Proceedings of the

17th Dutch Testing Day

Testing Evolvability

Editors: Mariëlle Stoelinga and Mark Timmer

November 29, 2011

University of Twente, Enschede

Programme Committee

Steering Committee

Organisation

Publication details

Proceedings of the 17^{th} Dutch Testing Day
Edited by Mariëlle Stoelinga and Mark Timmer

Published: February 2012
University of Twente, Enschede, The Netherlands

CTIT Workshop Proceedings Series WP12-01

Centre for Telematics and Information Technology
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

ISSN 0929-0672

Preface

These are the post-proceedings of 17th Dutch Testing Day, held on the 29th of November 2011 at the — recently completely renovated — campus of the University of Twente. These post-proceedings cover a selection of the material presented during the Dutch Testing Day.

For me personally, the synergy between academic and industrial testing activities is one of the most attractive aspects of the Dutch Testing Day: we have participants, speakers, and financial contributions from both worlds, fostering cross-fertilization and building bridges between them. It is widely known that industrial-academic partnership is a key driver to innovation and to remain at the competitive edge.

Therefore, I would like to thank everybody who made the 17th Dutch Testing Day a success: First of all, many thanks go to all participants: without participants, no Testing Day. Also, I would like to thank everybody who submitted an abstract, giving the Programme Committee a hard job in selecting the 7 best abstracts for presentation. Moreover, we are very much indebted to our sponsors: CIMSOLUTIONS, SQUERIST, AESTIS KMG, the Institute for Software Quality, ASML, Valori, the Centre for Telematics and Information Technology, NWO, Improve Quality Services, Better Be, Professional Testing, Refis and Collis. Their financial contributions allow the participants to attend the Dutch Testing Day free of charge and show the importance of the topic. I am grateful to Rector Magnificus Ed Brinksma for opening the Testing Day, and invited speaker Kim Larsen for giving the keynote speech.

Finally, I would very much like to thank the other members of the organizing team of the 17th Dutch Testing Day: Axel Belinfante, Joke Lammerink and Mark Timmer for their fantastic support.

Mariëlle Stoelinga

Formal Methods & Tools
University of Twente

Table of Contents

TOPAAS-model for software reliability analysis

Ed Brandt
Refis

Rijkswaterstaat is introducing probabilistic management to storm surge barriers and other objects. A risk analysis is the centre of this approach. Software failure is an important component in this. Quantifying software failure using existing methods, however, appeared to be not feasible. In order to address this problem, a group of known experts in the field of software reliability engineering from industry and academia has developed a factor driven model called TOPAAS. This article explains the necessity of software failure quantification, the use of expert opinion, the basics of TOPAAS as a property driven model and the work still to be done.

1. Introduction

Rijkswaterstaat is the Dutch Directorate for Infrastructure and Water Management. It is a public body responsible for building and maintaining the country's road and waterway infrastructure, including bridges, locks, weirs and movable storm surge barriers. A couple of years ago this public body introduced probabilistic management. A risk assessment is the centre of this approach. It gives direction to test intervals, maximal time to repair and modifications to the object. The ideal situation would be that Rijkswaterstaat provides different scenario's, with different cost estimates, that each will result in a different quality level and that the government is able to make a political choice.

Software plays an increasingly important role in these objects and systems. So to be able to quantify the risk of the entire system, quantification of software reliability is a necessity. Whereas for hardware reliability assessments and quantification is fairly common, for software it is not. Over the years several attempts have been made to introduce methods for this. In the next section some of these are discussed.

2. Existing methods for reliability analysis

Already in the early seventies of the previous century statistical models were described to predict the occurrence of failures in software. These Reliability Growth Models are based on the principle that the reliability of software grows when defects are found and fixed. This growth is not random but depends on properties of the system at hand. To determine an accurate failure probability, a statistically significant number of defects is required. With safety critical, high reliable systems as storm surge barriers, that appears to be a problem. When modules of those systems show many defects they are discarded all together.

The Monte Carlo approach of random testing and comparing the results with the predicted outcome is also not practically feasible. It would take excessive testing effort in complex systems as storm surge barriers. And then there are methods that help to achieve reliable software, like formal methods or the use of Safety Integrity Levels (IEC 61508). These methods however cannot be used to guarantee or quantify reliability of existing software afterwards.

The overall conclusion for Rijkswaterstaat must be that there are no adequate or feasible methods available. In order to address this problem, a group of known experts in the field of software reliability engineering conceived a new method called TOPAAS.

3. TOPAAS-model

TOPAAS is short for Task Oriented Probability of Abnormalities Analysis for Software. It is a static approach to analysis of reliability. This means it does not involve actual usage of the system at hand as in reliability growth modelling. But in stead it evaluates the circumstances that influence reliability and quantifies these factors.

Software failure in this approach is defined as the absence (for too long) of desired task execution, or the incorrect task execution, by a software module with respect to the mission of the overall system. Task execution is an individual action implemented in software that is observable on the outside (on a technical level) and that can be decided upon if it is performed correctly or incorrectly with respect to the overall mission of the system. A software module is then defined as a piece of software that is represented by a specific group of lines source code (or its graphical equivalent) with the following properties:

- A clear distinction can be made with respect to other pieces of code and there is clear separated functionality provided by the module that is required by the system;
- It exhibits observable behaviour with specific qualities (like timeliness, reliability, etc.);
- It isn't useful (in the light of the failure analysis on system level) or possible to make a further decomposition.

Applying TOPAAS starts with decomposing the system into software modules performing a Fault Tree Analysis. The probability of software failure of individual modules is then determined by investigating 15 factors divided into 5 dimensions. Answering the questions with every factor results in a failure probability per module in the FTA.

From a mathematical point of view:

- factor driven model provides *n* factors F_i to determine failure probability *P*

  $$P = PB \bullet F_1 \bullet F_2 \bullet \ldots \bullet F_n$$

  where
  - PB is the base failure rate (1 as a conservative default value, implying the assumption that software will fail if we cannot apply a reliability analysis method)
  - F_x is the impact of a specific factor based on a piece of knowledge

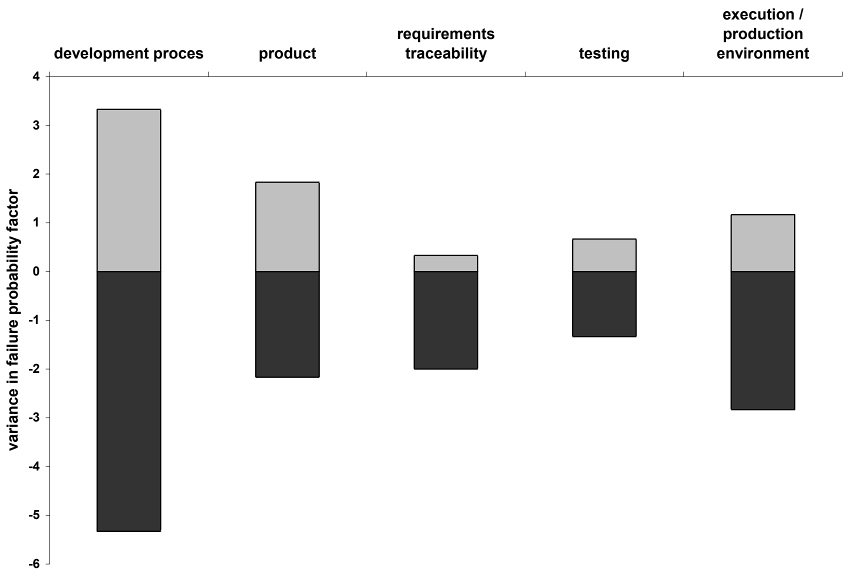
TOPAAS contains the following 5 dimensions and, in total, 15 factors:

- Development process
  - Safety Integrity Level
  - Inspections
  - Design modifications
  - Maturity organisation
  - Knowledge and experience
  - Cooperation
- Product properties
  - Complexity
  - Size
  - Transparency architecture
  - Certified compiler
- Requirements
  - traceability
- Testing
  - techniques and coverage
- Operational use
  - Multi processor
  - Field data available
  - Monitoring

The combination of these factors expresses the combined expert opinions that determine the probability of failure. By using these underlying factors, simulating the experts, the accuracy of the estimates and the transparency of the process are hugely increased. An example of the factor "Test techniques and coverage" is shown below.

12 Test techniques and coverage			
		Normal	**SIL3/SIL4**
1	Unknown	0	NVT
2	No documented test execution	0	NVT
3	Documented test execution, no techniques, unknown coverage	-⅓	NVT
4	Formal test techniques, low coverage	-½	⅔
5	Formal test techniques, medium coverage	-⅔	½
6	Formal test techniques, high coverage	-1	0
7	Formal test techniques, high documented coverage	-1⅓	-⅓

Note that TOPAAS makes a distinction between systems developed targeting a Safety Integrity Level of 3 or 4 and others. When a system is not developed targeting a SIL and formal test techniques were used but with a low coverage, the failure probability will be decreased by a factor $10^{-1/2}$ (answer 4). But when SIL 3 or 4 was targeted the failure probability in that case will increase by a factor of $10^{2/3}$!



Depending on the outcome of the underlying factors the TOPAAS dimensions may influence the base failure rate as follows (diagram).

4. To be done

Version 2 of the theory behind the model is published within Rijkswaterstaat and its suppliers and is already applied in several projects. It is available, but only in Dutch yet (http://www.refis.nl/media/artikelen.php). Publication in international magazines is being prepared. The model has been tested on several reference projects and compared to other methods of reliability analysis, including expert opinion.

It now needs further back up and calibration by statistical data and, based on that further review and referencing. Manuals and tools should be provided when broadening the access and usage. Also support of user forum and model maintenance should be organized.

A particular point of investigation is related to the crucial point of software module decomposition. Failure probability of modules in an FTA may interfere of influence each other. The correlation of modules in this respect needs further investigation resulting in guide lines for TOPAAS users.

TOPAAS authors and reviewers

Authors of the TOPAAS model are: Alessandro Di Bucchianico (TU/e), Jaap van Ekris (DNV), Jan-Friso Groote (TU/e), Wouter Geurts (Logica), Gerben Heslinga (Intermedion), Gea Kolk (Movares) and Ed Brandt (Refis).

Reviewers of the TOPAAS model are: Sipke van Manen (Bouwdienst RWS), Harry van der Graaf (Bouwdienst RWS), Peter van Gestel (Delta Pi) and Piet de Groot (NRG).

About the author

Ed Brandt is working in information technology since 1982. He specialized in software testing since 1996 and founded Refis in 2003, specialized in reliability engineering and software metrics. He is one of the authors of the Topaas-method. Please contact edbrandt@refis.nl for more information.

Constructing Formal Models through Automata Learning

Fides Aarts, Faranak Heidarian, and Frits Vaandrager
Institute for Computing and Information Sciences, Radboud University
Nijmegen P.O. Box 9010, 6500 GL Nijmegen, the Netherlands

Model-based system development is becoming an increasingly important driving force in the software and hardware industry. The construction of models typically requires specialized expertise, is time consuming and involves significant manual effort, implying that in practice often models are not available, or become outdated as the system evolves. In practice, 80% of software development involves legacy code, for which only poor documentation is available. Manual construction of models of legacy components is typically very labor intensive and often not cost effective. The solution is to infer models automatically through observations and test, that is, through black-box reverse engineering.
The problem to build a state machine model of a system by providing inputs to it and observing the resulting outputs, often referred to as black-box system identification, is both fundamental and of clear practical interest. A major challenge is to let computers perform this task in a rigorous manner for systems with large numbers of states.
The problem of learning state machines (automata) has been studied for decades. Many techniques for constructing models from observation of component behavior have been proposed. The most efficient such techniques use the setup of active learning, where a model of a system is learned by actively performing experiments on that system.
Tools that are able to infer state machine models automatically by systematically "pushing buttons" and recording outputs have numerous applications in different domains. They support understanding and analyzing legacy software, regression testing of software components, protocol conformance testing based on reference implementations, reverse engineering of proprietary/classified protocols, and inference of botnet protocols.
LearnLib [7], the winner of the 2010 Zulu competition on regular inference, is currently able to learn state machines with at most in the order of 10.000 states. During the last few years important developments have taken place on the borderline of verification, model-based testing and automata learning. By combining ideas from these three areas it will become possible to learn models of realistic software components with state-spaces that are many orders of magnitude larger than what state-of-the-art tools can currently handle.
Clearly, abstraction is the key for scaling existing automata learning methods to realistic applications. The idea of an intermediate component that takes care of abstraction is very natural and is used, implicitly or explicitly, in many case studies on automata learning. Aarts, Jonsson and Uijen [3] formalized the concept of such an intermediate abstraction component. Inspired by ideas from predicate abstraction, they defined the notion of a mapper $\mathcal{A}$, which is placed in between the SUT $\mathcal{M}$ and the learner, and transforms the interface of the SUT by an abstraction that maps (in a history dependent manner) the large set of actions of the SUT into a small set of abstract actions. By combining the abstract machine $\mathcal{H}$ learned in this way with information about the mapper $\mathcal{A}$, they can effectively learn a (symbolically represented) state machine that is equivalent to $\mathcal{M}$. Roughly speaking, the learner is responsible for learning the global "control modes" in which the system can be, and the transitions between those modes, whereas the mapper records some relevant state variables (typically computed from the data parameters of previous input and output actions) and takes care of the data part of the SUT.
A major challenge will be the development of algorithms for the automatic construction of

mappers: the availability of such algorithms will boost the applicability of automata learning technology. In [2], we provided a solid theoretical foundation for a generalization of the abstraction framework of [3]. The theory of [2] is (a) based on interface and I/O automata instead of the more restricted Mealy machines (in which each input induces exactly one output), (b) supports the concept of a learning purpose, which allows us to restrict the learning process to relevant interaction patterns only, and (c) supports a richer class of abstractions.

In [1], we presented our prototype tool Tomte, named after the creature that shrank Nils Holgersson into a gnome and (after numerous adventures) changed him back to his normal size again. Tomte is able to automatically construct mappers for a restricted class of scalarset automata, in which one can test for equality of data parameters, but no operations on data are allowed. The notion of a scalarset data type originates from model checking, where it is been used by Ip & Dill for symmetry reduction [6]. We use the technique of counterexample-guided abstraction refinement: initially, the algorithm starts with a very course abstraction $\mathcal{A}$, which is subsequently refined if it turns out that $\mathcal{M} \| \mathcal{A}$ is not behavior-deterministic.

Non-determinism arises naturally when we apply abstraction: it may occur that the behavior of a SUT is fully deterministic but that due to the mapper (which, for instance, abstracts from the precise value of certain input parameters), the system appears to behave non-deterministically from the perspective of the learner. We used LearnLib as our basic learning tool and therefore the abstraction of the SUT may not exhibit any non-determinism: if it does then LearnLib crashes and we have to refine the abstraction. This is exactly what has been done repeatedly during the manual construction of the abstraction mappings in the case studies of [3]. We formalized this procedure and described the construction of the mapper in terms of a counterexample guided abstraction refinement (CEGAR) procedure, similar to the approach developed by Clarke et al [4] in the context of model checking.

Using Tomte, we have succeeded to learn fully automatically models of several realistic software components, including the biometric passport, the SIP protocol and the various components of the Alternating Bit Protocol.

The Tomte tool and all the models that we used in our experiments are available via http://www.italia.cs.ru.nl/. Table 1 gives an overview of the systems we learned with the number of input refinement steps, total learning and testing queries, number of states of the learned abstract model, and time needed for learning and testing (in seconds). For example, the learned SIP model is an extended finite state machine with 29 states, 3741 transitions, and 17 state variables with various types (booleans, enumerated types, (long) integers, character strings,..). This corresponds to a state machine with an astronomical number of states and transitions, thus far fully out of reach of automata learning techniques. We have checked that all models inferred are bisimilar to their SUT. For this purpose we combined the learned model with the abstraction and used the CADP tool set, http://www.inrialpes.fr/vasy/cadp/, for equivalence checking.

System under test	Input refinements	Learning/ Testing queries	States	Learning/Testing time
Alternating Bit Protocol - Sender	1	193/3001	7	1.3s/104.9s
Alternating Bit Protocol - Receiver	2	145/3002	4	0.9s/134.5s
Alternating Bit Protocol - Channel	0	31/3000	2	0.3s/107.5s
Biometric Passport	3	2199/3582	5	7.7s/94.5s
Session Initiation Protocol	3	1755/3402	13	8.3s/35.9s
Login	3	639/3063	5	2.0s/56.8s
Farmer-Wolf-Goat-Cabbage Puzzle	4	699/3467	10	4.4s/121.8s
Palindrome/Repdigit Checker	11	3461/3293	1	10.3s/256.4s

Table 1. Learning statistics

Currently, Tomte can learn SUTs that may only remember the last and first occurrence of a parameter. We expect that it will be relatively easy to dispose of this restriction. We also expect that our CEGAR based approach can be further extended to systems that may apply simple or known operations on data, using technology for automatic detection likely invariants, such as Daikon [5]. Even though the class of systems to which our approach currently applies is limited, the fact that we are able to learn models of systems with data fully automatically is a major step towards a practically useful technology for automatic learning of models of software components.

References

1. F. Aarts, F. Heidarian, P. Olsen, and F.W. Vaandrager. *Automata learning through counterexample-guided abstraction refinement*, October 2011. Available through URL http://www.mbsd.cs.ru.nl/ publications/papers/fvaan/CEGAR11/.
2. F. Aarts, F. Heidarian, and F.W. Vaandrager. *A theory of abstractions for learning i/o-automata*, October 2011. Available through URL http://www.mbsd.cs.ru.nl/publications/papers/fvaan/.
3. F. Aarts, B. Jonsson, and J. Uijen. *Generating models of infinite-state communication protocols using regular inference with abstraction*. In A. Petrenko, J.C. Maldonado, and A. Simao, editors, 22nd IFIP International Conference on Testing Software and Systems, Natal, Brazil, November 8-10, Proceedings, volume 6435 of Lecture Notes in Computer Science, pages 188–204. Springer, 2010.
4. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. *Counterexample-guided abstraction refinement for symbolic model checking*. J. ACM, 50(5):752–794, 2003.
5. M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao. *The Daikon system for dynamic detection of likely invariants*. Science of Computer Programming, 69(1- 3):35–45, 2007.
6. C.N. Ip and D.L. Dill. *Better verification through symmetry*. Formal Methods in System Design, 9(1/2):41–75, 1996.
7. H. Raffelt, B. Steffen, T. Berg, and T. Margaria. *Learnlib: a framework for extrapolating behavioral models*. STTT, 11(5):393–407, 2009.

About the authors

Fides Aarts is a PhD student at the Radboud University in Nijmegen. After she has finished her Master thesis in Uppsala on inference and abstraction of communication protocols under supervision of Bengt Jonsson, she continued working on learning automata. Currently, she is involved in automatically generating abstractions of automata to extent inference to systems with large parameter domains.

Faranak Heidarian finished her PhD in Radboud University Nijmegen. She worked in MBSD group and her research was about abstraction refinement. She also fulfilled a research on formal analysis of synchronization protocols of wireless sensor networks. She has a bachelor degree in software engineering, and a Masters degree in Computer Science from Sharif University of Technology, Tehran, Iran.

Frits Vaandrager has a strong interest in the development and application of theory, (formal) methods and tools for the specification and analysis of computer based systems. In particular, he is interested in real-time embedded systems, distributed algorithms and protocols. Together with Lynch, Segala, and Kaynar he developed the (timed, probabilistic and hybrid) input/output automata formalisms, which are basic mathematical frameworks to support description and analysis of computing systems. He has been and is involved in a large number of projects in which formal verification and model checking technology is applied to tackle practical problems from industrial partners. Within the OCTOPUS project with Océ he is currently involved in the construction of the DSEIR toolset for model-driven design-space exploration for embedded systems. Recently, he has also become very interested in automata learning.

Managing the co-evolution of software artifacts

J.M.A.M. Gabriels[a], D.H.R. Holten[b], M.D. Klabbers[a],
W.J.P. van Ravensteijn[b], A. Serebrenik[c]

[a]Laboratory for Quality Software, Eindhoven University of Technology
[b]SynerScope B.V.
[c]Model-Driven Software Engineering, Eindhoven University of Technology

Software development projects are virtually always carried out under pressure. Planning and budgets are tight, room for errors is non-existent and the pressure to deliver is high. Natural questions for (test) managers arise, such as: "When have we tested enough?" and "How many tests do we have to redo for this new version?'". The naive answer would be: "when we have convinced ourselves through testing that all requirements are satisfied". Unfortunately, attaining maximal confidence with minimal effort is not easy.

In order to convince ourselves that the system does what it is supposed to do, tests are needed. Requirements, design and code change during the development of software. As a consequence, tests need to change as well. In the end we want to ensure that all requirements and risks are adequately addressed with tests. For this, tests at different levels of abstraction and for different software artifacts are required and need to be managed.

Traceability matrices are often used to relate user requirements, design, code, and tests. Traceability allows to link elements from different software artifacts, like requirements, design components and code components, to each other and to test cases. As a result, traceability can be used to analyze for example how well software artifacts are covered by test cases. Because a requirement leads to design components and eventually to code, tests are needed at each stage. Traceability can tell us how well test cases cover different software artifact elements. This information can be used to uncover mistakes in software artifacts at an early stage and actively manage the development and test efforts. Unfortunately, traceability information is often spread out over multiple artifacts and describes only the current situation.

TraceVis, a visual analytics tool based on the master thesis of Van Ravensteijn[1] combines the traceability information of multiple software artifacts in an interactive way. The tool was recently applied to fraud detection in financial transactions [2] and software model transformations[3], Figure 1 shows the traceability between four, vertically placed, hierarchical software artifacts: acceptance test plan (ATP), user requirements document (URD), software requirements document (SRD), and architectural design document (ADD). Hierarchy within each document is given by its division into chapters, sections and subsections. Each line between hierarchies represents a link between elements of two artifacts, e.g., user requirement being tested by an acceptance test or architectural component implementing a software requirement. The hierarchies can be collapsed and extended and risk levels and priorities can be

[1] W.J.P. van Ravensteijn, *Visual traceability across dynamic ordered hierarchies*, M.Sc. thesis, 2011, Eindhoven University of Technology

[2] SynerScope on-line demos: http://www.synerscope.com/demos, and, specifically, *SynerScope for Fraud* http://www.synerscope.com/content/SynerScope%20for%20Fraud1.pdf

[3] M.F. van Amstel, A. Serebrenik, & M.G.J. van den Brand, *Visualizing traceability in model transformation compositions*, 2011, Workshop on Composition and Evolution of Model Transformations, London: Department of Informatics, King's College London.

visualized by giving the elements different colors. The edge bundling technique bundles similar relations in the middle, clearly showing deviations. Furthermore, TraceVis provides a way of assessing the evolution of traceability between artifacts though a timeline (lower part of Figure 1). The timeline shows such events as addition, modification or removal of individual elements, e.g., user requirements or tests, as well as addition or removal of traceability links between the elements.

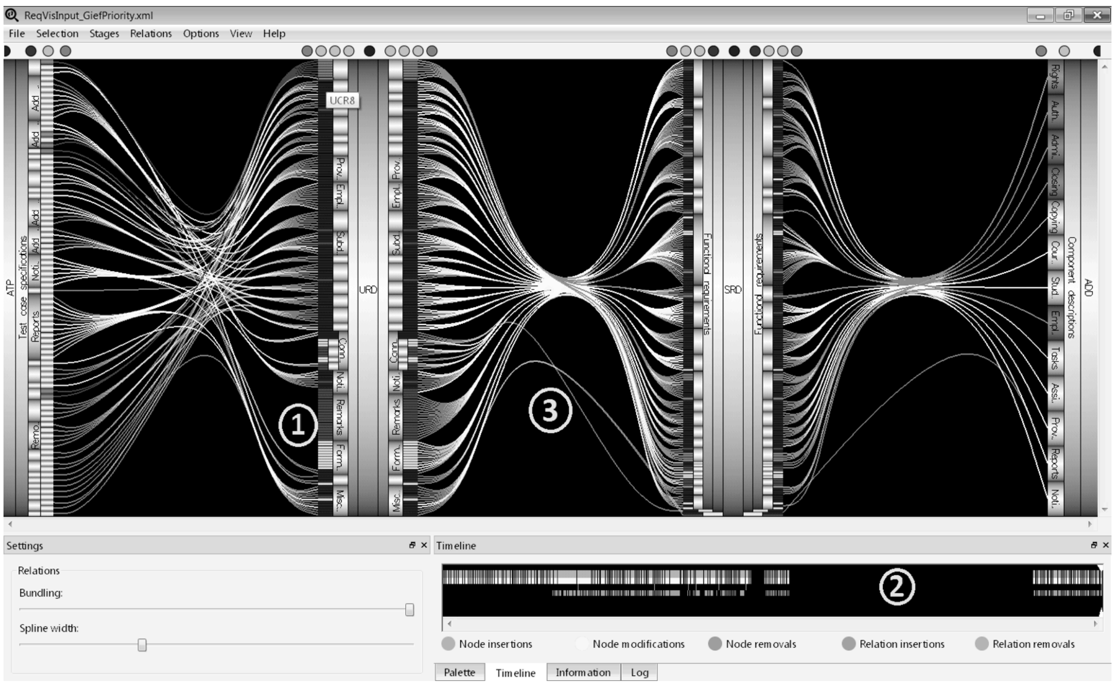


Figure 1: *TraceVis tool with traceability information from a student capstone project at the Eindhoven University of Technology. The edge bundeling technique makes it easy to spot deviations.*

Already at first glance, we can see points of attention in Figure 1: a gap in requirement coverage (1) and a gap in the timeline (2). The gap labeled (1) shows some medium and low priority user requirements not covered by acceptance tests. The gap labeled (2), located in the timeline, shows that test cases were added very late in the project. The gap itself relates to implementation activities in which there were no changes to the shown software artifacts.

By interactively inspecting the traceability information, we can discover points of interest. Figure 1 reveals, for example, that a selected user requirement (URC8) is tested by one acceptance test, corresponds to one software requirement and is implemented in one architectural component. While this does not seem to be problematic, further inspection of the evolution of user requirements closely related to URC8 tells an entirely different story. While grouped together in the URD, the corresponding software requirements are spread all over the SRD, and implementation involves five out of thirteen architectural components.

Figure 1 also shows outliers; grey lines running off-center between the URD and the SRD (3), and between the SRD and the ADD. The core part of the SRD consists of two parts: Requirements and Rights table. The Requirements part consists of Functional requirements further divided in groups (with 189 requirements in total) and Non-functional requirements (1 requirement). The Rights table part contains only one element, i.e., the rights table. This means that individual functional requirements are nested at depth four, the non-functional requirements at depth three and the rights table at depth two. Therefore the rights table is an "outlier" in the organization of the SRD.

The evolutionary traceability information allows us to see how well tests cover artifacts and whether risks are sufficiently tackled. It gives insight in the balance between tests, priorities, and risks and can support decision making in assigning test effort. Furthermore, it can help in determining which tests need to be redone when a certain component or requirement changes.
The insight in the co-evolution of software artifacts and associated tests makes it possible to actively manage test effort from an early stage on.

About the authors

Joost Gabriels (j.m.a.m.gabriels@tue.nl) received his M.Sc. in Computer Science from the Eindhoven University of Technology (TU/e) in 2007. Currently, he is a researcher and consultant at the Laboratory for Quality Software (LaQuSo), TU/e. His interests include software architecture and design, specification languages, program verification and the quality of software in general.

In his Ph.D., defended in 2009 at TU/e Dr. **Danny Holten** (danny.holten@synerscope.com) has developed techniques for the visualization of graphs and trees. He received the best paper award at IEEE INFOVIS 2006 and was nominated for the best paper award at ACMCHI 2009. He furthermore received the TU/e Doctoral Project Award for the best PhD dissertation in 2009. From 2009 to 2011, he worked as a postdoctoral researcher. Since April 2011, Danny Holten is CTO at SynerScope B.V., a Dutch visualization-research-inspired TU/e spin-off company that leverages his PhD research for "Big Data" analysis.

Martijn Klabbers (m.d.klabbers@tue.nl) received a M.Sc. in Computer Science from the Technical University Delft, The Netherlands in 1995. At the Technical University of Eindhoven, he is a senior consultant at LaQuSo (Laboratory for Software Quality). His research interests include certification, decision support systems, and user requirements.

In 2011, **Wiljan van Ravensteijn** (wiljan.van.ravensteijn@synerscope.com) received his Master's degree in computer science (with honors) from the Dept. of Mathematics & Computer Science at TU/e. His Master's thesis was titled "Visual traceability across Dynamic Ordered Hierarchies". Since August 2011, Wiljan van Ravensteijn is Software Developer at SynerScope B.V.

Dr. **Alexander Serebrenik** (a.serebrenik@tue.nl) is an assistant professor of Model-Driven Software Engineering at TU/e. He has obtained his Ph.D. in Computer Science from Katholieke Universiteit Leuven, Belgium (2003) and M.Sc. in Computer Science from the Hebrew University, Jerusalem, Israel. Dr. Serebrenik's areas of expertise include software evolution, maintainability and reverse engineering, program analysis and transformation, process modeling and verification.

Risk Based Testing. A piece of cake or not?

Jeanne Hofmans
Improve Quality Services

Risk based testing has been a popular approach to testing for a while. Based on the fact that it is impossible to test everything, the idea of focusing test effort on the most risky areas has been embraced by both testers and management. But even today many companies and projects are struggling with their risk based test approach. Especially since systems under test are becoming more complex the need to focus on the risky areas increases. And thus the need for a flexible and practical approach for identifying and managing risks. This article is about popular risk methods, their pitfalls and introduces a practical approach to risk based testing.

1. Product Risk Analysis

An important task in risk based testing is the product risk analysis (PRA). Product risk analysis is a method to identify and analyze risks. A risk is a factor that could have a negative consequence in the future and is usually expressed as impact and likelihood (ISTQB).

In testing one should focus on risks with high impact and likelihood. Risks with an extremely high impact should have more focus than risks with low impact. This does not mean that no effort is spent on other risks with a lower impact. It only means that less effort is spent.

2. Common PRA-methods

Several methods for product risk analysis exist. Popular methods are PRISMA®, PRIMA®, the risk method of TMAP Next® and the Failure Mode, Effect & Criticality Analysis (FMECA).

Methods like PRIMA® and the risk method of TMAP NEXT® divide the system under test in risk areas in which certain quality attributes are important. In the TMAP NEXT® risk analysis several other tables are constructed to determine the risk of smaller system components. This is done to make the risk concerned with certain quality attributes and both systems and components clear. The risk factor determines the test effort per component.

The PRISMA® method does not prescribe how to categorize risks, it is a method to identify risk quadrants, like shown in figure 1. Often it is used as a specification-based method in which each (part of the) specification represents a risk item. Each risk item is to be assigned a risk quadrant.

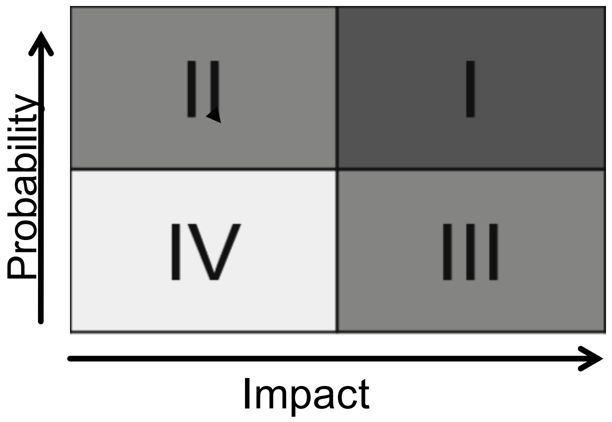


Figure 1: Four Risk quadrants as used in the PRISMA-method

In a Failure Mode Effect and Criticality Analysis a complete fault three is made to determine failure modes that could have a negative effect on a predefined top event. This is preferably done in the design stage in which the design can be altered, based on results of the FMECA.

3. Pitfalls using common PRA-methods

Fully applying a FMECA is very time-consuming. This is primarily done for high-risk systems like aircrafts. A solution is to only use FMECA for a risky part of the system.

TMAP Next® and PRIMA® give stakeholders and managers good insight in the high-risk areas of a system. However the resulting PRA is often not used by test engineers when deciding what to test and what not to test. Perhaps the PRA does not match the available specifications because the PRA is based on subsystems whilst the specification is function-based. Or perhaps the test engineer does not understand all the tables that are delivered in the risk analysis method TMAP Next®. It then becomes hard for a test engineer to test based on risks. In best case the information provided is used by the test manager as a checklist to see if nothing is missing in the current test set. In worst case the PRA is not used at all.

Using PRISMA® in a specification-based manner overcomes the problem of the PRA not being a practical source for test engineers. If chosen wisely the risk items in the PRA represent (sections of) the specifications. As testers use the specifications as a starting-point for testing it helps when the PRA simply states which parts represent the highest risk. The pitfall in this is that PRISMA® is only used in such a specification based manner. If test cases are only based on specifications, likely not all risks are covered.

4. A practical approach for Product Risk Analysis

A practical rule of thumb is to base the product risk analysis on three types of product risk:

- **S**pecification;
- **I**ntegral;
- **R**egression.

These three types of product risk are very much in line with the way testers (should) test. First of all the specification is tested, where more attention is paid to the parts which are identified as most risky. Then special attention is paid to risks that exist, but are not clear in the specification; the integral (or implicit) risks. Finally attention must be paid to the approach for regression testing.

For a system test, the specification and thus PRA can be based on functions or use cases. For an interface test the PRA and specification can be based on interfaces (per Interface Requirement Specification or for each interface in an Interface Requirement Specification). For a maintenance test the specification and thus PRA can consist of change reports. For a regression test, the set of existing test cases can be input to the PRA.

This rule of thumb, identifying three types of product risk, can very well be used in combination with PRISMA®. In this case a PRA is made for each test level. Some test levels can share a PRA if they share the same specifications or are highly related and traceable. For identifying integral or implicit risks brainstorm sessions, stakeholder interviews or FMECA's can be used. A FMECA can be done for even a complete system, but can also very well be applied to some function or subsystem. PRIMA® and the first part of the risk method of TMAP Next® can be useful to give an total, more static overview of the risky areas of the system. The PRA per test level will be used more intensively, as they represents a testers day-to-day's business. That is testing according to specs, testing the relevant risks and regression testing.

5. Product Risk Process

The most important thing to be said about the process of risk based testing is that it all depends on how well risks are being kept track of. Most of all this requires a lot of *discipline*. However when defining product risks in terms of three types (specification, integral and regression) this can very easily be translated to the test strategy of a test level. In a test level first the specification-based risks are covered, then or in parallel the integral risks and finally regression tests take place.

The *discipline* is in constantly keeping track of the status of existing risks and identifying new risks. It is easier to keep track of the status of risks when defining three types of product risk. At least, if using an accompanying test strategy…

The reward is that stakeholders and management become more involved. Involvement grows as they better understand what testers actually do and how testers can be helped to let the project deliver a system that supports the stakeholders and users in their needs .

6. Conclusion

The basic concept of risk based testing is still embraced by many testers, managers en stakeholders. If used properly it should direct testing in an efficient manner. Unfortunately very often risk based testing is not a piece of cake. In this article I discussed common methods for product risk analysis (PRA), and possible reasons why projects struggle. In worst case a PRA becomes shelfware.

In general it is smart to use a combination of risk analysis techniques like FMECA, a brainstorm, PRISMA, etc. In this way risks are determined using different viewpoints. These viewpoints are also represented in the three types of risk identified in this article: Specification, Integral and Regression. The strength of identifying tree types of product risk is that it is both understandable for a stakeholder and usable for a tester.

About the author

Jeanne Hofmans has studied Software Technology at Utrecht University and now is a test consultant at Improve Quality Services. She has participated in many projects in both the financial and the embedded domain where risk based testing was more or less successful. The suggested solution was implemented by Jeanne at a technical test center and presented at Testnet Spring Event (Nieuwe helden) and at a Supervision session at Rabobank International. Momentarily Jeanne works as a test manager for the Sluiskiltunnel and as an auditor for several other projects.

Jeanne Hofmans can be contacted at: jho@improveqs.nl

Testing Highly Distributed Service-oriented Systems using Virtual Environments

F. Nizamic[1], R. Groenboom[2], A. Lazovik[1]
[1] Johann Bernoulli Institute for Mathematics and Computer Science, Faculty of Mathematics and Natural Sciences, University of Groningen
[2] Parasoft Netherlands B.V.

Modern application landscapes are becoming more and more interconnected. The introduction of SOA and ESB technology has increased the dependency on other systems, both internal and external. This has severe consequences for testing SOA applications, their availability, and the constraints on different environments.

1. Testing challenges of service-oriented systems

On March 11, 2009, twenty-five residents of a little municipality in the West of the Netherlands unexpectedly received a certificate that they were married or had registered a newborn. What happened? To test the central information system of the country, data in the central people registry had been modified. While the data was supposed to be only a test for the system, it ended up updating the real population database and initiating the processes for informing the involved citizens. In fact, the change propagated even further: from the central population registry, it also affected the information system of the tax office and of the retirement administration.

This anecdote illustrates the challenges faced in testing service-oriented systems. From the perspective of a tester, most of the challenges arise from the need to prepare and execute tests in the shortest possible amount of time and without dependency on other processes. In order to work independently of other processes, dependencies such as dependency on availability of testing environment or dependency caused by sharing resources with others must be removed. Moreover, testers must start preparation of test scripts before a code that they will test is written. One of the reasons for this is that many software companies are working per Agile working methodology, which means that iterations in which developers and testers need to deliver an additional value to a software are much shorter. If testers could start the preparation of test scripts from day one of the sprint, their chances to prepare tests on time would significantly increase. In addition, testers usually have difficulties to produce all desired states of the external services that are part of their chain testing. In order to cover all code paths and to ensure that the system is fully functional, tester needs to have ability to mock-up some of the services or, in other words, simulate desired behavior of external services. Furthermore, external services can be available just for a limited time or not available at all.

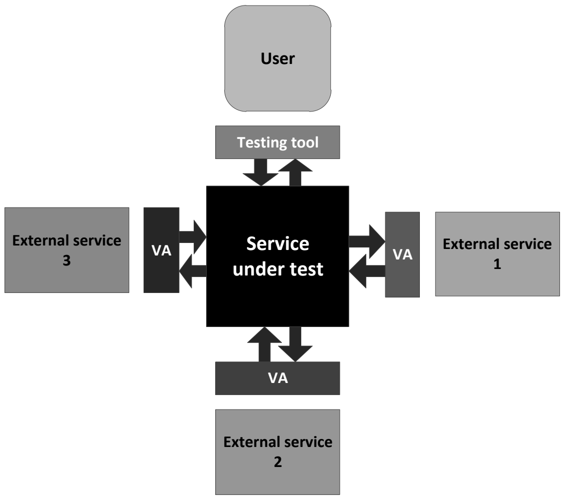
From the perspective of a company, besides quality of a software, major concerns are additional costs. These costs can be caused by acquiring additional equipment or human resources. Even higher additional costs lie in maintenance work that was unnecessary or in project delays which occurred. Such additional costs need to be reduced or if possible totally eliminated.

In short, all the above-mentioned issues point to the same goal: to increase the speed of a testing process and that way reduce the costs by removing dependencies and gaining the control over testing environments. Most of above-mentioned issues could be avoided if a simulation environment, behaving in the very same way as a real system, is used. Nowadays, generation of this kind of simulation environment is very feasible and will be further discussed in this paper.

2. The solution is in a virtualization

The facts are that software is continuously evolving and that in service-oriented systems the functionality is spread across the network. However, testing of software still implicitly assumes full control over a system. The reality is that the service under test depends on the service provided by external services (which are not owned by the subject performing testing) in order to provide its own service.

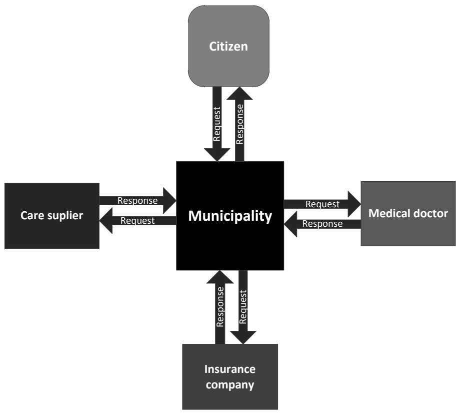
In this work we propose that simulation environments should be semi-automatically generated and thus gain control over external services and their data. Why?



As a part of integration testing, different versions of interdependent services need to be tested together as one system, in order to confirm that all functional aspects for this particular combination of services are covered. To prove this, data that is stored in each system need to be appropriate, so all important test cases can be executed. If external systems are not owned by us, which is usually the case, tests can be performed only with data that is available. Alternatively, virtual assets that mimic a behavior of external services can be created. A set of this kind of virtual assets, representing all external systems, would form a simulation environment. It is possible to generate this kind of environment by using currently available tools. Stubs or mock services are not something new, and nowadays they can be easily generated just by having a WSDL, log file or by some other way. Once virtual assets are generated/deployed, the behavior of the external service methods needs to be modeled. To make it even easier, only the behavior of certain methods which are needed in order to cover defined test cases needs to be modeled. Thus careful choice of the test cases permits saving time and resources. Behavior of a services can be modeled simply by defining expected outputs for certain inputs. However, this is not that simple for services that preserve a state. Examples of these kinds of service methods are ones that do not have input parameters but need to return an output (for example *createAccount* method). For this kind of method, it is difficult to map a set of inputs to set a set of outputs.

Besides generation of the simulation environment, the whole ecosystem should be populated with corresponding test-data. This procedure should be carried out only one time. After we have prepared environments that are populated with data that describe their behavior, real tests can be executed on the system under test. During execution, simulated external services return responses that are exactly the same as those that would be returned by real external services. Thus the creation of simulated systems, which can be easily maintained without a need for intervention of environment administrators, solves the problems of availability, dependency, and maintenance.

3. WMO case study



In order to demonstrate how this procedure could work in practice, we present a simulation environment of a real business process for the implementation of the *Wet Maatschappelijke Ondersteuning Law* (WMO). The WMO is a Dutch law for supporting people that have a chronic disease

or disability, so that these people can independently live in their homes and actively take part in everyday life despite their physical limitations. The support that is provided by the WMO typically includes transportation, a wheelchair, or a home modification (e.g. removing doorposts for people using wheelchairs). The responsibility for a WMO request lies with the Dutch municipalities; they handle the complete business process. The process accesses external parties, e.g., insurance companies and doctors for medical advice. In the WMO ecosystem, all of the issues mentioned in section 2 can be seen. External services such as “Medical doctor” are not available all the time and especially not available for sending test-data which would pollute the real database or send notifications to the real doctors. Furthermore, in case of testing a “Municipality” service, a system engineer would need to deploy the latest version of the external services code on test environments. Firstly, we may not have access to that code that is external to our organization. Secondly, we would depend on some other team (i.e. Ops team) for deployment. And thirdly, the testing environment might not be available. All of these issues can be avoided simulating all external services and modeling their desired behavior.
For this paper, we have used the *Parasoft Virtualize* tool to create virtual assets which would represent all three external services (Medical doctor, Insurance company, Care supplier). “Citizen” was simulated by *Parasoft SOAtest*, which we used to send the requests to the “Municipality” service. We have thus demonstrated that it is possible to create all needed components by using currently available tools. We have also modeled virtual assets’ behavior to cover basic scenarios and recorded desired requests and responses. The next step would be to find a way to semi-automatically generate all those virtual assets, populate them with the data and keep up-to-date.
As this was in a way a toy example, now we are approaching companies for appropriate systems that could be used as the real case study. We are aware that we have just scratched the top of the iceberg. Our goal is to have Agile support for a TestOps team which would enable simple switching between staged environments by merging testing and infrastructure knowledge.

About the authors

Faris Nizamic is a PhD student at the University of Groningen (RuG). His research interests cover the areas of software testing as well as service-oriented, distributed and cloud computing. He holds an MSc in Computing Science from the University of Sarajevo, Bosnia and Herzegovina. For the last two years, he has worked as a Senior Quality Assurance Engineer in NAVTEQ with the main focus on testing of web services and automation of testing processes.
Email: f.nizamic@rug.nl, Web: http://www.cs.rug.nl/~faris

Rix Groenboom is a consultant for Parasoft in the area of testing and virtualization of modern SOA, SaaS and cloud-based architectures. He has written a large number of technical articles and presented at many IT industry conferences in Europe and the USA. His core area of expertise is specification, design and validation of software applications. He holds an MSc and a PhD in Computing Science from the University of Groningen and published a thesis focusing on the formalization of domain knowledge.
Email: rix.groenboom@parasoft.nl

Dr. Alexander Lazovik is an assistant professor in the Distributed Systems group at the University of Groningen, after being an ERCIM fellow at CWI (NL) and INRIA (F), and an intern at IBM TJ Watson. He obtained a PhD in computer science from the University of Trento (I) in 2006. His research interests are in the areas of service-oriented and distributed computing. He is also actively interested in pervasive and ubiquitous computing, with an emphasis on dynamic coordination of context-aware embedded devices.
Email: a.lazovik@rug.nl, Web: http://www.cs.rug.nl/~lazovik

State-driven Testing: An Innovative Approach to Successful UI Test Automation

Jan de Coster
Micro Focus

In an IT world of constant change, shorter release cycles, iterative development processes, and increasing complexity of enterprise applications, testing is a business-critical step in the Software Development Lifecycle (SDLC). Automation of testing processes is a must-have for organizations that plan to remain competitive as these changes will continue to accelerate and further influence software development.

Building a successful test automation practice is a challenging task for many development organizations. Mature development organizations realize that simple UI test automation techniques like record/playback do not provide the ROI required for modern development projects.

Keyword-driven Testing (KDT) is a widely accepted test automation technique that many mature development organizations rely on to overcome the disadvantages of simple record/playback test automation. However beyond the advantages that KDT frameworks deliver, there are major disadvantages in manageability and complexity inherent in KDT. Many applications require that thousands of automation keywords be developed to make use of KDT. Navigating and constructing test cases based on these keywords can be cumbersome and unpractical.

Acceptance Testing Frameworks (ATF) such as FitNesse use a similar approach to structuring test cases (acceptance tests) using keywords that are implemented using coded fixtures. These frameworks do not support users in navigating all available actions (keywords). As with KDT frameworks, ATFs do not support developers in structuring fixtures (the implementation of the actions) so that they can be easily reused and maintained.

State-driven Testing (SDT) addresses the maintenance and complexity issues of KDT by providing a UI state-transition model. By defining state transitions of the user interface, the set of allowable UI actions (keywords) at any given point in a test case is reduced from thousands of allowable actions down to a manageable list of tens of allowable actions. SDT uses a domain specific language (DSL) to define the test automation framework which provides a highly maintainable and simple approach to structuring a test framework.

4. About the author

Jan de Coster is a testing guru with over 15 years of IT experience. He currently holds the position of Subject Matter Expert in Quality Solutions at Micro Focus. In this role, Jan provides domain expertise and thought leadership around Software Quality Assurance for Micro Focus' education organization as well as delivering consultancy and project services into the field throughout Europe.
Jan started his career in software development and project management. However, it soon became clear that he had a passion for the qualitative aspect of software delivery and for the last 11 years he has specialized in integrating processes and tooling to help organizations increase quality in their software. More recently, Jan has been more closely exploring quality in relation to Agile Test Management and Agile Test Automation – areas, he feels, of significant importance to the future success of software delivery.
For additional information please visit www.microfocus.com.