Structural and Functional Predicate Coverage Testing

Mario Winter, Michael Averstegge

Abstract

Predicate testing, also known as (branch) condition testing, is most often filed under structural (white box) test case design techniques. Nevertheless, contract-based specification techniques allow for adapting classical predicate testing as a functional (black box) design technique on the class resp. component test level, too.

In this paper we discuss various predicate testing coverage criteria known from the literature, adapt them for functional test case design focusing on “design by contract” specifications, and report on a tool supporting both structural and functional predicate coverage testing for Java programs with OCL specifications for invariants and pre- and postconditions.

Keywords

Software Testing, Test Coverage, Design by Contract, Computer Aided Software Testing

1 Introduction

Since humans tend to err, software, being among the most complex human artefacts, was, is, and – so far as foreseeable by us – will be deficient. Though the prevention of errors by far would be the most effective means to achieve software quality, we humbly have to cope with our limitations in applying any methods and techniques, thus analytical quality assurance, aka. “testing”, is here to stay.

We firstly sketch some terminology on software quality assurance [Ve04]. Firstly, an *error* is a human action that produces an incorrect result. An error may lead to a *fault* (a *“bug”*), which is a flaw in a component or system, e.g. an incorrect statement or data definition that can cause a component or system to fail to perform its required function. A fault, if encountered during execution, may cause a *failure* of the component or system, i.e. an observable deviation of the actual behaviour from the expected behaviour.

Testing may or may not depend on executing the software. If it doesn’t, one conducts *static testing* at specification or implementation level, e.g. reviews or static code analysis. If it does, we talk about *dynamic testing* which executes the software by stimulating it through some input, observing its actual behaviour, and comparing the observed behaviour with the expected one (Figure 1). Frankly, software testing aims at checking that the software does what it should do and that it doesn’t do what it shouldn’t.

In this context, a *logical test case* describes a set of input values, execution assumptions, expected results and execution effects, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement. A *concrete test case* specifies particular values satisfying the corresponding logical test case. A *test suite* is a set of test cases with the same concern.

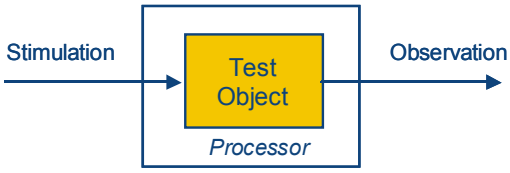

Figure 1. Dynamic Testing

Test cases are specified using some test design technique. In *functional* or *black box testing*, the input values and execution assumptions are derived from the specification, while in *structural* or *white box testing* the implementation is used as well. In both cases, the expected results and execution effects have to be derived by the specification, since only the specification tells what the software should and shouldn’t do.

Predicate testing, also known as (branch) condition testing, is most often filed under structural (white box) test design techniques. It aims at determining test cases which stimulate the logical decisions implemented in a program in particular ways (c.f. chapter 2). Like all structural testing techniques, predicate testing depends on a thorough understanding of the source code, thus in most cases it is bound to specially trained testers on the unit testing level. But also in unit testing one should

conduct functional testing first, since the primary task of testing is checking that the software does what it should do – and the "do's" are described in the specification.

Regarding the latter, a specification technique frequently referred to as "Design by Contract" is widely used in software development (c.f. chapter 3). In this paper we show that contract-based specification techniques allow for adapting predicate testing as a functional (black box) design technique on the class resp. component test level.

The sequel of the paper is structured as follows. In chapter 2 we discuss various structural predicate testing coverage criteria known from the literature. Chapter 3 is devoted to a concise introduction of the "Design by Contract" specification technique. In chapter 4 we adapt predicate testing for functional test case design based on contracts, and in chapter 5 we report on a tool supporting both structural and functional predicate coverage testing for Java programs with OCL specifications for invariants and pre- and postconditions.

2 Structural Predicate Testing

In all but trivial programs the statement execution order may vary due to some situations. Regarding structured programs, this is enabled by conditional branching statements (e.g. `if then else`, `case`) and loops (e.g. `for`, `while`).

Conditions denote boolean functions used to control branch- and loop-statements, i.e. functions which evaluate to either `true` or `false`. They can be *simple* like the access to a boolean variable `b`, a comparison `n > 0` or `n*m = 10` for variables `n` and `m` of type `integer`, or a call of an operation like `fb(x, y)` with two parameters `x` and `y` of arbitrary type which computes a single value of type `boolean`.

Simple conditions are often used to constitute *compound* conditions like `(a > 5) && (b == 3)` or `(fb(x, y) || (x < 0.5)) && (y*y > 10)`, where "`&&`" resp. "`||`" denote the *logical and* "$\wedge$" resp. the *logical or* "$\vee$". A (simple or compound) condition may be negated e.g. `!(a > 5)`, where "!" denotes the *logical not* "$\neg$". The evaluation order (i.e. `!` before `&&` before `||`) may be changed explicitly by using appropriate parentheses. In the following, a condition controlling a branching or loop statement is called a *predicate* (*decision, guard*).

If a predicate is incorrect, then according to [TV+94] one or more of the types of faults depicted in the table shown in Figure 2 may exist.

Fault Type	**Example**
Wrong boolean operator	`&&` instead of `||`
Wrong relational operator	< instead of <=
Parenthesis fault	`(a && !(b || c))` instead of `(a && (!b || c))`
Wrong arithmetic operator	`(a+b > 0)` instead of `(a*b > 0)`
Missing or extra ! operator	`a && !b` instead of `a && b`
Wrong boolean variable usage	`a && b` instead of `a && c`

Figure 2. Fault types in predicates

An incorrect predicate contains either a single fault or multiple faults of the same or different types. A test suite for a predicate *C* is said to detect the existence of faults in *C*, if an execution of *C* on at least one element of this test suite produces a failure.

To detect incorrect predicates in a program, various predicate testing methods are known. The simplest one, *predicate coverage (testing)*, also known as *branch coverage* (C1) or *decision coverage*, requires that each predicate is evaluated to `true` and `false` at least once. The number of test cases required for each predicate is two and does not depend on the complexity of this predicate.

In the sequel let *n* denote the number of simple conditions constituting a predicate. *Simple condition coverage* requires that each simple condition of a predicate is evaluated to `true` and `false` at least once. Here the minimum number of test cases required for a predicate is two. Depending on the structure of the predicate (especially if conditions are coupled via common variables, like e.g. `(x == 0) || (y == 1) || ((x * y) > 0)`), at maximum 2**n* test cases may be required. Since simple condition coverage does not subsume branch coverage, it can be slightly strengthened to *branch condition/predicate coverage,* which simply combines branch coverage and condition coverage.

Multiple condition coverage (MC) requires that each `true`/`false`-combination of all simple conditions of a predicate are forced at least once. The number of test cases required for each predicate is 2***n* and only depends on the number of simple conditions, but not on the structure of this predicate. Moreover, in case of coupled conditions not all combinations may be possible to achieve.

To cope with both the exponential complexity of multiple condition coverage and the structural properties of the predicates, the RTCA DO178B document [DO178B] defines *modified condition/decision coverage* (MC/DC) as follows: "*Every point of entry and exit in the program has been invoked at least once, every condition in a decision has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect the decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.*" Note that in the DO178B, decisions not only denote predicates controlling the execution of statements, but also boolean expressions on the right hand side of assignments or as actual parameters of operation calls. MC/DC for each predicate demands 2*n* test cases, if the predicate has taken both truth values under the truth value combinations of the conditions, else 2*n*+1.

Structural predicate coverage testing strategies has been shown of value in empirical studies (c.f. e.g.[TV+94], [WGM85], or [ZHM97]). Thus our idea was to exploit their power regarding functional testing, too. In the sequel we elaborate this idea regarding functional testing of component interfaces specified by the "design by contract" technique.

3 Design by Contract

According to Bertrand Meyer [Me97], for each software interface we need three kinds of specification elements:

- The *precondition* (of an operation) expresses the properties that must hold whenever the operation is called. Its clauses refer to input-parameters of the operation und the state of the component before the operation is activated.
- The *postcondition* (of an operation) expresses the properties that the operation guarantees when it returns (assuming its precondition was satisfied). Its clauses refer to output-parameters of the operation und the state of the component after the operation is completed, and may refer to the input parameters and the initial state, as well.

Both pre- and postconditions describe properties of individual operations, but often one wishes to specify some more general properties of the software component.

- To this end, the *invariant* (of the software component) expresses global properties of all instances of a component, which must be preserved by all operations. Its clauses refer only to the state of the component and must be satisfied before and after each execution of an operation, thus the invariant can be considered as logically conjugated ("and-ed") to all pre- and postconditions.

Together, pre- and postconditions of all operations of an interface together with the general properties stated by the invariant answer the questions of "What does the software component expect from its users?" "What does it deliver to them?" and "What does it maintain all time?" Bounded to the software components interfaces, they resemble a contract between the software component and its clients, saying: "If my clients promise to call me with the precondition satisfied then I, in return, promise to deliver a final state in which the postcondition (and my invariant) is satisfied".

As an example for design by contract we consider the well known stack interface with operations `top()`, `push()`, and `pop()`. In this example the size of the stack is bounded by an integer number of stackable objects which is settled by the only parameter of the constructor `stack(int n)`.

```
interface BoundedStack

    invariant@ self.size() >= 0 AND self.size() <= self.MAXSIZE()

    public BoundedStack (Integer maxSize)
       pre@  maxSize > 0
       post@ self.MAXSIZE() = maxSize@pre AND self.size() = 0

    public void push (Object item) throws FullStackException
       pre@ self.size() < self.MAXSIZE()
       post@ self.size() = self.size()@pre+1 AND self.top() = item@pre

    public Object top () throws EmptyStackException
       pre@ self.size() > 0
       post@ return <> null

    public void pop () throws EmptyStackException
       pre@ self.size() > 0
       post@ self.size() = self.size()@pre - 1

    public Collection all ()
       pre@ true
       post@     (self.size() > 0 and return.size() = self.size()) or
            (self.size() = 0 and return = null)
```

Figure 3. Contract for interface BoundedStack

Moreover, we consider an additional operation `all()` which returns an ordered collection containing (references to) all stacked objects ordered from top to bottom, thus the most recently stacked object occurs at the beginning of the returned collection. The contract for the resulting interface `BoundedStack` is sketched (in a frankly mixed Java/OCL notation) in Figure 3. Note that in OCL the equality operator is written as "=" (which is the assignment operator of Java).

The obvious observation is that clauses in contracts are formed by conditions and the resulting predicates maybe as complex as in the case of control flow decisions. Thus all fault types resp. error hypotheses listed in Figure 2 are applicable for contract specifications, too. So in the next chapter we transfer predicate testing techniques to functional testing.

4 Functional Predicate Testing

In functional testing one focuses on the functions of classes, subsystems, or systems. For the sake of simplicity we only consider the public operations of an interface for which contractual specifications are given. First we deserve an overall strategy, i.e. a global order in which the operations are tested. Characterizing the operations due to their gross effect reveals the following test order:

1. Constructor,
2. Simple observer like `getMember():value`,
3. Simple modifier like `setMember(value)`,
4. Complex observer (i.e. state preserving operations),
5. Complex modifier (i.e. state modifying operations), and
6. Destructor (if implemented).

Regarding the test input for an operation we consider its precondition, which should hold in case of conformance (or positive) tests regarding normal use and fail in case of robustness (or negative) tests regarding abuse. Without loss of generality, in the sequel we concentrate on conformance tests satisfying the preconditions of the operations under test, utilizing MC/DC as an appropriate test coverage metric.

In this case, for a simple condition `S` one specifies a single test case with `S == true`, for a negated simple condition `!S` one specifies a single test case with `S == false`. For a compound condition `S || T` each simple condition `S` and `T` must evaluate to `true` and `false` at least once, which raises two test cases `S == true, T == false` and `S == false, T == true`. For a compound condition `S && T` both simple conditions must evaluate to `true` which raises the test case `S == true, T == true`.

For each test case one then specifies the expected outcome. For contract based testing it suffices to compute expected truth values for each simple condition constituting the postcondition of the operation under test. This is done for all test cases specified so far. Afterwards one has to proof that all postconditions are covered w.r.t. the MC/DC criterion, i.e. all combinations analogous to those described for preconditions have to be covered. If not, more test cases have to be specified manually until full coverage is reached.

If state variables (attributes, member variables) occur in a precondition, normally some sequence of operation calls has to be activated before this precondition can be satisfied. To this end, due to the test order mentioned above often the resulting test cases may be staggered, i.e. test cases for complex observer operations may rely on test cases for simple modifier.

The table in Figure 4 summarizes the MC/DC-based logical conformance test cases for the `BoundedStack` interface presented in chapter 3. Each row corresponds to a simple condition, each of the right hand columns titled *A*, *B*, … represents one test case, i.e. a call of an operation with its precondition satisfied in the manner indicated as follows. Each entry denotes the evaluation of the corresponding simple condition to `true` (T) resp. `false` (F), whereas d.c. means “don’t care”.

The shaded entries indicate the operation under test, the entries for the preconditions of the other operations (and of the invariant) denote the expected values after the execution of the operation under test, thus aiding the operation call sequence problem mentioned above.

For example, column *A* indicates that the call of the constructor sets the stack object in a state satisfying the preconditions of operations `push()` and `all()`, but not those of `top()` and `pop()`. Columns *A*, *B*, and *C* reveal that a conformance test of `top()` may start with a call of the constructor with `maxsize > 0`, followed by a call of `push()` with an arbitrary object `obj`, followed by a call of `top()` and a check that the returned object equals `obj`.

interface BoundedStack	*A*	*B*	*C*	*D*	*E*	*F*
invariant@ self.size() >= 0 **AND**	T	T	T	T	T	T
self.size() <= **self**.MAXSIZE()	T	T	T	T	T	T
BoundedStack **pre@** maxSize > 0	T					
BoundedStack **post@ self**.MAXSIZE() = maxSize@**pre AND**	T					
self.size() = 0	T					
push **pre@ self**.size() < **self**.MAXSIZE()	T	T				
push **post@ self**.size() = **self**.size()**@pre**+1 **AND**		T				
self.top() = item@**pre**		T				
top **pre@ self**.size() > 0	F	T	T			
top **post@ return** != null			T			
pop **pre@ self**.size() > 0	F	T	T	T		
pop **post@ self**.size() = **self**.size()**@pre** - 1				T		
all **pre@** true	T	T	T	T	T	T
all **post@** (**self**.size() > 0 **and**					T	F
return.size() = **self**.size()) **or**					T	dc
(**self**.size() = 0 **and**					F	T
return = null)					dc	T

Figure 4. Contract based test case specification for interface BoundedStack

5 Tool Support

The elaboration of test cases regarding some predicate testing criterion is a time consuming, error-prone, and tedious task. So we’ve built a tool supporting the tester in the algorithmic parts of his work.

The resulting tool is called WeSUF (acronym from the German title “Werkzeug für strukturelle und funktionale Bedingungs-Überdeckungs-tests von Java-Programmen mit OCL-Spezifikationen”). It supports both functional and structural predicate coverage testing. Figure 5 depicts the tools architecture, which is described in the sequel.

Regarding functional predicate coverage testing, the tool extracts and works on predicates of contracts formulated in OCL (Object Constraint Language, [WK03]), which are contained as appropriately tagged comments in Java sources. Parsing of OCL is based on the *SableCC* compiler toolkit [GH98]. For structural predicate coverage testing, the tool is able to parse the Java sources in detail and extract all contained decisions. To this end, we rely on the *Barat* Java parsing framework [BS98].

After extracting and analysing the predicates, the tool offers several options, among them the checking for tautologies, some equivalence transformations up to the normalization of the predicates (disjunctive and conjunctive normal form), and the computation of (a minimal set of) truth value combinations regarding some selected predicate testing coverage criterion.

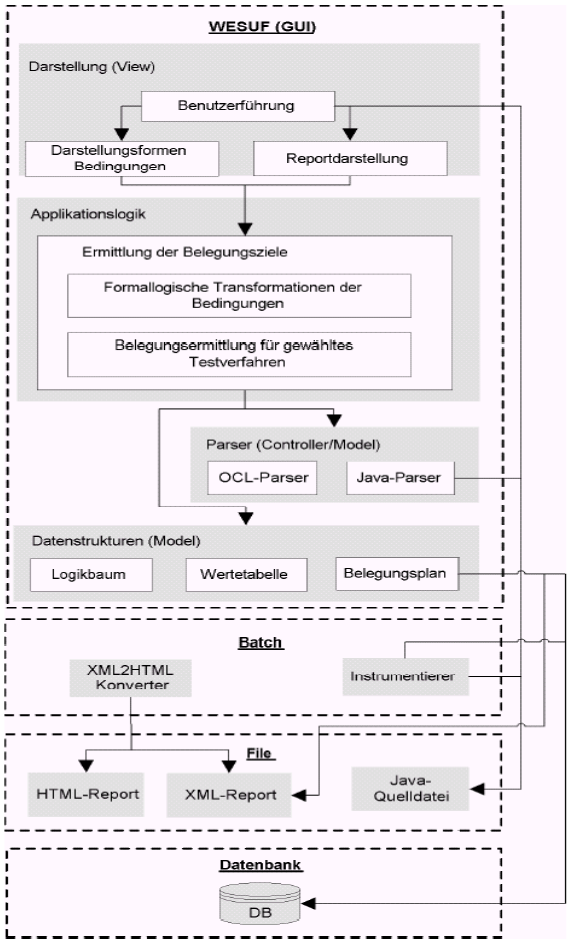

Figure 5. Architecture of the WeSUF-Tool

In order aid the tester in identifying the values needed to achieve the aimed coverage, WeSUF not simply computes a minimal set but the optimal set of logical values. Roughly spoken, the implemented optimisation criterion leads to a minimum of difference of the needed truth value combinations to achieve the aimed coverage (for more details see [Av04]).

Regarding structural predicate coverage and the specification of concrete test cases, we’ve coupled the WeSUF tool with a code instrumentation tool called DoIT (acronym from the German title “Datenbankgestützte Dokumentation instrumentierter Testläufe von Java-Programmen”, [We03], [WW03]). The statement, branch, and predicate coverage achieved by executing a test suite is reported into the DoIT database, which afterwards is analysed by WeSUF. Then all covered combinations of truth values are highlighted and listed together with the corresponding concrete test cases, the input values of the method containing the decision under test. With this information at hand, the tester may be able to infer test cases for the uncovered truth value combinations.

Furthermore, to support the specification of concrete test cases, WeSUF detects and reports the object state before and after each condition evaluation and, as a “side-effect”, warns on detecting side-effects.

WeSUF is able to generate XML- and HTML-based reports focusing on the decisions structure (e.g. to guide reviews), on the truth tables regarding a particular coverage criterion (as an aid for the tester, see [Av04]), and on the resulting coverage.

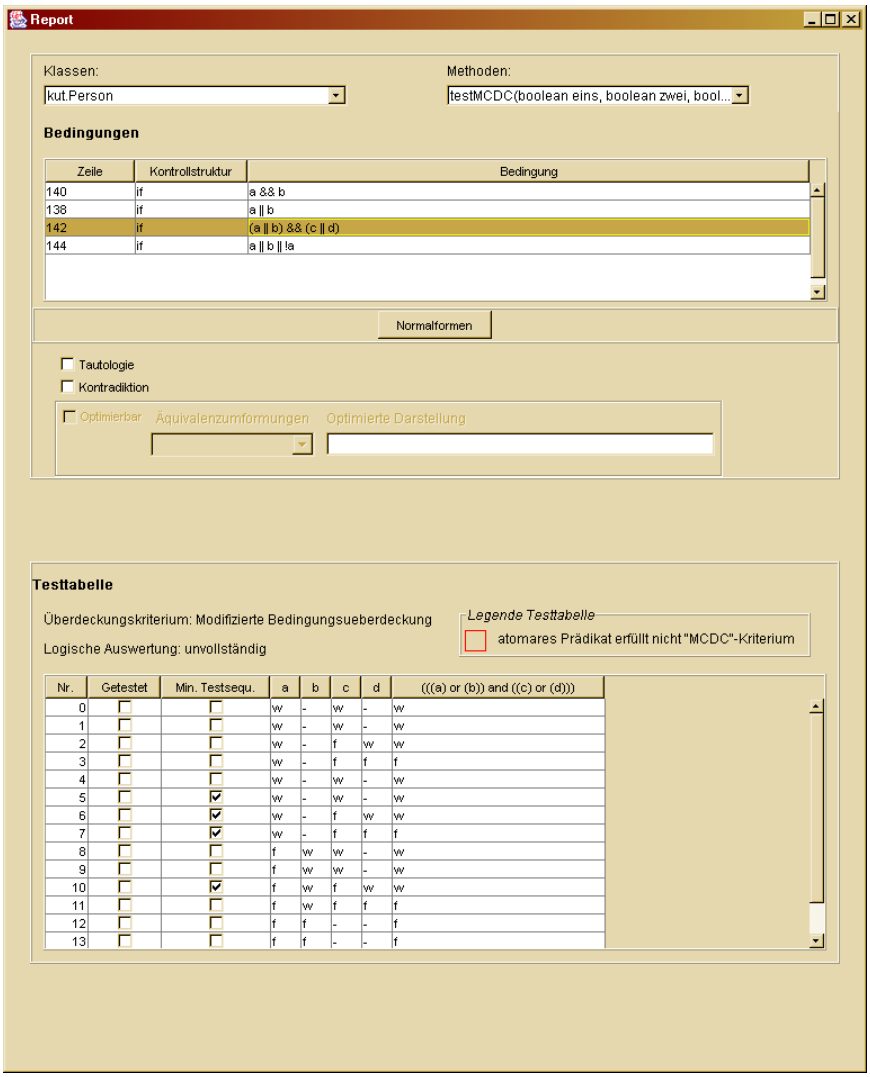

Figure 6. HTML-Report for a compound predicate

6 Related Work

In this chapter we sketch some recent related work on “contract testing”, i.e. test case specification techniques relying on contracts.

Groß reports on a built-in contract testing method for checking the pair-wise interactions of components in component-based software construction at integration and deployment time [Gr02a][Gr02b]. Functional test cases are specified mainly using state-based testing, but also “classical” structural testing techniques are applied.

Edwards describes a framework for practical, automated black-box testing of component-based software [Ed01]. He outlines a general strategy for automated black-box testing of software components that includes automatic generation of component test drivers, automatic generation of black-box test data, and automatic or semi-automatic generation of component wrappers that serve as test oracles. Test case generation is based on flowgraph techniques.

McGregor and Sykes construct test cases from contracts by identifying all possible combinations of situations in which a precondition can hold and postconditions can be achieved and adding test cases, to address what happens when a precondition is violated [MS01]. In [CM99], Cho and McGregor take pre- and postconditions as partial test oracles for the state-based testing of components.

None of the authors above utilize predicate coverage criteria for test case generation.

7 Conclusions

We discussed various predicate testing coverage criteria known from the literature, demonstrated how to adapt them for functional test case design focusing on “design by contract” specifications, and reported on a tool supporting both structural and functional predicate coverage testing for Java programs with OCL specifications for invariants and pre- and postconditions.

In our laboratory case studies we found that besides efficient test case specification another benefit of our approach and the tool was due to its predicate analysis and transformation abilities. Both tester and developer appreciated the corresponding support of contract specification in the design phase.

Until now the tool was only used in experimental settings, so before conducting industrial case studies we first have to evaluate (and maybe enhance) its robustness and scalability.

8 Bibliography

[Av04] Averstegge, M.: Konzeption und Realisierung einesWerkzeugs für strukturelle und funktionale Bedingungs- Überdeckungstests von Java-Programmen mit OCL-Spezifikationen. Diplomarbeit, Fachbereich Informatik, FernUniversität Hagen, November 2004.

[BS98] Bokowski, B.; Spiegel, A.: Barat – A Front-End for Java. Technical Report B-98-09; Freie Universität Berlin, Institut für Informatik; Dezember 1998.

[CM99] Cho, I.; McGregor, J.D.: Component Specification and Testing Interoperation of Components. Proc. IASTED’99, 3rd International Conference on Software Engineering and Applications, Honolulu, Hawaii, 1999

[DO178B] DO-178B / ED-12 B: Software Considerations in Airborne Systems and Equipment Certification. RTCA / EUROCAE, December 1992.

[Ed01] Edwards, S. H.: A framework for practical, automated black-box testing of component-based software. Journal of Software Testing, Verification, and Reliability, No.11, 2001:97–111.

[GH98] Gagnon, E.M., Hendren, L.J.: SableCC, an Object-Oriented Compiler Framework. In: 10th International Conference on Modelling Techniques and

Tools for Computer Performance Evaluation (TOOLS-98). Sable Research Group, School of Computer Science McGill University, Quebec, Kanada, 1998.

[Gr02a] Groß, H.-G.: Component+ Methodology, Built-In Contract Testing: Method and Process. IESE-Report No. 030.02/E, Oct. 2002.

[Gr02b] Groß, H.-G.: Component+ Methodology, Built-In Contract Testing: Technological Foundations. IESE-Report No. 073.02/E, Dec. 2002.

[KSW01] Kösters, G., Six, H.-W., Winter, M.: Coupling Use Cases and Class Models as a Means for Validation and Verification of Requirements Specifications. Requirements Engineering, Vol. 6, Nr. 1, Springer Verlag, London, 2001; pp. 3–17.

[MS01] McGregor, J.D.; Sykes, D.A.: A Practical Guide to Testing Object-Oriented Software. Addison Wesley, Boston, 2001.

[Me97] Meyer, B.: Object Oriented Software Construction. Prentice Hall, Upper Saddle River, 1997

[TV+94] Tai, K.-C., Vouk, M. A., Paradkar, A. M., Lu P.: Evaluation of a predicate-based software testing strategy. IBM Systems Journal, Vol. 33, No. 3, 1994, pp. 445-457.

[Ve04] Van Veenendaal, E. (Ed.): Glossary of terms used in software testing. Version 0.2, Int. Software Testing Qualification Board, 2004.

[WK03] Warmer, J., Kleppe, A.: Object Constraint Language 2.0. Pearson/Addison Wesley, Reading, 2003.

[We03] Wefels, H.-G.: Konzeption und Realisierung eines datenbankgestützten Testwerkzeuges zur Überdeckungsanalyse von Java-Programmen. Diplomarbeit an der Fernuniversität Hagen, Fachbereich Informatik, März 2003.

[WGM85] Weiser, M.D.; Gannon, J.D.; McMullin P.R.: Comparison of Structural Test Coverage Metrics. IEEE Software, Vol 2, No 2, March 1985, pp. 80-85.

[Wi01] Winter, M.: Testfallermittlung aus Komponentenschnittstellen. Imbus QS-Tag 01, Nürnberg, 2001.

[Wi04] Winter, M: Testing in the Component Age. Proc. 1st Int. Workshop on Software Quality (SOQUA04), LNI P-58, GI, Bonn, 2004

[WW03] Winter, M.; Wefels, H.G.: Überdeckungsmessung von Java-Programmen. GI Softwaretechnik Trends, Band 23, Heft 4, Nov. 2003.

[ZHM97] Zhu, H.; Hall, P.A.V.; May, J.H.R.: Software Unit Test Coverage and Adequacy. ACM Computing Surveys, Vol. 29, No. 4, December 1997.

9 Authors' biographies

Dipl.-Inform. Michael Averstegge

German Distance University of Hagen, Faculty of Electrical Engineering and Information Engineering, Universitätsstraße 27, D-58084 Hagen, Germany. Mailto: Michael.Averstegge@FernUni-Hagen.de.

Michael Averstegge is research assistant at the German Distance University of Hagen (since 2005) with the research fields Software Engineering and Software Quality Assurance. Since 1991 he worked as Developer, Projektleader, Senior-Consultant and Manager, at last (1998) SerCon (IBM) and (2000) Pluralis (Plönzke Holding).

Mr. Averstegge has a diploma in computer science (Dipl.-Inf.) from the German Distance University of Hagen (2004, study simultanous to his work) and a Magister (Master of Science) in Philosophy and German language from the University of Münster (1991).

Prof. Dr. Mario Winter

University of Applied Sciences Cologne, College of Computer Science and Engineering Sciences, Am Sandberg 1, D-51643 Gummersbach, Germany
mailto:winter@gm.fh-koeln.de

Mario Winter is professor in the computer science institute at the University of Applied Sciences Cologne, lecturing and researching on software development and project management with a special research focus on software quality. Since 1996 he is member of the GI (currently speaker of the GI SIG "Testing, Analysis and Verification of Software") and since 2004 he is also member of the GTB (German Testing Board).

From 1987 to 1994 Prof. Winter lead scientific software projects in control systems design at the University of Wuppertal and from 1983 to 1987 he was involved in industrial software projects (CAD/CAM, systems programming).

Mr. Winter received a PhD (Dr. rer.nat.) from the German Distance University of Hagen (FernUniversität) in 1999, having submitted a thesis on testing object oriented software. From 1994 to 2002 he was a research assistant at the German Distance University of Hagen with the research fields Software Engineering and Software Quality Assurance. Dr. Winter has a diploma in computer science (Dipl.-Inf.) from the German Distance University of Hagen (1994, study simultaneous to his work at the University of Wuppertal) and a diploma in electrical engineering/information technology (Dipl.-Ing.) from the University of Siegen (1983).