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. Putting measures such as KPIs simply on top of testing or the whole software development process 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 important KPI.
This paper combines value-based testing and model-based testing of nondeterministic systems and introduces new coverage metrics for this (e.g., quantified nondeterminism). Therewith, the quality of test suites are raised and value-based testing gets strongly supported, e.g., by new KPIs 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, quantification

1 Introduction

Testing already consumes up to 50% of the software development costs ([1]), so it needs to be financed and managed efficiently on all hierarchy levels. To enable sensible decisions, especially in higher management, conscise 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 consciesly 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. [15,13]). So they mislead management into bad decisions. In testing, for instance, *defect detection percentage* (i.e. bugs fixed / total bugs found) does not factor 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. [14]) or *coverage metrics* (cf. Section 3), require more context, i.e., a broader and 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 (cf. Section 3), model-based testing (MBT) and nondeterminism (cf. Section 4), and finally, Section 5 investigates how and which coverage metrics help for model-based testing and for KPIs. 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 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 their 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 as well as management activities closely related to software development:

- *VB requirements engineering* identifies success-critical stakeholders and prioritize requirements accordingly.
- *VB architecting* finds architectural solutions according to the prioritized requirements, and supports traceability back to them.
- *VB design and development* further refines the architecture and objectives. For this, traceability is important, to guarantee that the value considerations are really inherited.
- *VB verification and validation* (V&V) firstly checks that the previous activities really followed the value objectives. Secondly, the value considerations are applied to V&V itself by prioritization.

Two prominent techniques for VB V&V are *risk-based* and *value-based testing*.
- *VB planning and control* integrates value considerations into classical planing and control, and also contains techniques to plan and control the value itself (e.g., multi-attribute planing and decision support).
- *VB risk management* identifies, analyses and mitigates risks, and helps in prioritizations. It includes many VB techniques, for instance from risk-based analysis and predictability, risk-based simulation, risk-based testing and agile methods (cf. [10]).
- *VB quality management* prioritizes quality factors, e.g., by multi-attribute decision support techniques or VB approaches to security.

These elements reinforce each other and are enhanced by the contributions of this paper (cf. Section 6).

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 and test cases).

That way, testing can maximize its *return on investment (ROI)*, i.e., the KPI *(benefits - costs)/costs*. The cost of testing (cf. [12]) 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 strengths and weaknesses 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 (*probability of loss*) $\cdot$ (*size of loss*). These values are not only useful to prioritize tests, but give important 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, 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. [16,8]).

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. Which coverage criterion is best on the specification level varies, e.g., MC/DC needs not be as powerful as is often expected (e.g. MUMCUT detects more fault classes, cf. [18]).

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. It can be used as exit criterion, and afterwards to raise confidence that the implementation is complete.

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., checking 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)* of some kind. MBT can automate all kind of tests: unit, integration, system and acceptance tests. For unit tests, MBT's models must be sufficiently refined to give details at source code level. For acceptance testing, requirements must be integrated into the model. 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 are lower levels, such as the operating system and network, that 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 nondeterminism of the LTS itself: multiple identical labels from one state, or unobservable, internal transitions. These are often the result of composition.
- or multiple different outputs from one state.

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, used mainly by older tools, e.g., TGV and Spec Explorer (cf. [5]), only performs the first two steps. Thus test execution and evaluation is done subsequently with classical tools (e.g., a TTCN framework). Since a priori test generation does not know which nondeterministic choices the SUT will take, all choices have to be considered. If many long branches are discarded because the SUT does not actually follow them, off-the-fly is very inefficient. Other dynamic adaptations to the test generation are not possible either.

On-the-fly MBT, applied by many newer tools, e.g., TorX, UPPAAL TRON and also Spec Explorer (cf. [5,17]), uses the other extreme of strict simultaneity, i.e., all four steps are performed in lockstep for each single transition. Hence the exploration and test generation can be guided by the observations made of the SUT for preceding stimuli, but guidance itself and therefore test selection is strongly weakened. Thus all current tools explore the specification randomly (cf. [8] for details about the tools' coverage capabilities and their application). That might be the reason why nondeterministic coverage criteria for guidance have not been looked at closely. The results of doing this in Section 5, however, are not only useful for lazy on-the-fly MBT (see next paragraph), they also give general insight, can be applied to VBSE and to some extent for on-the-fly MBT.

Lazy on-the-fly MBT, currently developed by the author[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]).

[1] funded by Deutsche Forschungsgemeinschaft (DFG)

While backtracking, the method can harness dynamic information from already executed tests, e.g., nondeterministic coverage criteria (cf. Section 5). All on-the-fly tools have difficulties with guidance, hence this is the solution. Its improved test selection results in fewer, more meaningful and more flexible (and hence reproducible despite nondeterminism) tests.

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 reaching the desired coverage. For simpler systems, this has been practiced, e.g., via off-the-fly MBT and transition coverage. For more complex systems, two difficulties arise: Firstly, if the complete model cannot be built a priori, but only during on-the-fly MBT, the coverage metrics can only make statements about test progress on the part of the graph already traversed. This is to some extent improved using lazy on-the-fly, but an accurate measure in percentage related to the full graph is generally still not possible. The larger the explored graph gets, though, the more accurate the percentage becomes. Some parts of the graph might have to be re-executed multiple times, hence the percentage can become completely accurate. Secondly, since nondeterminism of the SUT cannot be controlled by the tester, some behavior might (almost) never be reached. Therefore, coverage criteria must be considered with a grain of salt and a desired level of classical coverage might not at all be achievable. To try to probe the different behaviors, test segments must be executed repeatedly. But how often? 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.

5.1 For deterministic LTSs

Coverage metrics for deterministic LTSs are easier than for nondeterministic LTSs since each trace leads to exactly one state, not a set of possible states. Uncontrollable nondeterminism via outputs is still present, so classical coverage metrics (e.g., transition) are not very meaningful, e.g., they can be misleading or not at all achievable.

To probe the possible different behaviors, test segments must be executed multiple times, but how often? Spec Explorer simply re-executes a constant number of times (without the automata being unwound explicitly), so counting how often a state s (with $s \in S$, the set of all states) is visited is sufficient. But this 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 via a mixture of statement and transition coverage, by what the author generally 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: $|\{\text{nondeterministic } s \in S|$ at least n different output transitions of s have been traversed$\}/|\{\text{nondeterministic } s \in S | s$ has at least n output transitions$\}$. This is a strong generalization of [11]. So *1-choices* coverage is closest to the classical statement coverage and measures how many nondeterministic states "have been visited" (and left via an output transition). For 1-choices to be sensible, S must be known from the beginning of testing. *2-choices* checks how many nondeterministic states really behave nondeterministically in the SUT. If we have special knowledge or constraints of the SUT, $n > 2$ might also be usefull, as well as *all-choices* $:= |\{\text{nondeterministic } s \in S|$ all different output transitions of s have been traversed$\}/|\{\text{nondeterministic } s \in S\}$. So all-choices measures to what degree the specified output really occurs in the SUT, i.e., how much underspecification we have (cf. [9]). Just as classical coverage metrics, these are also related, e.g., 100% transition coverage $\not\Longleftarrow$ 100% all-choices $\not\Longleftarrow$ 100% 2-choices $\not\Longleftarrow$ 100% 1-choices.

Although these coverage metrics give information about nondeterminism and can be used as exit criteria, they are in general not flexible enough to 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., $P[t] = count(t)/count(s)$ that a transition $t = s \rightarrow s'$ is taken when being in s.

Note that the probabilities used in Spec Explorer are not quantification for the nondeterministic choices, computed on the fly, but must somehow be provided by the tester and are used for prioritization.

In the author's approach, weights and probabilities are orthogonal: On the one hand, weights are associated to requirements. Requirements are used for test generation and can be traced back to. The weights can be used for automatic prioritization, such that more important requirements are tested earlier and more intensely. Size of loss can also be inferred from the weights, or associated to the requirements as additional value. On the other hand, in our quantification technique, probability is the quantified nondeterminism of the SUT and approximated on-the-fly. 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 implicitely 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[reaching requirement$_i$] $\cdot$ $Weight$[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. 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.

In seldom cases, an input behavior model that mimics a user or application domain is available. Then it makes sense to build a complete Markov chain for KPIs and to use this input model to drive testing, instead of using our guidance.

In conclusion, quantification strongly helps guidance and VBSE, in particular as KPI and for requirements-based and risk-based testing. But can this quantification be generalized for nondeterministic LTSs in the following section?

5.2 For nondeterministic LTSs

Because of nondeterminism of the LTS, we are not in a single state during traversal, but in a set of possible states $S_{current} \subseteq S$. 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. in Figure 1, state coverage is non-monotonic: After traversing b, the possibly covered states are S, but after ba_3, they are $\{init, s_2, s_3\} \subsetneq S$.

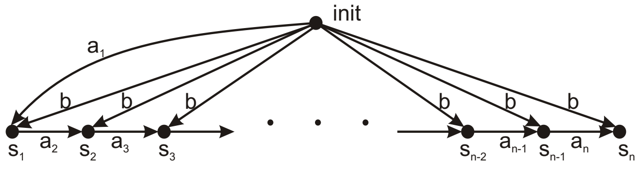

Fig. 1. Transition system with non-monotonic, misleading coverage metric

If the metric is also used as guidance, this becomes very misleading: The trace $a_1a_2 \cdots a_n$ would definitely visit all n states, but transition b is prefered since thereafter all S is possibly covered, although only $init$ and one further state have actually been visited.

Using on-the-fly determinization alternatively, e.g., coverage metrics on the state space 2^S instead of S, is not useful, since it gives almost no information about the coverage of the original artifacts, e.g., single states. Hence, we need to look into $S_{current}$ and also quantify the possible coverage of states. This uses the quantification (in P_{visit} below) from Section 5.1 and is similar to random walks: $\forall s \in S_{current} : P'_{covered}[s] := 1-(1-P_{covered}[s])(1-P_{visit}[s \text{ from } \mathrm{S}_{current}]) = P_{covered}[s] + (1-P_{covered}[s]) \cdot P_{visit}[s \text{ from } \mathrm{S}_{current}]$, with values set to 0 at the beginning. Therefore, the coverage gain of the last test step was $\Sigma_{s \in S_{current}} P'_{covered}(s) - P_{covered}(s) = P_{visit}[s \text{ from } \mathrm{S}_{current}] \cdot \Sigma_{s \in S_{current}}(1 - P_{covered}(s))$.

So for nondeterministic LTSs, quantification becomes very complex: Probability values are created on-the-fly and immediately used for successive quantifications and for guidance. Hence the method may be with a high propagation of uncertainty. It might even turn out that quantification for nondeterministic LTSs is too costly or too rough an approximation. But the benefit is large, as in Section 5.1, so this current research topic is worth looking into.

In conclusion: Research and industry have realized that nondeterminism has become necessary to cope with the complexity, so the emerging testing tools integrate it. But the tools can still be improved by using better coverage metrics that incorporate nondeterminism (cf. [7]), e.g., n-choices coverage or quantification. For nondeterministic LTSs, quantification is complex and a current research topic.

6 Summary

For today's complex software, testing and KPIs are also becoming very complex and difficult to handle. VBSE helps to focus on the business values, and to enforce them. Many aspects of VBSE are supported by MBT of nondeterministic systems with coverage criteria:

- Flexible modelling (with nondeterminism) helps VB requirements engineering, VB architecting, VB design and development and VB quality management to formalize their artifacts and priorities. The use of the prioritized artifacts can then be automated.
- VB testing and VB planning and control can also largeley be automated via MBT. The priorities are factored in by new coverage metrics, which also function as new KPIs. Traceability back to the requirements is facilitated by MBT, too.
- VB risk management is supported by the benefits from the points above and by coverage metrics that use quantification.

But not only VBSE improves, also the quality of the test-suite, i.e., shorter and fewer tests are sufficient, more revealing, and also increase reproducibility.

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. *SIGSOFT Softw. Eng. Notes*, 28:3–, March 2003.
5. Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-based Testing of Reactive Systems*, volume 3472 of *LNCS*. 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. 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.
12. F. M. Gryna. *Quality and Costs, Juran's Quality Handbook*. McGraw-Hill, 1999.
13. KPI Library. http://kpilibrary.com/home/. (November 2010).
14. Roman Pichler. *Agile Product Management with Scrum: Creating Products That Customers Love*. Addison-Wesley, 2010.
15. Jeff Smith. *The K.P.I. Book*. Insight Training & Development Limited, 2001.
16. Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard*. dpunkt, 3. edition, 2005.
17. Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 1 edition, 2007.
18. Yuen-Tak Yu and Man Fai Lau. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. *Journal of Systems and Software*, 79(5):577–590, 2006.