficial to base a testing method on scenarios of usage. The scenarios of usage are essentially sequences of method calls on the object under test and possibly on other objects, taking the system through a sequence of method calls from one state to another.

Test scenarios can be defined for different testing levels. At unit level (to test in detail the behavior of a class) we can base the generation of scenarios on *use cases* derived at the modelling stage or on sequences of method calls known to be error-prone from experience. At integration level (to test interactions between several objects) we can use *sequence diagrams*.

Apart from being based on scenarios of usage, test scenarios can also be systematically derived using various classical techniques, such as breadth-first search or depth-first search.

Various test coverages can be derived, determined by the strategy for combining method calls into scenarios. It is not our primary focus in this work to look closely at the testing strategies; this topic deserves close consideration and represents the subject of current research. Yet, we would like to present here some interesting coverage criteria, for illustrative purposes.

From the test case class (where methods are partitionned), we generate a set of correct scenarios (they are possible models of the specification) such that:

- **each scenario focuses on a test case**: for each test case, one scenario with the minimal number of method calls leading to the state satisfying the precondition of this method, followed by the method invocation;
- **all possible scenarios reaching a particular test case**, for each test case;
- **all possible scenarios between two test cases of interest**: all scenarios leading from one method to the other;
- **all possible scenarios**: all possible combinations of test cases.

Let us now present the formal definition of a test scenario and present rules for calculating correct scenarios.

6.2 Formalization and Correctness Issues

In Sec. 3.3 we have defined a test scenario as a sequence of test cases. In practice, it must be a terminating sequential composition of an initialisation, including object creation, followed by one or more method calls on this object. Initialisation can include initialisations of variables passed as input arguments to the object's methods.

When calculating scenarios of client interaction with an object, we should always start with object creation, followed by one or more method invocations on this object. Here we are interested in building correct sequences, indeed, possible sequences with respect to the specification. These sequences must be abstractions of the iterative choice of test cases, excluding abortions. More precisely, the preconditions of a method call must be etablished by the previous method call in the sequence. These requirements are expressed by the following rules which allow us to calculate incrementally some scenarios:

$$
\begin{aligned}
&\mathit{true} \subseteq C.g_i \quad \Rightarrow \quad C; \mathsf{do}\ \between_{i=1}^{n} g_i :: S_i\ \mathsf{od}\ \sqsupseteq_T\ C; \{g_i\}; S_i && (1)\\
&g \subseteq S.g_i\ \wedge\ C; \mathsf{do}\ \between_{i=1}^{n} g_i :: S_i\ \mathsf{od}\ \sqsupseteq_T\ C; \{g\}; S \quad \Rightarrow && (2)\\
&\qquad C; \mathsf{do}\ \between_{i=1}^{n} g_i :: S_i\ \mathsf{od}\ \sqsupseteq_T\ C; \{g\}; S; \{g_i\}; S_i
\end{aligned}
$$

Here C stands for initialisation of the state, including object creation which would typically involve invoking the corresponding class constructor; S is a sequence of one or more method calls on the object under test.

The first rule is a base case which allows us to construct scenarios of the form $C; \{g_i\}; S_i$ where initialisation is followed by one method invocation. The second rule is a step case which permits us composing a new scenario from an exsiting one through sequentially composing with it one more method invocation.

Note that, as true assertions can always be refined away, the rules (1) and (2) can also be stated without the assertions explicitly present, as follows:

$$
\begin{aligned}
&\mathit{true} \subseteq C.g_i \quad \Rightarrow \quad C; S_i\ \sqsubseteq_T\ C; \mathsf{do}\ \between_{i=1}^{n} g_i :: S_i\ \mathsf{od} && (3)\\
&g \subseteq S.g_i\ \wedge\ C; S\ \sqsubseteq_T\ C; \mathsf{do}\ \between_{i=1}^{n} g_i :: S_i\ \mathsf{od} \quad \Rightarrow && (4)\\
&\qquad C; S; S_i\ \sqsubseteq_T\ C; \mathsf{do}\ \between_{i=1}^{n} g_i :: S_i\ \mathsf{od}
\end{aligned}
$$

Effectively, the presented rules reduce the problem of calculating a valid scenario to checking whether two subsequent method calls in a scenario are composable. As this check requires, at most, a limited amount of theorem proving in (typically) propositional logic, it can be automated, requiring human intervention mostly for optimisation of performance.

6.3 Example: Calculating Scenarios for Testing Collections

Let us demonstrate now how the rules (1)-(2) allow us to calculate scenarios for our example. As an example, we choose to employ the testing strategy creating one scenario consisting of a partitioned method. We will derive scenarios that for each partitioned method use the minimal number of method calls establishing the precondition of this method. The creation and the following initialisation of a *tcol* object results in the *elems* attribute of this object being assigned an empty bag. Obviously, only the methods having the condition $tcol.\#elems = 0$ as the precondition can be sequentially composed with the initialisation. Accordingly, we get the following (elementary) scenarios testing individual methods $isEmpty_1$, $contains_1$, add_1, $remove_1$, $Iterator$, and $containsAll_1$:

1. $tcol =$ new $TestCollection()$; assert $tcol.\#elems = 0; b = tcol.isEmpty_1()$
2. $tcol =$ new $TestCollection()$; assert $tcol.\#elems = 0; b = tcol.contains_1(o_1)$
3. $tcol =$ new $TestCollection()$; assert $tcol.\#elems = 0; b = tcol.add_1(o_1)$
4. $tcol =$ new $TestCollection()$; assert $tcol.\#elems = 0; b = tcol.remove_1(o_1)$
5. $tcol =$ new $TestCollection()$; assert $tcol.\#elems = 0; i = tcol.Iterator()$
6. $c =$ new $TestCollection()$; $tcol =$ new $TestCollection()$;
   assert $c! =$ nil $\wedge\ tcol.\#elems = 0; b = tcol.containsAll_1(c)$

For clarity, we keep the assertions in the scenarios although these assertions refer to the internal attributes of the object under test, thus breaking encapsulation. In practice, these assertions would typically be removed while testing.

Scenario 3 produced a new postcondition $tcol.\#elems > 0 \wedge o_1 \in tcol.elems$, and therefore we can generate new scenarios by sequentially composing with it the methods those precondition is satisfied by this condition:

1.1. $tcol =$ new $TestCollection()$; assert $tcol.\#elems = 0; b = tcol.add_1(o_1)$;
     assert $tcol.\#elems > 0; b = tcol.isEmpty_2()$
1.2. $tcol =$ new $TestCollection()$; assert $tcol.\#elems = 0; b = tcol.add_1(o_1)$;
     assert $tcol.\#elems > 0 \wedge o_1 \in tcol.elems; b = tcol.contains_2(o_1)$
1.3. $tcol =$ new $TestCollection()$; assert $tcol.\#elems = 0; b = tcol.add_1(o_1)$;
     assert $tcol.\#elems > 0 \wedge o_2 \notin tcol.elems; b = tcol.contains_3(o_2)$
1.4. $tcol =$ new $TestCollection()$; assert $tcol.\#elems = 0; b = tcol.add_1(o_1)$;
     assert $tcol.\#elems > 0 \wedge o_1 \in tcol.elems; b = tcol.add_2(o_1)$
1.5. $tcol =$ new $TestCollection()$; assert $tcol.\#elems = 0; b = tcol.add_1(o_1)$;
     assert $tcol.\#elems > 0 \wedge o_2 \notin tcol.elems; b = tcol.add_3(o_2)$
1.6. $tcol =$ new $TestCollection()$; assert $tcol.\#elems = 0; b = tcol.add_1(o_1)$;
     assert $tcol.\#elems > 0 \wedge o_1 \in tcol.elems; b = tcol.remove_2(o_1)$
1.7. $tcol =$ new $TestCollection()$; assert $tcol.\#elems = 0; b = tcol.add_1(o_1)$;
     assert $tcol.\#elems > 0 \wedge o_2 \notin tcol.elems; b = tcol.remove_3(o_2)$
1.8. $c =$ new $TestCollection()$; $tcol =$ new $TestCollection()$;
     assert $c! =$ nil $\wedge\ tcol.\#elems > 0; b = tcol.containsAll_2(c)$

6.4 Class Specifications as Oracles

One of the widely acknowledged benefits of specificatiopn-based testing is the possibility of using formal specifications as test oracles for calculating the expected test results. Typically, in testing methods based on pre- and postcondition specifications, the postcondition plays the role of an oracle. In our approach, class specification is used as the oracle and the expected outcome of every method is calculated as the strongest postcondition of the method's definition with respect to the enabledness predicate of this method. An expected outcome of a scenario is calculated as the strongest postcondition of this scenario with respect to the initial precondition.

7 Conclusion

In this work, we propose a scenario-based method for testing object-oriented programs. We provide a mathematical foundation for this method, supporting the selection of test scenarios based on the specifications of class behaviour. We define a test driver as a generic object client capable of running all test cases in any order, any number of times; and we define a test scenario as an abstraction of the test driver. Finally, method specifications are used as test oracles for calculating the expected scenario results. For this, the strongest postcondition of the scenario with respect to the input predicate of the scenario is calculated.

Such a testing method provides a mathematical support to the current practice of scenario-based testing adopted by the industry. Besides, it improves non-regression testing and allows for easy adaptation. Apart from testing the behaviour of individual classes (using unit testing and integration testing), object-oriented programs require testing behavioural compatibility of classes and their subclasses, sometimes refered to as *incremental testing*. Although we do not specifically consider the problem of incremental testing, it is partially addressed in our method, because scenarios can be used in the same manner to test subclasses as they are used to test the original classes. For this, sequences of method calls are invoked on subclass objects instead of superclass objects and the results are verified against the same expected outcome.

The generation of test cases and the generation of a generic client do not require human intuition and guidance and, therefore, can be fully automated. The scenario generation is a human-directed process, based on the tester's experience, expertise, and a particular scenario-generating strategy relevant for a testing task at hand. To generate valid scenarios, the tester can start with creating an object and then further searching for methods calls to which could be included in the scenario leading to his test goal. Scenario generating rules that we provide reduce the problem of calculating a conformance scenario to checking whether two subsequent method calls in a scenario are composable. As this check requires, at most, a limited amount of theorem proving in (typically) propositional logic, it can also be automated, requiring human intervention mostly for optimisation of performance. In particular, a strongest postcondition of a particular method can turn out to be stronger than preconditions of some (partitioned) methods and thus could potentially be sequentially conjoined with them. But to verify that it is indeed stronger, a certain amount of theorem proving would be required.

We have not focused in this work on developing particular test scenario generating strategies, listing just a few for illustration purposes. This topic is an important and under-developed issue and represents the subject of current research.

References

1. B. K. Aichernig. Test-case calculation through abstraction. In *Formal Methods Europe 2001*, volume ? of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
2. R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.
3. R. J. R. Back, A. Mikhajlova, and J. von Wright. Reasoning about interactive systems. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of the World Congress on Formal Methods (FM'99)*, volume 1709 of *LNCS*, pages 1460–1476. Springer-Verlag, September 1999.
4. R. J. R. Back and J. von Wright. Contracts, games and refinement. In *4th Workshop on Expressiveness in Concurrency, EXPRESS'97*, volume 7 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1997.
5. R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, April 1998.
6. S. Barbey, D. Buchs, M.-C. Gaudel, B. Marre, C. Péraire, P. Thévenod-Fosse, and H. Waeselynck. From requirements to tests via object-oriented design. In *Published in DeVa Third Year Report*, pages 331–384, Dec. 1998.
7. B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.
8. R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison-Wesley, 1999.
9. J. Bloch. Java collections framework: Collections 1.2. `http://java.sun.com/docs/books/tutorial/collections/index.html`.
10. E. Brinksma. Formal methods for conformance testing: Theory can be practical. In *Proc. 11th International Computer Aided Verification Conference*, volume 1633 of *LNCS*, pages 44–46. Springer-Verlag, July 1999.
11. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 268–284. Formal Methods Europe, Springer-Verlag, April 1993. Lecture Notes in Computer Science 670.
12. E. W. Dijkstra. *A Discipline of Programming*. Prentice–Hall International, 1976.
13. L. du Bousquet. Feature interaction detection using testing and model-checking: Experience report. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99—Formal Methods, Volume I*, volume 1708 of *LNCS*, pages 622–641. Springer, 1999.
14. L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: A specification-driven testing environment for synchronous software. In *Proceedings of the 21st International Conference on Software Engineering (ICSE-99)*, pages 267–276, NY, May 1999. ACM Press.
15. R. Fletcher and A. S. M. Sajeev. A framework for testing object-oriented software using formal specifications. *Lecture Notes in Computer Science*, 1088:159–??, 1996.
16. M.-C. Gaudel. Testing can be formal, too. *TAPSOFT'95*, 915:82–96, 1995.

17. H.-M. Hoercher. Improving software tests using Z specifications. *Lecture Notes in Computer Science*, 967:152–??, 1995.
18. T. Jeron and P. Morel. Test generation derived from model-checking. In *Proc. 11th International Computer Aided Verification Conference*, pages 108–121, 1999.
19. S. H. Kirani and W.-T. Tsai. Method sequence specification and verification of classes. *Journal of Object-Oriented Programming*, 7(6):28–38, Oct. 1994.
20. R. ko Doong and P. G. Frankl. The ASTOOT approch to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, Apr. 1994.
21. J. D. McGregor and T. D. Korson. Integrated object-oriented testing and development process. *Communications of the ACM*, 37(9):59–77, Sept. 1994.
22. A. Mikhajlova. *Ensuring Correctness of Object and Component Systems.* PhD thesis, Åbo Akademi University, October 1999. http://www.tucs.abo.fi/publications/dissertations/.
23. A. Mikhajlova. Combining code with specifications: How to document and verify frameworks. *Special Issue of L'Objet on Formal Methods for Object Systems*, 6(1), 2000.
24. C. C. Morgan. *Programming from Specifications.* Prentice–Hall, 1990.
25. J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.
26. A. Morzenti, D. Mandrioli, and C. Ghezzi. A model parametric real-time logic. *ACM Transactions on Programming Languages and Systems*, 14(4):521–573, Oct. 1992.
27. L. Murray, D. Carrington, I. MacColl, J. McDonald, and P. Strooper. Formal derivation of finite state machines for class testing. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation, 11th International Conference of Z Users, Berlin, Germany, 24–26 September 1998*, volume 1493 of *Lecture Notes in Computer Science*, pages 42–59. Springer-Verlag, 1998.
28. D. J. Richardson and L. A. Clarke. Partition analysis: A method combining testing and verification. *IEEE trans. on Software Engineering*, SE-11, 12:1477–1490, 1985.
29. P. A. Stocks and D. A. Carrington. Test templates: A specification-based testing framework. In E. Straub, editor, *Proceedings of the 15th International Conference on Software Engineering*, pages 405–414, Baltimore, MY, May 1993. IEEE Computer Society Press.
30. J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Dec. 1992.
31. C. D. Turner and D. J. Robson. The Testing of Object-Oriented Programs. Technical Report TR-13/92, School of Engineering and Computer Science, University of Durham, Durham, GB, Nov. 1992.