Delft University of Technology
Software Engineering Research Group
Technical Report Series

Runtime Testability in Dynamic Highly-Availability Component-based Systems

Alberto González, Éric Piel,
Hans-Gerhard Gross, Arjan J.C. van Gemund

Report TUD-SERG-2010-025

TU Delft

SERG

TUD-SERG-2010-025

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Published at Valid'10

Runtime Testability in Dynamic High-Availability Component-based Systems

Alberto Gonzalez-Sanchez, Éric Piel, Hans-Gerhard Gross, and Arjan J.C. van Gemund
Department of Software Technology
Delft University of Technology
Mekelweg 4, 2628CD Delft, The Netherlands
{*a.gonzalezsanchez, e.a.b.piel, h.g.gross, a.j.c.vangemund*}*@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 cannot be performed. A prerequisite for runtime testing is the knowledge about to which extent the system can be tested safely while it is operational, i.e., the system's *runtime testability*. This article evaluates RTM, a cost-based metric for estimating runtime testability. It is used to assist system engineers in directing the implementation of remedial measures, by providing an action plan which considers the trade-off between testability and cost. Two testability case studies are performed on two different component-based systems, assessing RTM's ability to identify runtime testability problems.**

***Keywords*-Runtime testability; runtime testing; measurement; component-based system.**

I. Introduction

Integration and system-level testing of complex, high-available systems is becoming increasingly difficult and costly in a development-time testing environment because system duplication for testing is not trivial. Such systems have high availability requirements, and they cannot be put off-line to perform maintenance operations, e.g., air traffic control systems, emergency unit systems, banking applications. Other such systems are dynamic Systems-of-Systems, or SOA for which the sub-components are not even known a priori [1], [2].

Runtime testing [3] is an emerging solution for the validation and acceptance testing for such dynamic high-availability systems, and a prerequisite is the knowledge about which items can be tested safely while the system is operational. This knowledge can be expressed through the concept of *runtime testability* of a system, and it can be referred to as *the relative ease and expense of revealing software faults*. Testability enhancement techniques have been proposed either to make a system less prone to hiding faults [4], [5], [6], or to select the test cases that are more likely to uncover faults with the lowest cost [7], [8], [9], [10].

This paper evaluates the runtime testability metric (RTM) introduced in our earlier work [11], [12]. The metric 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 cost of the remedial measures that have to be applied. The main contribution of the paper is the empirical evaluation of the metric. In addition, scalable algorithm is introduced to calculate the near-optimal *action plan* which list by effectiveness the operations which must become testable to improve the RTM.

The paper is structured as follows. In Section II runtime testability is defined. Section III evaluates the RTM on two example cases. In Section IV, the action plan algorithm is presented. Finally, Section V presents our conclusions and future plans.

II. Runtime Testability

RTM addresses the question to which extent a system may be runtime tested without affecting it or its environment. Following the IEEE definition of testability [13], runtime testability can be defined as (1) the extent to which a system or a component facilitates runtime testing without being extensively affected; (2) the specification of which tests are allowed to be performed during runtime without extensively affecting the running system. This considers (1) the characteristics of the system and the extra infrastructure needed for runtime testing, and (2) the identification of which test cases are admissible out of all the possible ones. An appropriate measurement for (1) provides general information on the system independent of the nature of the runtime tests that may be performed, as it is proposed in [5], [6] for traditional testing. A measurement for (2) will provide information about the test cases that are going to be performed, as proposed in [7], [8], [9]. Here, we concentrate on (1), in the future, we will also consider (2).

Runtime testability is influenced by two characteristics of the system: *test sensitivity*, and *test isolation* [14]. Test sensitivity characterises features of the system suffering from test interference, e.g., existence of an internal state in a component, a component's internal/external interactions, resource limitations. Test isolation is applied by engineers in order to counter the test sensitivity, e.g., state duplication or component cloning, usage of simulators, resource monitoring.

In [11], [12], we define a numerical measurement for runtime testability as fraction of the features of the system that may be runtime tested for an affordable cost; i.e., in terms of the maximum test coverage attainable by the system testers under runtime testing conditions. This estimate is used to

indicate insufficient testing of some features of the system due to prohibitive costs during runtime testing, before any test is executed. Runtime Testability Measurement (RTM), is defined as $RTM = |C_r|$, and its associated relative metric, independent of the system, as $rRTM = \frac{|C_r|}{|C|}$ 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 at an acceptable cost.

Model of the System: In order to apply RTM, the system is modelled through a Component Interaction Graph (CIG) [15]. Example CIGs are shown in Fig. 1 and Fig. 2. Nodes or vertices of the CIG represent component operations annotated with a testability flag, i.e., small black testable, large red crossed untestable. Edges of the CIG represent (1) provided services of a component that depend on required services of that same component (intra-component); and (2) required services of a component bound to the actual provider of that service (inter-component). Edge information from within a component can be obtained either by static analysis of the source code, or by providing state or sequence models [15]. Inter-component edges can be derived from the runtime connections between the components.

Estimation of RTM: RTM is estimated in terms of impact cost of covering the features represented in the graph. We do not look at the concrete penalty of actual test cases, but at the possible cost of a test case trying to cover each element. Because CIG is a static model, assumptions have to be made on the actual behaviour of test cases. In the future, we will enrich the model with additional dynamic information to relax these assumptions. Because of lacking control flow information, there is no knowledge about edges in the CIG that will be traversed by a test case. In the worst case, the interaction might propagate through all edges, affecting all reachable vertices. For the moment, we assume this worst case.

Improving the System's RTM: Systems with a high number of runtime untestable features (i.e., low runtime testability) can be improved by applying isolation techniques to specific vertices, to bring their impact cost down to an acceptable level. However, not all interventions have the same cost, nor do they provide the same gain. Ideally, the system tester would plot the improvement of runtime testability versus the cost of the fixes applied, in order to get full information on the trade-off between the improvement of the system's runtime testability and the cost of such improvement. This cost depends on the isolation technique employed: adaptation cost of a component, development cost of a simulator, cost of shutting down a part of the system, addition of new hardware, etc. Even though these costs involve diverse magnitudes (namely time and money), for this paper we will assume that they can be reduced to a single numeric value: c_i.

III. Application Examples

Two studies were performed (1) AISPlot, a system-of-systems taken from the maritime safety and security domain, and (2) WifiLounge, an airport's wireless access-point system. These two systems are representative of the two typical software architectures: the first system follows a data-flow organization, while the second one follows a client-server organization. These cases show that RTM can identify parts of a system with prohibitive runtime testing cost, and it can help choose optimal action points with the goal of improving the system's runtime testability. The $CIGs$ were obtained by static analysis of the code. The inter-component edges were obtained during runtime by reflection. The runtime testability and fix cost information c_i were derived based on test sensitivity information obtained from the design of each component, and the cost of deploying adequate test isolation measures. AISPlot (WifiLounge) has 31 (9) components, 86 (159) vertices, and 108 (141) edges.

Example AISPlot: AISPlot is used to track the position of ships sailing a coastal area, detecting and managing potential dangerous situations. Messages are broadcast by ships, and received by base stations spread along the coast. Each message is relayed to a *Merger* component removing duplicates coming from different stations. A *Monitor* component scans all messages looking for inconsistencies in the ship data, and another component, *Vis* shows all ships on a screen. Fig. 1 shows the CIG for AISPlot.

Cost	Proposed fix					Testability	
	v33	v34	v35	v36	v42	RTM	$rRTM$
0						12	0.140
1		×				17	0.198
2	×				×	21	0.244
3	×	×			×	26	0.302
4	×		×	×	×	81	0.942
5	×	×	×	×	×	86	1.000

Table I
Testability analysis for AISPlot

Five *Vis* operations have testability issues (manually determined), displaying test ship positions and test warnings on the screen if not properly isolated. Table I shows that runtime testability is low. Only 14% of the vertices can be runtime tested. This poor RTM comes from the architecture of the system being organised as a pipeline, with the *Vis* component at the end, connecting almost all vertices to the five problematic vertices of the *Vis* component. We explored the possible combinations of isolation of any of these 5 vertices and computed the optimal improvement on RTM, assuming uniform cost of 1 to isolate an operation. Table I shows the best combination of isolation for each possible cost, × denoting the isolation of a vertex, and the RTM if these isolations were applied. The numbers suggest little gain in testability, as long as vertices v33, v35, v36 and v42 are not made runtime testable together. The four vertices appear at the end of the processing pipeline affecting the predecessor set of almost every vertex together. They must

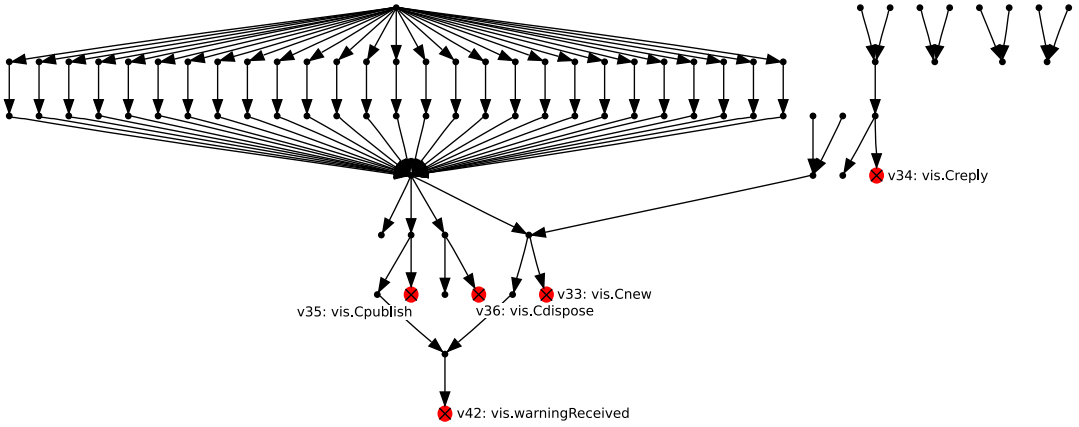

Figure 1. AISPlot Component Interaction Graph

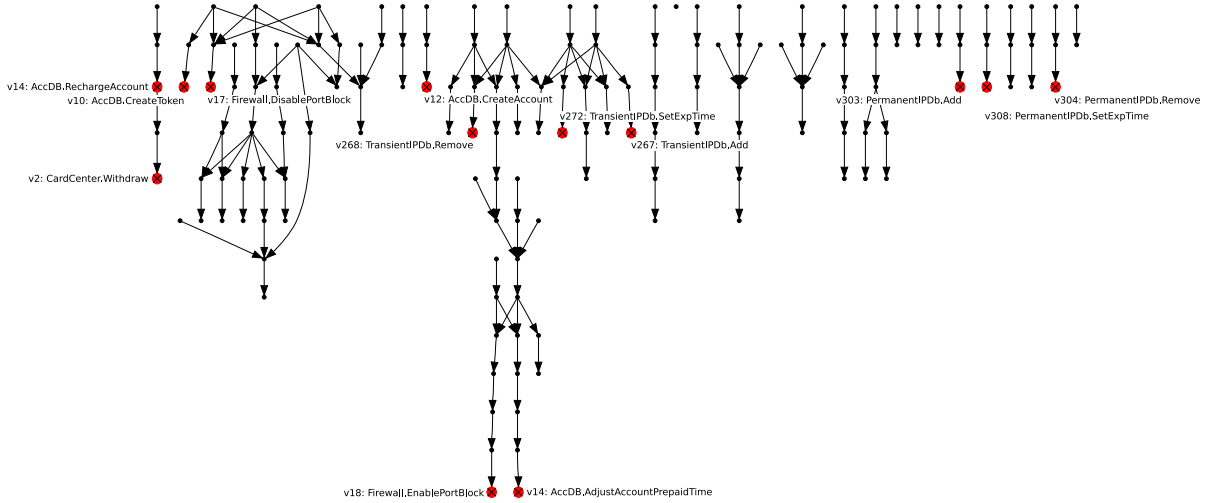

Figure 2. WifiLounge Component Interaction Graph

be fixed at once for any testability gain, leading to the jump at cost 4 for AISPlot in Figure 3 ($rRTM$ going from 0.302 to 0.942).

Example WifiLounge: When a client accesses the wifi network, a *DhcpListener* generates an event indicating the assigned IP address. All communication is blocked until authenticated. Business class clients are authenticated in the ticket databases of airlines. Frequent fliers are authenticated against the program's database, and the ticket databases for free access. Prepaid-clients must create an account in the system, linked to a credit card entry. After authentication, blocking is disabled and the connection can be used. Fig. 2 shows the CIG of *WifiLounge*.

Thirteen operations are runtime untestable, i.e., state modification operations of the *AccountDatabase*, *TransientIpDb* and *PermanentIpDb* components are considered runtime untestable because they act on databases. A withdraw operation of a *CardCenter* component is also not runtime testable because it uses a banking system outside our control. *Firewall* operations are also not runtime testable because this component is a front-end to a hardware element (the network), impossible to duplicate.

RTM is intermediate: 62% of the vertices can be runtime tested. This is much better than AISPlot, because the architecture is more "spread out" (compare both CIGs). There are runtime-untestable features, though they are not as interdependent as in AISPlot. We examined possible solutions improving RTM, displayed in Table II, and shown in Fig. 3 (Airport Lounge). The number of vertices that have to be made runtime testable for a significant increase in RTM is much lower than for AISPlot. Two vertices (v14 and v18) cause the "biggest un-testability." The other vertices are not so problematic and the value of RTM grows more linearly with each vertex becoming runtime testable.

Discussion: The two cases demonstrate the value of RTM. By identifying operations causing inadmissible effects, we can predict the runtime untestable features, leading to an optimal action plan for runtime testable features. These techniques are applied before running test cases.

Because of the static model, RTM represents a pessimistic

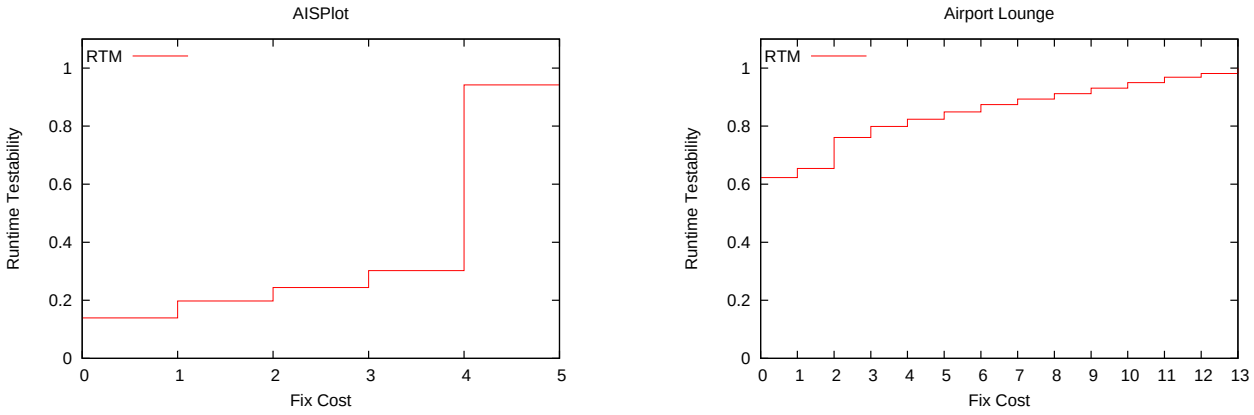

Figure 3. Optimal improvement of RTM vs. Fix cost

Cost	Proposed fix													Testability	
	v2	v10	v12	v13	v14	v17	v18	v267	v268	v272	v303	v304	v308	RTM	$rRTM$
0														99	0.623
1						×								103	0.648
2					×		×							121	0.761
3					×		×		×					127	0.799
4					×	×	×		×					131	0.824
5					×	×	×	×	×					135	0.849
6					×	×	×	×	×	×				139	0.874
7		×			×	×	×	×	×	×				142	0.893
8		×	×		×	×	×	×	×	×				145	0.912
9	×	×		×	×	×	×	×	×	×				147	0.925
10	×	×	×	×	×	×	×	×	×	×				150	0.943
11	×	×	×	×	×	×	×	×	×	×	×			153	0.962
12	×	×	×	×	×	×	×	×	×	×	×	×		156	0.981
13	×	×	×	×	×	×	×	×	×	×	×	×	×	159	1.000

Table II
TESTABILITY ANALYSIS OF AIRPORT LOUNGE

estimate, and we expect improvement by adding dynamic runtime information in the future, e.g., applying [16]. A high value, even if underestimated, is, nevertheless, a good indicator that the system is well prepared for runtime testing, and that the tests cover many system features. For instance, we looked at the dynamic behaviour of each system by executing an exhaustive test suite in terms of path coverage [12]. For both *AISPlot* and *WifiLounge*, about 30 to 40% of the test cases were considered touching untestable operation by the RTM definition, although in reality they were not. This is due to the complexity of the control flow. Many exclusive branch choices are not represented in the static model.

An interesting issue is the relationship between RTM and defect coverage. Even though the relationship between test coverage and defect coverage is not clear [17], previous studies have shown a beneficial effect of test coverage on reliability [18], [19].

IV. TESTABILITY OPTIMISATION

RTM analysis and action planning corresponds to the Knapsack problem [20], an NP-hard binary integer programming problem, which can be formulated as

$$\begin{aligned} &maximise: && RTM \\ &subject\ to: && \sum c(v_j) \cdot x_j \leq b, \quad x_j \in \{0,1\} \end{aligned}$$

with b = maximum budget available, and x_j = decision of including vertex v_j in the action plan. In this section, we present a way for an approximate action plan using the greedy heuristic method according to Algorithm 1, in which CIG is the interaction graph, U is the set of untestable vertices, and $H(v)$ a heuristic function to be used. For each pass of the loop, the algorithm selects the vertex in U with the highest heuristic rank, and removes it from the set of untestable vertices. The rank is updated on each pass.

Algorithm 1 Greedy Approximate Planning

```
function FixActionPlan(CIG, U, H(v))
    Sol ← ∅                          ▷ List to hold the solution
    while U ≠ ∅ do
        v ← FindMax(U, H(v))
        Append(Sol, v)
        Remove(U, v)
    return Sol
```

The method relies on heuristics that benefit from partial knowledge about the structure of the solution space of the problem. To motivate our heuristic approach, we analyse the properties of the RTM-cost combination space shown in Fig. 4. The dot-clouds show the structure and distribution of all the possible solutions for the vertex and context-

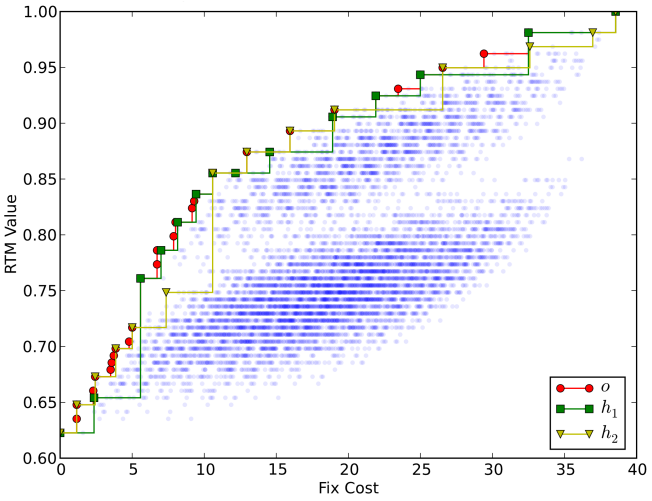

Figure 4. Optimal and heuristic RTM optimisations

dependence RTM optimisation problems of the *WifiLounge* system. On a system where the cost of fixing any vertex is uniform, and all the uncoverable vertices or paths come from only one untestable vertex, there would be only one cloud. However, we identified two interesting characteristics of the inputs affecting the structure of the solution space.

First, in most systems multiple untestable vertices will participate in the same uncoverable elements. This is he case for both examples. If a group of untestable vertices participates together in many un-coverable features, the solution cloud will cast a "shadow" on the RTM axis, i.e., any solution that includes those vertices will get a better testability. Second, vertices with exceptionally high cost will shift any solution that includes them towards the right in the cost axis, causing a separate cloud to appear. In this case, any solution which contains them will get a cost increase, due to space concerns not shown in the plots. An example of the first characteristic is in Figure 4, where the upper cloud corresponds to all the solutions which include vertices v14 and v18. We used the knowledge about these two situations to define heuristics to be used in Algorithm 1, based on the idea that dependent vertices are only useful if they are all part of the solution and expensive vertices should be avoided unless necessary.

Heuristics: First, we consider a pessimistic heuristic. It ranks higher the vertices with the highest gain on testability. The count is divided by the cost to penalise expensive nodes:

$$h_{pessimistic}(v_i) = \frac{1}{c_i}(RTM_{v_i} - RTM) \qquad (1)$$

with RTM_{v_i} = RTM after the cost for vertex v_i was spent. We expect this heuristic to perform well for low budgets, and poorly for higher ones. It should be noted that we can define this heuristic because RTM was proven to have ratio scale [12].

The second heuristic is optimistic. It ranks higher the vertices that appear in the highest number of P sets, i.e., the

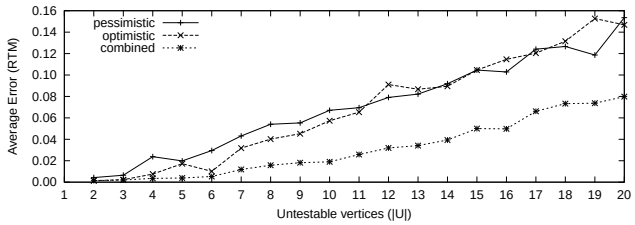

Figure 5. Performance of the approximate algorithms vs. the optimal

vertices that will fix the most uncoverable vertices assuming they only depend on the vertex being ranked. This value is also divided by the cost to penalise expensive nodes over cheaper ones:

$$h_{optimistic}(v_i) = \frac{1}{c_i}|\{v_j \mid v_i \in S_{v_j}\}| \qquad (2)$$

By ignoring the fact that an uncoverable vertex may be caused by more than one untestable vertex, and that the vertex may not be reachable through a testable path, this second heuristic will take very optimistic decisions on the first passes, affecting the quality of results for proportionally low budgets, but yields a better performance for higher ones.

Fig. 4 shows the performance for both heuristics (h_1 & h_2) for the *WifiLounge* (compared to the optimal solution o, obtained by exhaustive search). The steps in the optimal solution are not incremental and the action plans at each step could be completely different. The optimistic ranking skips many low-cost solutions (with curve much lower than the optimum), while the pessimistic heuristic is more precise for low cost, but completely misses good solutions with higher budgets. These shortcoming may be addressed through combining both heuristic rankings and taking the best results of both. However, the steps in the solution will not be incremental if the solutions intersect with each other (as in Figure 4).

Computational Complexity and Error: The time complexity of the *action plan* function depends on the complexity of the heuristic in Algorithm 1. Both heuristics perform a sum depending on the number of vertices. Hence, the complexity of the *action plan* function is $O(|V| \cdot |U|^2)$, i.e., polynomial.

Although polynomial complexity is much more appealing than the $O(2^{|U|})$ complexity of the exhaustive search, the error must be considered. Experiments were conducted to evaluate the approximation error of our heuristics. The graph structures of *AISPlot* and *WifiLounge* were used, randomly altering the untestable vertices, and the preparation cost information (chosen according to a Pareto distribution). The plot in Figure 5 shows the evolution of the relative average approximation error of RTM for our heuristics as a function of the number of untestable operations $|U|$.

The average error incurred by our heuristics is very low w.r.t. the processing time for their calculation. It is notable that the error has an increasing trend, and the pessimistic

and optimistic heuristics are similar. Combining the rankings created by both heuristics, choosing the maximum of either solution, reduces the error while maintaining the low computational complexity.

V. Conclusions and Future Work

We have studied and evaluated runtime testability metric (RTM), and have introduced an approach to improve a system's runtime testability in terms of a testability/cost optimisation problem. This approach is well suited for usage in an interactive tool, enabling system engineers to receive real-time feedback about the system they are integrating and testing at runtime.

Future work will extend the impact cost model with values in the real domain, and add more information about runtime behaviour of a system. As the RTM obtained by our method is a lower bound, to improve its accuracy, the model could be enriched with dynamic information in the form of edge traversal probabilities. Additional empirical evaluation using industrial cases and synthetic systems will be carried out.

Acknowledgement

This work is part of the ESI Poseidon project, partially supported by the Dutch Ministry of Economic Affairs under the BSIK03021 program.

References

[1] D. Brenner, C. Atkinson, O. Hummel, and D. Stoll, "Strategies for the run-time testing of third party web services," in *SOCA '07: Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 114–121.

[2] A. González, É. Piel, H.-G. Gross, and M. Glandrup, "Testing challenges of maritime safety and security systems-of-systems," in *Testing: Academic and Industry Conference - Practice And Research Techniques*. Windsor, United Kingdom: IEEE Computer Society, Aug. 2008, pp. 35–39.

[3] 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.

[4] 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.

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

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

[7] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, pp. 159–182, 2002.

[8] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[9] A. M. Smith and G. M. Kapfhammer, "An empirical study of incorporating cost into test suite reduction and prioritization," in *24th Annual ACM Symposium on Applied Computing (SAC'09)*. ACM Press, Mar. 2009, pp. 461–467.

[10] Y. Yu, J. A. Jones, and M. J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," in *International Conference on Software Engineering (ICSE 2008)*, Leipzig, Germany, May 2008, pp. 201–210.

[11] A. Gonzalez-Sanchez, É. Piel, and H.-G. Gross, "RiTMO: A method for runtime testability measurement and optimisation," in *Quality Software, 9th International Conference on*. Jeju, South Korea: IEEE Reliability Society, Aug. 2009.

[12] A. Gonzalez-Sanchez, É. Piel, H.-G. Gross, and A. J. van Gemund, "Minimising the preparation cost of runtime testing based on testability metrics," in *34th IEEE Software and Applications Conference*, Seoul, South Korea, Jul. 2010, (to appear).

[13] "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

[14] 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.

[15] 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.

[16] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," in *International Symposium on Software Testing and Analysis (ISSTA 2008)*, Seattle, Washington, Jul. 2008, pp. 189–200.

[17] L. Briand and D. Pfahl, "Using simulation for assessing the real impact of test coverage on defect coverage," in *Proceedings of the 10th International Symposium on Software Reliability Engineering*, 1999, pp. 148–157.

[18] X. Cai and M. R. Lyu, "Software reliability modeling with test coverage: Experimentation and measurement with a fault-tolerant software project," in *Proceedings of the 18th IEEE International Symposium on Software Reliability*. Washington, DC, USA: IEEE Computer, 2007, pp. 17–26.

[19] M. A. Vouk, "Using reliability models during testing with non-operational profiles," in *Proceedings of the 2nd Bellcore/Purdue workshop on issues in Software Reliability Estimation*, 1992, pp. 103–111.

[20] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, R. Miller and J. Thatcher, Eds. Plenum Press, 1972, pp. 85–103.

TUD-SERG-2010-025
ISSN 1872-5392