A Runtime Testability Metric for 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 Runtime Testability Metric (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. We perform a theoretical and empirical validation of RTM, showing that RTM is indeed a valid, and reasonably accurate measurement with ratio scale. 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 service-oriented systems for which the sub-components are not even known a priori [2], [3].

Runtime testing [4] 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*.

Figure 1 depicts this fundamental difference between traditional integration testing and runtime testing. On the left-hand side, a traditional off-line testing method is used, where a copy of the system is created, the reconfiguration is planned, tested separately, and once the testing has finished the changes are applied to the production system. On the right-hand side, a runtime testing process where the planning and testing phases are executed over the production system.

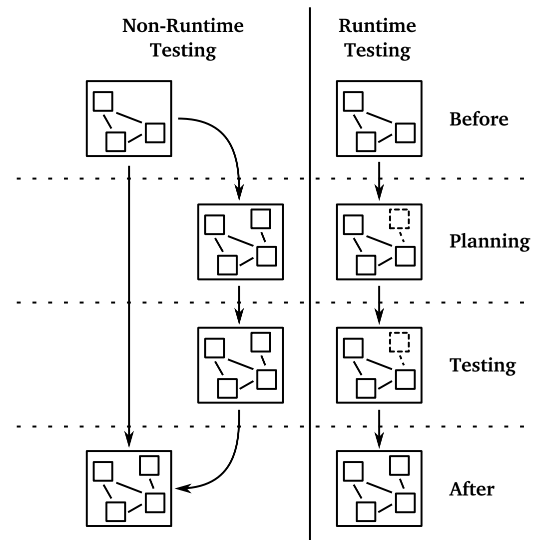

Figure 1. Non-runtime vs. runtime testing

Testability enhancement techniques have been proposed either to make a system less prone to hiding faults [5], [6], [7], or to select the test cases that are more likely to uncover faults with the lowest cost [8], [9], [10], [11]. However, they are not suited for the specific challenges posed by runtime testing, especially the cost that the impact tests will cause on the running system, which determines the viability of runtime testing, is not taken into account by those methods. Features of the system that need tests and whose impact cost is too high will have to be left untested, increasing the probability of leaving uncovered faults. Knowledge of the impact that runtime tests will have on the system will allow engineers to select and implement the appropriate needed measures to avoid interference with the system, or with its environment. As more features can be runtime tested, the probability of uncovering integration faults in the system increases.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

This paper evaluates the Runtime Testability Metric (RTM) introduced in our earlier work [12], [13]. 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 two main contributions of the paper are (1) the measurement-theoretical characterisation of RTM and its empirical validation, and (2) and evaluation of this metric on two industrial systems. In addition, scalable algorithm is introduced to calculate the near-optimal *action plan*, which list by effectiveness the operations that must become testable to improve the RTM.

The paper is structured as follows. In Section II runtime testability is defined. In Section III its theoretical characterisation is performed. An empirical validation of the metric is presented in Section IV. Section V evaluates the RTM on two example cases. In Section VI, an approximate, scalable algorithm is presented for the calculation of the action plan. Section VII describes an implementation of our metric into a component framework. Finally, Section IX 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 [14], 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 [6], [7] for traditional testing. A measurement for (2) will provide information about the test cases that are going to be performed, as proposed in [8], [9], [10]. 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* [15]. 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. Our approach consists in performing an analysis of which features of the system present test sensitivity, prior to the application of isolation measures.

Moreover, testability can also be affected by the design and code quality, which can be measured in terms of robustness, maintainability, flexibility... Here we do not take into account these additional factors and concentrate on the behavioural, and most prevalent, factor for runtime testing: test interference.

The generic aspect of RTM allows engineers to tailor it to their specific needs, applying it to any abstraction of the system for which a coverage criterion can be defined. For example, at a high granularity level, coverage of function points (as defined in the system's functional requirements) can be used. At a lower granularity level, coverage of the component's state machines can be used, for example for *all-states* or *all-transitions* coverage.

In the following, we will precisely define RTM in the context of component-based systems.

A. Model of the System

Component-based systems are formed by components bound together by their service interfaces, which can be either provided (the component offers the service), or required (the component needs other components to provide the service). During a test, any service of a component can be invoked, and the impact that test invocation will have on the running system or its environment is represented as cost. This cost can come from multiple sources (computational cost, time or money, among others).

Operations whose impact (cost) is prohibitive, are designated as untestable. This means that a substantial additional investment has to be made to render that particular operation in the component runtime testable.

In this paper we will abstract from the process of identifying the cost sources, and we will assume that all operations have already been flagged as testable or untestable. In reality, this information is derived from an analysis of the system design and it environment. This latter analysis is performed by the system engineers, who have the proper domain-specific knowledge. Future research will address the issue of deriving this cost information, and of deciding whether a certain impact cost is acceptable or not.

In order to apply RTM, the system is modelled through a Component Interaction Graph (CIG) [16]. A CIG is defined as a directed graph with weighted vertices, $CIG = \langle V, V_0, E, c \rangle$, where

- $V \equiv V_P \cup V_R$: vertices in the graph, formed by the union of the sets of provided and required operations by the components' interfaces.
- $V_0 \subseteq V$: input operations to the system, i.e., operations directly accessible to test scripts.
- $E \subseteq V \times V$: edges in the graph, representing dependencies between operations in the system. E.g., if $(v_1, v_2) \in E$, v_1 depends on v_2.
- $c : V \to \mathbb{R}^+$: function that maps a specific operation to the preparation cost that is going to be optimised.

Each vertex $v_i \in V$ is annotated with a testability flag τ_i, meaning whether the cost of traversing such vertex (i.e.,

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

invoking that service) when performing runtime testing is prohibitive or not, as follows:

$$\tau_i = \begin{cases} 1 & \text{if the vertex can be traversed} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Edge information from within a component can be obtained either by static analysis of the source code, or by providing state or sequence models [16]. Inter-component edges can be derived from the runtime connections between the components. In the case no information is available for a certain vertex, a conservative approach should be taken, assigning $\tau_i = 0$.

Example CIGs are shown in Fig. 6 and Fig. 7. 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).

B. 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: assume that all the vertices reachable from v_i, which we will denote as P_{v_i} (predecessors set), can be affected.

Following our assumptions, the total preparation cost needed to involve an operation in a runtime test, taking into account the individual preparation costs of all the operations it depends on, is defined as

$$C(v_i) = \sum_{v_j \in S_{v_i}} c(v_j) \quad (2)$$

where v_i and v_j are operations, and S_{v_i} is the set of successors of vertex v_i, i.e., all the vertices reachable from v_i including v_i itself.

Not all operations can be directly tested, only a subset of possible input vertices V_0 can be reached directly. Other operations are reached indirectly, via a sequence of operations, which necessarily starts with an operation in V_0. We model this by only counting operations that can be reached from a testable input vertex $v_0 \in V_0$, i.e., that $v_i \in S_{v_0}$ and whose $C(v_0) = 0$. RTM can be then defined as

$$\begin{aligned} RTM = &|\{v_i \in V : C(v_i) = 0 \wedge \\ &\exists v_0 \in V_0 : v_i \in S_{v_0} \wedge C(v_0) = 0\}| \end{aligned} \quad (3)$$

This value can be divided by $|V|$ in order to obtain a normalised metric $rRTM$ that one can use to compare the runtime testabilities of systems with different number of vertices. However, such application has important implications on the theoretical requirements on the metric, as we will observe in Section III.

C. 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. Some of those costs will be very small because they correspond to trivial fixes. However, there can be extremely high costs that will make providing a fix for that specific component prohibitive. For example, a test of an update of the software of a ship that can only be performed at the shipyard has a huge cost, because the ship has to completely abandon its normal mission to return to dry dock. 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. Theoretical Validation

In this section, we establish the characteristics of the RTM measurement from a measurement-theoretical point of view. It allows us to identify what statements and mathematical operations involving the metric and the systems it measures are meaningful and consistent. We will concentrate (1) on RTM's fundamental properties, and (2) on its type of scale.

A. Fundamental Properties

In this section, we will study the properties required for any measurement. These properties determine whether RTM fulfils the minimal requirements of any measurement, i.e., whether it actually creates a mapping between the desired empirical property and the characteristics of the system, that can be used to classify and compare systems. The properties were described by Shepperd and Ince in [17] through an axiomatic approach.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Axiom 1: It must be possible to describe the rules governing the measurement.

This is satisfied by the formal definition of RTM and the CIG.

Axiom 2: The measure must generate at least two equivalence classes.

$$\exists p, q \in CIG : RTM(p) \neq RTM(q)$$

In Figure 2 are assigned two different RTM values, therefore proving this axiom. A node marked with a $\times$ represents an operation where $c(v) > 0$.

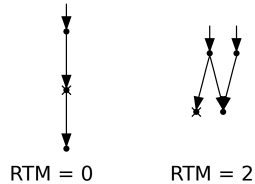

Figure 2. Two equivalence classes for RTM

Axiom 3: An equality relation is required.

This axiom is satisfied given that our measurement is based on natural numbers, for which an equality relation is defined.

Axiom 4: There must exist two or more structures that will be assigned the same equivalence class.

$$\exists p, q \in CIG : RTM(p) = RTM(q)$$

Figure 3 shows a collection of systems belonging to the same equivalence class, proving this axiom.

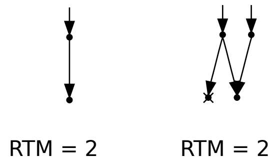

Figure 3. Equivalent CIGs for RTM

Axiom 5: The metric must preserve the order created by the empirical property it intends to measure. This axiom is also known as the Representation Theorem.

$$\forall p, q \in CIG : p \underset{rt}{\succeq} q \Leftrightarrow RTM(p) \geq RTM(q)$$

where $\underset{rt}{\succeq}$ represents the empirical relation 'more runtime testable than'.

This last axiom means that for any two systems, the ordering produced by the empirical property "runtime testability" has to be preserved by RTM. It is possible to find systems for which this axiom does not hold for RTM, because of the assumptions that had to be made (see Section II-A). However, we can empirically assess the effect of these assumptions on the consistency and accuracy of RTM. An empirical study about the accuracy of RTM is presented in Section IV.

B. Type of Scale

The theoretical characterisation of the metric's scale type (i.e., ordinal, interval, ratio, absolute) determines which mathematical and statistical operations are meaningful. This is important, because certain optimisation algorithms require specific mathematical operations that might not be meaningful for RTM, for example for defining heuristics as we will see in Section VI.

Assuming RTM satisfies Axiom 5, i.e., it preserves the empirical ordering of runtime testability, then, by definition, RTM defines a homomorphism from runtime testability to the Natural numbers. Therefore, by the ordered nature of Natural numbers, we can assert that RTM can be used as an *ordinal scale* of measurement. In practice this is true only for systems in which our assumptions about control flow and dependencies hold.

In order to be able to use RTM as a ratio scale measurement, in addition to the requirements for the ordinal type of scale being satisfied, a concatenation operation with an additive combination rule [18] must exist. A meaningful concatenation operation is creating the union of both systems by disjoint union of their CIG models. This operation, $\cup : CIG \times CIG \rightarrow CIG$, can be defined as $A \cup B = \langle V, V_0, E, c \rangle$, where

- $V \equiv V_A \cup V_B$
- $V_0 \equiv V_{0A} \cup V_{0B}$
- $E \equiv E_A \cup E_B$
- $c(v) = \begin{cases} c_A(v) & \text{if } v \in V_A \\ c_B(v) & \text{if } v \in V_B \end{cases}$

For this concatenation rule, the additive combination rule

$$RTM(A \cup B) = RTM(A) + RTM(B) \tag{4}$$

can be used. Therefore, RTM can be used as a ratio scale with extensive structure (e.g., like mass or length), with respect to the disjoint union operation.

C. Relative Values

If we divide the values of RTM by the total number of operations, $|V|$, we can obtain the relative runtime testability ($rRTM$), to compare systems in relative terms. This transforms the measurement from a count into a ratio. Ratios, as percentages have an absolute scale [18] and cannot be combined additively.

For the runtime testability ratio and disjoint union concatenation operator, we can define the combination rule

$$rRTM(A \cup B) = \alpha \cdot rRTM(A) + (1 - \alpha) \cdot rRTM(B) \tag{5}$$

where $\alpha = \frac{|V_A|}{|V_A| + |V_B|}$.

This combination rule is not additive, in order to state the effect of a combination of two systems we need more information than RTM, namely the size relation α.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

D. Summary and Implications

Because we proved that RTM fulfils the minimal properties of any measurement, RTM can be used to *discriminate* and *equalise* systems. Therefore, the statements '*system A has a different runtime testability than B*', and '*systems A and B have the same runtime testability*', are meaningful. Moreover, as we proved RTM has an ordinal scale type, RTM can be used to *rank* systems. The statement '*system A has more runtime testable operations than B*' becomes meaningful, and this enables us to calculate the median of a sample of systems, and Spearman's rank correlation coefficient.

Furthermore, by proving the ratio scale for RTM, it can also be used to *rate* systems, making the statement '*system A has X times more runtime testable operations than B*' a meaningful one. This allows performing a broad range of statistic operations meaningfully, including mean, variance, and Pearson's correlation coefficient.

RTM can also be used alone to reason about the composition of two systems. Due to its additive combination rule, '*systems A and B composed, will be* $RTM(A) + RTM(B)$ *runtime testable*' is a meaningful statement, provided that A and B are disjoint. This is not true for the relative $rRTM$, as additional information about the relationship between the systems (α, the size relation between the systems) is needed.

In order to support the theory, we are presenting an empirical validation with comparison with other metrics.

IV. Empirical Validation

In this section we conduct a number of experiments in order to empirically determine how accurate RTM is with respect to the empirical property of "runtime testability" (ERT).

A. Experimental Setup

To obtain the value of RTM, vertices are first classified into testable and untestable by means of $C(v)$ (see Eq. 2). Our goal is to assess the influence of the assumptions made when defining RTM, in the number of false positives and false negatives of this classification, and in the final value of RTM.

In order to have a baseline for comparison, the naive approach of just counting directly testable operations was used, defined as:

$$NTES = |\{v_i \in V : c(v_i) = 0\}| \quad (6)$$

We also use a previous proposition of RTM [15], which we name RTM_{old} and is defined as:

$$RTM_{old} = |\{v_i \in V : C(v_i) = 0\}| \quad (7)$$

Two systems were used in the experiment: AISPlot and WifiLounge, which are detailed in Section V. For the experiment, 500 variations of each system with different RTM values were generated by choosing the untestable vertices by randomly sampling in groups of increasing size from 2 to 30 untestable vertices.

The value of ERT to perform the comparison was obtained by creating and executing an exhaustive test suite in terms of vertices and execution paths. The set of operations covered when executing each test case was recorded. If a test case used any untestable operation, none of the operations covered by the test were counted. The test cases covered both systems completely, and redundantly, by exercising every possible path in the CIG from every input operation of the system. This way it was ensured that if an operation was not covered, it was not because a test case was missing, but because there was no test case that could cover it without requiring also an untestable operation.

B. Results

From each system and metric pair in this experiment, we recorded the following data:

- M_{set}: Set of operations classified as testable.
- Cov: Set of operations covered.
- $f_p = (|M_{set} - Cov|)/|M_{set}|$: false positive rate, i.e., operations wrongly classified as testable.
- $f_n = (|Cov - M_{set}|)/|Cov|$: false negative rate, i.e., operations wrongly classified as untestable.
- $\bar{e} = ||M_{set}| - |Cov||$: absolute error between the predicted and empirical runtime testabilities.

	System	f_p	f_n	$\bar{e}$
NTES	AISPlot	0.942	0.000	83.487
	WifiLounge	0.713	0.000	57.093
RTM_{old}	AISPlot	0.882	0.107	15.012
	WifiLounge	0.577	0.079	27.664
RTM	AISPlot	0.411	0.111	2.418
	WifiLounge	0.306	0.128	9.101

Table I
FALSE POSITIVE/NEGATIVE RATE, AND ERROR

Table I shows the rates of false positives and false negatives, along with the absolute error, averaged over 500 runs. The deviation between the predicted testability and the actual covered operations for each sample can be seen in the three plots in Figure 4. The dashed line represents the ideal target. Any point above it, constitutes an overestimation error, and below it, an underestimation error.

NTES: It can be seen that NTES has an extremely high error caused by its high false positive rate (94% and 71%). NTES has no false negatives as it classifies as untestable only the vertices that are directly untestable, disregarding dependencies.

RTM_{old}: By taking control flow dependencies into account, the false positive rate of RTM_{old} is lower than NTES, at the price of introducing a number of false negatives. False negatives appear because in some cases where the control flow does not propagate to all of the operations' dependencies, as we had assumed. Still, because of the assumptions

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

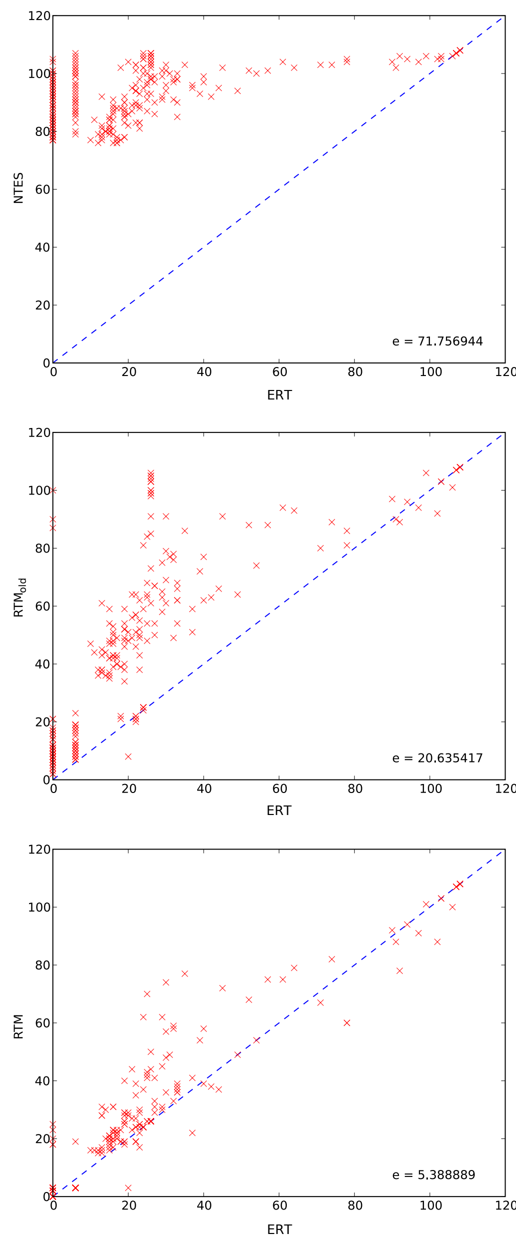

Figure 4. Accuracy of NTES, RTM$_{old}$ and RTM

that test interactions can start in any vertex, and that the test paths are independent, the number of false positives is considerable. The number of input vertices in WifiLounge is proportionally higher than for AISPlot. Hence, the number of false positives caused by this assumption is lower.

RTM: By taking input vertices into account, the amount of overestimation decreases dramatically for both systems. Still, the false positive rate is significant. Therefore, we conclude that assuming that paths are not dependent is not very reasonable and needs to be addressed in future work. The increase of false negatives makes more apparent the consequences of the assumption that control flow is always being transmitted to dependencies. The error caused by this assumption is augmented by the fact that it also applies to the input paths to reach the vertex being considered. Nevertheless, RTM correlates with the ERT, and did provide on average the minimal absolute error compared to the two other metrics.

After the theoretical and empirical validation of RTM, we will present application examples of it.

V. Application Examples

Two studies were performed on two component-based systems: (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. In order to keep the number of untestable vertices tractable, we considered that only operations in components whose state was too complex to duplicate (such as databases), or that caused external interactions (output components) would be considered untestable.

Table II shows the general characteristics for the architectures and graph models of the two systems used in our experiments, including number of components, vertices, and edges of each system.

	AISPlot	WifiLounge
Total components	31	9
Total vertices	86	159
Total edges	108	141

Table II
CHARACTERISTICS OF THE SYSTEMS

A. Example: AISPlot

In the first experiment we used a vessel tracking system taken from our industrial case study. It consists of a component-based system coming from the maritime safety and security domain. The architecture of the AISPlot system is shown in Figure 5. AISPlot is used to track the position of ships sailing a coastal area, detecting and managing potential dangerous situations. Messages are broadcast by ships

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

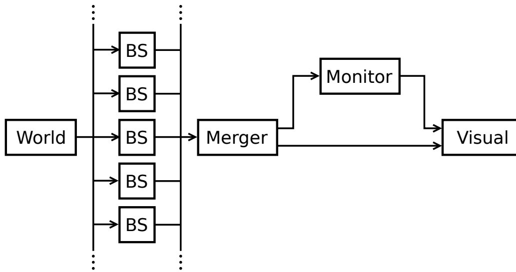

Figure 5. AISPlot Component Architecture

(represented in our experiment by the *World* component), and received by base stations (*BS* component) spread along the coast. Each message is relayed to a *Merger* component removing duplicates coming from different stations. Components interested in receiving status updates of ships, can subscribe to *Merger* to receive notifications. A *Monitor* component scans all messages looking for inconsistencies in the ship data, and another component, *Vis* shows all ships on a screen. Fig. 6 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 III
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 III 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 III 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 (corresponding respectively to operations in the visualiser for: new ships, status updates, disappearing ships, and warnings) are not made runtime testable together. This is caused by the topology of the graph: the four vertices appear at the end of the processing pipeline affecting the predecessor set of almost every vertex together. They must be fixed at once for any testability gain, leading to the jump at cost 4 for AISPlot in Figure 9 ($rRTM$ going from 0.302 to 0.942).

B. Example: WifiLounge

In a second experiment we diagnosed the runtime testability of a wireless hotspot at an airport lounge [19]. The component architecture of the system is depicted in Figure 8. Clients authenticate themselves as either business class passengers, loyalty program members, or prepaid service clients.

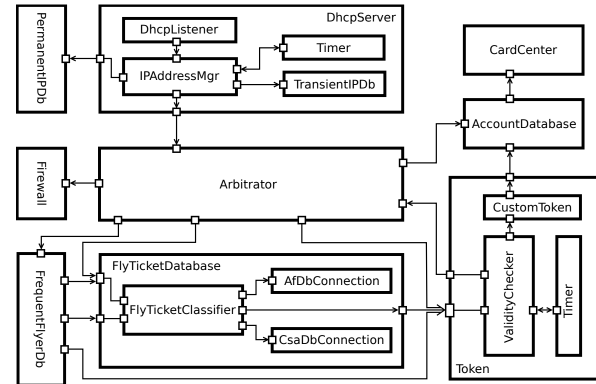

Figure 8. Wifi Lounge Component Architecture

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. 7 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 IV, and shown in Fig. 9 (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.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

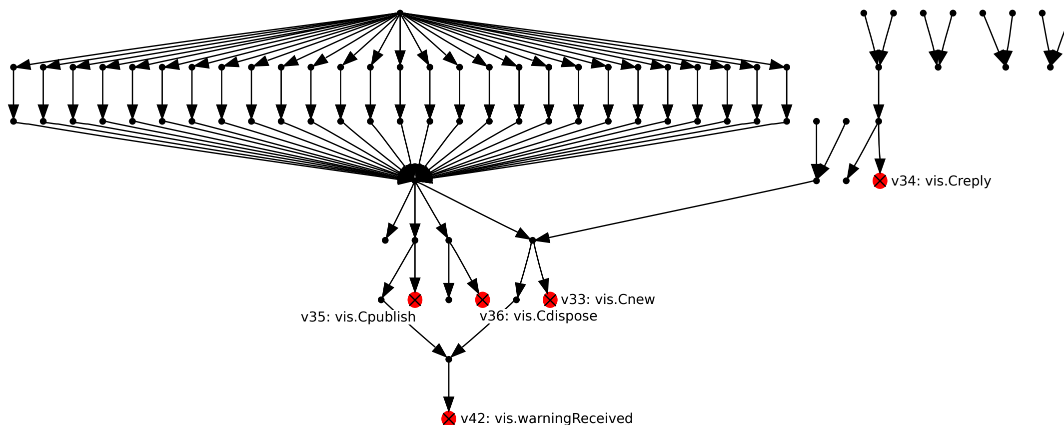

Figure 6. AISPlot Component Interaction Graph

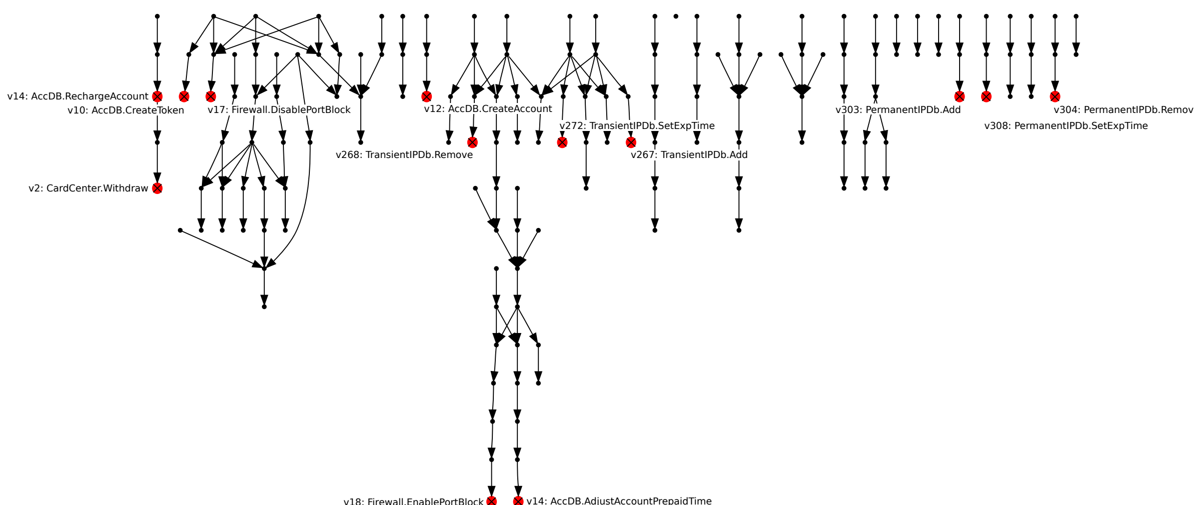

Figure 7. WifiLounge Component Interaction Graph

C. 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 estimate, and we expect improvement by adding dynamic runtime information in the future, e.g., applying [20]. 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. In future work, the value will be refined by providing dynamic information in the form of traversal probabilities, as proposed in the PPDG model presented in [20]. The design of the components on both systems was analysed in an effort to shed light on this issue. For instance, as we have seen in Section IV, 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 [21], previous studies have shown a beneficial effect of test coverage on reliability [22], [23].

VI. Testability Optimisation

RTM analysis and action planning corresponds to the Knapsack problem [24], an NP-hard binary integer programming problem, which can be formulated as

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

with b = maximum budget available, and x_j = decision of including vertex v_j in the action plan. In this section,

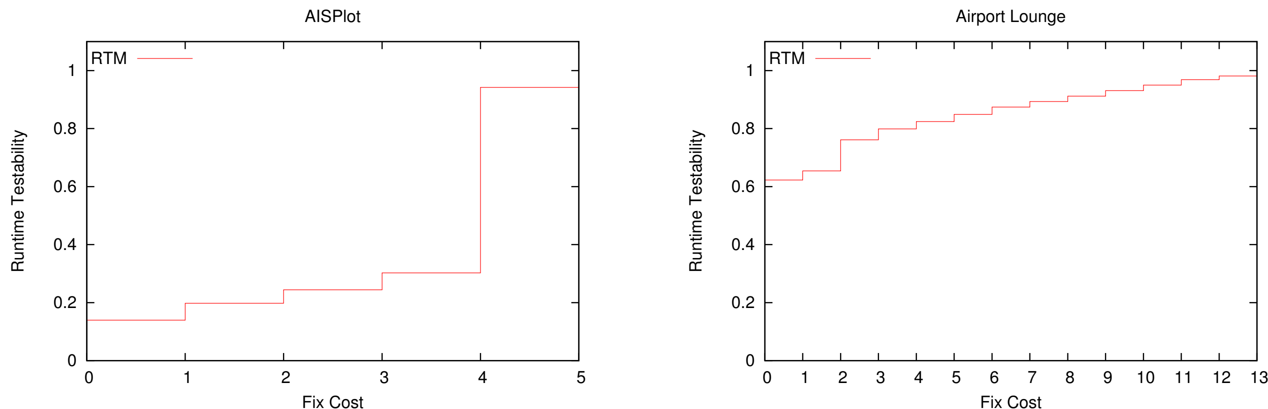

Figure 9. Optimal improvement of RTM vs. Fix cost

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 IV
TESTABILITY ANALYSIS OF AIRPORT LOUNGE

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
```

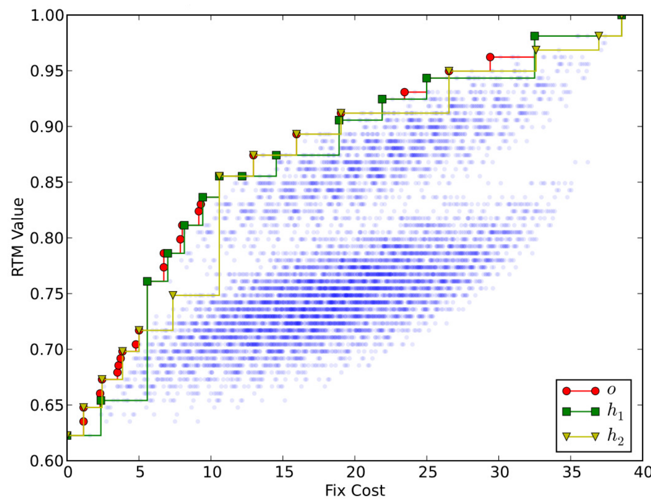

Figure 10. Optimal and heuristic RTM optimisations

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. 10. The dot-clouds show the structure and distribution of all the possible solutions for the vertex and context-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 that 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 10, where the upper cloud corresponds to all the solutions that 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.

A. 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:

$$h_{pessimistic}(v_i) = \frac{1}{c_i}(RTM_{v_i} - RTM) \tag{8}$$

with RTM_{v_i} = RTM after the cost for vertex v_i was spent. Given the pessimistic nature of this heuristic, 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 (cf Section III).

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

$$h_{optimistic}(v_i) = \frac{1}{c_i}|\{v_j \mid v_i \in S_{v_j}\}| \tag{9}$$

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. Although this heuristic ignores uncoverable elements that depend on multiple vertices of U, if two vertices appear together in many P_i sets, their ranks will be similar and will be chosen one after the other.

Fig. 10 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 10).

B. Computational Complexity and Error

The time complexity of the *action plan* function depends on the complexity of the heuristic in Algorithm 1. As in each pass there is one less vertex in U, the H function is evaluated $|U|, |U| - 1, \ldots, 1$ times while searching for the maximum. In total, it is evaluated $\frac{|U|^2}{2}$ times. 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 approximation 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 11 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 optimal solution function is obtained by exhaustive search.

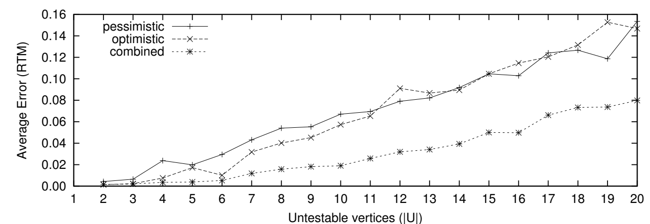

Figure 11. Performance of the approximate algorithms vs. the optimal

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. This can be seen in the *combined* error plot in Figure 11.

VII. Implementation

In order to further validate the applicability of the RTM in software projects, we have integrated the measurement of this metric into our component framework Atlas[1]. For a component-based system, in order to build the CIG, the

[1] http://swerl.tudelft.nl/bin/view/Main/Atlas

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

graph from which the metric is computed, three types of information are necessary:

- The external connections between components,
- The internal connections inside a component,
- The list of test-sensitive operations.

Within the Atlas framework, the connection between component interfaces is explicit, written in the architecture description file by the developer. In component frameworks where these connections are implicit and created dynamically, they can be still be determined at runtime automatically, as in such infrastructure the framework is always in charge of the connection of the components. Per definition, a component is seen by the framework as a black-box, and therefore the information on the internal connections cannot be directly determined. In our case, we have extended the architecture declaration language to permit encoding such information. The encoding is a simple list of tuples indicating which input interface might call which output interface. Nevertheless, this doesn't mean this cannot be automated. For instance, for the generation of this information for the two example systems presented previously in Section V, static analysis was used to discover most of the connections. A quick review by the developer is sufficient to correct the few missing connections. Finally, the information concerning the test sensitivity is the most delicate one to obtain. It is best to consider by default every operation, which is in this context an input interface, as test-sensitive. Then a review by the developer is necessary to define one-by-one which of the operation cannot cause any interference when tested at runtime. In our case, the architecture description language was extended to contain this information provided by the developer.

Figure 12 shows a screenshot of our framework after computation of the RTM. The user interface shows the components in the system on the left. At the request of the user, the CIG is displayed in the right pane. On the very right side, stands the rRTM for the system.

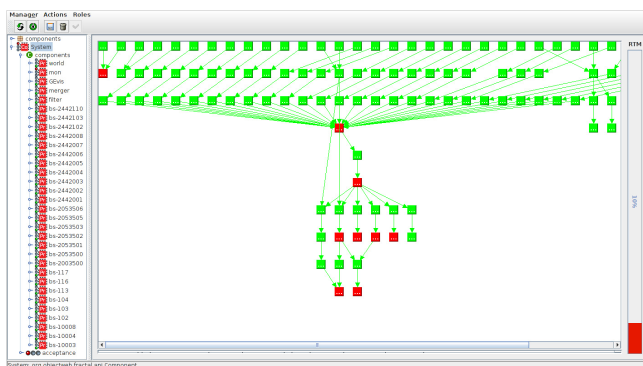

Figure 12. User interface of our component framework displaying the CIG and rRTM of a system.

VIII. Related work

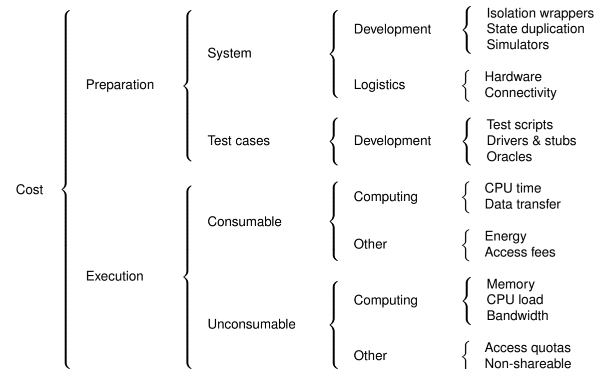

Figure 13. Taxonomy of Testing Costs

A number of research approaches have addressed testability from different angles. The cost of testing is actually complex and decomposed into multiple different types of costs.

Figure 13 presents a taxonomy of testing costs. On the upper branch of the diagram, costs associated with the preparation can be found. These costs correspond to testability improvement efforts which are performed before the tests are carried out. Test interferences are critical because, in the worst case, runtime tests will affect the system's environment in ways that are unexpected and difficult to control, or impossible to recover from, e.g., firing a missile while testing part of a combat system. Therefore the preparation costs include, for example, the adaptation of the system to support isolation and state duplication, or the development of simulators for some components. Preparation costs also include logistic provisioning, such as increasing the capacity of the system by using hardware capable of handling the operational and testing loads simultaneously. Of course, the development of the tests themselves, under the form of test cases, is also a large contribution to the preparation costs.

On the lower branch of the diagram costs associated to the execution of the test case on the system under test. Consumable costs are used up by executing tests, the most traditional example being time. Non-consumable costs, on the other hand, are "on loan" during the execution of the test, and returned when the test finishes. For example, CPU load is bounded (to 100%) and will not only depend on the quantity of tests applied but only on their "concentration": the CPU load might decreased simply by spreading tests over a longer period of time. These non-consumable costs are the one that can affect the non-functional requirements of the system when employing runtime testing.

Runtime testability has not been considered in depth so far. In such context, the system preparation cost for testing has a strong importance. To the best of our knowledge, our paper is the first to (1) define a measurement (RTM) in a

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

way that is conductive to an estimation of runtime testability, (2) consider testability improvement planning in terms of a testability/cost optimisation problem, and (3) to present a near-optimal, low-cost heuristic algorithm to compute the testability optimization plan.

Traditionally, test cost minimisation has considered only execution cost. To reduce the consumable costs test effort minimisation algorithms have been proposed, both for test time and coverage [8], [9], [10], [11] or test time and reliability [25], [26]. Test sensitivity and isolation, are introduced by Brenner et al. [4] to reduce the non-consumable costs, however no mention to nor relation with the concept of runtime testability were presented. On the same topic, Suliman et al. [27] discuss several test execution and sensitivity scenarios, for which different isolation strategies are advised. These two works form the base for our initial approach to runtime testability, presented in [12], [13], and extended and more thoroughly evaluated in this paper, which is an extended version of [1]. The factors that affect runtime testability cross-cut those in Binder's Testability [28] model, as well as those in Gao's component-based adaptation [29].

Other testability-related approaches have focused on modeling statistically which characteristics of the source code of the system are more prone to uncovering faults [6], [7] for amplifying reliability information [5], [30]. Preparation cost, understood as the compilation time overhead caused by the number of dependencies needed to test any other component, was addressed in [31]. They proposes a measurement of testability from the point of view the static structure of the system, to assess the maintainability of the system. Our approach is similar in that runtime testability is influenced by the structure of the system under consideration.

IX. Conclusions and Future Work

The amount of runtime testing that can be performed on a system is limited by the characteristics of the system, its components, and the test cases themselves.

In this paper, we have studied RTM, a cost-based measurement for the runtime testability of a component-based system, which provides valuable information to test engineers about the system, independently of the actual test cases that will be run. RTM has been validated from a theoretical point of view that it conforms to the notion of measurement, and that it can be used to *rate* systems. An empirical validation has shown that it provides with relatively good accuracy prediction of the actual runtime testability. Furthermore, we have introduced an approach to the improvement of the system's runtime testability in terms of a testability/cost optimisation problem, which allows system engineers to elaborate an action plan to direct the implementation of test isolation techniques with the goal of increasing the runtime testability of the system in an optimal way. We have provided a low-cost approximation algorithm which computes near-optimal improvement plans, reducing significantly the computation time. This algorithm 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 towards extending the impact cost model with values in the real domain instead of a boolean flag will be carried out. This work could benefit from the test cost estimation and reduction techniques cited in the related work, and be used to devise a runtime-test generation and prioritisation algorithm that attempts to achieve the maximum coverage with the minimum impact for the system. Moreover, because the RTM as obtained by our method is a lower bound, further work will encompass an effort to improve its accuracy, by enriching the model with dynamic information in the form of edge traversal probabilities. Finally, additional empirical evaluation using industrial cases and synthetic systems will be carried out in order to explore further the relationship between RTM and defect coverage and reliability.

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] A. Gonzalez-Sanchez, É. Piel, H.-G. Gross, and A. van Gemund, "Runtime testability in dynamic high-availability component-based systems," in *The Second International Conference on Advances in System Testing and Validation Lifecycle*, Nice, France, Aug. 2010.

[2] 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.

[3] 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.

[4] 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.

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

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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 Computer Software and Applications Conference*, Seoul, South Korea, Jul. 2010.

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

[15] 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.

[16] 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.

[17] M. Shepperd and D. Ince, *Derivation and Validation of Software Metrics*. Oxford University Press, 1993.

[18] H. Zuse, *A Framework of software measurement*. Hawthorne, NJ, USA: Walter de Gruyter & Co., 1997.

[19] T. Bures. (2011, Jun.) Fractal BPC demo. [Online]. Available: http://fractal.ow2.org/fractalbpc/index.html

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] C. Y. Huang and M. R. Lyu, "Optimal release time for software systems considering cost, testing-effort, and test efficiency," *IEEE Transactions on Reliability*, vol. 54, no. 4, pp. 583–591, 2005.

[26] K. Okumoto and A. Goel, "Optimum release time for software systems based on reliability and cost criteria," *Journal of Systems and Software*, vol. 1, pp. 315–318, 1980.

[27] D. Suliman, B. Paech, L. Borner, C. Atkinson, D. Brenner, M. Merdes, and R. Malaka, "The MORABIT approach to runtime component testing," in *30th Annual International Computer Software and Applications Conference*, Sep. 2006, pp. 171–176.

[28] R. V. Binder, "Design for testability in object-oriented systems," *Communications of the ACM*, vol. 37, no. 9, pp. 87–101, 1994.

[29] J. Gao and M.-C. Shih, "A component testability model for verification and measurement," in *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference*, vol. 2. Washington, DC, USA: IEEE Computer Society, 2005, pp. 211–218.

[30] 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.

[31] 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.

2011, © Copyright by authors, Published under agreement with IARIA - www.iaria.org