Delft University of Technology
Software Engineering Research Group
Technical Report Series

RiTMO: Runtime Testability Measurement and Optimisation

Alberto González, Éric Piel and Hans-Gerhard Gross

Report TUD-SERG-2009-016

TUD-SERG-2009-016

Published, produced and distributed by:

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

ISSN 1872-5392

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

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

Note: Accepted as short paper at QSIC 2009

RiTMO: A Method for Runtime Testability Measurement and Optimisation *

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

This paper introduces a method for the improvement of the runtime testability of a system, which provides an optimal action plan considering the trade-off between testability and implementation cost. The computation of the action plan is driven by an estimation of runtime testability, and based on a model of the system. Runtime testability is estimated independently of the test cases and focused exclusively on the test-relevant features of the system.

This method is used to direct integration- and test-engineers in the implementation of improvement measures for the runtime testability during the integration of a system. Furthermore, it can also be applied earlier, on the design stage, providing a Design for Testability method for dynamic systems. By means of an example, we demonstrate how our method enables an optimal expenditure of resources towards runtime testability during the integration phase of a system.

1. Introduction

Integration and system-level testing of complex, dynamic and highly available systems, such as Systems of Systems and Service Oriented Architectures, is becoming increasingly difficult and costly to perform in a dedicated development-time testing environment. Such systems cannot be duplicated easily, nor can their usage context. Moreover, in some cases, the components that will form the system are not available, or even known beforehand. Proper testing and validation of such systems can only be performed during runtime. *Runtime Testing* poses considerable runtime integration and testing challengesto engineers and researchers alike [4, 8].

A prerequisite for runtime testing is the knowledge about which items can be tested safely while the system is operational without disrupting the system's operation or its environment. This knowledge can be expressed through the concept of *Runtime Testability* of a system [9].

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 systems less prone to hiding faults [2, 6, 12, 14, 19], or which select test cases that are more likely to uncover faults [5, 15, 17]. However, these approaches are not suited for the specific challenges posed by runtime testing, especially, the interference which the tests will cause on the running system (which determines the viability of runtime testing), is not taken into account by these techniques. Features of the system which require tests whose interference is too high will have to be left untested, increasing the probability of leaving undetected faults. Knowledge of the impact that runtime tests will have on the system will allow engineers to select and implement appropriate measures to avoid interference with the system, or with its environment. Therefore increasing the probability of uncovering faults in the system, by making more features runtime testable.

The main contribution of this paper is RiTMO, a method to enhance the runtime testability of a system at design or deployment-time. RiTMO computes an action plan for the implementation of improvement measures of the system's runtime testability. For this purpose, it uses an estimated value of runtime testability based on the impact of runtime tests on the system [9], along with the cost of the remedial measures needed to reduce their im-

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

pact. 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. We present a detailed application example of RiTMO to a system taken from our industrial case study in maritime safety and security systems.

The paper is structured as follows. In Section 2 the concept of Runtime Testability is introduced. In Section 3, a method to improve runtime testability of a system is described. Section 4 evaluates the proposed method. In Section 5 related work is presented and compared to our research. Section 6 wraps up the paper and introduces some ideas for further research.

2. Runtime Testability

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 running system be tested without affecting its functionality 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 affected; (2) the specification of which tests are allowed to be performed during runtime without affecting the running system.*

An appropriate measurement and improvement method for (1) relies on general information about the system, independent of the nature of the runtime tests that may be performed, as it is proposed in [6, 19] for traditional testing. On the other hand, a measurement and improvement method for (2) will rely on information about the concrete test cases that are going to be performed, as proposed in [5, 15, 17] for regression testing. In this paper we will specifically concentrate on the first (system-centric) aspect of runtime testability.

Runtime testability is significantly influenced by two main characteristics of the system: *test sensitivity*, and *test isolation* [9]. 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. 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.

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 undetected faults. In order to test those features, extra cost has to be spent in addressing some of their sensitivity factors.

A numerical measurement for the runtime testability of a system can be defined in terms of what fraction of features of the system can be runtime tested without interfering with the system; 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. This supports engineers in taking decisions on whether more resources have to be spent improving the runtime testability of the system.

The Runtime Testability Measurement (RTM) was defined in [9], as the quotient between the number of features of the system which can be runtime tested without interfering with the system, and the total number of features, e.g., as determined by a test adequacy criterion:

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

where C is the complete set of features which have to be tested, and C_r is the subset of those features which can be tested at an acceptable cost.

We estimate the runtime testability of a system through an instantiation of the above generic definition of RTM to component-based systems, based on a static graph dependency model annotated with runtime testability information. This model is applied to estimate the (runtime impact) cost of invoking a specific feature of the system (e.g. a service or a specific interaction path), independently of the test cases used for it.

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

2.1. 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 (as decided by the system and test engineers), 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.

For now, we will abstract from the process of identifying the cost sources, and we will assume that all operations have already been classified as testable or untestable.

The system is modelled using a directed component dependency graph known as Component Interaction Graph (CIG) [20]. 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 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 test impact flag τ_i, representing whether the cost of traversing such vertex (i.e., invoking that service) when performing runtime testing is acceptable ($\tau_i = 1$) or not ($\tau_i = 0$). In Section 3, more details are given on how to derive the sets of vertices, edges and test impact costs.

2.2. Estimation of RTM

To estimate the runtime testability of a system, we will estimate the impact cost of covering each of the features of the graph, according to two different coverage criteria: *all-vertices*, and *all-context-dependence* [20]. The *all-vertices* criterion requires executing each method in all the provided and required interfaces of the components, at least once. On the other hand, the *all-context-dependence* criterion requires testing every possible invocation path (context) between two vertices.

The purpose is not to estimate the concrete impact of specific test cases, but the possible impact cost of any test case that tries to cover an element. Because of the lack of control flow information of the CIG model (it is a static dependency model of the system), we will assume that (1) the interaction starts directly at the point which we want to cover, and (2) that the interaction might propagate through all edges, affecting all reachable vertices.

For each vertex v_i or path $(v_i, v_j, v_k, \ldots)$ that we would like to cover, the test impact, $T(v_i)$, is calculated as

$$T(v_i) = \bigwedge_{v_j \in P_{v_i}} \tau_j \tag{2}$$

where P_{v_i} is the set of all the vertices reachable from v_i.

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

$$RTM_v = \frac{|\{v_i \in V \mid T(v_i) = 1\}|}{|V|} \tag{3}$$

$$RTM_{c\text{-}dep} = \frac{|\{(v_i, v_j, v_k, \ldots) \in CIG \mid T(v_i) = 1\}|}{|\{(v_i, v_j, v_k, \ldots) \in CIG\}|} \tag{4}$$

The runtime testability measurement provides an estimate of the runtime testing possibilities of a system, which is linked to the reliability of the system when it is constructed or updated at runtime. In practice, its main usage is to guide the system developers and integrators towards obtaining a more reliable system, i.e. with a high RTM.

3. RiTMO: a Method to Improve Runtime Testability

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

In a typical application scenario, a system with high availability and reliability requirements is being configured during run-time. In order to allow good testing coverage, integration engineers aim for a high RTM. By combining component design information with the system's architecture, the system integrator uses RiTMO in order to determine where the budget is best spent in order to increase runtime testability.

Another application scenario is a runtime system update. Then, the system integrator uses RiTMO to verify that sufficient runtime regression testing will be possible. If that is not the case, our method will point out which are the most worthy components of the system where where isolation measures will need to be applied before the testing of the update can be done.

Figure 1 depicts the five main steps which compose the RiTMO method, associated to the engineering roles in charge of performing the task. It must be noted that a

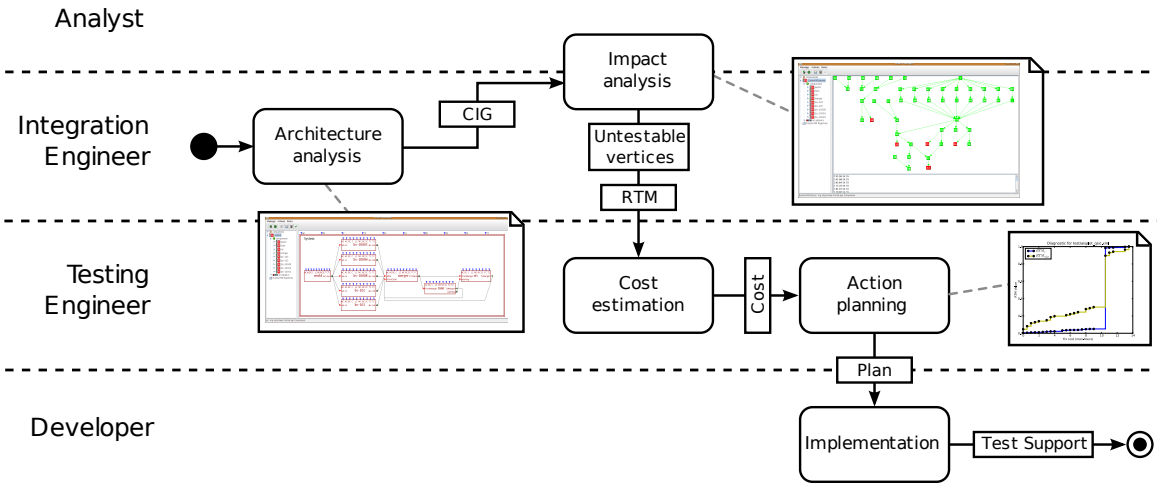

Figure 1. Complete workflow of RiTMO.

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

3.1. Step 1: Architecture Analysis

In order to compute the RTM of a system, the first step provides an architectural view of the system at the right level of granularity. In RiTMO, the architecture of the system is represented by a static dependency graph, the CIG, as depicted in Figure 2. This graph provides information on the call dependences between each method of each component in the system. A vertex of the graph represents a method, and an edge represents a call dependence of one method on another method.

Vertices in the CIG are obtained by examining the required and provided interfaces of each component instance. For each method of each interface, a vertex is created. If a method of a required interface is not used, i.e., no code of the component uses this part of the interface, the corresponding vertex is removed as it would add unreachable paths. It is important to note that when there exist multiple instances of a same component, each instance corresponds to different vertices, as their dependencies are different. Composite components are treated in a similar way: vertices are added for each of the methods in the interfaces.

Edges are generated in two independent steps. The first step of edge generation consists in finding the dependencies between the provided and required interfaces inside components. This task is challenging, because the very philosophy of component-based development hides the dependencies between interfaces: components are black boxes. There are, nevertheless, several ways to obtain this information:

- **From the specification.** If the models of the component behaviour are precise enough, it is possible to deduce for each provided functionality (interface method), which required method it depends on. For example, using sequence diagrams used to specify a component, the edge generation can be done automatically. The drawback is that even if the specification is precise enough, the implementation might be different due to optimisation (e.g. caching) or limitations (e.g. some functionalities are not implemented).
- **From the implementation, statically.** If the component's implementation is actually available, analysing the source code will allow to precisely determine which required methods are used for every provided method.

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

Table 1. Actors, inputs and outputs of each activity involved in RiTMO.

- **From the implementation, dynamically.** Even if neither a model nor the component's source are available, it is still possible to deduce the dependencies between the methods using a tracing mechanism [16]. The component can be exercised either by using available unit test cases, or by observing it running in the actual system. The drawback of this method is that some dependencies might not be detected if the calls are not triggered by the operational profile used during tracing.

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

Dependencies between components can be determined in two ways:

- **By introspection.** Runtime architecture reconstruction [13] can be applied if the component framework provides introspection functionality, such as EJB, CCM or Fractal. In this case, this step can be done entirely automatically.

- **From the specification.** If such functionality was not available, it is still possible to derive the edges either from a model of the system, such as a UML component model.

The CIG of an example composite component obtained by our method is depicted in Figure 2. The crossed out vertices and dashed edges represent elements that will be removed as they are unreachable (o_3 is never used, and $B.v$ cannot be used from outside C). Still, this way of deriving a CIG often leads to redundant vertices and edges, e.g., (o_1, u_1), or all the (i, m) pairs. This choice was made in order to keep the composition method of the CIG simple. Moreover, if there are modifications of the CIG due to an architectural reconfiguration, more paths will be kept compared optimised CIG, which will facilitate comparison with the old system.

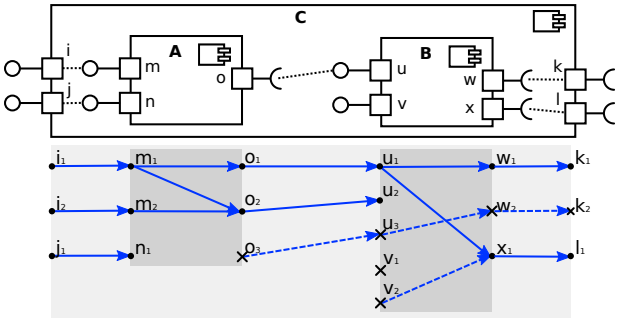

Figure 2. Example of CIG representation of a system.

Once this last step of the graph generation is done, the CIG model is complete, providing an architectural knowledge of the system with a fine granularity. The next step consists in determining which methods are not allowed to be executed during runtime testing due to their excessive impact on the running system or its environment.

3.2. Step 2: Impact Analysis and RTM

The goal of this step is to augment the CIG of the system with information on the impact of executing each method. In other words, it is necessary to annotate each vertex of the graph with the test impact flag τ_i, as de-

scribed in Section 2.2. The derivation of this information cannot be easily automated, and it is up to the integration engineer and analyst to apply their design and domain knowledge for deciding which method calls could disrupt the normal operation of the system. In the case no information is available for a certain vertex, a conservative approach should be taken, assigning $\tau_i = 1$.

Performing the impact analysis step requires inspecting each component specification or source code, checking if any activity in a testing context can have either impact on the state of the system or affect the system's environment in an undesired way. In case the system is large, this can be a laborious task, however it can be greatly simplified if component vendors associate test impact meta-information to their components, so that this information is easily reusable, and provided by those stakeholders, who have the most precise knowledge about the component.

Using this annotated graph, it is possible to compute the value of RTM for the chosen coverage criteria (vertex or context-dependence coverage), according to Equations 3 or 4. If this value is higher than the target coverage value for the system tests, the method can stop, as the system is sufficiently runtime testable. If this is not the case, improvements on the system must be performed.

3.3. Step 3: Cost Estimation

The third step of the RiTMO method consists in complementing the graph of the system with an estimation of the effort needed for the implementation of countermeasures to reduce the test impact of untestable vertices. For each vertex v_i marked with a $\tau_i = 0$, a cost, c_i, which represents the estimated cost of making it runtime testable, is associated. The cost unit is not fixed, but has to be uniform for the whole graph. Typically, the cost is expressed in terms of time (e.g., man-hours) or money.

Every untestable method is assessed by combining knowledge of the possible countermeasures (e.g., making the method test-aware) with knowledge about the developer team capabilities, in order to obtain an estimation of the implementation effort. Although this step cannot be easily automated, the estimation does not need to be very precise as long as the estimations are relatively consistent with each other so that the computed plan is still valid in relative terms.

3.4. Step 4: Computation of Action Plan

The completely annotated CIG contains information about the untestable methods and the cost to make them testable. It is then possible to compute an action plan automatically for obtaining the optimal RTM given a maximum budget. Conversely, it is also possible to compute an action plan with the minimal cost to reach a fixed RTM.

A general exhaustive search algorithm could be used to obtain the optimal plan. However, in practice, this approach is not usable for systems of realistic size due to its exponential complexity. This is the reason why a heuristic near-optimal algorithm was introduced in [10]. The resulting action plan in both cases consists of the list of methods to address, along with the final expected RTM value.

In order to have the most information available about the testability improvement, the optimal growth function of the system's RTM vs. fix cost can be calculated. From such graph it is easy to read what is the minimal budget required to reach a given RTM. An example is shown in Figure 5.

3.5. Step 5: Application of the Plan

The last step of the method consists in following the action plan by implementing the fixes for each of the method selected by the algorithm.

It is likely that the cost of addressing a vertex was under- (or over-) estimated. Therefore, it is advisable to address the methods in an order which will maximize the growth of RTM even when some of the vertices are not yet been made runtime testable, so that the most features have been made testable per unit of cost. The algorithm in [10] already provides this ordering, however, if the solution was obtained by the optimal algorithm, this order is still unknown Although obtaining the ordering is once again of exponential complexity, the heuristic algorithm could be applied to the solution's vertices to obtain a near-optimal ordering.

4. Application Example

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

4.1. System Setup and Architecture

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

The system is used to track the position of ships sailing a coastal area, detecting and managing potential dangerous situations. Position messages are broadcast through radio by ships (represented in our experi-

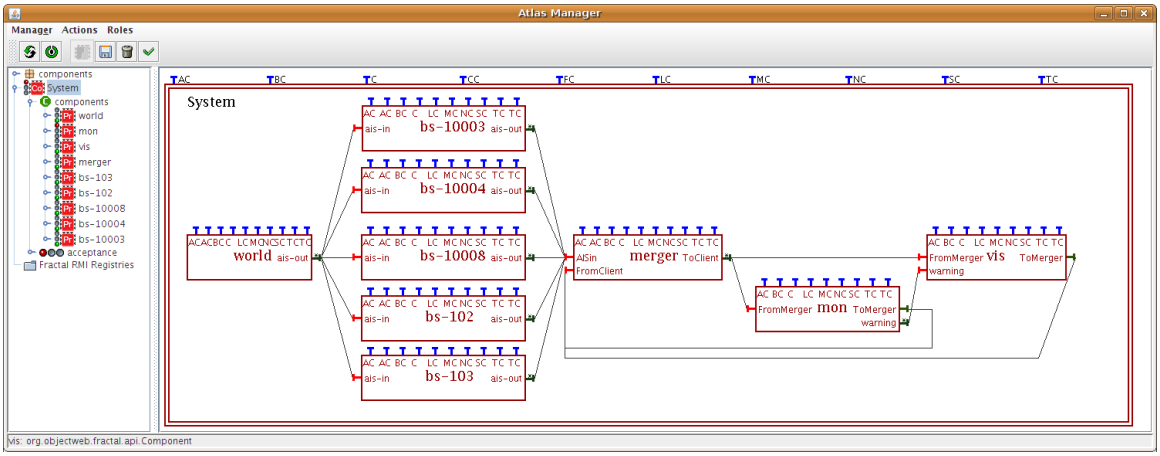

Figure 3. Runtime-reconstructed AISPlot Architecture

ment by the `World` component), and received by a number of base stations (`BS` components) spread along the coast. If a message received from the simulator belongs to the area the base is covering, it is relayed to the `Merger` component thorugh the `AISin` interface. `Merger` removes duplicates (some base stations cover overlapping areas), and offers a subscription service to client components for receiving updates on the status of vessels. Components interested in receiving these updates, can subscribe to receive notifications through the `Merger-Client` interface pair. The `Merger` interface offers the `Csubscribe` and `Cunsubscribe` operations for manging client subscriptions. The `Client` interface offers four operations for notification of ship status: `Cnew`, `Cpublish`, `Creply` and `Cdispose`. In the current system, there are two clients: a safety monitor, and a visualiser screen. 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, via the `Warning` interface.

4.1.1. Implementation. AISPlot is implemented in Java, on the ATLAS/FRACTAL runtime integration and testing research platform [8]. It is a dynamic, reflective component-based platform that allows the insertion, modification and removal of components at run-time. In addition, ATLAS/FRACTAL provides a way of enabling components to test the services they depend on, based on the principles of Built-In Testing (BIT) [11]. ATLAS provides components with the ability of requesting the execution of test cases associated with them, in order to check that the services on which they rely conform to their expectations, and of receiving a notification of when they are going to be involved in a test. At the time of this writing, ATLAS/FRACTAL does not provide means of interleaving test and nominal operations. Therefore, during a test, components only receive test invocations (this is referred to as *blocking* test execution context [9]).

For the purpose of this experiment, a number of new features were introduced into ATLAS/FRACTAL:

- A CIG model can be associated to primitive components, and retrieved at runtime.
- Composite components can compose their own CIG from the CIG's of their contained components.
- Special *test support* code can be injected at run-time into any component during tests. This allows the deployment of test isolation code into the components that need it.

4.2. Application of RiTMO

4.2.1. Architecture analysis. Because of the reflective capabilities of ATLAS/FRACTAL, the derivation of the set of required and provided interfaces into vertices, and the creation of the edges from external connections between components in the system, were a straightforward and completely automated process. Figure 3 shows the reconstructed view of the system's architecture (components and run-time links) obtained by querying the reflection interfaces provided by FRACTAL.

ATLAS/FRACTAL does not provide a way of reverse-engineering primitive components to extract edges induced by internal dependencies. Therefore, this process was performed semi-automatically by source code inspection. The *call hierarchy* functionality of Netbeans[1] was

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

used to automatically derive a call tree of each of the public operations in the implemented interfaces of each component, and locate calls to methods of required interfaces.

The CIG models obtained this way were stored as plain text files and associated to their components using the mentioned functionality of ATLAS/FRACTAL. During the system deployment, the management console of ATLAS/FRACTAL automatically composes all the CIGs of the primitive components to obtain the complete CIG of AISPlot, which can be seen in Figure 4.

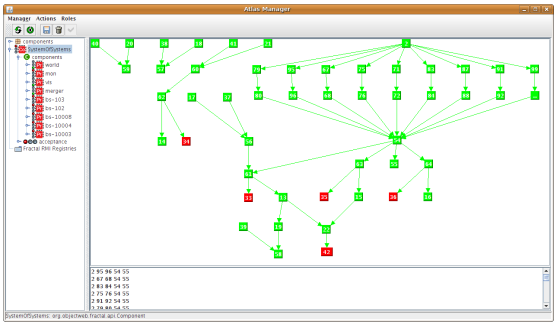

Figure 4. AISPlot CIG during impact analysis

4.2.2. Impact analysis. The `World` and `BS` components are stateless and have no interaction outside the system's boundaries. Therefore, no impact is caused by invoking their operations during a test.

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

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

The preliminary estimation of the runtime testability of the system can be seen in Table 2. Due to the pipelined nature of the system, only a very low number of vertices and paths are testable, those which are not related to the main pipeline path, hence the extremely low RTM values.

4.2.3. Cost estimation and action planning. A number of ATLAS/FRACTAL runtime test support artefacts had to be developed in order to address the test sensitive operations of the system. The development cost was estimated as effort in man-hours required to program the needed isolation code.

	Total	Testable	RTM
Vertices	86	4	0.046
Context-dependences	1621	6	0.003

Table 2. RTM results of the impact analysis

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

Table 3. Untestable operations in AISPlot

The test sensitive operations in `Merger` and `Monitor` require maintaining separate tables for vessel accounting (and subscription management in the case of `Merger`). As both the vessel table and the subscription table are implemented inside the components, the solution we selected was to insert another instance of the component as "test support instance". In ATLAS/FRACTAL, the test support functionality automatically redirects all the calls happening during testing to this special instance.

The solution required for `Visual` is more involved, as it requires isolating the low-level drawing commands that the visualiser is sending to the display. Moreover, it was decided that drawing commands resulting from a test should not be discarded but sent to a different display so that the observability of the system under test is enhanced.

Given our previous knowledge in implementing a set of analogous measures for a system with components of similar characteristics, the cost of implementing test support for the untestable operations in `Merger` and `Monitor` was estimated to be approximately 0.5 man-hours, and the implementation of the isolation and observation code for each operation in `Visual` would have an estimated cost of 2 man-hours. Table 3 summarises all the test sensitive operations in each component, along with suitable isolation techniques and their estimated costs.

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

plot, it would have been difficult to find this issue.

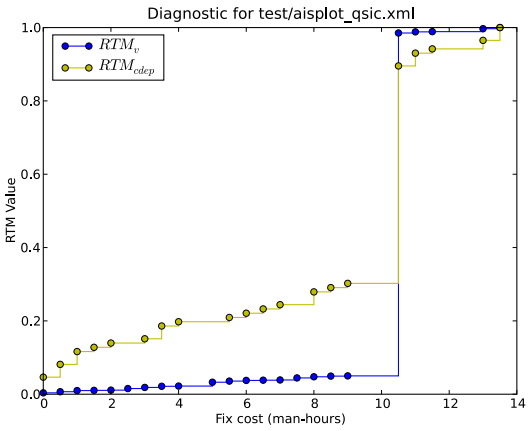

Figure 5. Evolution of RTM for different action plans in AISPlot

	Total	Testable	RTM
Vertices	86	77	0.895
Context-dependences	1621	1597	0.985
Plan	13,15,16,33,35,36,42,54,58		
Cost	10.5 man-hours		

Table 4. RTM results of the action plan

Table 4 shows the final value for the RTM of the system once all the countermeasures have been applied to the untestable vertices, along with the vertices which are part of the plan.

4.3. Discussion

In our example, we have shown how this method can be used in a component-based system, revealing a great testability gap (from 30% to 90% runtime coverage prediction) which would not have been easy to identify by simply looking at the architecture of the system.

Although in our example RiTMO is applied at a late stage of the system deployment, this is not the only possible application scenario. RiTMO can be applied to drive a *Design for Runtime Testability* phase in the earlier stages of the development life-cycle of a system, given that the knowledge about which components will be integrated, and their test sensitivities, is available. This way, the component's design can take into account runtime testability concerns from a much earlier stage, saving resources in the implementation of isolation measures later on the development life-cycle.

As final positive remark, mention that even though in our application example we have relied on specific support features provided by the ATLAS/FRACTAL platform, e.g., model meta-information, introspection and test code injection, RiTMO does not depend on these functionalities. It is still possible to apply it to systems where the platform does not provide runtime testing-specific features.

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

5. 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 work is the first to define a cost-driven method (RiTMO) to drive testability improvement efforts in terms of a testability/cost optimisation problem.

Two articles form the base for our approach to runtime testability, presented in [9], and complemented with the definition of RiTMO in this paper. In [4], Brenner et al. introduce the concepts of test sensitivity and isolation. However no mention to nor relation with the concept of runtime testability are done. In [18], Suliman et al. discuss several test execution and sensitivity scenarios, for which different isolation strategies are advised. The factors that affect runtime testability cross-cut those in Binder's Testability [3] model, as well as those in Gao's component-based adaptation [7].

Testability improvement efforts have focused on regression testing a well, selecting and prioritising the test cases more prone to uncover faults (i.e., those which contribute more test coverage) with the least cost [5, 15, 17]. Our approach to testability improvement is also cost-based. However, their approaches are focused on test cases, whereas our approach is focused towards the system.

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, 19] for amplifying reliability information [2, 12]. Jungmayr [14] proposes a method based on 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.

6. Conclusions and Future Work

In this paper we have presented RiTMO, a cost-driven method for the improvement of runtime testability based on RTM, a measurement for the runtime testability of a component-based system based solely on characteristics of the system under test. RiTMO enables integration engineers to identify critical situations of bad system runtime testability and to compute and execute an action plan to improve it. The next step in the development of our work is the integration of RTM and RiTMO into a CASE tool, enabling system engineers to receive timely feedback about the system they are designing.

Future work will also focus on an improvement of the RTM measurement itself, as well as the development of automated or semi-automated methods for performing the impact analysis and fix cost estimation. 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.

References

[1] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1990.

[2] 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.

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

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

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

[7] 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.

[8] 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.

[9] A. González, É. Piel, and H.-G. Gross. A model for the measurement of the runtime testability of component-based systems. In *5th Workshop on Advances in Model Based Testing (A-MOST 2009)*, Denver, USA, Apr. 2009. IEEE Computer Society (to appear).

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

[11] H.-G. Gross. *Component-Based Software Testing with UML*. Springer, Heidelberg, 2005.

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

[13] G. Huang, H. Mei, and F.-Q. Yang. Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Engg.*, 13(2):257–281, 2006.

[14] 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.

[15] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007.

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

[17] 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)*, pages 461–467. ACM Press, Mar. 2009.

[18] 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.

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

[20] 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*, pages 222–232, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

TUD-SERG-2009-016
ISSN 1872-5392