Runtime Testability in Dynamic Highly Available Component-based Systems

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

Report TUD-SERG-2009-009
Runtime Testability in Dynamic Highly Available Component-based Systems

Alberto González  Éric Piel  Hans-Gerhard Gross  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 paper defines a cost-based measurement for the estimation of runtime testability, based on a model of the system, independent of the test cases and focused exclusively on test-relevant features of the system. This measurement 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. A low-cost heuristic algorithm is introduced, which allows the computation of near-optimal testability optimisation plans in polynomial time, allowing the application of our approach to systems of realistic size.

Two testability analysis are performed on two different component based systems, showing how our approach is used to identify runtime testability problems on a system, and to provide an improvement plan to address their runtime-related testability problems.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: testing and debugging—test execution, test coverage of specifications

Keywords
Runtime testability, runtime testing, measurement, component-based.

1. INTRODUCTION
Integration and system-level testing of complex, highly available systems is becoming increasingly difficult and costly to perform in a development-time testing environment, since the systems cannot be duplicated easily, nor can their usage context. Examples of such systems are systems with a very high availability requirement, which cannot be put off-line to perform maintenance operations (such as air traffic control systems, systems of the emergency units, banking applications, etc.). Dynamic Systems of Systems, and Service Oriented Architectures, in which the components that will form the system are not available, or even known beforehand in some cases, pose also considerable runtime integration and testing challenges [5, 15] to engineers and researchers.

Runtime testing (a testing method that has to be carried out in the final execution environment of a system [6]) is emerging as the solution for the validation and acceptance testing for such systems, where traditional development-time integration and validation testing cannot be performed. A prerequisite for runtime testing 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.

Testability is commonly referred to as the relative ease and expense of revealing software faults. Testability enhancement techniques have been proposed which try to make system less prone to hide faults [3, 11, 16, 18, 22], or which intelligently select the test cases that are more likely to uncover faults with the lowest cost [10, 19, 20, 25]. However, they are not suited for the specific challenges posed by runtime testing, especially the cost which 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 which need tests 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. Therefore increasing the probability of uncovering integration faults in the system by allowing more features to be runtime tested.

In this paper, we propose to use an estimated value of runtime testability based on the impact cost of runtime tests on the system, along with the cost of the remedial measures needed to reduce the impact, to develop an action plan for
the improvement of the system’s runtime testability. 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 cost of the remedial measures that have to be applied.

The main contributions of this paper are:

- A model-based approach to estimate the runtime testability of a system, based exclusively on characteristics of the test requirements and independent of the characteristics of the test cases.
- A cost-driven optimisation method, yielding an action plan to enhance the runtime testability of the system, based on the trade-off between the cost of test interferences, and the cost of interference isolation measures.
- Two examples applied on two component-based systems, one coming from the maritime safety and security domain, and one used as an airport lounge internet gateway.
- A scalable approximation algorithm to generate the action plan allowing our approach to be applied in terms of an engineering tool to systems of realistic size.

The paper is structured as follows. In Section 2 runtime testability is defined, along with a measurement framework for the definition of architecture-based runtime testability measurements. Section 3 presents our particular, model-based coverage measurement, and the optimisation algorithm. Section 4 applies our analysis and optimisation approach to two example cases. In Section 5, an approximate, scalable algorithm is presented for the calculation of the action plan. In Section 6, background and related work on runtime testing and testability is discussed and compared to our research. Finally, Section 7 presents our conclusions and plans for future research.

2. RUNTIME TESTABILITY

One of the major challenges of runtime testing is the interference that it will cause on the system’s state or resource availability. In the worst case, runtime tests will affect the system’s environment in critical ways that are difficult to control or impossible to recover from, e.g., firing a missile while testing part of a combat system.

The fact that there is interference through runtime testing requires an indicator of how resilient the system is with respect to runtime testing, or, in other words, to which extent can the system be tested without affecting the system or its environment. The standard definition of testability by the IEEE [1] can be rephrased to reflect these requirements, as follows:

**Definition 1.** Runtime Testability is (1) the degree 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 definition considers both (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 of the system independent of the nature of the runtime tests that may be performed, as it is proposed in [11, 22] for traditional testing. On the other hand, a measurement for (2) will provide information about the concrete test cases that are going to be performed, as proposed in [10, 19, 20]. In this paper we will specifically concentrate on the system-centric aspect of runtime testability.

Runtime testability is significantly influenced by two main characteristics of the system: test sensitivity, and test isolation [14]. Test sensitivity characterises which fraction of the 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, resource limitations, etc. Conversely, test isolation techniques are applied by engineers to specific components to counter the test sensitivity, e.g., state duplication or component cloning, usage of simulators, resource monitoring, etc.

Ultimately, both the impact of the disturbances to the running system, and the implementation of isolation measures can be represented as a cost. All the sensitivity factors which impede runtime testing will prevent test engineers from assessing a certain feature or requirement, if their sensitivity cost is too large, increasing the probability of leaving uncovered faults. In order to runtime test those features, extra cost has to be spent in addressing some of those sensitivity factors.

A numerical measurement for the runtime testability of a system can be defined in terms of what fraction of the features of the system can 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 estimation can be used to indicate insufficient testing of some features of the system due to prohibitive costs during runtime testing, independent of the actual test cases, and before any test is actually run.

We define the Runtime Testability Measurement (RTM), as the quotient between the number of features of the system which can be runtime tested with an acceptable cost, and the total number of features, e.g., as determined by a test adequacy criterion:

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

(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 at an acceptable cost.

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 section, we will instantiate the above generic definition of RTM to component-based systems.

3. RUNTIME TESTABILITY OF COMPONENT-BASED SYSTEMS

In order to estimate the runtime testability of a component-based system, we will use a static graph dependency model, annotated with runtime testability information. This model is applied to estimate the (runtime impact) cost of in-
voking a specific feature of the system (e.g. a service or a specific interaction path), independently of the test cases used for it.

We choose to use a runtime dependency model as base abstraction for the definition of RTM for our experiments. On the one hand, it is detailed enough to identify key runtime testability issues to the individual operations of components that cause them. On the other hand, it is simple enough so that its derivation from the component’s design and the system’s runtime architecture is easy, and its computation is a tractable problem.

3.1 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.

The system is modelled using a directed component dependency graph known as Component Interaction Graph (CIG) [24]. A CIG is defined as a directed graph $\text{CIG} = (V,E)$. 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 a method of a provided or required interface of a certain component. Edges in $E$ account for two situations: (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).

Each vertex $v_i \in V$ is annotated with a testability flag $\tau_i$, meaning whether the cost of traversing such vertex (i.e., invoking that service) when performing runtime testing is prohibitive or not, as follows:

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

(2)

Edge information inside a component can be obtained either by static analysis of the component’s source code, or by providing some kind of model, such as state or sequence diagrams [24]. 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 $\tau_i = 0$.

3.2 Coverage Criteria

A number of architectural test coverage adequacy criteria have been defined based on CIG or other similar representations [17, 24]. We will measure the runtime testability of the system based on two adequacy criteria proposed in [24]: the all-vertices, and the all-context-dependence criteria.

The all-vertices criterion requires executing each method in all the provided and required interfaces of the components, which translates to traversing each vertex $v_i \in V$ of the model at least once.

On the other hand, the all-context-dependence criterion requires testing invocations of vertices between every possible context. A vertex $v_i$ is context-dependent on $v_j$ if there is an invocation sequence from $v_i$ that reaches $v_j$. For each of these dependences, all the possible paths $(v_i, v_{i+1}, \ldots, v_j)$ are considered viable, and need to be tested.

3.3 Estimation of RTM

To estimate the runtime testability of the system, we will estimate the impact cost of covering each of the context dependences or vertices in the graph, flagging as untestable those whose cost is prohibitive, in the same way as for individual operations.

We will not look at the concrete penalty of actual test cases, but at the possible cost of any test case that tries to cover each element. Because of the lack of control flow information of the CIG model (it is a static dependency model of the system), assumptions have to be made on the actual behaviour of test cases. In future work, the model will be enriched with additional dynamic information to relax these assumptions.

First, we will assume that the interaction starts directly at the vertex (for all-vertices coverage) or the first vertex of the path (for all-context-dependences coverage) which we want to cover. Second, because there is no control flow information, we cannot know which edges in the CIG will be traversed by a test case. We cannot assume that there exists a test case that only exercises the element we want to cover. In the worst case, the interaction might propagate through all edges, affecting all reachable vertices. Therefore, we will estimate the worst-case scenario of what could be traversed for each vertex $v_i$, or context dependence path starting at vertex $v_i$: assume that all the vertices reachable from $v_i$, which we will denote as $P_{v_i}$ (predecessors set), will be affected.

For each vertex $v_i$ or path $(v_i, v_j, v_k, \ldots)$ that we would like to cover, we calculate a testability flag $T(v_i)$ as

$$T(v_i) = \bigwedge_{v_j \in P_{v_i}} \tau_j$$

(3)

By considering as testable only those features whose $T(v_i) = 1$, Equation 1 can be rewritten for all-vertices and all-context-dependence coverage, respectively, as

$$RTM_{v} = \frac{|\{v_i \in V \mid T(v_i) = 1\}|}{|V|}$$

(4)

$$RTM_{v-dep} = \frac{|\{(v_{i+1}, v_{i+2}, \ldots) \in CIG \mid T(v_i) = 1\}|}{|\{(v_i, v_{i+1}, v_{i+2}, \ldots) \in CIG\}|}$$

(5)

3.4 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: $c_i$. In the following section we present two examples.

4. APPLICATION EXAMPLES

Two runtime testability case studies were performed on two component-based systems: (1) AISPlot, a system-of-systems taken from a case study in the maritime safety and security domain, and (2) WifiLounge, an airport’s wireless access-point system [8]. The objective of these cases is to show that our measurement can help identify which parts of a system have a prohibitive runtime testing cost, and that the same measurement can help to choose the optimal action points where resources have to be invested, with the goal of improving the system’s runtime testability, and hence, its quality and reliability.

In both experiments, the model of the intra-component CIG edges was obtained by static analysis of the primitive components’ source code. The edges between components were calculated dynamically by inspecting the dependencies at runtime using reflection. The runtime testability flag $\tau_i$ and fix cost information $c_i$ were added 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 which caused external interactions (output components) would be considered untestable.

Table 1 shows the general characteristics for the architectures and graph models of the two systems used in our experiments, including number of components, vertices, edges, and context-dependent paths of each system.

<table>
<thead>
<tr>
<th>AISPlot</th>
<th>WifiLounge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total components</td>
<td>31</td>
</tr>
<tr>
<td>Total vertices</td>
<td>86</td>
</tr>
<tr>
<td>Total edges</td>
<td>108</td>
</tr>
<tr>
<td>Context-dependent paths</td>
<td>1447</td>
</tr>
</tbody>
</table>

Table 1: Characteristics of the systems

4.1 Example System I: 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 1.

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 component) spread along the coast. Each message received by a base station is then relayed to the Merger component, which removes duplicates (some base stations cover overlapping areas). Components interested in receiving status updates of ships, can subscribe to receive notifications. The Monitor component scans all the received messages in search for inconsistencies in the data sent by the ships, in order to spot those which are less reliable and which therefore require more care from the operator. The Visual component draws the position of all ships on a screen in the control centre, and also the warnings generated by the Monitor component.

4.1.1 Testability Analysis

For AISPlot, five operations from the Visual component have testability issues, due to the fact that they will influence the outside world by printing ship positions and warnings from runtime tests on the real screen of the command centre if not isolated properly. Figure 2 depicts the Interaction Graph of AISPlot, with the untestable vertices marked with a larger, crossed-out dot.

As can be seen in the first row of Table 2, the runtime testability of the system is very low. Initially only 14% of the vertices and 1.2% of context dependences can be runtime tested. This extremely poor value of RTM is due to the fact that the architecture of the system is organised as a pipeline, with the Visual component at the end, connecting almost all vertices to the five problematic vertices of the visualiser component.

With the purpose of enhancing the runtime testability of the system, we examined possible solutions, consisting in modifying the problematic operations, by providing interaction isolation. As there are only five vertices responsible for the poor testability, we explored each of the possible combinations of fixes and computed the optimal improvement on RTM for both vertex and context dependence coverage, assuming a uniform cost for all vertices.

Table 2 shows the value of RTM for each of the possible fixes. The $\times$ symbol denotes an operation to which test isolation is applied. The cost column indicates a hypothetical cost of improving the testability of the system, assuming that the cost of addressing a single operation’s testability is uniform and equal to one. The values in the table suggest that there will be no practical gain in testability as long
Figure 2: AISPlot Interaction Graph

<table>
<thead>
<tr>
<th>Cost</th>
<th>Proposed fix</th>
<th>Testability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>v33 v34 v35</td>
<td>0.140</td>
</tr>
<tr>
<td></td>
<td>v36 v42</td>
<td>0.012</td>
</tr>
<tr>
<td>1</td>
<td>×</td>
<td>0.198</td>
</tr>
<tr>
<td>2</td>
<td>×</td>
<td>0.244</td>
</tr>
<tr>
<td>3</td>
<td>×</td>
<td>0.198</td>
</tr>
<tr>
<td>4</td>
<td>×</td>
<td>0.198</td>
</tr>
<tr>
<td>5</td>
<td>× × × × ×</td>
<td>0.942</td>
</tr>
</tbody>
</table>

Table 2: Testability analysis for AISPlot

4.2 Example System II: WifiLounge

In a second experiment we diagnosed the runtime testability of a wireless hotspot at an airport lounge [8]. The component architecture of the system is depicted in Figure 3. Clients authenticate themselves as either business class passengers, loyalty program members, or prepaid service clients.

When a computer connects to the network, the DhcpListener component generates an event indicating the assigned IP address. All communications are blocked by the firewall until the authentication is validated. Passengers of business class are authenticated in the ticket databases of a number of airlines. Passengers from a frequent flier program are authenticated against the program’s database, and the ticket databases to check that they are actually entitled to free access. Passengers using the prepaid method must create an account in the system, linked to a credit card that is used for the payments. Once the authentication has succeeded, the port block in the firewall is disabled so that the client can use the connection. The session ends when the user disconnects, or the authentication token becomes invalid. If the user is using a prepaid account, its remaining prepaid time will be updated.

4.2.1 Testability Analysis

A total of 13 operations are runtime untestable. Operations which modify the state of the AccountDatabase, TransientIpDb and PermanentIpDb components are considered runtime untestable because they act on databases behind the components. The withdraw operation in CardCenter is also not runtime testable because it is operating on a banking system outside our control. Finally, the operations that control the Firewall component are also runtime untestable because this component is a front-end to a hardware element (the network) impossible to duplicate. The interaction graph of the system can be seen in Figure 4 (with the untestable vertices marked with a larger, crossed-out dot).

The initial runtime testability of the system is intermediate (62% of vertices and 41% of context dependences can be covered). This value is far from the extremely low testability of the AISPlot system, because of the differences in the architecture of both systems. Even though there are more runtime-untestable vertices than in the previous case, they are not as dependent on each other as it was the case for AISPlot.

A runtime testability diagnostic and vertex address proposal was calculated for this second system as well. Table 3 shows the initial testability, and the proposed optimal fixes along with the increased testability.
in the right-hand-side plot in Figure 5. In this case, unlike what happened in AISPlot, the proportion of vertices that have to be fixed to get a significant increase in RTM is much lower. This is due to two vertices (v14 and v18) that cause a great deal of the untestability. The rest of the vertices are not so problematic and the value of RTM grows more linearly.

4.3 Discussion of the Experiments

Through these two experiments, we were able to demonstrate the value of RTM for a test engineer who faces a runtime testing situation. By identifying which operations of a system cause inadmissible effects, we are able to predict which features of the integration of the system cannot be tested at runtime. This permits the creation of an action plan to increase the number of testable features through applying isolation techniques. The testability analysis and accompanying action planning is performed before running any test case, therefore saving time and resources.

It is important to note we were using a static model to make estimations on something as dynamic as is the behaviour of test cases. Therefore, the assumptions we made, make the runtime testability value estimated with our method a lower bound. In this respect, a low RTM value may not be so low in practice because of the aforementioned underestimation problem. However, a high value, even if underestimated, is a good indicator that the system is well prepared for runtime testing and the tests will be able to cover a great deal of the system’s 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 [2].

The design of the components on both systems was analysed in an effort to shed light on this issue. On the one hand, the value of RTM for AISPlot is close to the actual runtime testability, because in the practice the processing will always generate an update in the Visual component upon receiving a ship tracking message. Each input will cause either new, publish or dispose to be invoked. Some testability underestimation is caused by the fact that even though warningReceived is not invoked for every input message, we assumed that all inputs would generate a warning. On the other hand, the RTM value for WifiLounge is underestimated to a higher extent because its control flow is more complex, and there are many exclusive branch choices that are lost in the static model. For example, a failed connection path is marked untestable because the graph has an edge to DisablePortBlock and CreateToken even if they are not called if the authentication fails. Furthermore, all disconnection paths will be considered untestable because of the AdjustAccountPrepaidTime operation, even in the case of business class or frequent flier accounts, which never

Table 3: Testability analysis of Airport Lounge

| Cost | v2 | v10 | v12 | v13 | v14 | v17 | v18 | v267 | v268 | v272 | v303 | v304 | v308 | RTM_{N} | RTM_{N-\text{deg}} |
|------|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----------|-----------------|
| 0    |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 99       | 0.623 307 0.412 |
| 1    |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 103      | 0.648 339 0.454 |
| 2    |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 121      | 0.761 618 0.826 |
| 3    |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 127      | 0.648 616 0.869 |
| 4    |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 131      | 0.858 672 0.901 |
| 5    |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 135      | 0.858 672 0.901 |
| 6    |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 139      | 0.874 696 0.933 |
| 7    |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 142      | 0.893 707 0.948 |
| 8    |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 145      | 0.912 713 0.956 |
| 9    |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 147      | 0.925 722 0.968 |
| 10   |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 142      | 0.912 713 0.956 |
| 11   |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 150      | 0.943 728 0.976 |
| 12   |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 163      | 0.962 734 0.984 |
| 13   |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 156      | 0.981 740 0.992 |
| 14   |    |     |     |     |     |     |     |     |     |     |     |     |     |     | 159      | 1.000 746 1.000 |

Figure 4: Wifi Lounge Interaction Graph

Table 3: Testability analysis of Airport Lounge
invoke that operation.

An issue to be considered is the actual relationship between RTM and defect coverage. Even though the relationship between test coverage and defect coverage is not clear [7], previous studies have shown a beneficial effect of test coverage on reliability [9, 13, 23].

5. TESTABILITY OPTIMISATION

The testability analysis and action plan proposal performed on the previous section corresponds to a Binary Integer Programming problem, also known as the Knapsack problem, which is NP-Hard. The top graph in Figure 7 shows the exponential time it takes to perform the calculation of an action plan for both Equation 4 and 5 on our experiment platform, averaged over 25 runs. As can be seen, the time employed in computing the optimal action plan grows exponentially and, therefore, its application to real systems with a large number of vertices is not practical, especially if we want the analysis and action planning to be available in real-time while the system is being deployed or modified.

In this section we present an approach to generate an approximate action plan by using the greedy, heuristic method according to Algorithm 1, where CIG represents the interaction graph of the system, 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

```plaintext
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. The dot clouds in both plots of Figure 6 show the structure and distribution of all the possible solutions for the vertex and context-dependence RTM optimisation problems, respectively, of the WiFiLounge system. On a system where the cost of fixing any vertex is more or less the same, and where all the uncoverable vertices or paths are such because of only one untestable vertex, there would be only one cloud. However, two interesting characteristics of the inputs which affect the structure of the solution space have been identified.

First, on the majority of systems, multiple untestable vertices will participate on the same uncoverable elements (as is the case in both of our case study systems). If a group of untestable vertices participates together in many uncoverabilities, the solution cloud will cast a “shadow” on the RTM axis: any solution that includes those vertices will get a testability boost. Second, vertices with exceptionally high costs 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 boost (due to space concerns this is not shown on the plots). An example of the first characteristic appears in Figure 6, 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 for the vertex and path problems 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.

5.1 Vertex Coverage Heuristics

In order to approximately solve the optimisation problem for vertex coverage we study two rank functions, one that rewards long-term gains, and another one that rewards short-term gains.

Our first proposed heuristic 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 then divided by the cost to penalise expensive nodes over cheaper ones. The heuristic is defined as

$$h_1(v_i) = \frac{1}{c_i} |\{v_j \mid v_i \in P_{v_j}\}|$$

By ignoring the fact that an uncoverable vertex may be caused by more than one untestability, this first heuristic
will take very optimistic decisions on the first passes that
affect the quality of results for proportionally low budgets.

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

Alternatively, we consider a second heuristic that takes
the opposite approach: it ranks higher the vertices that ap-
pear alone (as the only cause) on the highest number of \( P_i \)
sets, i.e. those which provide the highest gain on testability
on the next step. The count is also divided by the cost to
penalise expensive nodes. The heuristic is defined as:

\[
h_2(v_i) = \frac{1}{c_i} |\{ v_j \mid P_{v_j} \cap U \equiv \{ v_i \}\}| \quad (7)
\]

where \( U \) is the set of untestable vertices. Every time a vertex
is selected, it is removed from this set.

Figure 6 shows on the left hand side an example of the
progress of both of the vertex coverage heuristics. The curve
\( o \) represents the optimum. We can see how the \( h_1 \) rank
skips many low cost solutions between 0 and 5 cost units
(it’s curve being much lower than \( o \)), while the \( h_2 \) ranking is
more precise for low costs. However, this can also be seen
as greediness for instant RTM growth, which can make it
to completely miss a very good solution that depends on a
group of related untestable vertices, due to its lack of “long
v

5.2 Path Coverage Heuristics

For the problem of finding an optimal fix based on
\( RTM_{v-dep} \), we adopt the same ideas as for \( RTM_v \), and adapt
it to estimate the number of freed context dependence paths
instead of vertices. The value of the number of freed paths
is further approximated by the size of the \( P_i \) sets of each of
the vertices \( v_j \) that are freed in \( h_1 \), to avoid examining
each of the paths which are in a much greater number than
nodes.

The “optimistic” heuristic is defined as

\[
h_3(v_i) = \frac{1}{c_i} \sum_{v_j \in P_{v_j}} |P_{v_j}| \quad (8)
\]

On the other hand, the value of the “greedy” heuristic is
defined, respectively, as

\[
h_4(v_i) = \frac{1}{c_i} \sum_{v_j \in P_{v_j} \cap U \equiv \{ v_i \}} |P_{v_j}| \quad (9)
\]

where \( U \) is the set of untestable vertices.

The right hand side of Figure 6 shows an example of the
progress of both of the path coverage heuristics applied to
the same case as the vertex coverage plots on the left-hand-
side. The behaviour of \( h_3 \) follows the same as \( h_1 \), skipping
some of the initial, low cost solutions. Conversely, \( h_4 \) follows
the same trend as its vertex coverage sibling, with a very
good initial precision but a worse performance later on.

5.3 Computational Complexity and Error

The time complexity of the FixActionPlan function de-

pends on the time complexity of the heuristic function used
in the inner loop of 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 looking for the maximum. In total, it is evalu-
ated \( \Omega(2^{|U|}) \) times.

As all the heuristic functions perform a sum that depends
on the total number of vertices in the system, the complex-
ity of the FixActionPlan function is \( O(|V| \cdot |U|^2) \), and
therefore polynomial. The upper plot in Figure 7 shows the
experimental difference in complexity between the exact
method on our heuristic approach, plotted in a logarithmic
scale.

Although a polynomial complexity is much more appeal-
ing than the \( O(2^{|U|}) \) complexity of the exhaustive search,
the approximation error has to be taken into account. A
number of experiments were conducted to evaluate the ap-
proximation error of our heuristic method.

A tool to load CIG definitions in XML and to perform a
complete testability study was developed in Python using
the NetworkX\footnote{http://networkx.lanl.gov}, and SciPy\footnote{http://www.scipy.org} libraries. The experiments were run on a 2.6GHz AMD Athlon 64, on Linux.

The graph structure of the two systems in Section 4 was
used, although randomly altering the set of untestable ver-
tices, and the fix cost information of each of them. These

Figure 6: Heuristic RTMv and RTMv-dep analysis
costs were chosen according to a Pareto distribution. The lower plot in Figure 7 shows a comparison of the average approximation error, plotted in a logarithmic scale.

\[
\bar{e} = \frac{1}{m} \int_{0}^{m} (o(b) - a(b)) \, db
\]  

where \(o(b)\) corresponds to the optimal solution function for any budget, \(a(b)\) an approximate solution function using one of the heuristics, and \(m\) the cumulative cost of fixing all vertices.

The average error incurred by our heuristics is very low considering the processing time required for their calculation. It must be noted, however, that the error has an increasing trend which must not be overlooked. With respect to the vertex heuristics, both the errors of \(h_1\) and \(h_2\) are quite similar, whereas in the path heuristics case, the error for \(h_4\) is consistently worse than for \(h_3\). A possible explanation for this difference is the fact that a wrong vertex fix choice affects a greater number of untastable paths than vertices, and therefore, the error gets amplified.

Combining the rankings created by both the long and short term heuristics, by choosing the maximum of either solution, yields the best of both heuristics (i.e., the long term insight of \(h_1\) or \(h_3\), and the initial precision of \(h_2\) or \(h_4\)) while maintaining the low computational complexity. This can be seen in the \(e_{1-2}\) and \(e_{3-4}\) error plots in Figure 7.

6. RELATED WORK

A number of research approaches have addressed testability from different angles. However, runtime testability has not been considered in depth so far. To the best of our knowledge, our paper is the first to (1) define a measurement (RTM) in a 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 optimisation plan.

Test sensitivity and isolation are introduced by Brecker et al. [6], however no mention to nor relation with the concept of runtime testability were presented. On the same topic, Suliman et al. [21] 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 [14], and extended in this paper. The factors that affect runtime testability cross-cut those in Binder’s Testability [4] model, as well as those in Gao’s component-based adaptation [12].

Testability improvement efforts have focused on regression testing, selecting and prioritising the test cases more prone to uncover faults (i.e., those which contribute more test coverage) with the least cost [10, 19, 20, 25]. Our approach to testability improvement is also cost-based. However, their approaches are test-case dependent, whereas our approach is focused towards a test-case independent approach, only focusing on test-relevant features of the system. A test-case prioritisation approach (instead of a fix prioritisation approach) could be devised nevertheless, by using impact cost in the same way the cited methods use execution cost (time) to prioritise test cases.

Other testability-related approaches have focused on modeling statistically which characteristics of the source code of the system are more prone to uncovering faults (i.e., those which contribute more test coverage) with the least cost [10, 19, 20, 25]. Our approach to testability improvement is also cost-based. However, their approaches are test-case dependent, whereas our approach is focused towards a test-case independent approach, only focusing on test-relevant features of the system. A test-case prioritisation approach (instead of a fix prioritisation approach) could be devised nevertheless, by using impact cost in the same way the cited methods use execution cost (time) to prioritise test cases.

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 is planned, in order to explore further the relationship between RTM and defect coverage and reliability.

8. REFERENCES
