

Nondeterministic Coverage Metrics as Key Performance Indicator for Model- and Value-based Testing

David Faragó (farago@kit.edu)

Karlsruhe Institute of Technology, Institute for Theoretical Computer Science

Abstract. Assessing the testing or software development process by only using KPIs can easily be misleading. A better solution is a paradigm shift to value-based software engineering, which integrates value considerations into software engineering and offers a broader and more technical view on KPIs. Coverage metrics are such a technical view and a very helpful KPI.

This paper combines value-based testing and model-based testing of nondeterministic systems and introduces new coverage metrics for this. Therewith, the quality of test suites is raised and value-based testing gets strongly supported, e.g., by KPIs derived from nondeterministic coverage metrics and better requirements-based and risk-based testing.

Keywords. Value-based software engineering, value-based testing, key performance indicators, return on investment, model-based testing, coverage metrics, nondeterminism

1 Introduction

Testing already consumes up to 50% of the software development costs [1], so it needs to be operated and managed efficiently on all hierarchy levels. To enable sensible decisions, especially in higher management, concise information is needed. *Key Performance Indicators (KPIs)* deliver such: They are measures which evaluate the progress and degree of performance of a particular activity from certain viewpoints. So the value of all software development activities is mapped onto some measures. But concisely quantifying complex systems is very difficult. Hence often wrong KPIs or too few dimensions are considered (cf. [2]), or interpreted wrongly (e.g. *work in progress*, cf. [16,14]). So they mislead management into bad decisions. In testing, for instance, often *defect detection percentage* (i.e. bugs fixed / total bugs found) is used without factoring in the severity of bugs. Hence it cultivates (especially in combination with the KPI *lines of code per day*) quick and dirty coding and fixing many little bugs afterwards. More meaningful KPIs, e.g., *return on investment (ROI)* (cf. Section 2.2), *sprint burndown* (cf. [15]) or *coverage metrics* (cf. Section 3), require more context, i.e., a broader, deeper, more technical view on software engineering.

Hence Section 2.2 will give this broader view by introducing *value-based testing*. It will motivate looking into the following technical details: Coverage metrics (Section 3), model-based testing (MBT) and nondeterminism (Section 4), and finally, how and which coverage metrics help for model-based testing and for KPIs (Section 5). Section 6 gives a summary.

2 Value-based Software Engineering

2.1 Introduction

Value is more than money, it is the relative economic and utilitarian worth. With this general definition, everyone strives in his decisions and actions to maximize

his (personal) value. When KPIs are being used, business value of all development and testing activities are mapped onto some measures, so that higher management can sensibly steer the software development department. In contrast, *value-based software engineering (VBSE)* integrates business value considerations, mainly by prioritization, traceability and risk considerations, into the full range of software engineering principles and on all hierarchy levels. It offers means to better downstream value to all parties. Therefore, everyone involved – manager, executive, analyst, process engineer, software engineer, and tester – better understands the implications of his own decision. Thus all actions and decisions are monitored and synchronized to maximize the corporate values made explicit - they are enhanced to being *value-based (VB)*. Hence common mischiefs, e.g., development only striving for elegant design while marketing only valuing large functionality, are avoided.

So this paradigm shift investigates the value of all software development activities, how to measure, increase and enforce it. For this, the *VBSE Agenda* (cf. [4,2]) integrates value considerations into the software development process and into management activities closely related to software development: It covers *VB requirements engineering*, *VB architecting*, *VB design and development*, *VB verification and validation (V&V)*, e.g., via risk-based and VB testing (cf. Section 2.2), *VB planning and control*, e.g., by multi-attribute planing and decision support (cf. [2]), *VB risk management*, e.g., using agile methods (cf. [10]), and *VB quality management*. These elements reinforce each other and are enhanced by the contributions of this paper.

2.2 Value-based Testing

The value of test cases is usually a Pareto distribution, i.e., 80% of the value is covered by 20% of the test cases (cf. [6]). Hence a value-neutral approach,

which treats each artifact (e.g., path, scenario or requirement) equally important, is not efficient. Value-based testing aligns the test process and investments with the given value objectives by prioritizing the test basis and testware (e.g., requirements, test cases and bugfixes).

That way, testing can maximize its *return on investment (ROI)*, i.e., the KPI $(\text{benefits} - \text{costs})/\text{costs}$. The cost of testing (cf. [13]) can be partitioned into:

- the *costs of conformance*, for achieving quality, which includes prevention costs (e.g., for extended prototyping and modeling tools) and appraisal costs (e.g., for test execution).
- the *costs of non-conformance*, incurred because of a lack of quality, which includes internal failure costs (e.g., for defect fixing) and external failure costs (e.g., for technical support).

The benefit of testing are:

- either short-term, e.g., saved rework because of early bug detection and reduction of uncertainty in planning by assessing risks;
- or long-term, by detecting the strength and weakness of the development process.

Hence, a good ROI in testing means that costs of conformance + internal failure costs \leq savings in external failure costs (cf. [3]). VB testing mainly reduces costs of conformance by prioritization. Using the more formal MBT approach (cf. Section 4) helps to detect bugs early, so internal failure costs are reduced. Additionally, using nondeterminism can enhance bug prevention, i.e., reduce prevention costs, since the models can be more abstract and designed even earlier. Coverage metrics (cf. Section 3) improve test selection, resulting in fewer and more meaningful tests and therefore also decreasing the KPI *execution time per test case*, i.e., the appraisal costs.

The following practices are essential for putting value-base testing into effect:

- *Requirements-based testing*, to assure that the requirements are completely and accurately covered. Since they capture the business values agreed upon, this helps in VB testing. Requirements can be prioritized by associating weights to them. The best implementation of requirements-based testing is deriving black-box tests from the requirements (cf. [2]). This can be done automatically using MBT with the requirements included in the specifications. This automation also empowers traceability from tests to the original requirements, as well as early verification of requirements. The important prioritization of requirement tests can be put into effect by appropriate coverage criteria, which can factor in the weights of the requirements. Hence requirements-based testing is risk-oriented.
- *Risk-based testing* deals with *risk exposure*, which is $(\text{probability of loss}) \cdot (\text{size of loss})$. These values are not only useful to prioritize tests, but give im-

portant feedback - not only at the end of testing, but permanently as a KPI. It indicates, besides the progress of the project, areas that potentially contain errors and need to improve. By knowing the probability that a bug occurs, and its resulting loss (via weights), its fix can be prioritized. Section 5 will show how MBT with coverage metrics and quantification of nondeterminism implements risk-based testing.

- *Iterative and concurrent testing* copes with changing requirements and risks, caused by insights from development and testing, or from changing business values. Fast and flexibly responding to these changes is very important and the cause for most modern development processes being highly iterative. MBT's concise models support fast and flexible changes (cf. [10]). Nondeterminism also helps since it enables leaving undecided points open in the model (e.g., returning a `Collection` and later refining it to a specific `List`). This avoids unnecessary rework and reduces the initial effort, hence further increasing the ROI.

3 Coverage Metrics

Coverage metrics are the percentage of the code or the (functional or requirements) specification that the executed tests have covered so far. Depending on what artifacts should be covered, different metrics are formed, e.g., state(ment), transition, Modified Condition/Decision, or LCSAJ coverage (cf. [17,8]).

These metrics originate from white-box testing, but since formal specifications contain control flow, data and conditions, they can be applied at specification level, too. If the source code is present, it can still be used for additional coverage metrics.

A coverage metric is a KPI for testing: It can be used in management to decide whether and where quality assurance needs to improve and as exit criterion, either agreed upon in the test plan, or demanded by a required certificate (like DO178B). During development, a coverage metric informs about the progress, afterwards it raises confidence.

But a coverage metric can also have deep technical influence: When tests are being generated (manually or automatically), each test case should contribute as much as possible towards the goal of reaching the desired coverage. So the coverage metric helps prioritizing for test selection (cf. Section 5). Hence guidance by coverage metrics in test generation is absolutely value-based.

4 Model-based Testing of Nondeterministic Systems

4.1 Introduction

Model-based testing (MBT) originates from black box conformance testing, i.e., to check that the system under test (SUT) conforms to its functional specification.

If this reference behavior is given in a formal language, MBT can automatically generate conformance tests. The formal specification is often a *Labelled Transition System (LTS)*. Nondeterminism helps in specifying requirements, cf. Section 4.2. Section 4.3 describes the possible methods that MBT can apply.

4.2 Nondeterminism

Complex (e.g. distributed) SUTs (seemingly) behave nondeterministically, i.e., react varyingly after applying a fixed stimulus, e.g., occasionally with an exception. The reason is lower levels, such as the operating system and network, which are not under the testers control and too complex to model and monitor.

A tester can cope with this by also using nondeterminism when specifying the SUT, to cover all possible behaviors, for instance with the test code `if (NetworkException) {} else {}`.

Nondeterminism can be specified more concisely when MBT with formal specifications are used. It helps to model more efficiently via abstraction, i.e., describing several behaviors without having to care which one occurs (e.g. what data is present in the underlying database).

The possible input choices, i.e., the input transitions from a given state, are the *nondeterminism that is controllable* by the tester. *Uncontrollable nondeterminism*, i.e., *nondeterminism of the SUT*, is

- either multiple different outputs from one state,
- or unobservable nondeterminism of the LTS itself: multiple identical labels from one state, or internal transitions. These are often the result of composition.

4.3 Model-based Testing Methods

To generate tests, MBT

1. traverses paths through the graph of the specification. These paths are considered as test cases: The SUTs inputs on the paths are the stimuli, i.e. drive the SUT, the outputs are the oracles, i.e. check that the SUT behaves conformly.
2. If the test cases are too abstract for execution, they are refined to the SUT’s technical level.
3. These tests are then executed.
4. The results are finally analyzed, leading to the verdicts *pass*, *fail* and *inconclusive* (or the like).

MBT methods can be categorized depending on how they intertwine these steps:

Off-the-fly MBT (cf. [5]) only performs the first two steps. Since a priori test generation does not know which nondeterministic choices the SUT will take, all choices have to be considered.

On-the-fly MBT (cf. [5,18,7,8]), uses the other extreme of strict simultaneity, i.e., all four steps are performed in lockstep for each single transition. Hence the above deficiencies are no longer present, but guidance is very weak.

*Lazy on-the-fly MBT*¹, takes the happy medium: It executes subpaths of the model lazily on the SUT, i.e., only when there is a reason to, e.g., when a test goal, a certain depth, an inquiry to the tester, or some nondeterministic choice of the SUT is reached. Hence the method can backtrack within subgraphs of the model (cf. [9]). While backtracking, the method can harness dynamic information from already executed tests, e.g., nondeterministic coverage criteria (cf. Section 5).

5 Coverage Metrics for Model-based Testing of Nondeterministic Systems

In MBT, coverage metrics help to guide the traversal through the specification graph, such that the newly generated tests really contribute to the desired coverage, i.e., probe new behaviors. For the different nondeterministic behaviors, test segments must be executed repeatedly. But how often? Some behavior might (almost) never be reached. This section looks at modified coverage criteria that are suited for nondeterminism of the SUT: Section 5.1 for deterministic LTSs, Section 5.2 also for nondeterministic LTSs. For further details, see [11].

5.1 For Deterministic LTSs

Uncontrollable nondeterminism via outputs is still present. Current tools, e.g., Spec Explorer, simply re-executes test segments a constant k times. So counting how often a state s (with $s \in S$, the set of all states) is visited is sufficient to put this into effect. But k might be too often or too seldom. The method does not give any information about the nondeterminism, and hence can also not adapt to it.

A solution that is still simple but more revealing is to measure the coverage of nondeterminism of the SUT by what the author defines as *n-choices* coverage metrics: They measure the percentage of visited states with multiple outputs (so called *nondeterministic states*) where at least n output choices have been traversed: $|\{\text{nondeterministic } s \in S | \text{at least } n \text{ different output transitions of } s \text{ have been traversed}\}| / |\{\text{nondeterministic } s \in S | s \text{ has at least } n \text{ output transitions}\}|$. This is a generalization of [12]. So *1-choices* measures how many nondeterministic states have been visited (and left via an output transition). *2-choices* checks how many nondeterministic states really behave nondeterministically in the SUT. *all-choices* := $|\{\text{nondeterministic } s \in S | \text{all different output transitions of } s \text{ have been traversed}\}| / |\{\text{nondeterministic } s \in S\}|$ measures to what degree the specified output really occurs in the SUT, i.e., how much underspecification we have (cf. [9]). As in classical coverage metrics, there are many assumptions, e.g., 100% transition coverage

¹ currently developed by the author and funded by Deutsche Forschungsgemeinschaft (DFG)

$\not\Rightarrow$ 100% all-choices $\not\Rightarrow$ 100% 2-choices $\not\Rightarrow$ 100% 1-choices.

Although these coverage metrics give information about nondeterminism and can be used as exit criteria, they are in general not informative enough to efficiently guide the state space traversal such that nondeterministic states are visited optimally often for high coverage of the possible nondeterministic choices. The best solution is *quantification of nondeterminism*: By counting for all relevant artifacts (e.g., states and transitions) the number of times they are traversed or used (similar to the usual code instrumentation in white-box testing), the relevant probabilities can be approximated, e.g., $P[t] = \text{count}(t)/\text{count}(s)$ that a transition $t = s \rightarrow s'$ is taken in s .

These probability distributions over the nondeterministic output choices can be used to compute the probability of reaching some state or requirement, by interpreting the transition system as Markov chain. The missing probabilities for the input labels are implicitly set to one since their choice is under the control of the MBT tool. Thus coverage metrics can factor in both probabilities and the weights (e.g., $P[\text{reaching requirement}_i] \cdot \text{Weight}[\text{requirement}_i]$) when used for guidance. These values are not only useful for guidance, but also as approximation for other probability values, such as reaching some error, probability of loss or risk exposure. (The size of loss can be inferred from the weights of the requirements.) Hence they can be used as KPI and for requirements-based and risk-based testing, e.g., to reduce planning uncertainty and to guarantee lower variance of quality.

5.2 For Nondeterministic LTSs

This section investigates whether the solutions from the last section can be generalized to nondeterministic LTSs. Since nondeterminism of the LTS is not (immediately) observable, we are not in a single state at any one time during traversal, but in a set of possible states $S_{\text{current}} \subseteq S$. Hence, coverage criteria measure only *possibly* covered artifacts (such as states or transitions), since maybe a different path was taken. Therefore, coverage metrics are much less meaningful and rather complex (e.g., non-monotonic, cf. [11]). So using the metric as guidance can be very misleading.

Updating computations later on when choices of nondeterminism of the LTS become observable is very complex. Hence we try to decide instantly by also quantifying nondeterminism of the LTS. For this, S_{current} is replaced by the probability distribution P_{current} over S . P_{current} is updated by computing random walks. This requires a transition probability matrix. If we do not have probability values given for the nondeterminism of the LTS (e.g., from previously testing the components individually), we need to guess them for our matrix, e.g., equidistributedly. P_{covered} gives the probability that a state or other artifact is covered. It can be computed using P_{current} .

So for nondeterministic LTSs, quantification is unfortunately very costly and possibly too rough an approximation. Since the benefit is large (cf. Section 5.1), this current research topic is still worth looking into.

6 Summary

Many aspects of VBSE are supported by MBT of nondeterministic systems with coverage criteria: Tracability back to the weighted requirements, prioritization, and KPIs. These are obtained by new coverage metrics, e.g., quantified nondeterminism. The KPIs not only improve guidance, but offer valuable information, e.g., risk exposure. For nondeterminism of the LTS, quantification is intricate and a current research topic.

References

1. Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.
2. Stefan Biffl, Aybke Aurum, Barry Boehm, Hakan Erdogmus, and Paul Grnbacher, editors. *Value-Based Software Engineering*. Springer, Berlin, 2006.
3. Rex Black. *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*. John Wiley & Sons, Inc., NY, USA, 2nd edition, 2002.
4. Barry Boehm. Value-based software engineering: reinventing earned value monitoring and control. *SIGSOFT Softw. Eng. Notes*, 28, March 2003.
5. Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-based Testing of Reactive Systems*, volume 3472 of *LNCIS*. Springer, 2005.
6. J. Bullock. Calculating the value of testing. *Software Testing and Quality Engineering*, pages 56–62, June 2000.
7. Margus Veanes et al. Model-based testing of object-oriented reactive systems with spec explorer. In *Formal Methods and Testing*, pages 39–76. Springer, 2008.
8. David Faragó. Coverage criteria for nondeterministic systems. *testing experience, The Magazine for Professional Testers*, pages 104–106, September 2010.
9. David Faragó. Improved underspecification for model-based testing in agile development. *Volume P-179 - Proceedings of the Second International Workshop on Formal Methods and Agile Methods*, September 2010.
10. David Faragó. Model-based testing in agile software development. In *30. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV), Testing meets Agility*, Softwaretechnik-Trends, 2010.
11. David Faragó. Nondeterministic coverage criteria for model-based testing. *Technical Report 2011-7, Department of Computer Science, University of Karlsruhe*, 2011.
12. Gordon Fraser and Franz Wotawa. Test-case generation and coverage analysis for nondeterministic systems using model-checkers. In *ICSEA*, page 45. IEEE Computer Society, 2007.
13. F. M. Gryna. *Quality and Costs, Juran's Quality Handbook*. McGraw-Hill, 1999.
14. KPI Library. <http://kpilibrary.com/home/>. (November 2010).
15. Roman Pichler. *Agile Product Management with Scrum: Creating Products That Customers Love*. Addison-Wesley, 2010.
16. Jeff Smith. *The K.P.I. Book*. Insight Training & Development Limited, 2001.
17. Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard*. dpunkt, 3. edition, 2005.
18. Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 1 edition, 2007.