Delft University of Technology
Software Engineering Research Group
Technical Report Series

Reducing the Runtime Acceptance Costs of Large-Scale Distributed Component-Based Systems

Alberto González, Éric Piel and Hans-Gerhard Gross

Report TUD-SERG-2008-015

TUD-SERG-2008-015

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: Submitted for review at CBSE2008 at CompArch 2008 conference

© copyright 2008, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

Reducing the Runtime Acceptance Costs of Large-Scale Distributed Component-Based Systems *

Alberto González, Éric Piel, and Hans-Gerhard Gross

Software Technology Department, Delft University of Technology
Mekelweg 4, 2628CD Delft, The Netherlands
{a.gonzalezsanchez,e.a.b.piel,h.g.gross}@tudelft.nl

Abstract. Software Systems of Systems (SoS) are large-scale distributed component-based systems in which the individual components are elaborate and complex systems in their own right. Distinguishing characteristics are their short expected integration and deployment time, and the need to modify their architecture at runtime, while preserving the integrity of the system.
Integration testing is a commonly used technique employed in the acceptance processes of software SoS. In this paper, we propose a scheme to test a complete SoS at every reconfiguration, re-exercising the test cases of every updated component. In practice, re-executing all the test cases, whenever a modification takes place in one of the components, would be very costly. This is the case, in particular, when the system has to keep running all the time. Our proposal, therefore, encompasses several methods to limit the amount of test cases to be executed. The basis of all these methods is to rely on as much information as possible extracted from previous runs of the test cases. We illustrate our findings with an example SoS coming from the maritime safety and security domain.

1 Introduction

Maritime Safety and Security Systems of Systems (MSS SoS) represent a novel kind of large-scale distributed component-based systems, in which the individual components are elaborate and complex systems in their own right. They support authorities of states with their daily business of sensing issues at sea, understanding them, and disseminating forces for checking the situation, and, if required, intervening in critical situations. Typical tasks are collision detection and avoidance, offshore platform protection, shipping lane surveillance, deballasting dangerous items, harbour protection, pollution detection and prevention, submarine detection, intrusion detection and prevention, assistance for vessels,

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

search and rescue, and prevention of illegal activities such as illegal immigration, drug trafficking, piracy or terrorism [1].

In order to being able to support these tasks, MSS SoS are made of a range of technical sub-systems, such as coastal radar systems, and radar systems on patrol vessels, surveillance camera systems (on land, and at sea), surveillance aircrafts, and long range identification and tracking systems in satellites, satellite linking systems, meteorological stations, sonar systems, and AIS base systems for vessel identification, tracking and management, to name only a few. All these sub-systems, which are complex systems in their own right with their own operational and managerial issues, have to be integrated into a single combined operational entity, thereby forming a large System of Systems [2].

In this paper, we look at the acceptance processes that will be applied when first deploying such Systems of Systems, as well as when they have to evolve during runtime. The integration and acceptance processes deal with components from various sources and varying degrees of knowledge about their function and quality. In particular, we propose improvements to these methods that will take advantage of a specially built-in infrastructure which will allow a higher degree of integration and acceptance automation, and help reduce the costs of these operations, through limiting the amount of integration testing to be carried out.

The paper is structured as follows. Section 2 explains what characteristics of the development of large-scale Maritime Safety and Security SoS pose considerable software engineering challenges and describes the challenges we concentrate upon. Section 3 presents relevant related work. In Sect. 4 we propose a process in order to provide acceptance at the scale of the whole SoS. The usage of those processes are presented on an example MSS system. Section 5 describes our proposal for reducing the cost of the acceptance process whenever the system evolves. Finally, Sect. 6 summarizes and concludes the paper.

2 Challenges in the Acceptance of Large-scale Component-based Systems

The challenges presented by the integration of large-scale systems involve software engineering best practices for developing large-scale distributed component-based applications. These challenges entail methods, techniques, and tools for testing, monitoring and diagnosis. They also require work-flow modeling and realization for integration and acceptance processes, as well as automatic adapter generation for sub-system integration.

2.1 Specific Challenges of Systems of Systems

The challenges engineers are facing when building MSS SoS are, to a large extent, concerned with devising appropriate integration and acceptance strategies, given the large number of different components contributing to such a system. MSS SoS must achieve information interoperability among a wide range of heterogeneous sources. This must be done while maintaining constant operational

readiness, monitoring the system healthiness, while assuring the independence of its components. The dynamic nature of MSS SoS requires that components may be integrated in a brief period of time, shortly before deployment, i.e., for a maritime peace keeping mission, and thatHowever, it is possible to limit the amount of test cases to be executed for the second phase, the runtime testing phase, selecting only a part of the tests to be performed, by re-using the results obtained from the previous runs. The following subsections describe the two phases in detail, plus some complementary methods that permit to reduce the number of test cases to be executed. components may be removed, updated, replaced, or added during deployment, without threatening the integrity and quality of service of the overall system. In order to perform runtime integration of collaborating systems, the involved acceptance processes have to be able to deal with lack of total control over the operational and managerial aspects of the participating components, characteristic of Systems of Systems.

2.2 System Acceptance and Component Acceptance

System Acceptance refers to the process of checking the criteria that the system integrator and customer have agreed to be used to validate the system, and assess the compliance of the system with the customers' requirements. In a similar fashion, we refer to *Component Acceptance* as the process of assuring that a component included in an SoS environment will operate properly, according to both, the component's and the system's specifications.

The system and component acceptance criteria are employed in live processes that involve work before the system is deployed (development-time acceptance), during first deployment (integration-time acceptance), and during runtime (runtime and evolution acceptance). However, in this paper we concentrate on runtime deployment and on architecture evolution. More specifically, we focus on the usage of *testing* as the primary acceptance criterion. Central to this problem is the question of how we can ascertain that every reconfiguration of the system maintains the same level of certainty as during development time, assessed according to predefined test adequacy criteria. Here, questions to be asked are "what parts of the system should be re-checked", "how can we minimize the number of tests to be re-executed", and "how do we make sure that the tests that are left out would not have found any more errors", whenever the system is modified.

3 Related Work and Technologies Required

There is an active research community addressing the main topics of interest related to integrating and accepting parts of component-based applications. The following paragraphs provide a brief overview of related work relevant to the subject of this paper.

The Component Trust Problem. When connecting a foreign component into a component-based system, the problem of transferring the knowledge about the component's quality and expected behaviour from its developers to its potential users is known as the *component trust problem*. Morris *et al.* propose a framework for component certification through the developers [3], in contrast to having external component certification bodies issue certificates. They claim that components should provide standardized specifications plus test descriptions that have been applied by the component developers in their development environment. This provides an indication of the kind of quality assurance performed for system integrators. Gross *et al.* [4] take this idea of self-certification further, and propose to use test suites from the producer of a component and execute them in the new context of the system integrator (the component customer).

Built-in Integration Testing. Built-in testing (BIT) refers to techniques used for equipping components with the ability to check their execution environment, and their ability to be checked by their execution environment [5, 6], during runtime. BIT is typically implemented in terms of additional software artifacts permanently built into the components or their underlying runtime platform. They enhance the testability of a component. Among these artifacts are built-in testing interfaces, or built-in tester components that perform runtime tests of an assembly of components. Tester components may be invoked before deployment when a system is assembled, or during system updates, in order to verify the new configuration. Built-in testing also includes techniques to monitor the behaviour of components at runtime [7]. That way, components can perform much of the required system validation effort automatically and by "themselves" [8].

Architecture Evolution in Component-Based Systems. Because of the inherent dynamic nature of MSS Systems of Systems, they will be subject to frequent architectural changes [9] as their components evolve, join and leave. The MSS SoS architecture has to support safe run-time reconfigurations. Before and after each reconfiguration, the consistency of the system has to be checked. Matevska-Meyer and Hasselbring [10] propose a method that relies on a "consistency manager", which could be combined with the idea of "multi-versioning connectors" [11], and a test request execution and isolation strategy based on BIT [12] to avoid interfering with the business functionality of already existing components.

Regression Testing. The regression testing method will play an important role as primary technique used to re-evaluate acceptance criteria for every (runtime) modification of the MSS SoS architecture. Approaches that require access to the source code cannot be applied in an MSS SoS context, since source code of the components will hardly ever be available. Although model-based approaches can be a solution [13, 14], the complexity of the models of the components to be integrated, can often amount to intractable resulting combined models. A solution that can be used when a model is not available, or the available model is too complex, is to dynamically derive state models from usage traces [15]. As an advantage, these models will be smaller, as they will be restricted to the way

the component has been used earlier in the SoS, thus, leaving out all the features of the component not relevant in the current context. There exist also some methods [16], that address this problem from a formal point of view, providing a way to measure system updates and finding conflict-free configurations.

4 Integration Testing of Systems of Systems

As stated in Sect. 2, we will employ testing as our mean of providing acceptance throughout the main stages in the life-cycle of an MSS. In this section we will present our proposed acceptance process and the infrastructure that is required in order to make it possible. Furthermore, we will apply this process to an example MSS system taken from the vessel traffic management domain.

4.1 Hierarchical Organisation of the SoS

Typically, an SoS is organized as a hierarchical composite structure, in which components are coordinated and associated into a composite, in which they perform mutual tasks and share information and services. These composites take part in bigger composites, and so on. As a concrete example of such a composite structure, we use a vessel tracking system comprised of two sub-systems: ship's AIS identification (signals that all ships provide themselves) coming from the coastguard [17], and radar information, coming from the port authorities. Fig. 1 displays the structure of this system. It is organized in terms of three hierarchical components: the main *MSS system* on the top level, the port authority, and the coastguard. On the MSS level, a *Track Fusion* component processes the information provided by the radar and AIS services. From this information it builds a shared picture of ship tracks which are displayed on the *Alert Screen*. This shared picture can be used to coordinate the work of both organisations.

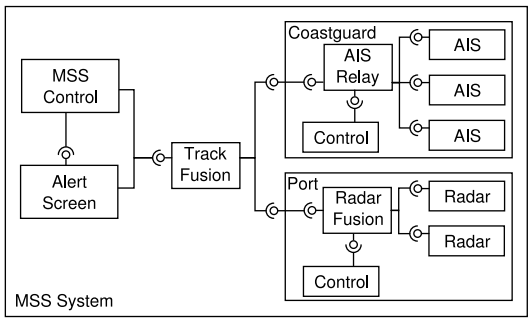

Fig. 1. Structure of our example MSS System.

By using this example, we will illustrate our proposed process to verify the acceptance of the whole system with respect to integration testing as primary acceptance criterion.

4.2 An Infrastructure for Integration Testing

In order to test the integration of an SoS, it is necessary, first, to be able to test the integration of two components together. The basis of our process is that sets of interacting components can check themselves for proper for integration. More specific, in [18], a framework was devised, in which each component can have a set of test cases associated with it, in order to check that the components on which it relies behave as expected. This technique, we call it *provider acceptance*, applies the principles of built-in testing (BIT), according to [5]. It introduces two special interfaces (called *controllers*) to access the built-in testing features:

- The **Acceptance controller** through which test cases can be listed or executed for a particular *required* interface.
- The **Testing controller** through which the necessary functions are provided for manipulating a *component under test*. This is required, especially, in the context of runtime testing, where the component must be tested for a future configuration, at the same time, as it must continue to operate in its current configuration.

Our proposed process relies on a second, complimentary technique which we call *composition acceptance*. It assigns to each composite component its own set of built-in test cases, that perform the validation of the integration of the sub-components contained in a composite component. This is required for checking interactions that can only be seen from the composition point of view. For example, if a signal is sent through another binding as a result of an operation. These techniques will help to distribute the responsibility of the component acceptance to the parts that are concerned: to the client/server level and to the composite/sub-component level. Figure 2 represents the system's architecture with the BIT controllers added. The TC interface corresponds to the testing controller, while the AC interface corresponds to the acceptance controller. In terms of implementation, the AC is, in fact, a normal component, as it contains a test suite, and it can be bound to other components. That is why it is represented as a small component in Fig. 2. We have also indicated a tester component in Fig. 2, that is *Radar Fusion Tester*. However, typically there are more than only one tester components. In this configuration, the tester component is used to perform the provider acceptance of one instance of *Radar* through *Radar Fusion*.

Based on these techniques, the system acceptance process can be devised. It is sub-divided into two phases:

- An initial phase, which permits to assure that the original configuration is acceptable.

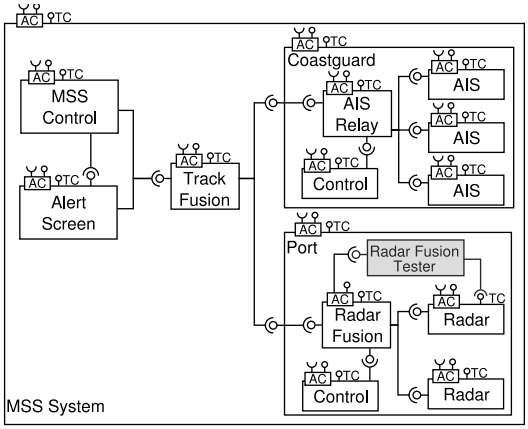

Fig. 2. Structure of our example MSS System with built-in testing controllers and one tester component represented.

– A reconfiguration phase, which validates every modification done to the running system.

The first phase cannot be optimized in terms of testing effort, because the whole system has to be tested completely. This means that all (built-in) unit and integration tests from the development phase of the system will be performed.

Although the method we propose for system evolution already limits the parts of the system that are re-checked, it is possible to limit the amount of test cases to be executed even further by re-using the results obtained from the previous runs. The following subsections describe the two phases in detail, leaving the complementary methods that permit to reduce the number of test cases to be executed for Sect. 5.

4.3 First System Deployment

When deploying the system for the first time, we must ensure that the integrated components accept the context they are being deployed in, and that the system as a whole is acceptable. We perform integration-time system and component acceptance by using built-in tests in the components. As mentioned earlier, this is performed

– by executing test cases contained in one component on all the other components to which it is associated, and on which it relies (down and across the compositional graph), and
– by executing test cases on all composite components (up the compositional graph).

This way, every component of the SoS is able to check its direct environment according to its own acceptance criteria. In the case of composite components,

their built-in tests will check the part of the system acceptance that concerns the composition of the components they contain.

It should be noted, that this phase usually happens at first deployment, when all the components can be executed but the system is not yet operating. However this is not a requirement. In particular, some other parts (sub-components) of the SoS could, in fact, already be running while the SoS as a whole is deployed for the first time. The framework supporting the testing has to handle this kind of integration testing at runtime. For instance, in our MSS example, the *Coastguard* component and the *Port* component are already running and cannot be stopped when the complete MSS system is integrated and deployed for the first time. In this case, the initial testing phase will take place while these two sub-components are already operating normally.

In this phase of acceptance, the acceptance process executes every single test case available. The order of tester components executed is not relevant. In particular, it must be possible to run concurrent tests on several components, as long as there is no dependency between the tested components (neither on the client/server level nor on the composite/sub-component level). However, for diagnosing the errors found, with as much detail as possible, our proposed acceptance process is carried out in a constructive, bottom-up way, guided by the dependencies between the components. In other words, the acceptance process starts by testing the components which have no *requires* dependencies and which are leaves in the compositional tree. Next, those components are tested, whose dependencies have been tested already, and so on. Complete composite components are tested only when all their sub-components have already performed their acceptance tests. This process is repeated, until the root is tested, and the entire MSS system can be accepted. It is important to note, that this order cannot always be respected: there might be cyclic dependencies between the components. In this case, one of the components will have to be selected first for testing. Here, a problem is that it is not possible to determine exactly in which component the acceptance failed.

For example, the deployment-time acceptance process of our MSS system performs first the provider acceptance of the three *AIS* components, the *AIS Relay*, and then the *Control* components of *Coastguard*. Then, the composition acceptance of *Coastguard* will be performed. The process for *Port* is done in the same way, and simultaneously. Finally, the provider acceptance of *Coastguard*, *Port*, *Track Fusion*, *Alert Screen* and *MSS Control* (in that order) will be performed, followed by the composition acceptance of the whole *MSS System* component.

If the component acceptance fails, the system architect will have to amend the part where the error has been detected, or take the decision to force the integration of the component, with a warning, or in a degraded mode. This depends on the trade-off between the thoroughness of the acceptance criteria applied, and the requirements for a prompt deployment. If the surrounding components do not depend on the degraded functionality, this may not even affect their or the system's acceptance.

4.4 System Runtime Evolution

MSS Systems of Systems are dynamic in nature. Components will leave the system and others will join. Therefore, we must ensure that after each runtime modification [9], the acceptance criteria for the whole system are still satisfied.

Modifications range from simple alterations, such as changing bindings without adding or removing any component, to more complex ones where multiple components are added and removed, and numerous bindings are affected at the same time. Modifications on the acceptance contracts themselves must also be taken into account. We impose the restriction that reconfigurations are treated in an atomic way, that is, they have to be accepted/rejected as a whole, not only the components whose acceptances have passed/failed.

Reconfigurations occur at runtime. Furthermore, the typical systems under consideration cannot be stopped as a whole, not even for a short period of time. Therefore it is a requirement that all the test cases can be executed while the current configuration stays operational. The exact way how this is achieved, is out of the scope of this article, but we can mention that either this will be handled by the testing framework (by using sandboxing for instance), or explicitly by the component under test (by using the testing controller to determine their testing capabilities) [12].

System evolution involves three steps:

1. Perform the acceptance of the new components, if any;
2. Determine which of the other participating components are affected;
3. Re-evaluate the acceptance of the affected components.

The re-evaluation of acceptance criteria in the affected components is performed bottom-up, in the same way as it was done during first deployment.

In order to determine the components affected by a reconfiguration, it is necessary to look at the components' *requires* dependencies. These are the bindings between all other associated client and server connections in the architecture of the SoS. Every component is considered as being affected by runtime evolution, that

- has one of its bindings modified,
- has a dependency with a modified component,
- has a dependency with an affected component,
- contains an affected or a modified component.

Please note, that this is recursive.

For each of the affected components the acceptance procedures are then executed. The execution order of the tests is the same as the dependencies browsing: starting from the modified component, going through each dependant component, until the topmost level is reached. Figure 3 shows this process for one modification of our example MSS SoS. The system architect requests the modification of the AIS component indicated by a star (e.g., a version upgrade). The dotted curve shows the detection of the affected components. Black circles are drawn on components for which provider acceptance will be performed, while

black squares are drawn on components for which composition acceptance will be performed.

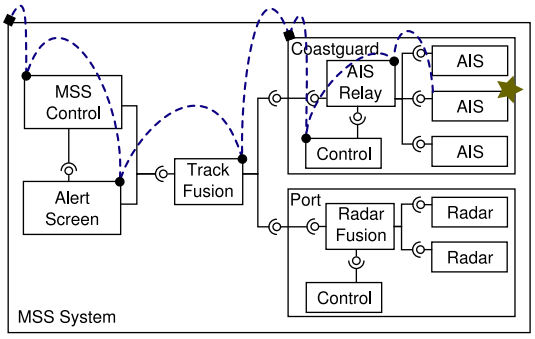

Fig. 3. Acceptance re-checking steps triggered by the replacement of an AIS component.

5 Reducing the Cost of Acceptance

As an implication of the testing sequence introduced in the previous subsection, whenever a modification occurs in Fig. 3, it is necessary to re-run a significant part of the acceptance process. This is not acceptable for runtime component updates, especially in the context of large-scale systems.

5.1 Reducing the Number of Components to Test

Reducing the cost of the acceptance process at system modification can be achieved through limiting the number of affected components to re-test. Constraining the scope of redoing the acceptance process relies on two fundamental observations on the properties of MSS SoS:

1. The fact, that MSS are comprised of loosely coupled components. This constrains their inter-dependencies considerably.
2. The knowledge, that in the current configuration of the system, every component has passed its tests.

The hierarchical, composite structure of the system will act as a containment barrier for repeating unnecessary acceptance processes to be performed during runtime, thus, reducing the scope and amount of acceptance checks that have to be done after dynamic system updates.

When devising the list of components affected by a dynamic update, for every composite component it must be determined if *executing test cases from the upper components will permit the detection of faults that cannot be detected by*

the test cases provided by the lower-level components. How this can be done automatically is still an open research question. Nevertheless, it is possible to refer to one obvious and meaningful condition: if the test cases for the composition acceptance have 100% specification coverage of the component under consideration, and they have all passed, then, any more testing on a higher level of the compositional hierarchy will be redundant. In a general automated scenario, the function to determine whether higher-level testing may be redundant, would have to consider four parameters:

- the model of the modification of the system,
- the specification coverage criterion,
- the test cases already performed on the lower level, and
- the test cases of the higher levels.

This function would return true or false, depending on whether or not repeating the acceptance process can be stopped on a given hierarchical level, or not. This function could be used to indicate also whether new test cases are needed in the current level to avoid upwards propagation of the acceptance process. Fig. 4 shows the acceptance process on the MSS SoS example with this method in use. As in Fig. 3, one AIS component marked by a star is modified, and the first affected component, in this case *AIS Relay*, is, therefore, identical. However reaching the Coastguard component, the function determines that the test cases from the higher components can be omitted, thus, constraining the number of components to be tested to the *Coastguard* subsystem.

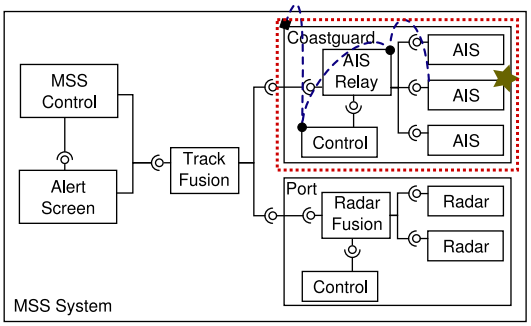

Fig. 4. Reduction of the number of components to re-test by pruning the highest levels of hierarchy from the integration acceptance process.

5.2 Reducing the Number of Test Cases

The cost of acceptance when the system evolves can also be reduced by minimizing the number of test cases to be executed per built-in tester component. The basis of this approach is to apply regression testing (as introduced in Sect. 3),

although, on the component level. Typically in an SoS, every elementary component is a black-box, the source code is not available. The finest level of detail, which can be managed by the runtime framework is, therefore, the elementary component. Typical code-based test criteria may not be applied in our case. Test case selection techniques for regression testing are readily available in the literature, e.g. [19].

During the initial phase of acceptance (and previous evolutions of the system), traces for each test case are recorded. A trace permits to associate a given test case with the components and bindings exercised during execution. When performing the integration acceptance of the new system's configuration, only the test cases which exercise the modified components have to be run again. This applies to both, provider and composition acceptance test cases.

Figure 5 illustrates this method. The provider acceptance is performed for *Track Fusion* on *AIS Provider*. The diagram on the left-hand side shows the traces of three example test cases associated with *Track Fusion*. They were obtained by the component runtime framework on first deployment of the system. One of the test cases exercises *AIS Relay* and *AIS* only, another one exercises all the three sub-components, and the third test case involves only *AIS Relay* and *Filter*. The diagram on the right-hand side depicts the test case reduction when *Filter* is modified. Only the second and third test cases are required to be rerun.

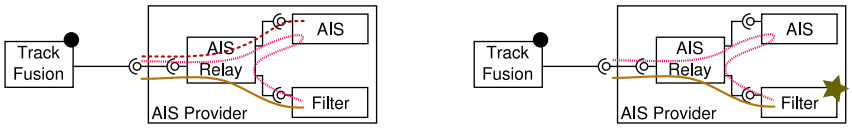

Fig. 5. Reduction of the test cases to run by pruning those not involved in testing the modified components.

6 Summary, Conclusions and Future Work

Every modification of a part of an MSS SoS may cause high runtime costs for repeating the acceptance processes. If the system has to stay operational at all times, or if the reconfiguration has to be done quickly, this cost, has to be minimized while maintaining the same kind of acceptance assurance on the newly integrated systems as has been achieved at first deployment. The integration acceptance process we proposed in this article relies on built-in provider acceptance and composition acceptance in order to achieve this goal.

The first contribution of this paper is the distribution of the integration acceptance procedures of the entire SoS over its sub-components, and re-evaluation when the architecture changes. By using BIT technology, it is possible to keep

the test cases synchronized with the current components constituting the system. Moreover, from the chain of related components affected by a modification, it is directly possible to constrain the set of test cases to be re-executed, to the test cases contained in this particular chain.

Another contribution of the paper is the proposition to limit the propagation of the acceptance process upwards in the composite hierarchy without compromising the assurance on the correctness of the system. By determining whether the test cases from the higher-level components are able to detect faults that could not have been detected by the lower-level tester components, decision can be made to propagate further up, or to stop the acceptance process. In case that the system architect want the propagation to stop on a certain level of hierarchy, while the test cases do not provide enough coverage compared to the higher level, additional test cases may have to be devised. In such a case, either the system architect is requested to provide more test cases, or the framework can automatically generate test cases out of the model of system.

Reduction of the integration acceptance cost can also be achieved by reducing the number of test cases to run. By observing the execution of every test cases at deployment time, it is possible to have precise knowledge of which component is exercised during which test case. When a modification happens on the system, only the test cases involving precisely the modified components have to be re-executed.

A challenging question for future work is: "How can we determine automatically, whether higher-level test cases can detect defects that have not been detected by the test cases run so far?" When the test cases already reach full coverage, it is easily possible to omit the other higher-level tests. In practice, full coverage is difficult to achieve. The higher level components use only a part of the functionalities provided by the component under test, so, logically, their test cases only cover this limited part. If this part of the functionalities has been 100% covered by the previous test cases, the acceptance process can stop. Therefore, one way to answer the question is to get into the position to compare the parts covered by two sets of test cases and determine if one is included in the other.

As future work, we are also planning to implement and automate these proposed processes on the Fractal [20] component platform, so that it can be integrated with the reconfiguration tools we have so far. In particular it will be interesting to evaluate to which extent acceptance process can be distributed over the available components. In the context of MSS SoS, the fact that each component requests by itself the test cases to be run would avoid having to share the information of the component's architecture with the rest of the system (a security requirement). That way, we could define the acceptance criteria to be independent from the various sub-systems (needed to keep freedom on the management of each sub-system).

Moreover, a study of the applicability of these techniques to a publish-subscribe data distribution service is also planned. This kind of runtime system presents its own challenges with respect to integration testing. In particular the

dependencies between each component are not explicit, each component receiving or sending a particular type of data. In addition, the communication between the components are always asynchronous, which leads to difficulties in defining the test cases.

References

1. Thales Group: Maritime safety and security. http://shield.thalesgroup.com/offering/port_maritime.php (2007)
2. Maier, M.W.: Architecting principles for systems-of-systems. Systems Engineering **1**(4) (1998) 267–284
3. Morris, J., Lee, G., Parker, K., Bundell, G.A., Lam, C.P.: Software component certification. Computer **34**(9) (2001) 30–36
4. Gross, H.G., Melideo, M., Sillitti, A.: Self-certification and trust in component procurement. Science of Computer Programming **56**(1–2) (April 2005) 141–156
5. Gross, H.G., Mayer, N.: Built-in contract testing in component integration testing. Electronic Notes in Theoretical Computer Science **82**(6) (2004) 22–32
6. Gross, H.G.: Component-Based Software Testing with UML. Springer, Heidelberg (2005)
7. Deveaux, D., Collet, P.: Specification of a contract based built-in test framework for fractal (2006)
8. Brenner, D., Atkinson, C., Malaka, R., Merdes, M., Paech, B., Suliman, D.: Reducing verification effort in component-based software engineering through built-in testing. Information System Frontiers **9**(2–3) (2007) 151–162
9. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE '98: Proceedings of the 20th international conference on Software engineering, Washington, DC, USA, IEEE Computer Society (1998) 177–186
10. Matevska-Meyer, J., Hasselbring, W.: Enabling reconfiguration of component-based systems at runtime. In van Gurp, J., Bosch, J., eds.: Proceedings of Workshop on Software Variability Management, University of Groningen (February 2003) 123–125
11. Rakic, M., Medvidovic, N.: Increasing the confidence in off-the-shelf components: a software connector-based approach. In: SSR '01: Proceedings of the 2001 symposium on Software reusability, New York, NY, USA, ACM (2001) 11–18
12. Suliman, D., Paech, B., Borner, L., Atkinson, C., Brenner, D., Merdes, M., Malaka, R.: The morabit approach to runtime component testing. In: 30th Annual International Computer Software and Applications Conference, 2006. COMPSAC '06. Volume 2. (Sept. 2006) 171–176
13. Orso, A., Do, H., Rothermel, G., Harrold, M.J., Rosenblum, D.S.: Using component metadata to regression test component-based software. Software Testing, Verification and Reliability **17**(2) (2007) 61–94
14. Muccini, H., Dias, M., Richardson, D.J.: Reasoning about software architecture-based regression testing through a case study. In: COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 2, Washington, DC, USA, IEEE Computer Society (2005) 189–195
15. Mariani, L., Papagiannakis, S., Pezze, M.: Compatibility and regression testing of cots-component-based software. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2007) 85–95

16. Stuckenholz, A., Zwintzscher, O.: Compatible component upgrades through smart component swapping. In Reussner, R.H., Stafford, J.A., Szyperski, C.A., eds.: Architecting Systems with Trustworthy Components. Volume 3938 of Lecture Notes in Computer Science., Springer (2004) 216–226
17. International Telecommunication Union: Recommendation ITU-R M.1371-1. technical characteristics for a universal shipborne automatic shipborne automatic identification system using time division multiple access in the VHF maritime mobile band (2001)
18. Gonzalez, A., Piel, E., Gross, H.G.: Runtime integration and acceptance of distributed maritime safety and security systems. Technical Report TUD-SERG-2008-007, Delft University of Technolgy, Software Engineering Research Group (2008)
19. Graves, T.L., Harrold, M.J., Kim, J.M., Porter, A., Rothermel, G.: An empirical study of regression test selection techniques. ACM Transactions on Software Engineering and Methodology (TOSEM) **10**(2) (April 2001) 184–208
20. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: An open component model and its support in java. In Crnkovic, I., Stafford, J.A., Schmidt, H.W., Wallnau, K.C., eds.: CBSE. Volume 3054 of Lecture Notes in Computer Science., Springer (2004) 7–22

TUD-SERG-2008-015
ISSN 1872-5392