A Model for the Measurement of the Runtime Testability of Component-based Systems

Alberto González Éric Piel Hans-Gerhard Gross
Delft University of Technology, Software Engineering Research Group
Mekelweg 4, 2628 CD Delft, The Netherlands
Email: {a.gonzalezsanchez,e.a.b.piel,h.g.gross}@tudelft.nl

Abstract

Runtime testing is emerging as the solution for the integration and validation of software systems where traditional development-time integration testing cannot be performed, such as Systems of Systems or Service Oriented Architectures. However, performing tests during deployment or in-service time introduces interference problems, such as undesired side-effects in the state of the system or the outside world.

This paper presents a qualitative model of runtime testability that complements Binder's classical testability model, and a generic measurement framework for quantitatively assessing the degree of runtime testability of a system based on the ratio of what can be tested at runtime vs. what would have been tested during development time. A measurement is devised for the concrete case of architecture-based test coverage, by using a graph model of the system's architecture. Concretely, two testability studies are performed for two component based systems, showing how to measure the runtime testability of a system.

1. Introduction

Runtime testing is emerging as the solution for the validation and acceptance of software systems for which traditional development-time integration testing cannot be performed. 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.), or dynamic Systems of Systems [14], and Service Oriented Architectures [3], where the components that will form the system are not known beforehand in some cases. Integration and system testing for such systems is becoming increasing difficult and costly to perform in a development time testing environment, so that a viable option is to test their component interactions during runtime.

Although runtime testing solves the problem of the availability of unknown components, and eliminates the need to take the system off-line, it introduces new problems. First, it requires the test infrastructure, like test drivers, stubs, oracles, etc. to be integrated into the runtime environment. Second, the components themselves must assume some of the testing infrastructure in order to take advantage of runtime testing. Third, and most important, it requires knowledge of the likely impact of tests on the running system in order to elude interference of the runtime activities with the operational state of the system, and taking the appropriate measures in order to minimise these disturbances to the lowest possible degree.

The knowledge of the runtime test's impact on a system requires a measurement of what can be tested during runtime compared with what would have been tested if the system was checked off-line in a traditional development time integration test without causing any disturbance in the running system. In this paper this measurement is defined as *Runtime Testability*. Questions regarding runtime testability include "What tests are safe to run?", "Which parts of the system will have to be left untested?" or "How can the runtime testability of a system be improved?".

This paper devises a qualitative model of the main facets of runtime testability, and a framework for the definition of measurements of runtime testability to derive a number of metrics according to the test criteria applied. Furthermore, it defines one of such measurements, based on architectural test coverage on a graph representation of the system. This measurement will be used to estimate the Runtime Testability of two component-based systems, one taken from a case study in the maritime safety and security domain, and an air-

port lounge Internet gateway.

The paper is structured as follows. In Section 2 runtime testing is presented, and in which cases it is necessary. In Section 3, background and related work to runtime testing and testability is discussed and related to our research. Section 4 defines runtime testability in the context of the IEEE's definition of Testability, describes the main factors that have an influence over it, and the generic framework for the measurement of runtime testability. Section 5 presents our particular model-based coverage measurement and describes two examples of the calculation of runtime testability and their results. Finally, Section 6 presents our conclusions and ideas for future research.

2. Runtime Testing

Runtime Testing is a testing method that has to be carried out on the final execution environment [4]. It can be divided on two phases: *deployment-testing* (when the software is first installed), and *in-service-testing* (once the system is in use).

Deployment-time testing is motivated because there are many aspects of a system that cannot be verified until the system is deployed in the real environment [4, 21]. Also, on Systems of Systems and Service Oriented Architectures, engineers will have to integrate components that are *autonomous*, (i.e. the components or services that are integrated have a separate operational and managerial entity of their own [9, 19]). The service or component integrator does not have complete control over the components that he or she is integrating. Moreover, in many cases these components will be remote, third-party services over which he or she will have no control at all, let alone control for accessing a second instance of the system for testing purposes [3, 6].

In-service testing derives from the fact that Component-based systems (such as Systems of Systems and Service Oriented Architectures) can have a changing structure. The components that will form the system *may not be available, or even known* beforehand [3]. Every time a new component is added, removed or updated there will be a number of tests whose results will no longer be valid and will have to be re-verified. The only possibility for testing, thus, is to verify and validate the system after it has been modified and the missing components become available. This is relevant for self-managing autonomic systems as well, which dynamically change the structure or behaviour of components that may be already operating in an unpredictable environment.

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.

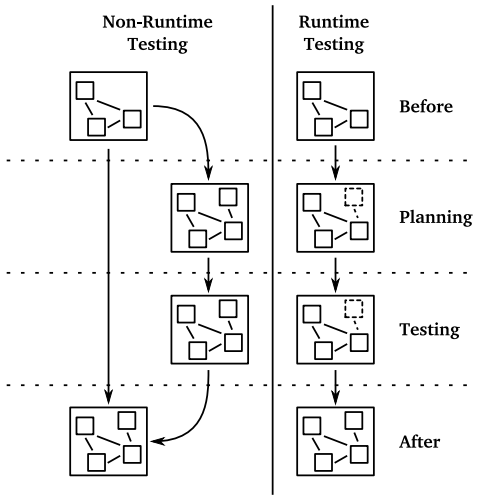

Figure 1. Non-runtime vs. runtime testing

3. Related Work

3.1. Related work on Runtime Testing

Architectural support (special sets of interfaces and/or components and the activities associated to them) for testing a system once it has been deployed have already been introduced for component-based systems [4, 13] and autonomic systems [18].

Many of the concepts inherent to runtime testing are introduced through Brenner et al. [4] without relating them explicitly to the concept of runtime testability. Examples are awareness of when tests are possible from the application logic and resource availability point of view, or the necessity for an infrastructure that isolates application logic from testing operations applying the appropriate countermeasures. Related to this, Suliman et al. [21] discuss several runtime test execution and isolation scenarios, for which, depending on the characteristics of the components, different test isolation strategies are advised. The properties introduced will be applied in our testability model presented in Section 4.

In addition, Brenner et al. [3] also introduce runtime testing in the context of Web Services. The au-

thors argue that traditional component-based testing techniques can still be applied to Web Services under certain circumstances. Different testing strategies for runtime testing are proposed, depending on the services type (stateless, per-client state, pan-client state). However, in this paper neither test sensitivity nor test isolation are mentioned.

3.2. Related work on Testability

According to the IEEE's Standard Glossary, testability is defined as: (1) The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met; (2) The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met.

A number of research efforts are focused on modeling statistically which characteristics of the component, or of the test setup, are more prone to showing faults [10, 22]. These probabilistic models can be used to amplify reliability information [1, 15].

Jungmayr [17] proposes a measurement of testability from the point of view of the architecture of the system, by measuring the static dependencies between the components. This measurement does not intend to extract reliability information, but to maximize the system's maintainability, by minimizing the number of components needed to test another one, and by minimizing the number of affected components after a component changes.

The model presented in this paper can be seen as a complement to Binder's model of Testability for object-oriented systems [2], which is based on six main factors, that contribute to the overall testability of the system: (1) Representation; (2) Implementation; (3) Built-in Test; (4) Test Suite; (5) Test Tools; and (6) Test Process. A very similar model of Testability, adapted to component-based systems, is presented in [11].

Although the testability issues taken into account by these works are a concern when performing runtime testing, they do not explicitly refer to test sensitivity and isolation as first class concerns of runtime testing. This paper provides a complementary model of testability that can be used to assess the ability of a system of being tested at runtime, without interfering with the production state of the system: its *runtime testability*.

4. Runtime Testability

Runtime testing will interfere with the system state or resource availability in unexpected ways, as the production state and data of the system will mix with the testing. Even worse, test operations may trigger events outside the system's boundaries, possibly affecting the system's environment in critical ways that are difficult to control or impossible to recover, 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, what adverse effects can be caused by tests on the running system. The standard definition of testability by the IEEE 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.

Runtime testability is based on two main pillars: *test sensitivity*, and *test isolation*. We will introduce the main factors that have an impact on both of them. Figure 2 depicts a fish bones diagram of them.

4.1. Test Sensitivity

It characterises which operations, performed as part of a test, interfere with the state of the running system or its environment in an unacceptable way. In this section we will describe four of the main factors that have an influence on the test sensitivity of a component: a component having internal state, a component's internal/external interactions, resource limitations, and system availability.

4.1.1. Component State. Knowing if the component exhibits some kind of external state (i.e. the result of an input does not depend only on the value of the input itself, but also on the value of past inputs) is an important factor of test sensitivity. In traditional "off-line" testing, this is important because the invocation order will have an effect on the expected result of the test. In the case of runtime testing, knowing if a component has state is important for two additional reasons. Firstly because the results of runtime tests will be influenced by the state of the system if not handled correctly, and secondly, because the state of of the system could be altered as a result of a test invocation.

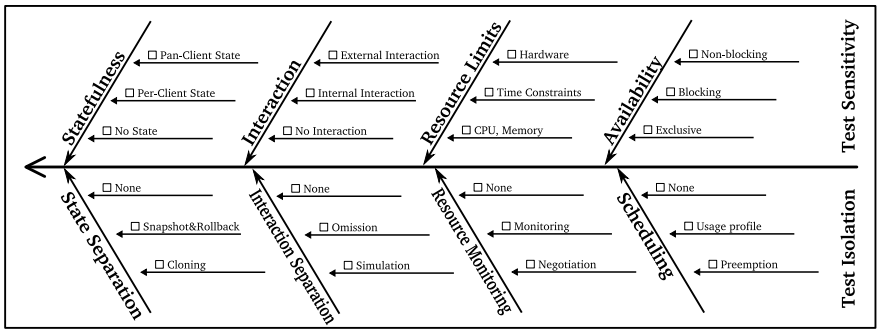

Figure 2. Qualitative factors that affect runtime testability

An interesting distinction, made in [3], is whether the state of the component under test can influence in the states of other components other than the tester component (i.e. each user sees the same common state) or not (i.e. each user sees a different state).

4.1.2. Component Interactions. On many occasions, components will use other components from the system, or interact with external actors outside the boundaries of the system. These interactions have the potential of initiating other interactions, and so forth. This means that the runtime testability of a component depends on the runtime testability of the components it interacts with during a test.

All these interactions will likely cause interferences with the state of the running system by changing the state of any of the components in the collaboration. In some cases, some of these interactions will cross the boundaries of the system and affect the states of other systems, and this may be difficult to prevent and fix. In the worst case, the interaction will reach "the outside world" by sending some output that will enable a physical output that may be impossible to undo, for example, firing a missile.

The main implication of interactions, besides altering the real world, is that the runtime testability of a system cannot be completely calculated if the some of the components are not known (something very common in a dynamic system like a Service Oriented System). Therefore, runtime testability has to be studied online, when the components that form the system are known.

4.1.3. Resource Limitations. The two previous sensitivity factors are mainly affecting functional requirements of the system. However, runtime testing may also affect non-functional requirements. Because runtime tests will be executed on the running system, the load of these tests will be added to the load caused by the normal operation of the system. In some cases it will exceed the available resources of the system such as processor or memory usage, timing constraints, or even power consumption restrictions.

Appropriate measures such as the ones presented later on this paper, must be implemented to ensure that runtime tests do not affect the availability of resources for the components that need them, or that, if the availability is impaired, the affected components can recover.

4.1.4. Availability. The availability requirements of the system in which the testing is going to be performed is also a factor. There exist two possibilities: if the component is going to be active only for testing purposes (exclusive usage), or for both the testing and normal service (shared usage). In a shared configuration two distinctions can be made: blocking and non-blocking. The first means that production operations will be blocked or rejected while the test is being performed, impairing the availability of the services provided by the components. If a component has a high availability requirement, runtime testing under this circumstances cannot be performed. In the second case, test invocations can be interleaved with production invocations and the component is able to distinguish between testing and production requests.

4.2. Test Isolation

Test isolation techniques are the means test engineers have of preventing test operations from interfering with the state or environment of the system, and of the production state and interactions of the system from influencing test results. Therefore, they represent the capability of a system to enable runtime testing by providing countermeasures for its test sensitivity.

A prerequisite for applying test isolation techniques is a prior study of test sensitivity in order to find the runtime testability issues that will make runtime testing to (partially) fail. In the following paragraphs we present some ideas for the development of test isolation

techniques that could be evaluated in further work, by using our measurement in terms of testability gain and implementation effort.

4.2.1. State Separation. State separation techniques are the counterpart of state sensitivity. They aim to separate the in-service state of a component from the testing state. If the component has an observable state, Suliman et al. [21] propose a solution based on three levels of sophistication. The first level consists of blocking the component operation while it is being tested. The second level proposes cloning the component with special support from the runtime environment. The last level relies on special testing sessions supported by the tested components to provide state isolation on components that cannot be cloned easily.

4.2.2. Interaction Separation. Interaction separation can be applied to component interactions that propagate through the system and affect other components, and, in particular, the external environment of the system. When interactions cross the system boundary two possible isolation solutions can be foreseen: omission of the output, or simulation. Omission of an output consists of suppressing the output, as if it had never occurred. This is possible only if the rest of the test does not depend at all on the effects that output might have. If a response is expected, in the form of an input or external event, then the external system will have to be replaced by a simulator, or a mock component.

4.2.3. Resource Monitoring. To prevent test cases from exhausting the resources of the system, resource monitoring techniques can be applied. A simple monitoring solution would be to deny or postpone the execution of the test case if this needs more resources than currently available, for example if system load grows over a certain threshold [4]. A more advanced possibility would be to allow components and tests to negotiate the resources needed for specific tests.

4.2.4. Scheduling. Tests can be scheduled to preserve the availability of the components, aiming to control how, in what number, and at what moment test cases are allowed to be executed in the system. For example, some test cases would only be executed when the component is less needed, akin to what was proposed for the reconfiguration phase in [20]. If a component is blocked by a test, but there is a certain service operation that cannot be put on hold anyway, the test case could be pre-empted from the system to satisfy the service call.

4.3. Runtime Testability Measurement

Ultimately, all the test sensitivity factors which impede runtime testing will prevent test engineers from assessing a certain feature or requirement that could otherwise be performed under ideal conditions of unlimited resources and full control of the running system. This is the main idea used in this section to obtain a numerical measurement of the Runtime Testability Measurement (RTM) of a system.

Let M^* be a measurement of all those features or requirements which we want to test and M_r the same measurement but reduced to the actual amount of features or requirements that can be tested at runtime, with $M_r \leq M^*$. The Runtime Testability Measurement (RTM) of a system is defined as the quotient between M^*, and M_r.

$$RTM = \frac{M_r}{M^*} \tag{1}$$

Although generic, the simplicity of RTM allows engineers to tailor it to their specific needs, applying it to any abstraction of the system for which they whose features they would like to asses by runtime testing. In this paper, we will further instantiate RTM in terms of test coverage, to estimate the maximum test coverage that a test engineer will be able to reach under runtime testing conditions. This measurement of runtime testability can be used to predict defects in test coverage, even in the complete absence of test cases, and correct this situation by showing the testability improvement when the issues causing the untestabilities are addressed.

Given C, the set of all the features that a given test adequacy criterion requires to be covered, and C_r, the set of features which can be covered at runtime, the value of RTM, based on Equation 1, is calculated as

$$RTM = \frac{|C_r|}{|C|} \tag{2}$$

This definition is still generic enough so that it can be used with any representation 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.

5. Case Study

As a concrete case, this paper presents the application of a graph dependency model of the system, anno-

tated with runtime testability information. This model is used to assess the cost of covering a specific feature with potential runtime test sequences, and to remove those whose cost is unacceptable.

A runtime testability study is performed on two component-based systems: a system-of-systems taken from a case study in the maritime safety and security domain, and an airport's wireless access-point system. The objective of our experiments is to show that our model can help identify systems not apt for runtime testing, and taking decisions on how to address this situation, with the final goal of improving the quality and reliability of the integrated system.

5.1. 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 component to provide the service). During a test, any service of a component can be invoked in a component, although at a cost. In the case of a runtime testability analysis, the cost we are interested in is the impact cost of the test invocation on the running system or its environment, derived from the sensitivity factors of the component in Section 4. These costs can present themselves in multiple magnitudes (computational cost, time or money, among others).

Operations whose impact cost is prohibitive have to be avoided, designating them as untestable. In this paper we will abstract from the process of identifying the cost sources and their magnitudes, and assume that all operations have either testable (no cost) or untestable (infinite cost).

For the purposes of this paper, the system will be modelled using a directed component dependency graph known as Component Interaction Graph (CIG) [24]. 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 is easy and its computation is a tractable problem.

A *CIG* is defined as a directed graph $CIG = (V, E)$. The vertex set, $V = V_P \cup V_R$, is formed by the union of the sets of provided and required vertices, where each vertex represents a method of an interface of a certain component. Edges in E are created from the vertices corresponding to the required interfaces to the vertices of provided of interfaces for inter-component dependencies, and from the provided to the required interfaces for intra-component dependencies.

Each vertex $v_i \in V$ is annotated with a testing penalty information τ_i, meaning whether it is possible to traverse such vertex when performing runtime testing or not, as follows:

$$\tau_i = \begin{cases} 0 & \text{if the vertex can be traversed} \\ \infty & \text{otherwise} \end{cases} \quad (3)$$

Edge information 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]. In the case no information is available for a certain vertex, a conservative approach should be taken, assigning infinite weights to it.

5.2. Coverage Criteria

A number of architectural test coverage adequacy criteria have been defined based on *CIG* or other similar representations [16, 24]. We will measure the runtime testability of the system based on two adequacy criteria proposed in [24]:

The *all-vertices* adequacy 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 our 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_j is context dependent on v_i if there's an invocation sequence from v_i that reaches v_j. For each of this dependences, all the possible paths $(v_i, v_{i+1}, \ldots, v_j)$ are considered viable, and need to be tested.

5.3. Value of RTM

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 do not look at the penalty of actual test cases, but at the possible, worst-case penalty of any test case that tries to cover the elements of C.

We will assume that the interaction starts at the vertex (for *all-vertices* coverage) or the first vertex of the path (for *all-context-dependences* coverage) that we want to cover. Because edges in the *CIG* represent interactions that might happen or not (without any control-flow information), we cannot assume that when trying to cover a path only the vertices in the path will be traversed. In the worst case, the interaction could propagate through all vertices reachable from the vertex where the interaction starts. Therefore, to estimate the worst-case penalty of covering a vertex v_i or a context dependence path starting at vertex v_i, the calcula-

	AISPlot	WifiLounge
Total components	31	9
Total vertices	86	159
Total edges	108	141
Context-dependent paths	1447	730

Table 1. Characteristics of the experiments

tion has to take into account all the vertices reachable from v_i, which we will denote as P_{v_i}.

For each vertex v_i or path $(v_i, v_j, v_k, \ldots)$ that we would like to cover, we calculate a penalty value $T(v_i)$ similar to the one for individual vertices:

$$T(v_i) = \sum_{v_j \in P_{v_i}} \tau_j \tag{4}$$

By considering as testable only those features whose $T(v_i) \neq \infty$, Equation 2 can be rewritten for *all-vertices* and *all-context-dependence* coverage, respectively, as

$$RTM_v = \frac{|\{v \in V \mid T(v) \neq \infty\}|}{|V|} \tag{5}$$

$$RTM_{c\text{-}dep} = \frac{|\{(v_i, v_j, v_k, \ldots) \in CIG \mid T(v_i) \neq \infty\}|}{|\{(v_i, v_j, v_k, \ldots) \in CIG\}|} \tag{6}$$

A possibility for future research is to use finite valued penalties, establishing finite upper limits for the traverse penalty.

5.4. Experimental Setup

In both experiments, intra-component *CIG* edges were derived by static analysis of the primitive components' source code. The edges between components were derived by inspecting the dependencies at runtime using reflection. The runtime testability flag τ_i was added based on the test sensitivities found on the source code of each component. In order to keep the number of untestable vertices at a tractable size, 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 characteristics of the system architecture and graph model of the two systems used in our experiments, including number of components, vertices, edges, ad context-dependent paths of each system.

5.4.1. AISPlot. For the first experiment we will use 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, code-

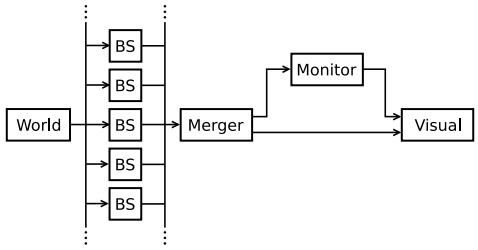

Figure 3. AISPlot Component Architecture

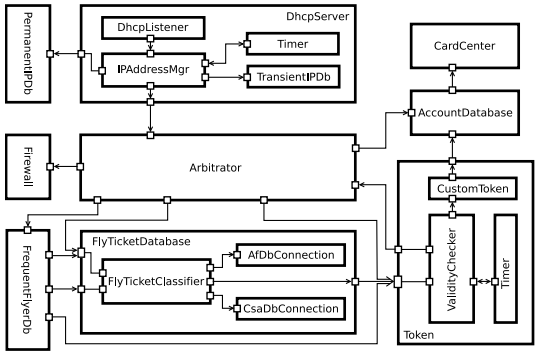

Figure 4. Wlfi Lounge Component Architecture

named AISPlot. The architecture of the AISPlot system can be seen in Figure 3.

Position messages are broadcast thourgh 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 received messages in search for inconsistencies in messages to detect potentially dangerous situations, e.g. ships on collision course. 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.

5.4.2. Airport Lounge. In a second experiment we diagnosed the runtime testability of a wireless hotspot at an airport lounge [7]. Clients authenticate themselves as either business class passengers, loyalty program members, or prepaid service clients. The component architecture of the system is depicted in Figure 4.

When a computer connects to the network, the `DhcpListener` component generates an event informing of the assigned IP address. All communications are blocked by the firewall until the authentication is validated. Passengers of business class are authen-

	RTM_v	$RTM_{c\text{-}dep}$
AISPlot	0.14	0.012
WifiLounge	0.62	0.41

Table 2. Runtime testabilities of both systems

ticated against a number of fly ticket databases. Passengers from a miles program are authenticated against the frequent flyer program 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.

5.5. Testability Diagnostic

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 on the real screen if not isolated properly. Figure 5 depicts the Interaction Graph of AISPlot, with the problematic vertices marked with a larger dot.

As summarized in Table 2, the RTM of the system is initially 14% for vertex coverage, and 1.2% for context dependence coverage. This extremely poor value is due to the fact that the architecture of the system is organised as a pipeline, with the `Visual` component at the end. Almost all vertices are connected to the five problematic vertices of the visualiser component, so they will appear in the P_{v_i} of almost every vertex.

On the other hand, for the Airport Lounge system, 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 impossible to duplicate. The interaction graph of the system can be seen in Figure 6.

The runtime testability of the system is intermediate, 62% for vertex coverage, and 41% for context dependency coverage. This value is far from the extremely low testability of the AISPlot system, because even though there are more runtime-untestable vertices than in the previous case, they are not part of as many P_{v_i} as it was the case for the previous experiment.

5.6. Discussion of the Experiments

Once the model of the system and the measurement is obtained, the potential applications of this measurement are multiple. It can be used to evaluate the gain in testability when the test sensitivities of a set of vertices are addressed. An algorithm can be devised to find the optimal solution in terms of cost and testability gain, for example by providing fix costs for each vertex or component, and evaluating all the possible fix combinations.

Moreover, even though the relationship between test coverage and defect coverage is not clear [5], previous studies have shown a beneficent effect of test coverage on reliability [8, 12, 23]. Furthermore, coverage is a widely used measurement as a quality indicator by the industry. For this reasons, and as our measurement is as an indicator of the maximum test coverage attainable of a runtime-tested system, our measurement can be used as a indicator of the quality of a system, for example by using a coverage-based reliability models to account for the fact that coverage will be imperfect when performing runtime testing.

6. 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. A measurement for this limitations is what we have defined as runtime testability.

In this paper, we have presented a qualitative model of the main factors that affect the runtime testability of a system. Furthermore, we have provided a framework for the definition of quantitative coverage-based runtime testability measurements, and proposed a concrete application of our measurement to a number of coverage criteria of the Component Interaction Graph of the system. This model is very suitable for many types of systems, such as data flow or client-server architectures.

Further work will include using this model and measure to find the optimal fix at the lowest cost, and a possible refinement of the model, for example to include state information to give better estimations. The evaluation of accuracy of the predicted values and of the effect of runtime testability on the system's reliability is also left for future work. More validation using industrial cases and synthetic systems is also planned.

Acknowledgements: This work has been carried

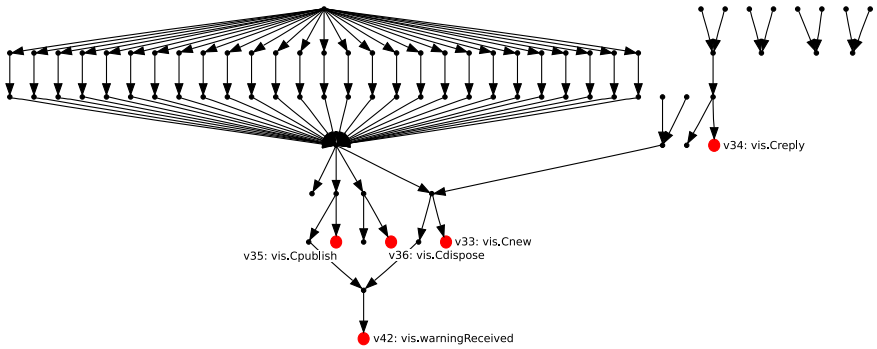

Figure 5. AISPlot Interaction Graph

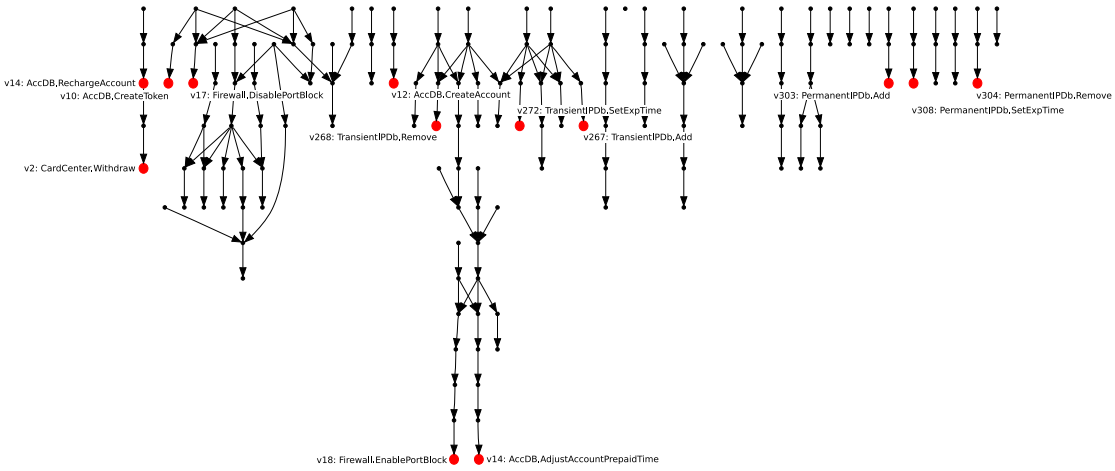

Figure 6. Wifi Lounge Interaction Graph

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

References

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

[2] R. V. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, 1994.

[3] 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*, pages 114–121, Washington, DC, USA, 2007. IEEE Computer Society.

[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*, 9(2-3):151–162, 2007.

[5] 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*, pages 148–157, 1999.

[6] A. Bucchiarone, H. Melgratti, and F. Severoni. Testing service composition. In *8th Argentine Symposium on Software Engineering*, Mar del Plata, Argentina, 2007.

[7] T. Bures. Fractal BPC demo. http://kraken.cs.cas.cz/ft/doc/demo/ftdemo.html.

[8] X. Cai and M. R. Lyu. Software reliability modeling with test coverage: Experimentation and measurement with a fault-tolerant software project. In *ISSRE '07: Proceedings of the The 18th IEEE International Symposium on Software Reliability*, pages 17–26, Washington, DC, USA, 2007. IEEE Computer Society.

[9] D. Fisher. An emergent perspective on interoperation in systems of systems. Technical Report CMU/SEI-TR-2006-003, Software Engineering Institute, 2006.

[10] R. S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, 1991.

[11] 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*, volume 2, pages 211–218, Washington, DC, USA, 2005. IEEE Computer Society.

[12] S. S. Gokhale and K. S. Trivedi. A time/structure based software reliability model. *Annals of Software Engineering*, 8(1-4):85–121, 1999.

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

[14] 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 (TAIC PART'08)*, pages 35–39, Windsor, United Kingdom, Aug. 2008. IEEE Computer Society.

[15] D. Hamlet and J. Voas. Faults on its sleeve: amplifying software reliability testing. *SIGSOFT Software Engineering Notes*, 18(3):89–98, 1993.

[16] N. L. Hashim, S. Ramakrishnan, and H. W. Schmidt. Architectural test coverage for component-based integration testing. In *QSIC '07: Proceedings of the Seventh International Conference on Quality Software*, pages 262–267, Washington, DC, USA, 2007. IEEE Computer Society.

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

[18] T. M. King, D. Babich, J. Alava, P. J. Clarke, and R. Stevens. Towards self-testing in autonomic computing systems. In *Autonomous Decentralized Systems, 2007. ISADS '07. Eighth International Symposium on*, pages 51–58, mar 2007.

[19] M. W. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284, 1998.

[20] J. Matevska and W. Hasselbring. A scenario-based approach to increasing service availability at runtime reconfiguration of component-based systems. In *EUROMICRO '07: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 137–148, Washington, DC, USA, 2007. IEEE Computer Society.

[21] 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*, volume 2, pages 171–176, Sept. 2006.

[22] J. Voas, L. Morrel, and K. Miller. Predicting where faults can hide from testing. *IEEE Software*, 8(2):41–48, 1991.

[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*, pages 103–111, 1992.

[24] 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*, volume 0, page 0222, Los Alamitos, CA, USA, 2001. IEEE Computer Society.