# Model-Based Testing for the Cloud

Antonia Bertolino, Wolfgang Grieskamp, Robert M. Hierons,
Yves Le Traon, Bruno Legeard, Henry Muccini, Amit Paradkar,
David S. Rosenblum, Jan Tretmans

June 26, 2010

**Abstract**

Software in the cloud is characterised by the need to be highly adaptive and continuously available. Incremental changes are applied to the deployed system and need to be tested in the field. Different configurations need to be tested. Higher quality standards regarding both functional and non-functional properties are put on those systems, as they often face large and diverse customer bases and/or are used as services from different peer service implementations. The properties of interest include interoperability, privacy, security, reliability, performance, resource use, timing constraints, service dependencies, availability, and so on. This paper discusses the state of the art in model-based testing of cloud systems. It focuses on two central aspects of the problem domain: (a) dealing with the adaptive and dynamic character of cloud software when tested with model-based testing, by developing new online and offline test strategies, and (b) dealing with the variety of modeling concerns for functional and non-functional properties, by devising a unified framework for them where this is possible. Having discussed the state of the art we identify challenges and future directions.

Keywords: cloud computing, model based testing, non-functional properties

## 1 Introduction

In this paper we first discuss cloud computing and what makes testing particularly difficult. We then focus on several important aspects of cloud computing. For each aspect we review the state of the art, outline key challenges, and propose possible solutions and research directions.

## 2 The nature of The Cloud

Include a scenario in this section.



Dagstuhl Seminar Proceedings 10111
Practical Software Testing : Tool Automation and Human Factors
http://drops.dagstuhl.de/opus/volltexte/2010/2625

# 3 Scalability Requirements

Scalability is a particularly important consideration for many large-scale, networked systems, and as described in this section, it introduces a host of new challenges to model-based testing.

Despite its importance, scalability is a concept that has been poorly defined and poorly understood. Indeed, scalability is not even included in the main taxonomies of non-functional requirements appearing in industrial standards and the software engineering literature [20, 2, 7, 22, 23]. And the relatively few attempts in the literature to define the term tend to associate scalability solely with performance.

In recent work, we have defined a framework to characterize and analyze a system's scalability in precise terms [12, 11]. We define scalability as *a quality of a system characterizing its ability to maintain the satisfaction of its quality goals to levels that are acceptable to its stakeholders when characteristics of the application domain and the system design vary over expected ranges.* There are two important aspects of this definition worth highlighting: First, scalability is about *variation* of phenomena that arise in two places, namely the application domain or execution environment (over which developers typically have very little or no control) and the system design (over which developers have a great deal of control). Examples of the former are the average and peak rate at which users submit queries to a system, while examples of the latter are the size of a message buffer and the number of cores used for execution. Second, scalability is a quality that *relates to other qualities*, such as performance, resource consumption, availability, security, and others. Furthermore, scalability typically relates to multiple such qualities, which must be traded off in a Pareto-optimal fashion.

**** more detailed examples can be provided that illustrate the above concepts ****

In our earliest work on scalability, we applied our framework to a comparative characterization and analysis of the scalability of two consecutive designs for a real-world financial fraud detection system called IEF, with the earlier design employing a pure memory cache in a key component, and the latter design employing a hybrid memory/disk cache. The second design was developed in order to address scalability problems that existed in the first design. In particular, we ran test executions of existing, deployed implementations of the designs, varying the executions over ranges of values for the key characteristics of the application domain (and system design that govern the execution of IEF. An example of a key characteristic of the application domain is the number of distinct "business entities" represented in the transaction data processed by IEF, while a key system design characteristic is the size of the thread pool used for concurrent processing of transactions. For each execution we measured the key system qualities of interest, namely throughput, memory consumption and disk consumption, and we combined these measurements via a utility function provided by the stakeholders of IEF to a single scalar value representing the scalability of the execution. Finally, we plotted the utility values against the

corresponding ranges of values for the application domain and design characteristics, resulting in what is in essence a "scalability profile" for the two designs, allowing us to demonstrate that the latter design was indeed the more scalable of the two designs.

**** details and plots can be provided for the study described above ****

In our most recent work we have developed a systematic method for elaborating and analyzing scalability requirements, which among other things provides the basic information needed to undertake an analysis of a deployed system of the kind described above. Our method is an extension of the KAOS goal-oriented requirements engineering method, which relates business goals, software requirements and domain assumptions, and which supports the management of conflicts between goals and the resolution of obstacles to the satisfaction of goals [23]. More specifically, we have extended the conceptual framework of KAOS with a precise characterization of the concepts of *scalability goals*, *scalability requirements* and *scaling assumptions* and their roles in goal models. We then deal with scalability concerns during the goal-obstacle analysis steps of the goal-oriented elaboration process [29]. It starts from the observation that during a typical requirements elaboration process, most goals are elaborated first without considerations for scalability. The specification of such goals implicitly assumes that system agents, including human agents, software components and hardware devices, have infinite capacities for satisfying them. In the running system, such goals will be violated when some application domain quantities scale above the capacities of the system agents. We characterize such conditions as *scalability obstacles*; existing obstacle analysis techniques [29] do not consider scalability obstacles. Our approach to elaborating scalability requirements consists of systematically identifying and dealing with such scalability obstacles at requirements elaboration time, instead of later in the development process or (even worse) during system usage. Anticipating scalability obstacles at requirements level provide greater freedom to modify or extend the system design to deal with the problems. We resolve identified scalability obstacles by modifying and consolidating the system requirements specification in a precise, quantitative manner so that it specifies all relevant scalability goals, scalability requirements and scaling assumptions needed to inform later stages of development.

**** examples illustrating the method described above can be provided ****

So as a result of our previous work, we have the means to specify scalability requirements in the earliest stages of development, and the means to analyze the scalability of the finished product in the final stages of development. What we lack are systematic ways of progressing from the former to the latter. In particular, we lack

1. formal modeling languages that allow us to express architectures and designs in a way that makes scalability concerns explicit;

2. techniques for analysis and simulation of such architecture and design models that would allow early detection of the violation of scalability requirements and the identification of additional scalability problems;

3. techniques for systematic generation of test cases from such models; and

4. techniques for executing the test cases and evaluating their results across a range of values for relevant application domain and design characteristics, in a way that produces useful and accurate test results even if the execution environment cannot be fully and faithfully reproduced.

In this paper we are interested in addressing the first, third and fourth of these items, which together would constitute an approach for *model-based testing of scalability requirements*.

Development of such an approach requires addressing (at least) three major challenges. First, a wide variety of modeling, analysis and testing approaches have been developed for the different system qualities one might be concerned with in evaluating the scalability of a system. For instance, modeling and analysis of performance characteristics typically involves specification of probability distributions for waiting times and service times. Such information can be specified as extensions to UML design models, from which queueing network models can then be derived for analysis of overall system performance. On the other hand, testing and estimating the reliability of a system typically involves sampling the system input space according to an operational profile, executing the tests under a controlled time schedule, and executing enough tests for a statistically meaningful estimate, so that an accurate picture of reliability growth can be developed. This first challenge is thus to find a way of reconciling, relating and (ideally) unifying these disparate approaches to modeling and testing in order to facilitate a uniform means of scalability testing.

A second and related challenge relates to the fact that the qualities one considers in scalability typically compete in a Pareto-optimal fashion. As a simple example, a network communication component may be able to achieve increased throughput at the cost of increased memory consumption for message buffers. Or vice versa, the component may be able to reduce memory consumption at the cost of increased latency. Scalability testing thus requires the generation of test cases that allows one to explore such tradeoffs in a way that allows sensible decision-making for their resolution.

The third challenge relates to the fourth item listed above, which alludes to the need in scalability testing to "fake" a system into experiencing the effects and extremes of a variety of deployment environments even when those environments cannot be faithfully reproduced at test time. This is already a problem faced in performance testing and load testing, and testers sometimes resort to clever "tricks" involving transformation or compression/expansion of test data and test parameters (and we employed such techniques in early versions of the IEF study described earlier). Having multiple system qualities to test simultaneously, as is required typically in scalability testing, only compounds the problem, and thus scalability testing begs for a more rational approach that replaces trickery with mechanics that can produce test results having predictable and provable accuracy.

# 4 MBT for adaptive and evolving systems

# 5 Security

Security is a key issue in modern software intensive systems driving the everyday life of billions of people all around the world. If we consider the use of clouds, one of the main obstacle for the adoption of clouds by large companies is the security assurance guaranteed by providers.

Software systems deployed in clouds must be available 24/7 with a high level of security. Among different security concerns (user authentication, data encryption, etc), access control plays a critical role. It ensures that users, depending on their roles, can only access the resources they are supposed to access.

From the viewpoint of the cloud provider, it should be able to ensure the access control of the cloud as a whole. From the client of a cloud viewpoint, security will focus on the internal accesses permissions. Both policies have to be synchronised, especially in case of intrusion detection (unexpected event) or when one of the stakeholders decides to modify the access rights. Due to the distributed nature of a cloud, the complexity of the potential discrepancies between all these viewpoints may be high. Since the system deployed on a cloud, as well as the computing resources, are not frozen and may evolve in parallel, inconsistencies may occur which must be detected each time an adaptation is performed.

The solution is to use the standard architecture that involves designing a dedicated security component, called the policy decision point (PDP), which can be configured independently from the rest of the implementation containing the business logic of the application. The execution of functions in the business logic includes calls to the PDP (called PEPs policy enforcement points), which grant or deny access to the protected resources/functionalities of the system.

The objective of the test cases in a partly automated generation process (PDP and PEPs) is to check that the security policy of the application is fully synchronised with the security model. Errors in the security policy can have several causes. They can be caused by an error in the policy definition, by an error when translating the policy into an executable PDP or by an error in the definition of PEPs (calls to the PDP at the wrong place in the business logic, calls to the wrong rule, missing call, etc.). In order to increase the efficiency of the validation and verification tasks, we investigate an integrated approach based on two facets. First, we want to ensure as much quality as possible by construction. For that purpose we propose an MDE process based on a specific modeling language for security policies and automatic transformations to integrate this policy into the core application. Second, we develop generic verification and validation techniques that can be applied early in the development cycle and that are independent of any particular security formalism.

The main limitation of the current access control implementation techniques is that they neither support flexible access control mechanisms nor the alignment of the system security policy with the distributed resources of the clouds. There

is therefore a need for adaptive access control mechanisms and MBT techniques to test that any change of the models have been correctly propagated to the system implementation in the specific deployment on a cloud.

# 6 MBT for non-functional properties

# 7 Distributed testing

When testing a physically distributed system we may have to supply inputs and make observations at the separate interfaces (ports). Typically there is no global clock and so it is not possible to establish the ordering of the sequence of observations (global trace) made. Instead, we know the order of events at each port and potentially are able to synchronise at certain points in the global trace.

Two effects of distributed testing have been discussed. First, there can be additional controllability problems; the tester $t_p$ at a port $p$ may not know when to send an input because it did not observe events at the other ports [26]. Let us suppose, for example, that the tester $t_q$ at port $q$ is to supply the first input $x_q$, this should lead to output $y_q$ at port $q$ only and we then want the tester $t_p$ at port $p \neq q$ to supply input $x_p$. Here $t_p$ does not know when to send its input since it does not observe the previous events. As a result, if we try to execute this test then we cannot be sure that the SUT receives $x_q$ before $x_p$.

A second effect, called an observability problem, occurs when we expect a global trace $\sigma$ to occur but there is another global trace $\sigma' \neq \sigma$ that is indistinguishable from $\sigma$ when making local observations [9, 10]. Here $\sigma$ and $\sigma'$ are indistinguishable if for every port $p$ we have that the projections onto $p$ of $\sigma$ and $\sigma'$ are identical. Let us suppose, for example, that the tester $t_q$ at port $q$ is to supply the first input $x_q$, this should lead to output $y_q$ at port $q$ and $y_p$ at $p \neq q$, this is to be followed by $x_q$ at $q$ and we expect output $y_q$ at $q$ only. Then the tester at port $p$ should observe $y_p$ and the tester at $q$ should observe $x_q y_q x_q y_q$. This is also the case if the SUT produces $y_q$ only in response to the first $x_q$ and produces $y_q$ and $y_p$ in response to the second $x_p$. Essentially, fault masking has occurred: The response to each input is not that specified but the two differences have masked one another. Thus, the differences are not found when running this test but might later be exhibited in use if a sequence in which there is no fault masking is used.

Many methods for avoiding or overcoming controllability and observability problems have been reported in the context of testing from a deterministic finite state machine (DFSM) [6, 5, 28, 18, 25, 27]. However, distributed systems are rarely deterministic and so the DFSM model is usually too restrictive for this domain. In addition, it is straightforward to produce examples in which we cannot achieve test objectives, such as executing particular transitions, if we restrict to controllable testing. Thus, these methods are use a formalism that is not suitable for distributed systems and also, even within the DFSM formalism, lack generality.

Controllability and observability problems arise from the inability of the testers to synchronise their actions. However, this can potentially be overcome if the testers can communicate with one another through sending coordination messages [4]. Again let us suppose that the tester $t_q$ at port $q$ is to supply the first input $x_q$, this should lead to output $y_q$ at port $q$ only and we then want the tester $t_p$ at port $p \neq q$ to supply input $x_p$. We have seen that there is a controllability problem since the tester at $p$ does not observe the events at $q$ and so cannot know when to send its input. However, if the testers can communicate then $t_q$ can send a message to $t_p$, after it has supplied its input. Similar schemes can be used to overcome observability problems. However, the establishment of an external network through which the testers can communicate can incur costs and the need to wait for coordination messages can make testing slower and so more expensive. In addition, there can be tests that cannot be run in this manner if there are timing constraints [21]. Other approaches that help the tester synchronise their actions include the use of monitors [1, 8, 13, 30] and methods for establishing an approximation to a global clock through the exchange of messages between agents (the testers) [4]. However, these approaches introduce additional costs and complexity into the testing process.

A separate line of research has explored conformance relations, for distributed testing, that reflect the observational power of the testers. Conformance relations were initially defined in the context of DFSMs [19] but more recent work has defined conformance relations for input/output transition systems [15, 16]. Conformance relations have also been defined for the case where there are distributed ports but global observations can be made [3, 14, 24].

The key problems introduced by distributed testing are controllability and observability. Controllability can make it difficult to design a test that achieves a given objective, such as reaching a state or executing a transition. Specifically, we may not be able to achieve a test objective without introducing controllability problems and these make it hard to know what sequence of inputs was actually applied to the SUT and to guarantee that the inputs are received in the desired order. Thus, controllability problems make it hard to apply a test that achieves an objective.

Observability problems can lead to fault masking. In addition, conformance relations that reflect the nature of distributed testing can be rather different from traditional conformance relations. For example, the conformance relation usually used when testing from a DFSM is an equivalence relation but this is not the case in distributed testing [19].

Having run a test, our observation is a set of sequences; one local trace for each port. If we wish to check whether the observed behaviour is consistent with a model $M$ then we need check this set of local traces against $M$. This potentially introduces a combinatorial explosion even if $M$ is deterministic since controllability problems can mean that there is an exponential number of orders in which the SUT could have received the input and so potentially we need to compare the local traces with an exponential number of global traces of $M$.

These problems are all potentially significant when testing any distributed system. For cloud computing, however, we have extra complexities in terms

of non-functional requirements. In addition, by their nature cloud systems are likely to be large and so we require methods that scale.

There are a range of approaches that might help distributed testing of cloud systems. These include the following.

1. Include some synchronisation in order to simplify test execution and the oracle problem and to increase the observational power of testing. However, there is a cost to synchronisation and so there is the challenge to introduce enough to help testing but not too much. In particular, it will be important to find the right places to apply synchronisation but also tests that require relatively little synchronisation. There is a trade-off here: synchronisation has a cost but it simplifies testing.

2. Further develop notions of correctness that represent limited observational ability. Here, it is particularly important to include non-functional properties. Some initial work has defined a conformance relation for probabilistic systems [17] but it appears that other non-functional aspects have not been investigated.

3. Adaptive automated testing for distributed testing. Testing must be adaptive but we might aim to produce tests that achieve probabilistic objectives. For example, rather than requiring a test to be guaranteed to reach a state we might be happy with it having a high probability of reaching the state. Ideally, the tests are such that we can determine after testing whether the objective was achieved and so re-run the test if it was not.

4. We might use results from game theory in test generation, since we can see testing as a game with incomplete information.

# 8 Conclusions

# References


[1] A. Bauer, M. Leucker, and C. Schallhart. Model-based runtime analysis of distributed reactive systems. In *17th Australian Software Engineering Conference (ASWEC 2006)*, pages 243–252. IEEE Computer Society, 2006.

[2] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[3] E. Brinksma, L. Heerink, and J. Tretmans. Factorized test generation for multi-input/output transition systems. In *FIP TC6 11th International Workshop on Testing Communicating Systems (IWTCS)*, volume 131 of *IFIP Conference Proceedings*, pages 67–82. Kluwer, 1998.

[4] L. Cacciari and O. Rafiq. Controllability and observability in distributed testing. *Information and Software Technology*, 41(11–12):767–780, 1999.

[5] J. Chen, R. M. Hierons, and H. Ural. Resolving observability problems in distributed test architectures. In *Formal Techniques for Networked and Distributed Systems (FORTE 2005)*, volume 3731 of *Lecture Notes in Computer Science*, pages 219–232. Springer, 2005.

[6] W. Chen and H. Ural. Synchronizable checking sequences based on multiple UIO sequences. *IEEE/ACM Transactions on Networking*, 3:152–157, 1995.

[7] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[8] P. S. Dodd and C. V. Ravishankar. Monitoring and debugging distributed real-time programs. *Software, Practice and Experience*, 22(10):863–877, 1992.

[9] R. Dssouli and G. von Bochmann. Error detection with multiple observers. In *Protocol Specification, Testing and Verification V*, pages 483–494. Elsevier Science (North Holland), 1985.

[10] R. Dssouli and G. von Bochmann. Conformance testing with multiple observers. In *Protocol Specification, Testing and Verification VI*, pages 217–229. Elsevier Science (North Holland), 1986.

[11] L. Duboc. *A Framework for the Characterization and Analysis of Software Systems Scalability*. PhD thesis, Department of Computer Science, University College London, February 2010.

[12] L. Duboc, D. Rosenblum, and T. Wicks. A framework for characterization and analysis of software system scalability. In *ESEC-FSE 2007: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 375–384, New York, NY, USA, 2007. ACM.

[13] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. Netlogger: A toolkit for distributed system performance analysis. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 267–273. IEEE Computer Society, 2000.

[14] L. Heerink and J. Tretmans. Refusal testing for classes of transition systems with inputs and outputs. In *Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE X/PSTV XVII)*, volume 107 of *IFIP Conference Proceedings*, pages 23–38. Chapman & Hall, 1997.

[15] R. M. Hierons, M. G. Merayo, and M. Núñez. Controllable test cases for the distributed test architecture. In *6th International Symposium on Automated Technology for Verification and Analysis (ATVA 2008)*, Lecture Notes in Computer Science, pages 201–215. Springer, 2008.

[16] R. M. Hierons, M. G. Merayo, and M. Núñez. Implementation relations for the distributed test architecture. In *20th IFIP TC 6/WG 6.1 International Conference on the Testing of Software and Communicating Systems (TestCom/FATES 2008)*, volume 5047 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2008.

[17] R. M. Hierons and M. Núñez. Testing probabilistic distributed systems. In *Joint 12th IFIP WG 6.1 International Conference, FMOODS 2010 and 30th IFIP WG 6.1 International Conference, FORTE 2010*, volume 6117 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2010.

[18] R. M. Hierons and H. Ural. Checking sequences for distributed test architectures. *Distributed Computing*, 21(3):223–238, 2008.

[19] R. M. Hierons and H. Ural. The effect of the distributed test architecture on the power of testing. *The Computer Journal*, 51(4):497–510, 2008.

[20] ISO/IEC. Software product evaluation—quality characteristics and guidelines for their use. Technical Report ISO/IEC TR 9126, 1991.

[21] A. Khoumsi. A temporal approach for testing distributed systems. *IEEE Transactions on Software Engineering*, 28(11):1085–1103, 2002.

[22] P. Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[23] A. v. Lamsweerde. *Systematic Requirements Engineering - From System Goals to UML Models to Software Specifications.* John Wiley & Sons, 2008.

[24] Z. Li, J. Wu, and X. Yin. Testing multi input/output transition system with all-observer. In *16th IFIP International Conference on Testing of Communicating Systems (TestCom 2004)*, volume 2978 of *Lecture Notes in Computer Science*, pages 95–111. Springer, 2004.

[25] G. Luo, R. Dssouli, and G. v. Bochmann. Generating synchronizable test sequences based on finite state machine with distributed ports. In *The 6th IFIP Workshop on Protocol Test Systems*, pages 139–153. Elsevier (North-Holland), 1993.

[26] B. Sarikaya and G. v. Bochmann. Synchronization and specification issues in protocol testing. *IEEE Transactions on Communications*, 32:389–395, April 1984.

[27] K.-C. Tai and Y.-C. Young. Synchronizable test sequences of finite state machines. *Computer Networks and ISDN Systems*, 30(12):1111–1134, 1998.

[28] H. Ural and C. Williams. Constructing checking sequences for distributed testing. *Formal Aspects of Computing*, 18(1):84–101, 2006.

[29] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, 26(10):978–1005, 2000.

[30] M. Zulkernine and R. E. Seviora. A compositional approach to monitoring distributed systems. In *International Conference on Dependable Systems and Networks (DSN 2002)*, pages 763–772. IEEE Computer Society, 2002.