Software Testing: 
Testing new Software Paradigms and new Artefacts

Henry Muccini
Dipartimento di Informatica
University of L’Aquila
Via Vetoio, 1 – L’Aquila, Italy
muccini@di.univaq.it

Abstract. Software testing research is continuously moving towards different challenges and obstacles driven by different forces. What we believe are two main forces driving software testing evolution are the progress in software development paradigms and languages, and the artefacts along the software development process to be used for testing purposes.

Based on this specific perspective, this work wants to provide an ample presentation of issues and assessments related to testing nowadays software. Ongoing and (expected) future research directions on software testing are also outlined.

Keywords: Software testing, code-based testing, object-oriented testing, component-based testing, software architecture-based testing, specification-based testing, formal testing, model-based testing, MDA testing, testing software product lines, testing service-oriented systems.

1 Introduction to the Chapter

The purpose of software testing is to improve the quality of the software product by detecting and removing as many failures as possible, thus increasing the developer confidence in the proper functioning of the software. For this purpose, software testing is not meant to be an exhaustive technique for software verification and validation. It can show that software defects are present, while not being able to show their absence [1].

Software testing is centered on the concept of selecting some of the (fault enabling) inputs from a possibly infinite input domain, to perform system runs driven by such inputs, and to compare the expected results with the real ones. If expected and observed behaviours differ, a failure is manifested due to a system fault which needs to be fixed.

A successful test is the one that uncovers undiscovered errors.

By taking a look at the evolution of software testing in the past decades, we can recognize that two (among the others) can be considered the main sources of advances in software testing: i) testing techniques have been applied over new software development paradigms, languages, and applications, and ii) testing techniques have been applied over new artefacts (other than the source code).

This chapter wants to present current testing techniques along those two directions.
Testing new Software Paradigms: Over the course of the past fifty years, the way software has been produced is greatly changed. We have moved from procedural code, towards object-oriented systems, to component-based development (as pictorially described in Figure 1); from thousands to millions of lines of code with real-time, reliability, safety, and performance requirements. Software testing has had to change accordingly. Traditional techniques, adopted for testing procedural code, had to be extended for testing object-oriented (OO) software: inheritance, polymorphism, dynamic binding and other OO characteristics required new testing features. With the introduction of component-based systems, component-based testing has been introduced for testing the components in isolation or the assembly.

![Figure 1: From Procedural programming to Component-based development](image)

Testing new Artefacts: In traditional approaches to software testing, specific methodologies are used to select test cases based on the source code of the program to be tested [2]. The main practical drawback related to (purely) source code testing is that since the code is produced at the latest step in the software production process, testing activities are left to the end of the software life cycle. In consequence, schedule slippage, time-to-market pressures, and cost-constraints result in neglected testing.

Nowadays, source code is no longer the single source for selecting test cases and we can apply testing techniques all along the development process, by basing test selection on different pre-code artefacts [3]; the test selection phase can be based on system specifications (formally or informally defined), on architectural high-level design, or on component-based or object-oriented specifications.

All these testing techniques deserve consideration. Recent studies have shown that different test selection techniques target different classes of faults (e.g., [4]), and that the combined use of diverse techniques is always more effective than concentrating the effort on only one technique (even though proved to be the most efficacious) [5].

This chapter: The goal of this chapter is then to provide a view on software testing techniques from this specific perspective, as graphically described in Figure 2. It discusses code-based testing (Section 2) with a specific focus on testing object-oriented software (Section 2.1). It covers component-based testing (Section 3), distinguishing
between component-testing and component-based testing. It analyzes specification-based
testing (Section 4) (with particular attention to formal testing and model-based testing,
and software-architecture-based testing).

![Diagram](image_url)

Fig. 2. Testing new Software Paradigms and new Artefacts

The chapter wants to provide the most comprehensive treatment of the testing sub-
ject from the specific perspective presented above, describing the state of the art and
providing references for further readings and expected directions for future work. Being
the chapter oriented to a wide audience of readers, a glossary of testing terms has been
provided in Appendix A to facilitate the reading to inexperienced readers. Being the
study limited in size, references are provided to those best sources an interested reader
may refer to for further detailing the concepts outlined in this work. The focus has also
been oriented to mature research, to practices of current use, and to current trends, while
limiting the discussion on the latest speculative research that, while promising, might
never work out. Moreover, this chapter focuses on systematic testing techniques, while
not considering purely experience-driven, not repeatable, testing techniques.

The remaining sections are structured so to provide a Problem Statement, informa-
tion on the main Achievements, links to Tool Support, and references to Further
Readings.

Further Readings: M.J. Harrold [6] (Conference paper, 2000): the ICSE 2000 Fu-
ture of Software Engineering track paper on Software Testing. It provides a clear and
wide roadmap on fundamental research topics to be further investigated in order to ob-
tain practical testing methods, tools, and processes to develop high-quality software. A.
track paper on Software Testing. It provides a roadmap which structures and classifies
software testing into three main aspects: achievements (where we are today), dreams
(where researchers would like to go), and challenges (as a way to move from achieve-
Testing and Analysis, covering both the basic concepts and techniques, testing methods,
and processes. ISSTA [9], ICST [10], TESTCOM [11] (Conferences): among the main
conferences on software testing.
2 Code-based Testing: testing the source code

Problem Statement: Code-based testing (also known as structural testing or white box testing) assumes visibility into internal data and structures of the system implementation. Test cases are selected according to the structure of the program and executed in order to exercise program statements.

The code structure is (typically) represented through a graph: traditional white box analysis techniques use a control flow or data flow graph representation of a program. In control flow graphs, nodes correspond to sequentially executed statements while edges represent the flow of control between statements. A data flow graph extends the content of a control flow graph by adding information on variables accessed and modified by program statements.

Achievements: The test selection phase in code-based testing consists of selecting the minimum number of test cases able to cover as much as possible of the flow graph. Designers have been applying white box techniques for a long time and several coverage criteria are available today. Some criteria are based on the control flow, while others on the data flow:

Statement coverage: this criterion reports whether each executable statement is encountered. This coverage selects a test set $T$ such that, by executing a program $P$ for each test in $T$, each elementary statement of $P$ (i.e., each node in the graph) is executed at least once. The chief characteristic of this measure is that it provides the weakest coverage: if compared with other coverage criteria (see Figure 3) statement coverage is subsumed by any of the structural coverage criteria.

Branch coverage: it measures the coverage of all blocks and case statements that affect the control flow. Boolean expressions are evaluated for both true and false conditions. This criterion selects a test set $T$ such that, by executing $P$ for each test in $T$, each of $P$’s control flow graph is traversed at least once (i.e., each branch of the control flow graph has to be executed by at least one test case). This measure has the advantage of simplicity, but may ignore branches within Boolean expressions (see condition coverage below), or relevant combinations of branches (considered in path-based criteria).

Condition coverage: it measures the subexpressions independently of each other, allowing for a better analysis of the control flow. It forces the exploration of possible conditions of a boolean expression in a branch, covering different combinations of the individual conditions in a compound boolean expression. This coverage criterion selects a test set $T$ such that, for each test in $T$, each edge of $P$’s control flow is traversed at least once and all possible values of the constituents of compound conditions are exercised at least once. There exist many variations of the condition coverage criterion (see Reference [8] chapter 12 for a detailed explanation). Here it is worth mentioning the modified condition/decision coverage (MCDC), required by many certification agencies: it enhances the condition/decision coverage criterion requiring that each condition independently affecting the outcome of the corresponding decision has to be tested [12]. MCDC is an attractive compromise between number of required test cases and thoroughness of the test, since with a complexity of $n+1^1$ (in the best case), it

---

1 $n+1$ test cases for a decision with $n$ inputs
achieves many of the benefits of the multiple-condition testing which has a complexity of \(2^n\).

Path coverage: it requires all paths to be covered. Path coverage is similar to condition/decision coverage, but it handles multiple sequential decisions. As the number of paths soon becomes unfeasible, several variations of this criterion are considered to limit the number of loops. This technique selects a test set \(T\) such that, executing \(P\) for each test in \(T\), all paths leading from the initial to the final node of the \(P\)'s control flow graph are traversed at least once.

Data-Flow coverage: data-flow testing [13–15] offers a family of criteria for unit testing of programs represented as data-flow graphs. It improves control-flow testing by identifying how the execution of a certain statement can affect the system computation. Data flow testing criteria pair variable definitions with uses, ensuring that each computed value is actually used, and thus selecting from among many execution paths a set that is more likely to propagate the result of erroneous computation to the point of an observable failure [8]. Many are the coverage criteria based on data flow, the most important being: all DU pairs adequacy criterion (it requires each DU pair to be exercised in at least one program execution), all DU paths adequacy criterion (extends the all DU pairs criterion by requiring each simple DU path to be traversed at least once), all definitions adequacy criterion (which requires pairing each definition with at least one use).

Let \(P\) be the following program:

```plaintext
1 function P return INTEGER
2 begin
3    X, Y:INTEGER;
4    READ(X); READ(Y);
5    while (X > 10) loop
6       X := X - 10;
7       exit when X = 10;
8    end loop;
9    if (Y < 20 and then X mod 2 = 0) then
```
The corresponding control flow graph is shown in Figure 4.

![Control Flow Graph](image)

**Fig. 4.** The Control Flow Graph of the P program above

Some structural tests follow:

*All-statement coverage:* Inputs: \((X = 20, Y = 10)\) and \((X = 20, Y = 30)\), where \((X = 20, Y = 10)\) covers nodes <1,2,3,4,5,6,8,9>, and \((X = 20, Y = 30)\) covers nodes <1,2,3,4,5,7,9>.

*All-branch coverage:* Inputs: \((X = 20, Y = 10)\), \((X = 15, Y = 30)\), and \((X = 20, Y = 15)\) where \((X = 20, Y = 10)\) covers nodes <1,2,3,4,5,6,8,9> and branches in <2T,4T,5T,6T>, \((X = 15, Y = 30)\) covers nodes <1,2,3,4,2,5,7,9> and branches in <2T,4F,2F,5F>, and \((X = 20, Y = 15)\) covers nodes <1,2,3,4,5,6,7,9> and branches in <2T,4T,5T,6F>.

It is important to remark that a 100% coverage is typically unfeasible since it may require the execution of unreachable code (referred as “infeasibility” problem).

Many other coverage criteria are used in practice. For more details please refer to [16–19]. How coverage criteria have been adapted to concurrent programs may be found in [20]. An analysis of coverage criteria costs may be found in [21]. A comparison of structural testing criteria can be found in [8].

**Tool support:** As far as concerns tool support, the Open Source Testing Tools [22] web site presents a list of open source testing tools (for white-box testing), while the Software QA Testing and Test Tool Resources [23] site presents a list of software quality assurance and testing tools.

**Further Readings:** M. Pezzé and M. Young [8] (Book, 2007): chapter 12, entitled “Structural Testing”, presents the various structural testing techniques with examples; chapter 13, entitled “Data Flow Testing”, presents theory and examples on data flow testing.
2.1 Testing Object-Oriented Software: testing the source code

Problem Statement: Testing object-oriented (OO) software resembles testing procedural software, where the focus moves from unit, to integration and system testing, and regression testing [24]. However, object-oriented software has some specific characteristics which make testing object-oriented software diverse from other testing strategies and requires specialized techniques. In particular, as remarked in [25] and thoroughly discussed in [8, 26], the unique facets of testing object-oriented software are that each method must be tested in the context of its class and inherited features, objects are stateful and need to be tested in the different states, encapsulation, dynamic binding, inheritance, and exceptions make the system behaviour unpredictable so that an apparently harmless modification may affect different portions of the program.

Achievements: In code-based testing of OO software\(^2\), new criteria are introduced (extending the classical statements, branches, conditions, and decisions coverage) in order to cope with state-dependent behaviour.

When considering unit testing of OO code, a unit is the class, and unit testing aims at testing the functions belonging to a class. As noticed in [8], while an individual method might be considered as a unit, testing a single method can become unpractical, since methods (in a class) interact by modifying object states and affect other methods behaviour. For this reason, unit testing of OO software is usually referred as intra-class testing. When performing intra-class testing of the source code, a graph model (typically, a control flow graph - CFG) is built for each method, and CFGs of the various methods in a class are joined together in order to specify intra-class method calls. This joined graph is usually referred to as a Class Control Flow Graph (CCFG) [27]. CCFGs are then used to compute def-use pairs, by first annotating the CCFG with definitions and uses, and then traversing it to compute all pairs.

When different classes are integrated, inter-class testing checks interactions among objects of different classes. Souter and Pollock [28] define a contextual def-use pair \((d, u)\) of a variable \(o\) to be used in inter-class testing. Buy, Orso and Pezzé [29] combine dataflow analysis with symbolic execution and automatic deduction to generate test cases that satisfy data flow criteria. In [30,31], it is recognized that a representation based on simple CFGs is inadequate for inter-class testing of Java programs. To adequately handle all Java language constructs and features (such as inheritance, polymorphism and dynamic binding, and exception handling), a Java Interclass Graph (JIG) representation is provided. A JIG extends the CFG to handle variable and object type information, internal or external methods, interprocedural interactions through calls to internal or external methods from internal methods, interprocedural interactions through calls to internal methods from external methods, and exception handling. The JIG is used in [30, 31] for regression testing of Java programs.

Data Flow testing criteria for computing definitions and uses of object attributes have been proposed in the context of unit and integration testing of OO software (as discussed in [26]).

\(^2\) While this section will focus on structural testing of OO software, model-base testing techniques will be described in Section 4.
Tool Support: As far as concerns tool support for testing object-oriented code, a recent study on tool support for white-box testing of OO software seems to be missing. Other than the popular JUnit unit testing framework for Java code [32], many other tools (typically, for structural intra class testing of Java or C++ programs) can be found at Reference [33–36].


3 Component-based Testing

Roughly speaking, a component-based software system is an assembly of reusable components, designed to meet the quality attributes identified during the architecting phase [38] while a component is defined as “a unit of composition with contractually specified interfaces and explicit context dependencies only” [39]. Components are specified, designed and implemented with the intention to be reused, and are assembled in various contexts in order to produce a multitude of software systems. The main peculiarities in component-based software development are that components can be white-box or black-box (e.g., if components are Off-the-Shelf, their implementation is not accessible), they can be produced in house or bought from external vendors, they can be written in different programming languages and run in different execution environments, a detailed specification may exist or not.

Problem Statement: Component-based testing consists of two main tasks: testing the individual components (i.e., component testing), and testing the component-based systems built by assembling components.

In component testing, the goal is to detect software errors and validate the quality of the software components considered as the smallest test unit [40]. Testing of components becomes extremely important for the success of the entire CBS, being the quality of the entire system strongly affected by the quality of each single component. In testing component-based systems, instead, the main goal is to test the assembly of heterogeneous components, possibly written in different programming languages, distributed across the network and executed in various platforms.

While many are the issues specific to component testing and testing component-based software (see [41] chapters 2.3 and 4.2, and [42]) two can be considered the most relevant ones, common to both aspects of component-based testing:
Availability of information about components: basic ingredient of any testing strategy is the availability of information/representation describing relevant aspects of the component to be tested. While in traditional software development processes, the organization has a full control on the system under development (from requirements down to implementation), in component-based development the source code is in the general case not available. In the specific context of Commercial-Off-The Shelf (COTS) components, components are retrieved from the market and come to the users without the source code. This lack of information poses particularly difficult questions to the final user for what concerns testing.

Presence of different stakeholders: different stakeholders take place in the development of a component-based system, belonging to even unrelated organizations. Each of these stakeholders has to define and follow a specific testing process while pursuing, in general, quite different objectives. In this paper we mainly differentiate between two main roles: the component developer (who implements software components to be easily and widely reused) and the component user (who integrates the component in a final system).

The component developer starts the development of a component having in mind possible reuse scenarios that will make the component worthy to be bought by a possible user. In the general case, the process adopted will follow a “traditional” approach and the testing process will not show major novelties.

The component user is the developer of a complex system that decided to manage the complexity by reusing existing components. In case of COTS components the unavailability of the source code makes all the traditional code-based techniques not applicable. Moreover the scarcity of information makes also black-box approaches harder to apply.

Therefore developers and users of components are completely different actors from a testing point of view, having the first a complete view of the component internal detail but little information on the final deployment environment, and the second little information on the inner parts of the component but a precise understanding of the final deployment environment.

In summary, component-based testing strongly relies on the information available to any possible stakeholder interested in carrying on a testing campaign on a software component or a set of them. At the same time it is necessary to understand how the available information can be fruitfully used for testing purposes.

Achievements: In the following part of this section, existing component-based testing approaches are classified according to two main criteria: who defines the test to be executed, and which information are sued to define the test cases. Alternative classifications can be found in [43, 41, 44, 45].

Following the proposed classification, three main classes have been identified:

- Developer defined test cases;
- Developer defined component models suitable for analysis and testing derivation by the user;
- User defined test cases.
In developer defined test cases, test cases are defined by the developers and provided as they are to the component user, who can just run them (i.e., the user has no control on the test selection phase).

Built in Test (BIT) approaches and testable architectures-based CB testing are typical examples of approaches in this category. Built in testing is a generic approach to testing where the component is augmented with executable test cases that are built into the component, together with its normal functions. BIT requires component developers to embed tests in software component implementation to support self-testing. By running the embedded test cases, the component user can thus validate in the final environment the hypotheses made by the component developer. Several techniques have implemented this philosophy: Edwards [46] has proposed a framework to provide BIT wrappers for component testing, using the specification language RESOLVE as an example. Wang et al. [47] make use of the BIT for enhancing component-based software maintainability. They build tests in component source code as extra member functions. Other BIT techniques are used in [48] and [49]. More details on BIT approaches can be found in [43, 41].

The testable architectures-based CB testing approach can be seen as a special case of BIT where component developers equip the component with a specific testable architecture that allows the component user to easily execute the test cases. The test information is appended by the developer in the form of specifications, instead of enclosing them in the component itself. One of the most known approaches is the one presented by Momotko and Zalewska [50] which propose a testable architecture based on the Component+ built in testing (C+ BIT) technology for testing component interactions with the environment at run-time.

In developer defined component models suitable for analysis and testing derivation by the user, the component developer provides users with (behavioural) models she can use for test case selection. All forms of additional information appended to the software component, (either by the developer, the user or a third-party tester, so as to simplify software testing), can be regarded as forms of metadata.

Wu et al. [51] define a test-model called CIG (Component Interaction Graph) for integration testing. As a first step, test elements are identified to build a CIG, and test cases are generated for each test element in the CIG. Another interesting approach, likewise relying on the definition of a particular framework for component testing, has been proposed by Atkinson and Gross [52]. The framework is not intended for the execution of generic test cases, but rather focuses on providing the component user with specific test cases derived from contract specifications.

In user defined test cases, the user has full control on the test selection phase. Three are the main approaches that can be assigned to this class: interface probing [53], where the component user derives a set of test cases, executes the component in accordance with them, and analyzes the outputs produced; the Component Deployment Testing (CDT) approach where Bertolino and Polini [54] propose an integration testing framework for easing the execution of test cases derived from user architectural specifications; the Behavior Capture and Test Framework proposed by Mariani and Pezzé [55] where the saving and monitoring of test results in one environment can be useful for evaluating component behaviour in another environment.

4 Specification-based Testing

A software specification describes the expected behaviour and characteristics of a software product as outlined by Poston in his book [57].

In specification-based testing (also called functional testing or black box testing), the internal structure and behaviour of the program is not known (i.e., the program is considered as a black box, since we cannot see what internally happens). What is assumed to have is a model of the system under test and the objective is to analyze whether the system correctly behaves with respect to the specifications, i.e., to find discrepancies between the actual behaviour of the implemented system’s functions and the desired behaviour, as described in the functional specifications. Test sets are derived from the system specification and the test selection phase can start even before the code is available (while test execution still needs the existence of the system implementation).

Manifold are the motivations for making use of a specification-based testing approach: (i) Code unavailability: if source code is not available (not yet developed or not publicly released) test cases can be produced out of a specification. (ii) Scalability: even if source code is available, it becomes rapidly unfeasible to provide complete coverage of complex systems. Being specifications more abstract than source code, specification-based testing is certainly more scalable. (iii) Accuracy: specification-based testing complements code-based testing techniques: while source code testing is typically structural and at a syntactic level (i.e., source code structure coverage), specification-based testing is more functional and at a semantic level (i.e., system functions coverage). (iv) Effectiveness: a specification-based testing plan can start before the code is available, which avoids having to leave the testing activities to the end of the software life cycle, when schedule slippage, time-to-market pressures and cost-constraints may result in neglected testing.

The rest of this section will cover the topics of model-based testing based on formal specifications (Section 4.1) (when the model is generated from a formal specification or the formal specification itself is used as the test input), and model-based testing based on diagrammatic specifications (referring to UML-based testing and testing based on models designed through diagrammatic tools). Section 4.3 presents existing work on Software Architecture-based testing (to be considered as a form of specification-based testing).
4.1 Model-based testing based on Formal Specifications

Problem Statement: A formal specification can be regarded as “the expression in some formal language and at some level of abstraction, of a collection of properties some system should satisfy” [58]. A formal specification language consists of a clearly defined syntax (the notation), fully specified semantics (the specifiable objects), and some satisfies relation (the semantics associated to the syntax). While a formal specification can be of difficult use (if compared with informal specifications), it has a precise, unambiguous semantics, which enables mathematical precision of the analysis of systems and the reasoning about them [59]. Moreover, a specification written in a formal language can be processed by automated tools and formal specification-based testing guarantees higher accuracy, objectivity and repeatability than ad hoc test derivation from informal specifications.

Achievements: Testing from formal specifications has received much attention in the last decades. The first approaches for specification-based testing were proposed in the ’80s. Since then, many specification-based testing approaches have been proposed, based on formal languages such as Z, VDM, CSP, CCS, LOTOS, SDL, and Petri Nets [60]. More recently, conformance testing approaches based on Labelled Transition Systems or Finite State Automata (or their variations) have been proposed [61–63]. Here, we restrict our attention to test selection and execution from such formal specifications, for which a quite mature theory of conformance testing now exists, together with automated tool support.

The aim of a formal conformance testing framework is to define a conformance relation between the implementation under test IUT and the (formal) specification S. Such a relation precisely establishes when IUT is a correct implementation of S and is based on the test hypothesis, i.e., the IUT can be modelled by a formal object MOD, such that all the observations that we can make on the IUT and on MOD (along the executions of all defined test cases) cannot be distinguished [64]. In such a way, we can formally define an “implementation relation” (imp) that correlates S with MOD. IUT conforms to S iff MOD is imp-correct with respect to S.

Figure 5 graphically summarizes how a conformance testing framework works.

---

Fig. 5. The Conformance Testing Framework
In Tretmans’ approach [62, 65], both the specification $S$ and the model $MOD$ of the implementation $I$ are expressed using Input/Output Transition Systems (IOTSS), an extension of the classical LTS model, in which the set of actions are partitioned into the Input actions and the Output actions. The implementation relation $imp$ used is of the form $\text{loc}$, that is a relation holding when $MOD$ can never produce an output which could not have been produced by $S$ after the same sequence of actions, or trace [66].

Similar derivations are obtained by Bochmann et al. [63] who use Finite State Machines to derive the system specification, its implementation and “conformance relations” (equivalence, quasi-equivalence, reduction) to generate test sequences. Fernandez et al. [67], use IOLTS in their approach for an automatic, on-the-fly generation of test cases.

**Tools:** TorX, TGV, and TVEDA are well known tools for automating the test selection and execution process in formal testing.

TorX [68] allows an on-the-fly test generation and execution (i.e., test derivation and test execution occur simultaneously). The input language is LOTOS or Promela, and the test selection can be random or manual. In TGV [61] the tester can use his/her knowledge of the implementation under test and of the context to guide the test selection through the notion of a test purpose [61, 66] (which permits the focusing on testing relevant interactions, while hiding the others). The TGV input languages are LOTOS or SDL, and test cases are derived in TTCN [69]. TVEDA [70] derives TTCN tests from SDL specifications.

Differently from TGV and TVEDA, TorX allows test derivation and execution in an integrated manner.

**Further Readings:** C. Jard and T. Jeron [71] (Article, 2005): this is one of the most recent papers on TGV. It describes the tool and the theory behind it. TGV [72] (Web, 2008): this is the reference web site for TGV. J. Tretmans [59] (Notes, 2002): this paper, written as the “Test and Verification” course notes, covers in an easy-to-understand and complete way the theory behind conformance testing.

### 4.2 Model-based testing based on Diagrammatic Specifications

**Problem Statement:** A model-based testing approach accepts two main inputs, a model of the software under test and a set of test generation directives (which guide the test cases selection), and outputs a test specification (which includes a set of stimuli the tester should introduce in the system together with expected responses) [73]. This section will focus on model-based specifications in the Unified Modelling Language (UML) [74].

**Achievements:** Binder surveys how each and every UML diagram could be tested (Reference [25], chapter 8). While UML structural diagrams are usually employed for testing design consistency (grey-box testing), behavioural diagrams are used for functional testing, precisely, interaction diagrams for integration testing (interactions among objects), and state diagrams for functional testing of objects.
This section will focus especially on a representative subset of scenario-based and state-based testing approaches. Figure 6 (taken from Leila Navlasky work [75]) summarizes the test generation, test execution and evaluation, coverage analysis, and regression testing activities implemented by automated model-based testing approaches.

The test selection activity receives models as input and produces test scripts as output. The inputs are textual or graphical scenarios, state-based specifications, or other specifications (structural specifications, such as class or component diagrams, activity diagrams). Sometimes, the code itself becomes an input of the test generation activity. The outputs are test scripts (sometimes referred to as test sequences), and comprise: a set of steps to be followed when testing a program, and input and output values. Test scripts are either abstract or concrete. Abstract test scripts describe the steps a tester should follow when using the system, the inputs to provide and the outputs to expect. Concrete test scripts can be compiled and automatically executed. They consist of calls to methods in the code, the inputs to provide, and the outputs to expect. The evaluation is either made manually by the tester, or automatically by an oracle.

The test execution and result evaluation activity receives as input abstract or concrete test scripts outputted by the test generation activity. Test execution runs test scripts
over the source code. The observed behaviours are evaluated against expected results, as expressed by the specification models.

Coverage analysis can be automatically performed to evaluate how much of the code has been covered. Regression testing can be applied to re-test changed models.

In the study conducted in [75] and summarized in [76], eleven automated model-based testing approaches were evaluated. Six of them are based on scenarios, two on state-based models, two on both scenarios and state machines, and one on scenarios or state machines. Figure 7 summarizes the input artefacts utilized by the eleven surveyed model-based testing techniques.

![Figure 7. The main Artefacts used in Model-based Testing](image)

By taking a look at other surveys on the topic, Utting et al. [77] place model-based testing approaches into a seven-dimension orthogonal taxonomy. The dimensions characterize the approaches with regard to the nature of the model used (e.g. what is modelled, notation used), to the nature of the test generation techniques used (e.g. test selection coverage) and to the nature of the test execution (e.g. on-line, or off-line).

In their work, Prasanna et al. [78] survey test case generation approaches. They classify these approaches into two categories: specification-based approaches and model-based approaches.

**Tools:** Hartman [73] presents a survey on model-based test generation tools. He distinguishes between test generators and model-based input generators. He also distinguishes between test generators and test automation framework, where the automation framework executes the test sequences without human supervision. The objective of Hartman’s survey is to place the AGEDIS [79] project and tool in relation to other tools. He groups the tools into academic and commercial and succinctly describes each of them. It is therefore a good reference to a list of tool supported approaches.

In their book chapter [80], Belinfante et al. provide an in-depth evaluation of test case generation tools. The book, however, is on model-based testing for reactive systems, where models are in their majority described with formal specification languages.

**Further Readings:** R. V. Binder [25] (Book, 1999): mostly on model-based test design techniques (based on combinational models, state machines and the UML) for OO

4.3 Software Architecture-based Testing

A Software Architecture (SA) is the first design artefact that can be produced during the software development. It breaks down the system in a number of cooperating elements (components, connectors, data elements). While components represent the core artefacts, a software architecture consists in the study of how such components can be integrated in order to satisfy desired functional and non functional requirements.

According to Bosch and Stafford [38], software architecture represents an integral part of any component-based software system and software architecture and components are two sides of the same coin. A component-based design can be i) component-driven, if existing components are integrated via a component framework, according to a certain architecture, or ii) architecture-driven, if components are produced or acquired in order to satisfy architectural requirements.

Problem Statement: A growing interest in software architectures characterizes the last decade. While most of the initial effort has been on how to specify an architecture, how to select the right architecture has become one of the most relevant challenges in recent days. In particular, the importance of the role of SA in testing and analysis is becoming evident [83].

When referring to software architecture and testing, we need to distinguish between SA-based testing and testing SAs. In SA-based testing, the goal is to use the SA specification as a reference model to extract suitable test classes for the higher levels of testing and to refine them into concrete tests at the code level [84]. In testing SA, instead, the main goal consists in assessing the SA quality with respect to expected requirements it shall meet.

Two are the major issues to be solved in architecture-based testing: traceability and test execution.

Traceability consists in “relating the abstract values of the specification to the concrete values of the implementation” [85]. Since test cases are selected from the (intentionally abstract) architectural model, they have to be refined into more concrete test sequences in order to be able to execute these tests on the code. Similar problems can be found in specification-based testing (discussed in Section 4) but they are typically solved by making use of detailed and lower-level specifications.

Test execution entails forcing the Implementation Under Test (IUT) to execute the specific sequence of events that has been selected. However, A problem arises with
concurrent programs which, starting from the same input, may exercise different sequences of interactions (among several concurrent processes) and produce different results. This problem has already been analyzed in the literature, and nondeterministic and deterministic-testing [86] approaches have been proposed.

Achievements: Several authors have advocated the use of architectural models to drive testing, and in particular to select relevant behaviours of interactions between system components, based on the early SA specification. In [87], the authors analyze the advantages in using SA-level testing for reuse of tests and to test extra-functional properties. In [88], the authors define six architecture-based testing criteria, adapting specification-based approaches. Those two papers represent the first attempts in software testing based on an architectural specification.

In [89], the authors present an architecture-based integration testing approach that takes into consideration architecture testability, simulation, and slicing. In [90] Harrold presents an approach for effective software architecture regression testing, and in [91] she also discusses the use of software architecture for testing. In [91] Rosenblum adapts his strategy for component-based systems testing to SAs. The testing approach is based on architectural models, that could be simulated or executed or used to guide integration or regression testing of the implemented system. The author also shows how formal models of test adequacy can be used in conjunction with architectural models to guide testing.

In [92] the authors propose a technique to test data flow and control flow properties at the architectural level. Six architecture relations among architecture units are defined and then used to define architecture testing paths. Five architecture-level testing criteria are proposed. However, to the best of our knowledge, the approach in [84] is still the only comprehensive attempt to tackle the whole cycle of SA-based testing.

Reference [93] presents a framework for SA-based regression testing, coping with two main types of evolution: architectural evolution and code evolution.

Figure 8 shows a synthetic table of SA-based testing techniques, pointing out the specific goals of each given paper/approach.


5 Ongoing Research in Software Testing

While previous sections have discussed consolidated testing techniques based on different development paradigms and based on different pre-code artefacts, this section wants to introduce some ongoing research that, while not consolidated, is being used in academic and practical domains and are being considered key technologies. We here focus on Service oriented testing (Section 5.1), on the UML Testing Profile (Section 5.2), and on Product Line Testing (Section 5.3).
5.1 Testing Service-based Applications

Service Oriented Computing aims to provide the basis for building software by assembling independent, loosely coupled services. Services are understood as autonomous platform-independent computational entities that can be described, published, categorized, discovered, and dynamically assembled for developing massively distributed, interoperable, evolvable systems and applications. These characteristics pushed service-oriented computing towards its present widespread success, demonstrated by the industrial investments on this technology.

Problem Statement: Web Services represent a concrete instantiation of the service-oriented paradigm, where web services are distributed and integrated via Web standards. Being web services considered today a key technology in software production, how to test them is becoming a relevant task in many industrial organizations.

The main issues and challenges related to testing Web services resemble those related to component-based testing (as already analyzed in Section 3 and remarked in [95]), with some further extensions as in the following:

- **Lack of control:** as in component-based development components can be produced by different vendors, services can be produced by different stakeholders. However, while component-based applications are typically produced by a single organization by integrating reusable components, service oriented systems are never under the full control of a single organization.

- **Different Stakeholders:** while two are the main stakeholders involved in component-based testing, five are the main actors who might be concerned with testing of services: service developers, service providers, service integrators, third-party certification authorities, and final users. Each of them can thus require specific testing techniques, related to the level of knowledge/control they do have on the service and on the testing purpose.
Different stages: a web service can be tested at three different stages: before deployment, at registration time, at execution time.

A service can be tested before its deployment by using off-line testing. At this time, the service developer or provider tests the service by simulating possible usage scenarios. Successively, at registration time the test can be executed in a real execution environment providing more realistic results. The discovery and directory services can then be reinforced with testing functionalities. Eventually, at execution time the system can be monitored so as to discover run-time service failures.

Different ways of integrating services: services can be composed by following two complementary views: orchestration and choreography. In the orchestration, an orchestrator defines the interaction order among components. A choreography, instead, specifies the interaction among all the services taking place in the system. Different interaction primitives and languages are utilized by the two views, which impose clear distinctions among choreography-based and orchestration-based testing [96].

Achievements: Many approaches have been proposed so far for testing web services. The survey study in [97], takes into consideration 12 web services testing frameworks and classify them according to many parameters, the most relevant being: the testing web service architecture, the stakeholders involved in the testing activity, the web services specification languages, and the testing strategies. The table in Figure 9 summarizes the survey main results.

In Reference [95] a comparison among three web services testing tools (Parasoft SOATest, Mindreef SOAPScope, and PushToTest TestMaker) is reported according to the following (main) parameters: support for functional testing, support for custom scripting, type of licence, type of interface, support for simulation and recording, support for extra-functional testing.

Further Readings: L. Baresi and E. Di Nitto [98] (Book, 2007): this book contains contributions from leading academic and industrial research groups on static analysis, testing, monitoring, and non functional analysis of web services.

5.2 MDA-based Testing: the UML Testing Profile

Problem Statement: With the widespread adoption of software models and model-based testing techniques, new challenges appear. A multitude of different model-based testing approaches are available today (as outlined in Section 4.2), each one focusing on different UML models, different level of abstractions, and different test selection/execution/coverage criteria. As soon as a model evolves, related models and the test themselves shall evolve to keep the specification and test specification consistent. Explicit support for models alignment is rarely available. Moreover, it is not uncommon during evolution that only the low-level design and implementation are changed to meet tight deadlines, while the model specification is not updated to track the changes being made to the implementation [99]. As soon as the specification “drifts” out of
conformance with the implementation, it is no longer representative of the expected behaviour.

Explicit relationships need to be devised between specifications and their implementations (sometimes called mapping), and between specifications and test results (sometimes called traces) (as initially discussed in [85]).

**Achievements:** In order to handle such problems and bridge the gap between models and implementation, in 2001 the Object Management Group (OMG) has released the UML Testing Profile Request For Proposal. As a result, the UML Testing Profile (UTP) [100] has become an official OMG standard since late 2005, with the main goal of allowing a conformance testing activity to take place in parallel with the system development. The main objectives of the UTP are to provide an ontology of testing-related artefacts needed for testing purposes, and model transformation algorithms to move towards different models. The UTP implements the MDA philosophy, by providing a common framework for system development and testing. The UTP profile extends the UML 2.0 meta-model in order to introduce test-specific concepts. The analogy to MDA has been illustrated in Figure 10. The PIT is the Platform Independent Test, generated through model-to-model transformation techniques from the Platform Independent Model (PIM). Following the MDA philosophy, as the PIM is transformed into a Platform Specific Model (PSM), the PIT is transformed into a Platform Specific Test (PST), i.e., a test executable on the PSM. While vertical transformations permit to move from

<table>
<thead>
<tr>
<th>Paper Title</th>
<th>Stakeholder (Developer/Provider/Standard body)</th>
<th>Specification Language</th>
<th>Type of Testing</th>
<th>Orchestration</th>
<th>Local Choreography</th>
<th>Global Choreography</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coyote: An xml-based framework for web services testing</td>
<td>P</td>
<td>WSDL</td>
<td>Integration/Regression</td>
<td>Integration</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Web Service’s Test Harness: A Functional, Load, and Performance Testing Framework for Web Services</td>
<td>P</td>
<td>WSDL</td>
<td>Functional, Load, Stress, CHD</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Scenario-based web service testing with distributed agents</td>
<td>D, P, S</td>
<td>WSDL</td>
<td>Unit, Integration, Stress</td>
<td>Unit, Integration</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Using Progressive Group Testing</td>
<td>D, P</td>
<td>WSDL</td>
<td>Unit, Integration, Stress</td>
<td>Unit, Integration</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Automatic conformance testing of web services</td>
<td>P, S</td>
<td>WSDL, GT Rules</td>
<td>Functional, Stress</td>
<td>Unit</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Towards contract-based testing of web services</td>
<td>P, S</td>
<td>WSDL, Contracts</td>
<td>Functional, Integration</td>
<td>Integration</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>The audition framework for testing web services interoperability</td>
<td>D, P, S</td>
<td>WSDL, PSM</td>
<td>Integration</td>
<td>Integration</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Speakeasy unit testing: Framework and implementation</td>
<td>P</td>
<td>BPEL</td>
<td>BPEL, Unit Process</td>
<td>Unit</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Verification of WS-CDL Choreography</td>
<td>P, S</td>
<td>WS-CDL, CDL</td>
<td>Functional</td>
<td>Unit, Integration</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Towards a bpel unit testing framework</td>
<td>D</td>
<td>WSDL, BPEL, extension mechanisms</td>
<td>BPEL, Unit Process</td>
<td>Unit</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A model checking based test case generation framework for web services</td>
<td>D, P</td>
<td>WSDL, BPEL</td>
<td>Unit, Integration</td>
<td>Unit, Integration</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WebSob: A Tool for Robustness Testing of Web Services</td>
<td>D</td>
<td>WSDL</td>
<td>Unit, Robustness</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Fig. 9. Comparing Testing Web Services Frameworks
the PIT, to the PST and test code, horizontal transformations permit to move back and forth from development to testing.

![Fig. 10. The UTP contextualization in MDA](image)

The main goals and principles of UTP are:

- **To interweave software development and testing.** The main principle implemented in the UTP is that software testing needs to be carried out during the entire software development, and should be connected as much as possible with the system development models. The UTP thus provides a unique notation, based on the UML, to be used by both software developers and software testers, and the ability to move from the modelling domain to the testing one;

- **Conformance Testing:** the UTP has been proposed in order to provide an answer to the need of solid conformance testing, certification and branding;

- **Structure + behaviour Modelling:** the UTP takes into consideration both structural and behavioural diagrams. Structural diagrams are utilized to specify the testing components. Behavioural diagrams are utilized for specifying test cases and for identifying the test cases execution order;

- **Black-box testing:** UTP focuses on black-box, conformance testing. For this reason, the profile itself does not consider any information on the internal structure of the system under test.

The UTP contributes to the aforementioned goals in two different ways; it provides an ontology of model-based testing concepts to produce a testing model, and it describes some transformation rules for moving from development models (PIM and PSM) to testing models (PIT and PST). The ontology of concepts has been categorized into four conceptual packages: test architecture, test behaviour, test data, and time. Each package has a specific goal and contains meta classes identifying the UTP ontology. Each package is described in detail in [100].
5.3 Testing Software Product Lines

Software product lines (SPLs) represent a more strategic approach to reuse, where components and other assets are identified in relation with specific application domains, architectures and scopes (opposed to software reuse schemes which try to create assets as general as possible). Quoting [105], “a software product line is a set of software-intensive systems sharing a common managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way”. The idea behind a system-family approach is to build a new system or application from a common set of assets (e.g., component, known requirements or design elements, models, artefacts) in the same domain.

The main concepts which characterize an SPL are those of:

- **domain-specific core assets**: instead of creating libraries of general purpose components, hoping they will be used in future development, SPLs make a predictive analysis of such artefacts which may be reused in the development of a specific domain-related product [106];

- **commonality and variability**: commonality defines what different products have in common and guides the production of domain-specific core assets. Variability is the ability to change or customize a software system [107] (i.e., to distinguish one product from another in the SPL), and requires to delay design decisions such that many decisions are left open until a certain product is selected out of the product line. A variation point identifies one or more locations at which the variation will occur;

- **domain engineering and application engineering**: taking a look at an SPL software process, two main phases are always contemplated: domain engineering and application engineering. In the former phase, commonality and variability of the entire family are analyzed, while during application engineering an individual application, member of the SPL, is selected .

When testing a SPL, traditional testing techniques have to be adapted so as to take into consideration the SPL peculiarities described above. The main challenge is in investigating how the entire SPL can be used to automatically generate testing information which may be effectively reused to test each derivable product, as discovered in [108].

Future Directions in Software Testing

Mary Jean Harrold [6] and Antonia Bertolino [7] in their ICSE 2000 and ICSE 2007 (respectively) Future of Software Engineering (FOSE) papers on software testing, highlight a number of working directions and challenges in software testing. While sharing most of them, I would like to briefly discuss here those areas I personally believe will be of major relevance in next generation software testing techniques.

Testing evolving at run-time software: An increasingly important requirement for software-intensive systems is the ability of changing over time, due to the need to dynamically add/remove new features, the need to provide a more dependable system, or the need to protect the system from run-time incoming attacks or deficiencies. This newly acquired level of dynamicity, while enhancing the flexibility of a software system which can become self-adaptable, imposes new degrees of complexity when certain qualities of the system must be guaranteed. Continuously evolving systems become difficult to be analyzed with techniques used over traditionally static systems. In the context of such highly dynamic systems, the focus moves from validating the static configuration to validating the changing over-time design. Thus, while in traditional static systems the verification can be done once before system deployment, for dynamic architectures the validation becomes a perpetual activity to be performed during system execution. On-line testing and monitoring techniques are currently being analyzed for testing such types of software (see for example [95, 111]).

Automation in Software Testing: Practitioners regard software testing as the central means for ensuring that a system behaves as expected. The introduction of automated testing techniques has strongly facilitated the introduction of software testing into practice; automation has reduced the amount of effort spent on technical testing activities and has also increased the precision of activities, like result evaluation, often performed by humans and thus more error-prone. However, testing in industrial projects can be effective only when the testing effort is “affordable”: the testing approaches should support automatic creation of test plans sooner, they should automate most of the testing activities, testing and development shall proceed hand-by-hand being automated in the same development and testing framework.

As a side effect, software testing automation would also improve technology transfer of testing and analysis research. Technology transfer is fundamental for making research useful in industrial domains and for bridging the gap among academic research and industrial needs, as remarked by Marat Boshernitsan [112] in his ISSTA 2006 invited speak. Alan Hartman in [113] describes his experience, the challenges and dreams in making model-based testing an industrial practice. Test automation is remarked as one of the main means to make model-based testing an industrial practice.

Integration of Testing and Analysis Techniques: Integration of analysis techniques is a topic which has recently been receiving some attention in the software engineering community (e.g., Reference [114]). This problem can be addressed from two different viewpoints: integrating analysis techniques during the development process (from
requirement-based to source code analysis), and integrating different analysis techniques at the same stage in software development (e.g., design-based testing and model-checking).

**Anti-model based Testing:** Specification-based testing is certainly useful and effective; however, there can be several reasons why such an approach cannot be applied or is too expensive for deployment in a specific context. One of the main obstacles relies on the assumption that a complete and consistent model of the software system exists, while for certain classes of systems (e.g., component-based systems) we cannot a-priori assume that a specification or the source code are available. In such cases, model-based testing is not applicable, or would be too costly.

This is the rationale for an “anti-model-based testing approach”: while model-based testing starts from an a-priori established model and tries to execute some sequences derived from this model, in “anti-model based” testing the opposite direction is taken: the implementation is executed on some sample executions, and by observing the traces of execution an abstract model of the system is inferred/synthesized a-posteriori.

**Self-testing:** An increasingly important requirement for software systems is the ability to self-manage, i.e., to autonomically manage themselves. According to Kephart and Chess [115], four are the main autonomic areas: self-protection, self-configuration, self-optimization, and self-healing. Other colleagues describe other forms of self*-techniques, such as self-adaptivity, and self-organization. In the context of self*-systems, self-testing seems to be a natural way to move forward, where the system has the ability to autonomically test (and repair) itself.

7 Concluding Remarks

This work has tried to provide an ample view on software testing from two different viewpoints: the development technology to be tested and the artefacts to be used for testing purposes. A detailed list of references on high quality research work has been provided to help the reader acquiring a more advanced knowledge on each area covered.

Acknowledgments

I wish to thank the anonymous reviewers, Mauro Pezzé and Antonia Bertolino for their useful feedback on a preliminary version of this paper. I thank you Andrea Polini for his help within the component-based testing section of this chapter, and Damien Andrew Tamburry for is thorough proof reading.

References

1. Edsger W. Dijkstra: Chapter I: Notes on structured programming. (1972) 1–82
9. ISSTA: International symposium on software testing and analysis. (http://issta08.rutgers.edu/, last visited: Jan 2008)


68. TorX: TorX Test Tool Information. (http://fmt.cs.utwente.nl/tools/torx/introduction.html, last visited: Jan 2008)
79. AGEDIS: AGEDIS Project on Automated Generation and Execution of Test Suites for Distributed Component-based Software. (http://www.agedis.de/index.shtml, last visited: Jan 2008)
82. Forum: Model-based testing forum. (http://tech.groups.yahoo.com/group/model-based-testing/, last visited: Jan 2008)
100. UTP: (Omg: Uml 2.0 testing profile. final adopted specification, formal/05-07-07, july 2005)
Appendix A: A Glossary of Software Testing terms

This section provides an explanation of the main terms used in software testing. In order to make the explanation more practical, some of the described concepts will be applied to the BuyOnline fictitious web application, which allows users to register, to select products from a list, to visualize their descriptions and information, and to buy them online.

Test Case: A test case is a set of inputs, execution conditions, and a pass/fail criterion [116]. A test case thus includes not only input data but also any relevant execution conditions and procedures, and includes a way of determining whether the program has passed or failed the test on a particular execution [8].

A single test case $t_1$ for the BuyOnline application might be: select product with id 563AD, put it in the chart, buy it by credit card.

Test Suite: A test suite is a collection of test cases.

Further test cases can be generated to test the BuyOnline application (by changing the selected products, by removing a product from the chart, or by changing the payment type). All together they will form a test suite for the BuyOnline system.
**Test Process:** The testing process consists of different activities, the most important being:

- **Test Selection:** it consists in selecting a suitable and finite set of test cases from the possibly infinite set;
- **Test Execution and Evaluation:** it consists in executing the code accordingly to the selected test cases and comparing real and expected results;

The test selection activity provides guidelines on how to select test cases. It is driven by a “test criterion” and has to produce “suitable” test cases:

*Test Criterion:* A test criterion provides the guidelines, rules, and strategy by which test cases are selected. In general, a test criterion is a means of deciding which shall be a “good” set of test cases [117].

*Suitability:* A test case is suitable if it contributes to discovering as many failures as possible, according to a test criterion.

Test cases for the BuyOnline application can be selected randomly (i.e., by randomly selecting products, payments types, and functions to be run) or through a more systematic approach, like code coverage (i.e., test cases are selected so to cover all the functions, or the code statements/branches/paths.)

The test execution activity consists in executing the code according to test case inputs. Typically, it describes how to bring the software system in a state so that the test input can be given.

When executing the BuyOnline application according to test case tc1, the user has to be registered first, then the test case inputs can be provided.

The test oracle evaluates whether the observed outputs comply to expected behaviours. The test oracle can be the tester herself: based on her knowledge of the expected system output, she can predict the expected behaviour and thus compare the oracle with the real behaviour. Otherwise, the oracle can be automatically generated e.g., from an existing specification or from a previous version of the system.

The test oracle for tc1 may state that if product 563AD is the one selected, it is the one delivered and the credit card is charged for its correct amount.

The test adequacy criteria permit to judge whether the test campaign is sufficient. In general, the test adequacy should state that a test suite is adequate when it allows the identification of any failures. However, since the number of failures is not known a priori, we need to approximate the intuitive concept of adequacy. Thus, a test criterion can be used to define a stopping rule for evaluating the adequacy of the selected test suite.

The testing campaign of the BuyOnline is adequate as soon as the selected test suite covers, for example, the amount of source code defined in the test criterion.

**Further Readings:** A. Bertolino [3] (Guide, 2004): this chapter, part of the Guide to the Software Engineering Body of Knowledge, provides a compendium and guide to the body of knowledge on software testing as developed in the past four decades.