2009 Ninth International Conference on Quality Software

RiTMO: A Method for Runtime Testability Measurement and Optimisation

Alberto Gonzalez-Sanchez, Eric Piel and Hans-Gerhard Gross

Software Engineering Research Group
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
Email: {a.gonzalezsanchez,e.a.b.piel,h.g.gross}@tudelft.nl

***Abstract*—Runtime testing is emerging as the solution for the integration and assessment of highly dynamic, high availability software systems where traditional development-time integration testing is too costly, or cannot be performed. However, in many situations, an extra effort will have to be invested in implementing appropriate measures to enable runtime tests to be performed without affecting the running system or its environment.**

This paper introduces a method for the improvement of the runtime testability of a system, which provides an optimal implementation plan for the application of measures to avoid the runtime tests' interferences. This plan is calculated considering the trade-off between testability and implementation cost. The computation of the implementation plan is driven by an estimation of runtime testability, and based on a model of the system. Runtime testability is estimated independently of the test cases and focused exclusively on the architecture of the system at runtime.

I. Introduction

Integration and system-level testing of complex, dynamic and highly available systems, such as Systems of Systems and Service Oriented Architectures, is becoming increasingly difficult and costly to perform in a dedicated development-time testing environment. Such systems cannot be duplicated easily, nor can their usage context. Moreover, in some cases, the components that will form the system are not available, or even known beforehand. Proper testing and validation of such systems can only be performed during runtime. *Runtime Testing* poses considerable runtime integration and testing challenges to engineers and researchers alike [1], [2], [3].

A prerequisite for runtime testing is the knowledge about which parts of the system can be tested while the system is operational without interfering with the system's operation or its environment. This knowledge can be expressed through the concept of *Runtime Testability* of a system [4].

Although runtime testing is also influenced by traditional testability factors [5], [6], [7], [8], [9], the interferences between the tests that are executed and the normal operation of the system (which determines the viability of runtime testing) are never taken into account. Features of the system which will produce unacceptable interference when tested, will have to be left untested, increasing the probability of leaving undetected faults.

The main contribution of this paper is RiTMO, a method to enhance the runtime testability of a system at deployment-time, before the test cases are executed. RiTMO computes an implementation plan for the application of isolation measures to avoid interferences caused by runtime tests, and therefore, improve the system's runtime testability. For this purpose, RiTMO uses an estimated value of runtime testability based on an annotated view of the system's runtime architecture, where the untestable features of the system have been identified [4], combined with the cost of implementing isolation measures needed to counter the interferences at the root of the untestability. Our approach reflects the trade-off that engineers have to consider, between the improvement of the runtime testability of the system after some interferences are addressed, and the implementation cost of the measures that have to be applied. We present a detailed application example of RiTMO to a system taken from our industrial case study in maritime safety and security systems.

The paper is structured as follows. In Section II the concept of Runtime Testability is introduced. In Section III, a method to improve runtime testability of a system is described. Section IV evaluates the proposed method. In Section V related work is presented and compared to our research. Section VI wraps up the paper and introduces some ideas for further research.

II. Runtime Testability

The fact that there is interference through runtime testing requires an indicator of the extent to which the running system can be tested without affecting its functionality or its environment. The standard definition of testability by the IEEE [10] can be rephrased to reflect this requirement, as follows:

Definition 1: Runtime Testability is (1) the degree to which a system or a component facilitates runtime testing without being affected; (2) the specification of which tests are allowed to be performed during runtime without affecting the running system.

Appropriate measurement and improvement methods for the first point should rely on general information about the system, *independent of the nature of the runtime tests that*

1550-6002/09 $26.00 © 2009 IEEE
DOI 10.1109/QSIC.2009.56

may be performed, whereas measurement and improvement methods for the second point should rely on dynamic information about the concrete test cases that are going to be performed as well. In this paper we will specifically concentrate on the first (system-centric) aspect of runtime testability, by using a model that captures the runtime architecture of the system right at the moment when tests will be executed.

Runtime testability is significantly influenced by *test sensitivity* [4]. Test sensitivity characterises which features of the system will cause interference between the running system and the test operations, e.g., a component having internal state, a component's internal/external interactions, or resource limitations.

A. Runtime Testability Measurement

Ultimately, components affected by sensitivity factors will impede runtime testing of certain features or requirements, increasing the probability of leaving undetected faults. The Runtime Testability Measurement (RTM) was defined in [4], as the quotient between the number of features of the system which can be runtime tested without interfering with the system, and the total number of features, i.e., as expressed by a test adequacy criterion:

$$RTM = \frac{|C_r|}{|C|} \tag{1}$$

where C is the complete set of features which have to be tested, and C_r is the subset of those features which can be tested without interference.

In this paper, the value of RTM is calculated for an instantiation of Equation 1 for component-based systems, based on a runtime dependency graph model annotated with runtime testability information, that captures the runtime architecture of the dynamic system at the moment of testing, known as Component Interaction Graph (CIG) [11].

A CIG is defined as a directed graph $CIG = (V, E)$. The vertex set, $V = V_P \cup V_R$, is formed by the union of the sets of provided and required vertices, where each vertex represents an operation either provided or required by a certain component's interface. Edges in E account for two situations: (1) provided operations of a component that depend on required operations of that same component (intra-component); and (2) required operations of a component bound to the actual provider of that service (inter-component).

Vertices in the CIG are annotated with a testability flag, τ_i, which represents whether using the operation the vertex symbolises during a test will cause interferences or not. In Section III, more details are given on how to derive the sets of vertices, edges and testability annotations.

This runtime dependency graph abstraction is detailed enough to link key runtime testability issues to the individual operations of components that cause them, and, on the other hand, it is simple enough so that its derivation from the component's design and the system's runtime architecture is an easy task, and its computation is a tractable problem.

The runtime testability measurements proposed in [4] provides an estimate of the impact on the reliability that runtime testing will have on a system. Knowledge of this impact will allow selecting test isolation techniques to be implemented by engineers into specific components to counter their test sensitivity. Examples of these techniques are state duplication, component cloning, usage of simulators and resource monitoring [1], [4].

III. RiTMO: a Method to Improve Runtime Testability

In this Section we introduce RiTMO (*RuntIme Testability Measurement and Optimisation*), a method for improving the runtime testability of a given system based on RTM. This method helps developers in elaborating an implementation plan, identifying parts of a system where an effort has to be invested in the implementation of test isolation measures, in order to maximise the RTM for a given budget. It can also be used to compute the minimal budget required to reach a target RTM.

Typical usage scenarios are the deployment or the update of systems with high availability and reliability requirements during run-time. In order to allow good runtime test adequacy, integration engineers aim for a high RTM. By combining component design information with the system's architecture, the system integrator uses RiTMO in order to determine where the budget is best spent in order to increase runtime testability.

Figure 1 depicts the five main steps which compose the RiTMO method, associated to the engineering roles in charge of performing the task. It must be noted that a single person can have multiple roles. This figure is complemented by Table I, in which details on the inputs and outputs of each activity are provided. The first step consists in analysing the dependencies between each part of the system. The second step aims at determining which operations cannot be tested at runtime. At this point, the RTM of the system in its current implementation can be computed. The third step determines the cost of applying an adequate isolation technique to each of the untestable operations. The fourth step uses our algorithm to obtain the optimal action plan for increasing the RTM. The fifth and final step consists in applying this action plan to effectively obtain a more runtime testable system. Each of these steps is described in the following subsections in more detail.

A. Step 1: Architecture Analysis

In order to compute the RTM of a system, the first step is to provide a view of the runtime architecture of the system at the right level of granularity, by means of a CIG as depicted in Figure 2.

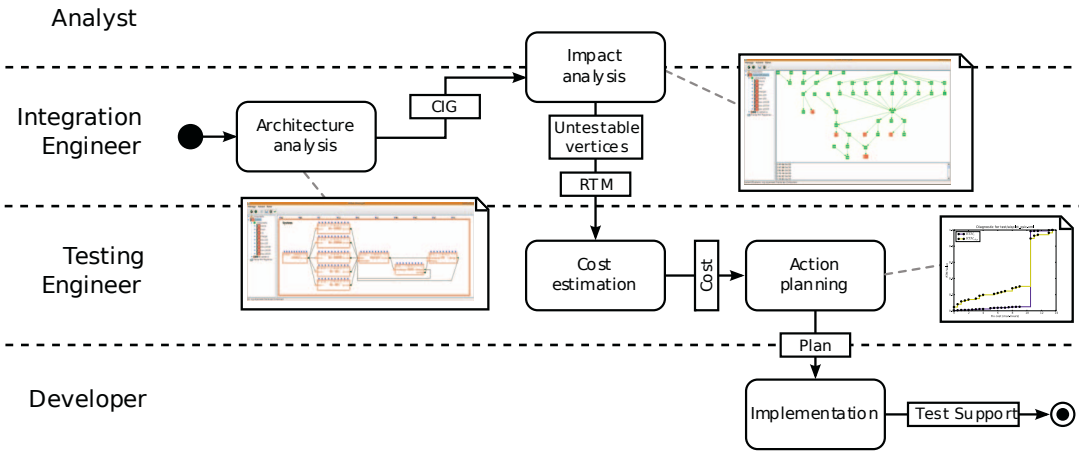

Figure 1. Complete workflow of RiTMO.

Activity	Actors	Inputs	Outputs
Architecture analysis	Integration Engineer	Component designs Bindings	CIG
Impact analysis	Integration Engineer Analyst	CIG Domain knowledge Design knowledge	Untestable vertices Initial RTM value
Cost estimation	Test Engineer	CIG Untestable vertices Component designs Isolation techniques	Vertex fix proposal Vertex fix costs
Action planning	Test Engineer	CIG Untestable vertices Vertex fix costs Target Budget or RTM	Action plan Estimated cost Estimated RTM value
Implementation	Developer	Action plan Vertex fix proposal	Test support code

Table I
ACTORS, INPUTS AND OUTPUTS OF EACH ACTIVITY INVOLVED IN RITMO.

Vertices in the CIG are obtained by inspecting the required and provided interfaces of each component. For each method of each interface, a vertex is created. Vertices for unused operations are removed as they would add unreachable paths. Composite components are treated in a similar way: vertices are added for each of the operations in their interfaces.

Edges are obtained in two steps. First, internal edges representing dependencies between provided and required operations inside components have to be derived. If models are available and detailed enough, internal edges can be derived from a component's model. Otherwise, internal edges can be either derived by static analysis of the component's implementation or, if not available either, by dynamic analysis of the component running on the actual system [12].

Secondly, the CIGs of each component instance are composed, by adding edges from required to provided operations, which can be directly obtained from the runtime bindings between the component instances in the system. For each connection of a required interface to a provided interface, an edge is added between the corresponding vertices. Similarly, all delegation dependencies between an interface of a composite component and the interface of the subcomponent in which it delegates, are also represented as edges.

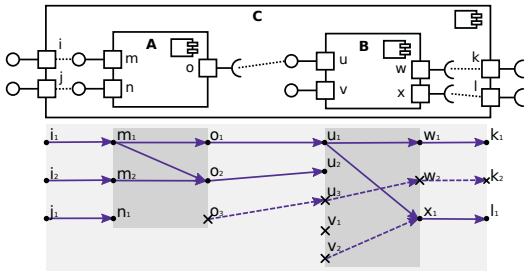

Figure 2. From components to CIG.

B. Step 2: Impact Analysis and RTM

The goal of this step is to annotate the just obtained CIG with the test impact flag τ_i. The derivation of this information cannot be easily automated, and it is up to the integration engineers and analysts to apply their design and

domain knowledge for deciding which method calls could disrupt the normal operation of the system.

In case the system is large, inspecting each component for runtime test sensitivity factors can be a laborious task. However, it can be greatly simplified if component vendors associate this information to components, as it is easily reusable.

The value of RTM is computed as described in [4]. If this value is higher than the target adequacy criteria for the system tests, the system is sufficiently runtime testable and the method finishes. If this is not the case, improvements on the system must be performed.

C. Step 3: Cost Estimation

The third step of the RiTMO method consists in complementing the graph of the system with an estimation of the effort needed for the implementation of countermeasures to allow testing a certain untestable vertex. For each vertex marked with a $\tau = 0$, a cost, c_i, which represents the estimated cost of implementing a suitable test isolation measure. Typically, the cost is expressed in terms of time (e.g., man-hours) or money.

Untestable operations are assessed by combining knowledge of the possible test isolation measures countermeasures (e.g., state duplication, cloning, simulation) with knowledge about the developer team capabilities, in order to obtain an estimation of the implementation effort.

D. Step 4: Computation of Implementation Plan

The completely annotated CIG contains information about the untestable methods and the cost to make them testable. It is then possible to compute a plan automatically for the implementation of the isolation measures in the components that will yield the optimal RTM given a maximum budget. Conversely, it is also possible to compute a plan with the minimal cost to reach a fixed RTM.

A general exhaustive search algorithm is not applicable in practice to systems of realistic size due to its exponential complexity. This complexity can be tackled with a heuristic near-optimal algorithm [13]. The resulting implementation plan consists of the list of operations to address, the chosen isolation technique, and the final expected RTM value.

E. Step 5: Application of the Plan

The last step of the method consists in following the implementation plan by implementing the isolation measures for each of the operations selected by the algorithm. Once this step is complete, the final deliverables of the method are ready and can be applied to the system's components. The test process can then begin.

IV. Application Example

In this section we will describe an application example of RiTMO to a component-based subsystem taken from our industrial case study.

A. System Setup and Architecture

The system used in our integration experiment is a vessel tracking system taken from our industrial case study, codenamed AISPlot. It is part of a component-based system from the maritime safety and security domain. The architecture of the AISPlot system is shown in Figure 3.

The system is used to track the position of ships sailing a coastal area, detecting and managing potential dangerous situations. Position messages are broadcast through radio by ships (represented in our experiment by the `World` component), and received by a number of base stations (`BS` components) spread along the coast. If a message received from the simulator belongs to the area the base is covering, it is relayed to the `Merger` component through the `AISin` interface. `Merger` removes duplicates (some base stations cover overlapping areas), and offers a subscription service to client components for receiving updates on the status of vessels. `Merger` offers the `Csubscribe` and `Cunsubscribe` to components interested in receiving these updates. `Clients` then have to support four operations for notification of ship status: `Cnew`, `Cpublish`, `Creply` and `Cdispose`. `Monitor` scans all the received messages in search for inconsistencies in the data sent by the ships. `Visual` draws the position of all ships on a screen in the control centre, and also the warnings generated by the `Monitor` component, via the `Warning` operation.

AISPlot is implemented in Java, on the ATLAS/FRACTAL runtime integration and testing research platform [2]. It is a dynamic, reflective component-based platform that allows the insertion, modification, removal and testing of components at run-time.

B. Application of RiTMO

1) Architecture analysis: To derive the internal edges, the *call hierarchy* functionality of Netbeans[1] was used, automatically deriving a call tree of each of the public operations in the implemented interfaces of each component, and locate calls to methods of required interfaces.

Because of the reflective capabilities of ATLAS/FRACTAL, the derivation of the set of required and provided operations into vertices, and the creation of the external edges were a straightforward and completely automated process. Figure 3 shows the reconstructed view of the system's runtime architecture (components and runtime bindings) obtained by querying the reflection interfaces provided by FRACTAL.

During the system deployment, the management console of ATLAS/FRACTAL automatically composes all the CIGs of the primitive components to obtain the complete CIG of AISPlot, which can be seen in Figure 4.

2) Impact analysis: The `World` and `BS` components are stateless and have no interaction outside the system's

[1] http://www.netbeans.org

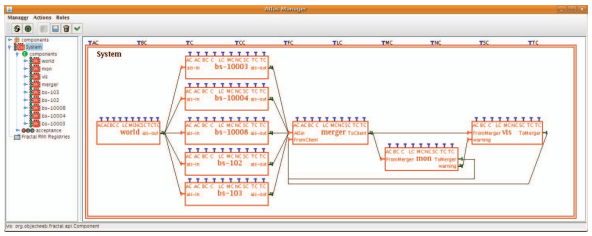

Figure 3. Runtime-reconstructed Architecture

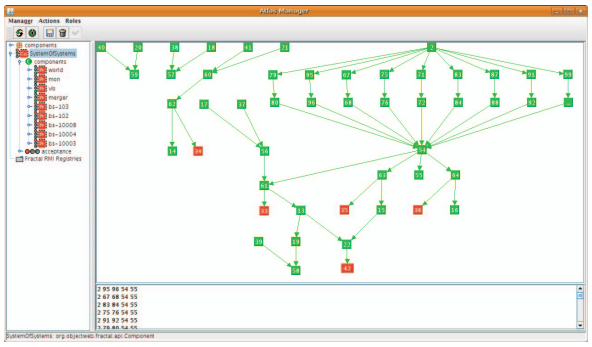

Figure 4. AISPlot CIG during impact analysis

boundaries. Therefore, no impact is caused by invoking their operations during a test.

On the other hand, `Merger` and `Monitor` are stateful components, as both of them store tables of received messages and vessel information internally. This information will be altered by test operations, and therefore these components are test sensitive. The merger component has to manage vessel subscriptions from clients as well. Because the set of subscribed ships will be altered during testing, this is also a source of test sensitivity.

The visualiser component stores internally a list of observed vessels (ships to which it has subscribed in `Merger`), which is already a source of test sensitivity. Moreover, the visualiser component is interacting with a real display through a socket connection. This interaction must be isolated in some way so that users of the system do not see the test vessels and warnings drawn on the display.

The preliminary value of RTM can be seen in the *before* row of Table II. Due to the pipelined nature of the system, all the operations preceding the untestable ones will be indirectly untestable, and only a very low number of vertices will remain testable: those which are not related to the main pipeline path, hence the extremely low RTM values.

3) Cost estimation and planning: Given our previous knowledge in implementing a set of analogous measures for a system with components of similar characteristics, the cost of implementing test support for the untestable operations in `Merger` and `Monitor` was estimated to be approximately 0.5 man-hours, and the implementation of the isolation and observation code for each operation in `Visual` would have

	Total	Testable	RTM
Before	86	4	0.046
After	86	77	0.895
Plan	13,15,16,33,35,36,42,54,58		
Cost	10.5 man-hours		

Table II
RTM RESULTS OF THE IMPACT ANALYSIS AND IMPROVEMENT PLAN

v_i	Operation	Sensitivity	Isolation	Cost
13	monitor.Cnew	state	separate state	0.5
14	monitor.Creply	state	separate state	0.5
15	monitor.Cpublish	state	separate state	0.5
16	monitor.Cdispose	state	separate state	0.5
33	visual.Cnew	state, interaction	separate, redirect	2
34	visual.Creply	state, interaction	separate, redirect	2
35	visual.Cpublish	state, interaction	separate, redirect	2
36	visual.Cdispose	state, interaction	separate, redirect	2
42	visual.Warning	interaction	redirect output	2
54	merger.MessageIn	state	separate state	0.5
58	merger.Csubscribe	state	separate state	0.5
59	merger.Cunsubscribe	state	separate state	0.5

Table III
UNTESTABLE OPERATIONS IN AISPLOT

an estimated cost of 2 man-hours. Table III summarises all the test sensitive operations in each component, along with suitable isolation techniques and their estimated costs.

Figure 5 shows the RTM as a function of the cost spent developing test support artefacts, as calculated by our testability optimisation tool. Because of the pipelined architecture of the system, all the untestable vertices in the pipeline have to be fixed to obtain a substantial improvement of testability. This can be seen on the big jump in testability in Figure 5 when 10.5 man-hours are dedicated to testability improvement. Without the information in the plot, it would have been difficult to find this issue.

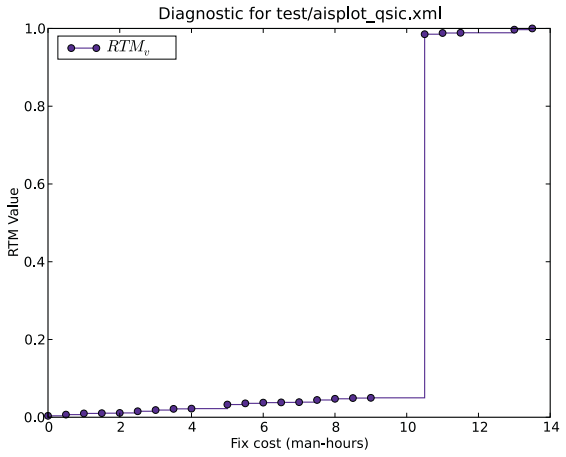

Figure 5. Evolution of RTM as the plan progresses

The *after* row in Table II shows the final value for the RTM of the system once all the countermeasures have been applied to the untestable vertices, along with the vertices which are part of the plan.

C. Discussion

There are number of issues that have to be considered concerning the applicability of the method. First, it must be taken into account that the quality of the plan depends greatly on the quality of the predicted costs. Care should be taken in obtaining a development cost estimation model to obtain realistic (and therefore, useful) improvement plans. Second, as a system can have a very large number of components and vertices, a (semi-)automatic tool that supports all the steps of RiTMO is needed. Although Figures 3 and 4 show our preliminary implementation work, it is in a very early stage, especially with respect to finding the components that contain test sensitive features, which still has to be done manually. Finally, for better cost estimation, the process for defining isolation costs could be refined to allow dependencies between vertex fixes, as often several operations of a component have to be fixed as a whole.

V. Related work

Our approach to runtime testability, presented in [4], and complemented with the definition of RiTMO in this paper is influenced by the work presented in [1], where the concepts of test sensitivity and test isolation are introduced. However, no mention of, or relation to the concept of runtime testability are done. A method based on a measurement of testability, from the point of view the static structure of the system is presented in [8] to assess the maintainability of the system. Our approach is similar in that runtime testability is influenced by the architecture of the system under consideration, although during runtime instead of during compilation time.

VI. Conclusions and Future Work

In this paper we have presented RiTMO, a cost-driven method for the improvement of runtime testability based on RTM, a measurement for the runtime testability of a component-based system based solely on characteristics of the system under test. RiTMO enables integration engineers to identify critical situations of bad system runtime testability and to compute and execute an implementation plan to improve it. Future work will focus on the improvement of the RTM measurement itself, as well as the development of automated or semi-automated methods for performing the impact analysis and fix cost estimation, and the support of RiTMO by a CASE tool. Finally, additional empirical evaluation using industrial cases and synthetic systems is planned, in order to explore further the relationship between RTM and defect coverage and reliability.

Acknowledgement

This work has been carried out as part of the POSEIDON project under the responsibility of the Embedded Systems Institute (ESI), Eindhoven, The Netherlands. This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK03021 program.

References

[1] D. Brenner, C. Atkinson, R. Malaka, M. Merdes, B. Paech, and D. Suliman, "Reducing verification effort in component-based software engineering through built-in testing," *Information Systems Frontiers*, vol. 9, no. 2-3, pp. 151–162, 2007.

[2] A. González, É. Piel, and H.-G. Gross, "Architecture support for runtime integration and verification of component-based systems of systems," in *1st International Workshop on Automated Engineering of Autonomous and run-time evolving Systems (ARAMIS 2008)*. L'Aquila, Italy: IEEE Computer Society, Sep. 2008, pp. 41–48.

[3] C. Murpy, G. Kaiser, I. Vo, and M. Chu, "Quality assurance of software applications using the in vivo testing approach," in *ICST '09: Proceedings of the 2nd international Conference on Software Testing*. IEEE Computer Society, 2009.

[4] A. González, E. Piel, and H.-G. Gross, "A model for the measurement of the runtime testability of component-based systems," in *Software Testing Verification and Validation Workshop, IEEE International Conference on*. Denver, CO, USA: IEEE Computer Society, 2009, pp. 19–28.

[5] A. Bertolino and L. Strigini, "Using testability measures for dependability assessment," in *ICSE '95: Proceedings of the 17th international conference on Software engineering*. New York, NY, USA: ACM, 1995, pp. 61–70.

[6] R. S. Freedman, "Testability of software components," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 553–564, 1991.

[7] D. Hamlet and J. Voas, "Faults on its sleeve: amplifying software reliability testing," *SIGSOFT Software Engineering Notes*, vol. 18, no. 3, pp. 89–98, 1993.

[8] S. Jungmayr, "Identifying test-critical dependencies," in *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 404–413.

[9] J. Voas, L. Morrel, and K. Miller, "Predicting where faults can hide from testing," *IEEE Software*, vol. 8, no. 2, pp. 41–48, 1991.

[10] "IEEE standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, 1990. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=159342

[11] Y. Wu, D. Pan, and M.-H. Chen, "Techniques for testing component-based software," in *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems*. Los Alamitos, CA, USA: IEEE Computer Society, 2001, pp. 222–232.

[12] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proc. 30th Int'l. Conf. on Softw. Eng. (ICSE'08)*, Leipzig, Germany, May 2008, pp. 501–510.

[13] A. González, E. Piel, H.-G. Gross, and A. van Gemund, "Runtime testability in dynamic highly available component-based systems," Delft University of Technolgy, Software Engineering Research Group, Tech. Rep. TUD-SERG-2009-009, 2009.