Architecture Support for Runtime Integration and Verification of Component-based Systems of Systems

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

Abstract

Systems-of-Systems (SoS) represent a novel kind of system, for which runtime evolution is a key requirement, as components join and leave during runtime. Current component integration and verification techniques are not enough in such a dynamic environment. In this paper we present ATLAS*, an architectural framework that enables the runtime integration and verification of a system, based on the built-in test paradigm.* ATLAS *augments components with two specific interfaces to add and remove tests, and to provide adequate testability features to run these tests. To illustrate our approach, we present a case study of a dynamic reconfiguration scenario of components, in the Maritime Safety and Security domain, using our implementation of* ATLAS *for the Fractal component model. We demonstrate that built-in testing can be extended beyond development-time component integration testing, to support runtime reconfiguration and verification of component-based systems.*

1. Introduction

The commission of the European communities has recently pushed for the establishment of a European Network for Maritime Surveillance [6]. Such a network will provide safe and secure usage of the seas around Europe, providing border control, law enforcement assistance, and detection of maritime pollution, and illegal activities. This will need the cooperation and coordination between the concerned Member States' security agencies, and an efficient usage and integration of already existing systems.

This new kind of large-scale component-based system, in which the components have an operational entity of their own, and usually a managerial entity as well, is known as "System-of-Systems" (SoS) [14]. SoS present considerable engineering challenges that have been acknowledged by the Dutch Embedded Systems Institute and Thales Nederland. They have set up the Poseidon research project [5], committed to devising engineering best practices for developing, integrating and deploying such maritime safety and security (MSS) systems. Current approaches for system integration and testing are mainly static, inappropriate for the highly dynamic nature of MSS SoS, where components join and leave, and where even the system's requirements evolve at runtime.

In this paper we present an architectural framework, called ATLAS, based on the paradigm of Built-In Testing [8] (BIT). ATLAS allows the SoS integrator to add and remove test cases from components at runtime, and whenever an architectural change happens, inform all the potentially affected components that their execution context must be rechecked. We have implemented ATLAS over Fractal [3], a component model with dynamic, introspective capabilities. To realize ATLAS' distinguishing, dynamic features, we extend the Fractal component model with two extra component interfaces. Furthermore, we have created an adapter to run integration tests written for JUnit in the ATLAS framework, so that runtime test cases can be easily defined. To illustrate our approach, we present a case study based on the context of MSS systems, demonstrating and assessing how well runtime testing can be performed during dynamic reconfiguration, i.e. joining and leaving of components.

The paper is structured as follows. In Section 2 we study the challenges of SoS. Section 3 outlines relevant related work to our research. Section 4 introduces and describes ATLAS, and its implementation. Section 5 illustrates how ATLAS is used in a current MSS implementation, discussing the initial solution and its limitations. Finally, section 6 summarizes and concludes the paper.

2. The Challenges of MSS Systems

Runtime integration and verification strategies are amongst the most obvious challenges in building and evolving large-scale MSS SoS, given the large number of different systems contributing to them. Most sub-systems in an SoS have operational and managerial independence [7, 14]. This implies that parts of the SoS may be changed without the SoS integrator having too much to say in the decision. In some cases, a sub-system might not even provide detailed information about the modification (due to political or business reasons). Even in such cases, the integrity of the entire SoS must still be guaranteed.

The fact that MSS SoS evolve dynamically during operation time also brings implications for quality assurance, in particular for testing. Systems can join or leave the SoS, meaning that offered services may vary in terms of functionality, as well as quality. When a sub-system joins or leaves the SoS, the other sub-systems may have to be reconfigured to take advantage of new services and improved quality of service, or they may have to be notified that services are degraded. This process should be mostly seamless for the system operators and should be executed within a short time, without any major disruption of the rest of the SoS.

The tests used to verify the system *have to evolve simultaneously as the SoS evolves*. In particular, functionalities of a component which were not exercised in the initial configuration of the SoS may be required by components inserted at runtime. These functionalities have to be tested before being used, even though no tests were originally provided to verify them. Moreover, when a new version of a component is created to fix a misbehaviour, tests to detect the misbehaviour might be introduced at the same time to ensure no regression happens in the future. Verification techniques need not to be restricted to testing, nor to a fixed set of techniques. Therefore, the platform must support *different types of verification techniques* (static contracts, monitoring of resources, etc.), and *dynamic insertion and removal* of these, in the same way it supports joining and leaving of components.

By their very nature, SoS are large-scale systems, with a large number of components, contained in the sub-systems. After each reconfiguration, the integration of the SoS has to be verified again. It is, therefore, necessary to devise an appropriate verification strategy that not only achieves this goal, but also *minimizes* the cost of checking after each modification. Re-checking must be as little disruptive as possible for the running configuration and the latency between the moment a reconfiguration is requested and the moment it is accepted and deployed must be minimal.

Due to the huge size of the SoS, the limited access to the system's code or executables, the need to keep SoS always available, or the fact that some components use resources that cannot be duplicated, (a) testing will have to be executed concurrently with the working configuration, and (b) some component instances will be shared between the tested and the working configurations. Therefore, runtime testing [19], the ability to test a component while it is also performing normal work, is the only realistic option when verifying an MSS SoS at operation time.

3. Related Work

To our best knowledge, there is no visible research being carried out specifically addressing the domain of MSS Systems-of-Systems. However, there has been an active research community addressing the main topics of interest related to integrating and verifying component-based systems, since the publication of Weyuker's landmark article on component testing [20].

Built-In Testing is an important paradigm, key in understanding our approach in the following sections. BIT refers to any technique used for equipping components with the ability to check their execution environment, and their ability to be checked by their execution environment [8], during runtime. These built-in tests may be invoked before deployment when a system is assembled, or during system updates, when existing components are replaced or new components added, so that the pair-wise client-server relations can be assessed. That way, the components can perform much of the required system verification effort automatically and by "themselves" [2]. This distribution of the responsibility of verifying the component's environment to the components themselves is very interesting for MSS SoS. It can help us to maintain the independence of each of the participating systems.

Having components carry information useful for verifying the context they are being deployed in, has already been proposed [4, 9, 16], and has been extended to perform this verification at runtime [18]. However, these approaches are static, and do not allow for redefinition or evolution of the tests if required, or are mostly focused on monitoring. With ATLAS, we tackle the *dynamic evolution* of test cases, and the verification of these on the running system. Moreover, although the main focus of our contribution is to be able to detect integration problems between a component and the other components *before* it is used in the system, ATLAS can also accommodate monitoring, as it is an integral part of the integration verification process.

In the line of dedicated infrastructure to automate integration testing, research work has also been demonstrated before [1, 12].

4. The ATLAS Framework

In order to address the challenges of run-time integration of verification presented in Section 2, we have devised the ATLAS component verification framework. Before testing the integration of a SoS, it is necessary, first, to be able to test the integration of two components together. Therefore, the basis of ATLAS is that interacting components will check their peers for proper for integration. Each component in ATLAS has a set of test cases associated with it for this purpose.

This technique, we call it *provider acceptance*, applies the principles of built-in testing, according to [8]. More specifically, we assume that every component in our system provides two additional built-in interfaces that are used for verification purposes:

- A **Testing Interface** through which the necessary functions are provided for manipulating a *component under test*.
- An **Acceptance Interface** through which test cases can be listed, managed and their results invalidated for a particular component. This interface is an extension of traditional BIT [8].

4.1. Artefacts of Atlas

As defined in BIT, components have a *Testing* port that offers functionality for querying and exercising a component's testability features through the `TestingController` interface [8]. 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 continues to operate in its current configuration. However, in ATLAS, the tests are not permanently built into the components. Components in the ATLAS framework also provide an *Acceptance* port with an `AcceptanceController` that offers enhanced capabilities over BIT. The acceptance controller has two roles, (a) dynamic addition and removal of the tests that the component will use to verify the context it is deployed in, and (b) receiving notifications informing that a part of the architecture has been changed and needs re-checking. Figure 1 shows the basic UML representation of an ATLAS component, with a port for testing and a port for acceptance. The normal service ports and interfaces (i.e. the component's proper functionality) are component-dependant. They are labelled `Provided` and `Required`.

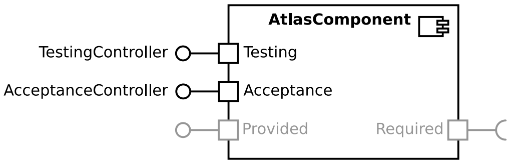


Figure 1. A generic UML Atlas component

In order to make the definition of tests as flexible and generic as possible, ATLAS also defines the concept of *Acceptance Providers*: special components dedicated to testing other components. Each acceptance provider is available for all the components in the system, and it is targeted to one specific type of test (e.g. JUnit, TTCN-3). What exactly an acceptance provider can test, depends on the technology it relies on. It can be designed to test functional (the outputs generated from specific inputs), as well as non-functional (e.g.: execution time, memory usage) properties.

The generic UML representation of an acceptance provider and test component is depicted in Figure 2. The testing port interface, is used for set up and clean up tasks in the component under test. The `Provider` port is designed to be accessed by the framework components that issue the tests. In our current implementation this is the task of the Management Console. No assumption is made neither about the way the provider and test components interact, nor how test components look like. In the figure we show a test component that provides and requires the same interfaces as the ATLAS component that requires this test (e.g. `AtlasComponent` in Fig. 1), and "impersonates" the component during testing. This corresponds to the way in which tests in the example in Section 5 are designed.

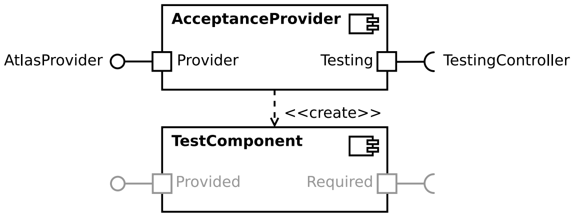


Figure 2. Atlas Provider in UML

The usual usage scenario starts after a reconfiguration, with a component that needs to check the components on which it depends. The system integrator will see this, and use the Management Console to issue requests to the appropriate acceptance providers. The providers will create the corresponding test component, set up the test environment, fire the test case, and collect the results when this has finished.

The tests run by acceptance providers can be hand-coded specifically for one component, or generated from component models and specifications. It is also possible to write or generate the tests in a notation dedicated to testing, such as TTCN-3 or the UML Testing Profile, and then have a generic tester component read these specifications and run the tests [10].

4.2. Interface Specifications

The most important extension of BIT in the ATLAS framework is the `AcceptanceController` interface. It defines three operations to query, add and remove test cases.

Associated to this interface definition are descriptions of test cases that have to be run when something changes. The descriptions contain information about what test component to use, results of past runs, and whether the test is pending, i.e. it has to be tested again or not.

It further defines two other operations: `isAccepted` and `notifyChange`. The first one returns true if no test case is pending, and all of them have passed in their last run. Calling the second operation informs the component that there has been a change in its context. A call to this operation invalidates all the test cases' results, setting them as pending.

The `AtlasProvider` interface is much simpler. It has only one operation: `check`, that receives as parameters the test case to be checked (which is provider-dependent), and the component that issues the test request (so that its context can be passed to the test component).

The `TestingController` provides a number of operations to let a component be aware of the testing process, so that testing does not interfere with the normal working of the component. In particular, it permits the component to tell whether a call originates from the working or the testing configuration. It also provides operations such as `begin` and `end` for setup and cleanup of the component's internal state.

4.3. Implementation in Fractal

An implementation of the ATLAS framework specification has been developed based on the Fractal component model [3]. Instead of implementing the acceptance and testability features directly as interfaces in the functional components, we have exploited the possibility to wrap components using "component membranes" offered by Fractal. Membranes encase components, adding special controller objects that can be used to manage infrastructural aspects of a component, such as binding, content and life cycle. Membranes provide a convenient way of adding our special ports to the Fractal component platform, so that component developers do not have to care about these additional aspects, consequently enabling a more clear separation between functionality and testability.

The current implementation of ATLAS takes into account the set of architectural modifications descbired in [17], to trigger notifications. This has been achieved by programming an extension of the behaviour of the standard Fractal *Binding* and *Content* controllers [3], so that they call `notifyChange`.

Currently, to keep the implementation simple, the component framework only allows modifications of the architecture if the components are in *stopped* state (paused). Therefore, to ensure all tests have passed in the current implementation, we do not allow components to transition to *started* state unless all of their associated test cases are marked as passed.

In this paper, we focus on devising and validating the architecture of ATLAS. The testing port of the ATLAS model, used to make the component test-aware during runtime testing, has not yet been fully implemented. For now, the components under test are duplicated by the acceptance provider so that the tests do not disrupt the working configuration. It should be possible to provide a user interface which displays in parallel the current production version and the modified version which still has to be fully validated.

ADL Extension. In order to allow an easy association of components to test cases, we extended Fractal's Architectural Description Language (ADL) parser to permit the association of test cases to components in a convenient way. Each of these associations is defined as an additional item in the component's ADL definition. An example of a component's description and associated test cases is presented in Listing 1. It shows how to associate test cases (the `test` tag), indicating which test infrastructure has to be used (the `provider` attribute) and which test definition has to be loaded by the provider (the `definition` attribute) In this example we are using an tester component called `MonitorTest`, which is supported by the JUnit acceptance provider explained in the following Section.

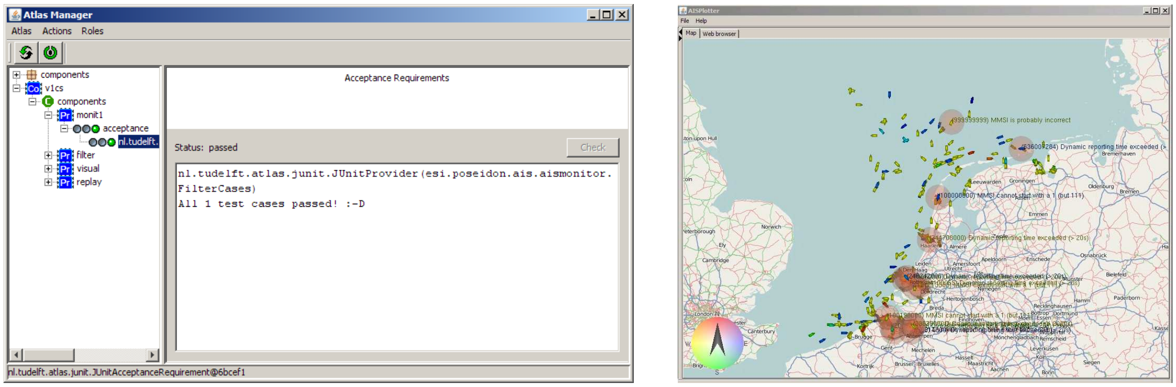
```
<definition name="Visualiser">
  <interface name="monitor" role="client"
    signature="AISMonitor" />
  <test provider="JUnit"
    definition="DupTest" />
  <test provider="JUnit"
    definition="MonitorTest" />
  <content class="Visualiser" />
</definition>
```

Listing 1. Associating of test cases in the ADL file

JUnit Provider. JUnit[1] is a well-known and widely supported testing framework. We have implemented an acceptance provider based on it, that adapts the way test cases are normally written for it. This adapts the way in which tests are run in JUnit, to the requirements of runtime integration testing.

Traditional JUnit tests are different in comparison with integration tests, in that they have to create their own test system, whereas in runtime testing, the test system is already running. Hence, a new type of test case was defined, that is able to access the running system. Note that, in the

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



Status: passed

```
nl.tudelft.atlas.junit.JUnitProvider(esi.poseidon.ais.aismonitor.
FilterCases)
All 1 test cases passed! :-D
```

nl.tudelft.atlas.junit.JUnitAcceptanceRequirement@6bcef1

(999999999) MMSI is probably incorrect

(636007284) Dynamic reporting time exceeded (>

(100000000) MMSI cannot start with a 1 (but 111)

(244706000) Dynamic reporting time exceeded (> 20s)

(100190000) MMSI cannot start with a 1 (but 111

(a) Management console

(b) AIS Visualiser screen

Figure 3. Screen captures of the Management Console and the example system interface.

current implementation, the system they access is a copy of the original one, created by the provider. By simulating how the component would use its context, the JUnit test is then able to do integration testing. The results obtained can be further analysed by tools cooperating with JUnit such as IBM's Eclipse [2].

Management Console. Finally, in order to provide a graphical overview of the acceptance status for all the components of the system, we have extended the Fractal management console to support querying the acceptance controller interface, and interpreting test results from JUnit. Our tool also provides a way to access all the aspects that are not yet automated, like instantiating the acceptance providers, querying the acceptance controllers and starting tests. The management console is shown in Figure 3a. In the left pane, the acceptance status of each component is displayed. A detailed report of the selected test case is shown on the right hand side pane. How the console is used during testing, is depicted in Figures 4b and 5, and explained in the next Section.

5. Case Study: Join and Leave of AIS Monitors

To demonstrate the process of integration testing and to validate our proposal on a realistic scenario, we will present an experiment with a sub-system component leaving the MSS SoS and a new one joining it. This scenario represents a system re-configuration during operation. We show the integration and verification procedure and the artefacts to be provided by the framework in order to be able to perform the modification of the system at runtime. This experiment focuses on the integration verification of one given component in a new context.

5.1. AIS: Automatic Identification System

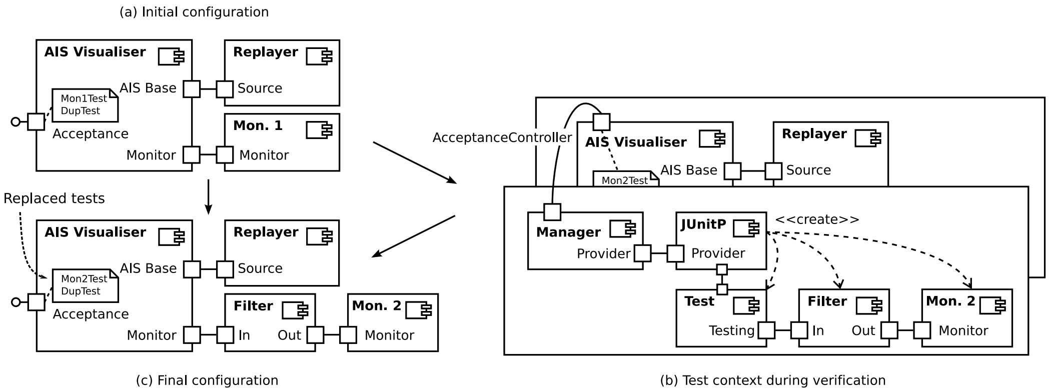
The Automatic Identification System (AIS) is a worldwide adopted International Telecommunication Union standard used for vessel identification [13]. Ships broadcast over radio information about their status and position with a variable report rate that depends on various parameters of the ship. Several AIS base stations are distributed along the coast of The Netherlands. The messages received by these stations are then relayed to the coast authority, who can then use the data for traffic control, collision avoidance, and assistance. One particular automated task is the monitoring of the messages to identify ships with a malfunctioning AIS transponder, or ships whose captain has forgotten to correctly adjust the transmitted information.

For our experiment, we will use a simulator which replays a dump of a week's worth of AIS data covering the whole coast of The Netherlands. Because AIS messages are broadcast and can be received by many base stations, duplicates of the same message will be received. Our recorded data is no exception to this.

5.2. Case Study

In our experiments we used an example system composed initially of three components, shown in Figure 4a. It comprises a replayer component, a visualiser component and an AIS conformance monitor. The visualiser receives AIS data coming from the base stations through the `AIS Base` port, and draws the position of the ships on screen. It has an additional `Monitor` port for connecting a monitoring component that detects anomalies in the AIS messages. Anomalies in the AIS data will be displayed on the map as a warning message next to the ship. The anomalies being watched are of two kinds: inconsistencies in the data, and incorrect transmission rates of the AIS messages. Figure 3b shows what an operator would see in the control room.

[2] http://www.eclipse.org


(a) Initial configuration

(c) Final configuration

(b) Test context during verification

Figure 4. Component diagram before, during, and after the replacement of the monitor.

5.2.1 Initial Deployment

Using the ATLAS management console (shown in Figure 3a), the three initial system components are instantiated. Two test cases are associated with the visualiser via the acceptance interface, as can be seen in the UML 2.0 diagram note in Figure 4a. The first one checks that when the AIS stream contains no duplicates, the right warnings are generated; the second one checks that the monitor can correctly handle duplicate messages.

Before initiating the system operation by starting every component, both test cases have to be executed. As both tests have been successfully executed, the system is allowed to start. Should a test fail, it is still the architect's responsibility to decide what actions to take to solve the situation. A scenario of this situation is described in Section 5.2.2.

5.2.2 Rejected Modification

The first version of the monitor had been developed to check strict compliance to the AIS standard. However, in practice, the monitor overwhelms the visualiser with too many warnings, rendering it useless for the operator. This comes as the result of the combination of a number of factors, that include skewed timing information caused by delays in the relaying between the base stations and the central coastguard facilities, and AIS transponder misuses and abuses by the ship's captains [11]. In this case, we cannot make the rest of the system adapt to the component since we have no authority over ships, nor the AIS relay network. Instead, we decide to replace the monitor by a more relaxed one, that only reports in case of a clear violation of the AIS regulations. This represents a runtime evolution of the system.

Because the expectations on the warnings generated by the monitor change, the test cases have to change accordingly. Therefore, the first test suite is removed, and a new, adapted one is inserted using the acceptance controller of visualiser. Replacing the first monitor by the second monitor causes the acceptance interface of the visualiser to receive the `notifyChange` call, which invalidates the test case's results. If we want to start the components again, we must re-check them. Unfortunately, when they were checked again, the second test case was not satisfied, because the new monitor does not handle duplicates correctly. As pointed out in Section 5.2.1, a decision must be made on which part of the system should be adapted. In this case, inserting a new component to perform the missing functionality, as seen in Section 5.2.3.

Due to space constraints, this intermediate configuration is not shown in Figure 4. A sequence diagram showing the interactions during the testing process can be seen in Figure 5. As noted on Section 4, the Management Console queries the acceptance interface of the component, creates the corresponding provider and requests the test to be checked. When the check finishes, the result of the test is stored back in the component. This is done in a completely automated way.

5.2.3 Accepted Modification

In order to solve the component rejection problem, the duplicate filtering functionality of the first monitor was extracted as a separate component, and added as a preprocessing component before the monitor. This invalidates the test case's results in the visualiser once more (although one of them was already invalid as it failed in the previous configuration).

After this change, the testing process is run again. Figure 4b shows the relationship between the manager and the acceptance controller of the visualiser, as well as the testing context set up by the provider during the testing process. It depicts the current status of the implementation, in which the testing context is replicated for each test case. In this case, both test cases are satisfied. The modification is therefore accepted and the operation of the system is allowed to resume under this third configuration. The final configuration of the system corresponds to the one in Figure 4c.

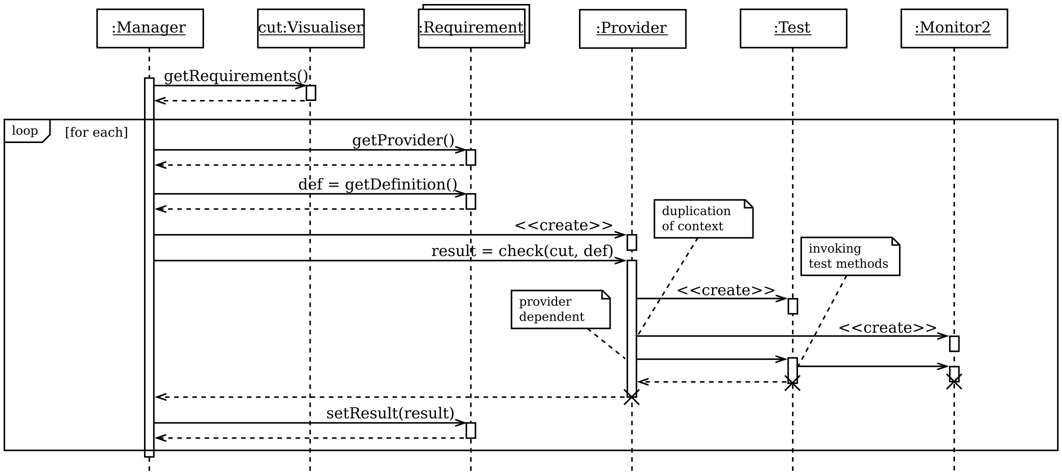


Figure 5. Sequence diagram of the test process in the current implementation.

5.3. Evaluation

The experiments performed must be regarded as a proof of concept. Although they are simple, they demonstrate the usability of our proposed process to verify the integration of a given component into a specific context, made of other components. Each component can have a set of cases easily associated with it, and those can be automatically managed by the same platform that manages the component integration. The framework also handles the initialisation of the system under test, which simplifies the definition of test cases, focusing on the test scenario. Still, there are plenty of issues that need further attention, especially with regard to runtime testing and scaling our approach to a complex system.

We have shown how ATLAS and its extension of the BIT paradigm provide a base to automate the runtime integration and verification of an entire component-based MSS System-of-Systems. Still, the creation of the test cases, is not automated.

An advantage of this approach, as shown in the second experiment, is that the acceptance interface allows us not only to query and check the required tests at instantiation of a component, but also to replace them, as it is in the case of the second experiment. The acceptance controller can be used to add new test cases or remove obsolete ones during the execution of the system, therefore making sure the SoS requirements are able to evolve and change during runtime. The implementation of components and test cases can, therefore, be performed separately and in a loosely coupled way.

Adding our verification infrastructure in a transparent way has proved to be fairly straight-forward thanks to Fractal's introspection and extensibility capabilities. The fact that the automatic integration and verification mechanisms are added at runtime by Fractal to the components saves development effort and cleanly separates testability and acceptance mechanisms from the functionalities of the components. That said, ATLAS can be implemented in other component frameworks without much effort, as it relies on constructs present in most of them, such as interfaces, ports, method calls and component instantiation.

6. Summary, Conclusions and Future Work

In this paper, we have presented some of the challenges of the integration and verification of large-scale component-based systems, such as MSS systems. We have described ATLAS, our solution architecture based on Built-In Testing. ATLAS not only permits to associate each component with a set of test cases and dedicated interfaces to request testing of other components, but also permits these test cases to evolve dynamically with the system. While the original idea of BIT was component integration testing at development-time configuration, before deployment, we have shown in our example case that, with ATLAS, this can be done during deployment of a system, as runtime integration testing.

Although, in the current implementation we have avoided the problems caused by runtime testing by replicating the components, these effects are important and must be considered. When components are tested during runtime it is likely that side effects of the tests will propagate into the rest of the system. Our future work on the implementation will ensure that every component can be made test-aware and that it is an easy task for the developer.

In our future work we will also study the usage of our approach in large-scale systems formed by a large number of components contained within relatively independent subsystems. The following primary issues will have to be considered.

The first issue concerns the "domino effect": when a binding or a component is modified, the modification can potentially affect every component directly or indirectly linked to it, as well as all the composite components that contain this affected components. This means devising a mechanism to notify components other than those adjacent to the modification, in a scalable and distributed way.

Secondly, since analysing and performing the necessary verification of the whole system is a costly operation, minimizing the disturbance caused by runtime testing will be essential. This means reducing the number of tests being performed. One of the possibilities is to exploit the fact that between two system configurations, the difference is little and the previous configuration has already been entirely tested. If the notifications carry fine-grained information about what part of the system has changed, the re-checking of test cases can be reduced to only those which exercise the affected part, reducing the cost of the runtime regression testing.

It is also possible to entirely automate the testing process. After a change notification has been received, the testing can be started right away by components themselves (instead of by the system integrator). However, several aspects must be taken care of. If the reconfiguration involves multiple modifications, usually only the final configuration has to be tested. The testing must also be ordered in a way that allows to find errors more efficiently, reason about them more effectively, and avoid the system to be suddenly flooded with tests.

Another work direction is the inclusion of support for verification of non-functional properties, such as resource consumption, execution time, etc. Finally, we will also ensure the applicability of ATLAS to architectures other than client-server, such as publish-subscribe architectures, which present their own challenges in the testing process [15].

Acknowledgements. This work has been carried out as part of the Poseidon project under the responsibility of the Embedded Systems Institute (ESI), Eindhoven, The Netherlands. We want to thank Niels Willems of the visualization group of Eindhoven University of Technology, for letting us use his AISPlotter visualiser component in our example. This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK03021 program.

References

[1] A. Bertolino and A. Polini. A framework for component deployment testing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 221–231, Washington, DC, USA, 2003. IEEE Computer Society.

[2] 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 System Frontiers*, 9(2–3):151–162, 2007.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in java. In I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. C. Wallnau, editors, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2004.

[4] D. Deveaux and P. Collet. Specification of a contract based built-in test framework for fractal, 2006.

[5] Embedded Systems Institute. The poseidon project. http://www.esi.nl/poseidon, 2007.

[6] EU Commission. An integrated maritime policy for the european union. European Commission, Maritime Affairs, Oct. 2007.

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

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

[9] H.-G. Gross, M. Melideo, and A. Sillitti. Self-certification and trust in component procurement. *Science of Computer Programming*, 56(1–2):141–156, Apr. 2005.

[10] H.-G. Gross, I. Schieferdecker, and G. Din. Model-based built-in tests. *Electronic Notes in Theoretical Computer Science*, 111(1):161–182, 2005.

[11] A. Harati-Mokhtari, A. Wall, P. Brooks, and J. Wang. Automatic Identification System (AIS): Data reliability and human error implications. *Journal of Navigation*, pages 373–389, 2007.

[12] J. Hartmann, M. Vieira, H. Foster, and A. Ruder. A UML-based approach to system testing. *Innovations in Systems and Software Engineering*, 1(1):12–24, 2005.

[13] International Telecommunication Union. Recommendation ITU-R M.1371-1, 2001.

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

[15] A. Michlmayr, P. Fenkam, and S. Dustdar. Specification-based unit testing of publish/subscribe applications. In *ICDCSW '06: Proceedings of the 26th IEEE International Conference Workshops on Distributed Computing Systems*, page 34, Washington, DC, USA, 2006. IEEE Computer Society.

[16] J. Morris, G. Lee, K. Parker, G. A. Bundell, and C. P. Lam. Software component certification. *Computer*, 34(9):30–36, 2001.

[17] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[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. Vincent, G. King, P. Lay, and J. Kinghorn. Principles of Built-In-Test for Run-Time-Testability in component-based software systems. *Software Quality Journal*, 10(2):115–133, 2002.

[20] E. J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Softw.*, 15(5):54–59, 1998.