Testing Challenges of Maritime Safety and Security Systems-of-Systems

Alberto González[1] Éric Piel[1] Hans-Gerhard Gross[1] Maurice Glandrup[2]
[1]Delft University of Technology, Software Engineering Research Group
Mekelweg 4, 2628 CD Delft, The Netherlands
{a.gonzalezsanchez,e.a.b.piel,h.g.gross}@tudelft.nl
[2]Thales Nederland.
Haaksbergerstraat 49, 7550 GD Hengelo, The Netherlands
maurice.glandrup@nl.thalesgroup.com

Abstract

Maritime Safety and Security systems 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. Two distinguishing characteristics are their ability to evolve during runtime, that is, joining and leaving of components, and the need for high reliability of the system.

In this paper, we identify the challenges that will have to be addressed, given the current state of the art in component-based software engineering in order to build such system-of-systems. In particular, we highlight the specific difficulties regarding acceptance and testing. A first group of testing challenges is raised by the need of accepting the integration of such large systems, and the ability to reconfigure them at runtime. A second group of testing challenges comes from the fact that, generally, not all the sub-systems are designed along the same kind of architecture (e.g. client-server vs. publish-subscribe architecture). Devising an integration testing process for such hybrid architecture is inherently harder than for a homogeneous one.

1. Introduction

The commission of the European communities has recently pushed for the establishment of a European Network for Maritime Surveillance [4]. Such a network will provide safe and secure usage of the seas around Europe, integrated and coordinated maritime planning, research, climate control, and sustainable development.

This network will require the cooperation of the Member States' security agencies, and an efficient usage and integration of not only existing navigation, monitoring and tracking systems, but also the systems in the respective operations and control centres.

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) [9]. 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 [3], committed to devising engineering best practices for developing, integrating and deploying such maritime safety and security (MSS) systems.

In this "industrial/academic challenges" paper, we will outline the general software engineering challenges that providers of MSS solutions such as Thales Nederland are facing, and we will focus on the particularities that make the quality assurance and acceptance of this kind of system-of-systems a challenging line of research. In particular, the highly dynamic nature of these systems and high reliability required are of interest. Some other challenging issues, in a more technical realm, stem from the choice of runtime model used to develop the system.

In the next section (Sect. 2), we will briefly outline the more general challenges encountered when building MSS SoS. Sections 3 and 4 will focus more on the specific issues related to integrating and accepting MSS SoS, and Section 5 summarizes the paper and gives an outline of current and future work.

2. Challenges of MSS Systems

The challenges of Maritime Safety and Security systems are defined by the typical requirements of highly flexible, adaptable and evolvable large-scale systems. This implies [3]:

- The development of adaptable and evolvable SoS architectures

- An integration and acceptance method for reliable dynamic reconfigurations
- Trustworthy fusion and processing of information sources in terms of type, role, syntax, and semantics
- Intelligent analysis of the available information to solve application tasks.

Although, the last two items are more concerned with domain-specific issues of surveillance-oriented SoS, they do have implications on how such systems are built and deployed.

Systems-of-Systems comprise a large number of autonomous interoperable nodes which cannot be anticipated completely and precisely [8]. Most sub-systems in an SoS have operational and managerial independence [6, 9]. These imply that parts of the SoS may be changed without the SoS integrator having too much to say in the decision, or may not even be notified. 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.

A specific requirement for MSS SoS, is *geographic distribution*. An example scenario is a UN peace keeping operation along a foreign coast line, in which different surveillance ships with different kinds of systems have to cooperate in a meaningful way in order to provide situational awareness of the controlled coast. Distribution of an MSS SoS, in this case, over various frigates from a number of different countries brings up diverse security and confidentiality issues that must be dealt with by the complete SoS.

The following paragraphs shortly summarize the challenges to be considered (with emphasis on integration) when building MSS systems.

Dynamic Reconfigurability. Systems can join or leave the SoS, meaning that offered services may vary in terms of function, 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. The reconfiguration must be compatible with the need for high reliability of the MSS system. The problem space covers, amongst others, fault detection, system degradation, fault recovery, architectural completeness, and system evolution. In the UN peace keeping scenario, frigates may join and leave the task-force. This process should be mostly seamless for the system operators and be executed within a short time, without any major disruption on the operation of the rest of the SoS.

Interoperability. Since an SoS usually consists of systems of different vendors, their integration may not be straight-forward. This could have various reasons, ranging from mismatching hardware connections to different electrical signals, or deviating protocols for transmitting service requests and data. Our primary focus is on the last item. Typically, when setting up a connection between systems, the semantics of the transmitted data is different, and the protocols are not compatible. In that case, adaptation must be performed, either manually, or it can be assisted by means of automatic protocol adapter generation [12]. In the UN-peace keeping scenario, frigates exchange information about high value objects, for instance fighter aircrafts, of participating countries in the mission area. For each nation the services that manage the information on the frigates use different data models and different communication protocols.

Trustworthiness. There may be constraints on subsystems that allow them to use, or prevent them from using information coming from other sub-systems, or pass this information on to other parties. This is known as trustworthiness and integrity of supplied services and information [1]. When a system joins the SoS, its collaboration must be accepted by all other systems, and the data and services which it may access have to be determined. In case of the UN peace keeping scenario, the quality of all services must be re-evaluated when a frigate of another nation joins, simply because of specific treaties between the nations constraining information to be passed.

In the two subsequent sections, the paper will concentrate on the testing-related challenges of MSS SoS.

3. Integration, Reconfiguration and Testing

Systems-of-Systems rely heavily on runtime techniques for integration, evolution, and for ensuring the quality of the system. SoS cannot be completely predicted at design time. This prevents the systems from being fully tested during development or during first deployment.

3.1 Integration and Reconfiguration

A typical MSS SoS will undergo many changes during its operation. There are several different scenarios that lead to a reconfiguration of the system. One of those scenarios is the late arrival of a vessel into the system (evolution of the SoS). When this vessel arrives, the information obtained through the various sensors it supports must be transmitted to the rest of the SoS. Similarly, this new vessel will need to access the aggregated data of the SoS.

Another scenario is the update of one of the software components within one of the sub-systems (evolution of a system within the SoS), for instance, an improved anomaly detector in the harbour command centre. In this case, one will have to assure that no regression has been brought in by the modified component.

Evolution of the Acceptance Requirements. The evolution problem is part of the *dynamic reconfigurability* challenge and also induced by the strong autonomy of the sub-systems. As new components and updated versions of previous components can be inserted at runtime, the tests used to ensure the integration of 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. Acceptance requirements need not to be restricted to testing, nor to a fixed set of requirements. Therefore, the platform must support different types of acceptance checks (static contracts, monitoring of resources, etc.), and dynamic insertion and removal of these, in the same way it supports join and leave of components. So far, our approach has been to rely on Built-In Testing (BIT) [7], since it permits close association of the integration tests with a component, and even with a specific version of a component. However, it will be useful to extend the approach with more dynamic BIT capabilities, in order to allow for removing or adding requirements independently of the component, for example.

Cost of the Acceptance Process at Runtime. This problem is part of the *dynamic reconfigurability* challenge. By their very nature, SoS are large-scale systems, with a large number of components, contained in the sub-systems. As after each reconfiguration, the integration of the SoS has to be re-accepted, one of the main goals of the Poseidon project is to achieve the highest degree of automation of the acceptance process possible. We will address this problem, firstly, by devising a framework that automates most of the testing process, and secondly, reduces the number of test cases by adequately selecting and prioritizing them, therefore minimizing the cost of testing after each modification by testing only the altered parts of the SoS, and by finding defects with the lowest possible number of test cases. Moreover, a model-based approach could be used to detect incompatibilities between components without resorting to testing [2, 10, 13]. This makes the verification of a modification as little disruptive as possible for the running configuration and reduces the latency between the moment a reconfiguration is requested and the moment it is deployed.

The fact that SoS are made of relatively independent sub-systems will likely be a useful property on which one can depend for restraining the number of components to test after a reconfiguration.

Limitation of Shared Information and Testing. This problem is part of the *trustworthiness* challenge. The managerial independence between sub-systems comes also with the need to limit the amount of information shared between pairs of sub-systems. Typically, this can be due either to political reasons or to business reasons. The main implication is that data exchanged between the sub-systems has to be scrutinized.

Although this challenge concerns the computer security community, rather than the software testing community, it is also important that during the execution of test cases, no confidential information can leak, for instance, by accessing a component via normally unused testing interfaces, or by requesting specific information to be used for testing that should not be shared. Moreover, the fact that a particular part of a system has been modified might also be considered confidential. In this case, the modified sub-system might have to avoid sending messages to the rest of the SoS such as "this particular component and bindings have been modified in this way." Rather, it should only send information such as "the sub-system has been modified".

3.2 Runtime Testing

Testing usually relies on creating a testing instance of the component or system under test (for the case of integration testing). 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 to the working configuration and b) some component instances will be shared between the tested and the working configurations. Therefore, *runtime testing* [15] is the only realistic option when integrating and reconfiguring MSS SoS.

Test Isolation. The test isolation problem is induced both by the *interoperability* and the *dynamic reconfigurability* challenges. Testing involves interactions with the System Under Test (SUT), sending stimuli to verify that the SUT responds as expected. Runtime testing bears the danger that testing interactions with the component will affect its other users. We must ensure that test operations and data stay in the testing realm and do not affect the other clients of a component or sub-system. We refer to this as *test isolation* [14].

Test Awareness. The test awareness problem is part of the *interoperability* challenge. In order to achieve test isolation, two related aspects have to be taken into account: *test sensitivity* [14], and *test awareness*. On the one hand, components are test-sensitive if interactions originated in the "testing world" can have effects on the "production world". These effects comprise modification of the component data or state, generation of new data or events that will be received by other components, or repercussions in the "real world", e.g. controlling some peripheral. On the other hand, components are test-aware if the execution environment provides a way to "tell the difference" between test and non-test invocations and data, for example by using two ports, or two versions of the same port, one used for working and other for testing. Test-insensitive components do not need to be test-aware. In contrast, test-sensitive components have to be test-aware. However, this imposes extra responsibilities on the developers that we would like to minimize as much as possible, moving most of the runtime testing responsibilities to the SoS runtime platform.

Fault Detection. An MSS SoS is under constant risk of sabotage or terrorist attacks against the system itself. For instance in the UN peace keeping scenario, a series of radars on the coast will suddenly stop reporting data, or all the data provided by a frigate will be incorrect and incoherent with the data provided by the other systems. Such attacks should be detected automatically by the system. Moreover, it would be helpful to differentiate faults happening accidentally (real faults), from the other faults, caused by malicious attacks, as, depending on the case, the reaction of the system to be taken will be very different.

Component monitoring and diagnosis, also termed *fault localization*, will be necessary to detect incorrect data, component misbehaviour, and to pin-point the origin of a problem. However, higher level of analysis (e.g. pattern recognition) will be required in order to be able to estimate the chances that a particular fault is caused by malicious action. One specific issue for MSS SoS is that the components are geographically distributed. Thus, providing a component framework which allows to take the physical position of a faulty component into account can help in distinguishing the likely causes correctly.

4. Design-originated Challenges

MSS systems will have to integrate systems from heterogeneous sources. The design of those components (e.g. the interaction model) will also pose challenges that do not directly derive from the requirements of the system. For instance, the client-server model, or its more recent incarnation, the service oriented computing model, seem like natural choices when implementing services inside a command and control center. This means accepting systems programmed in CORBA, CCM or Enterprise JavaBeans. However, sensor networks will play an important role in the MSS, as they will provide the necessary input from the real world to construct the situational awareness picture. Furthermore, many of the already existing systems in the maritime world are data-centric. Publish and subscribe platforms [5], such as the OMG data distribution service (DDS) [11], are, therefore, readily used in this domain. Other component models (for instance, Peer-to-Peer) may also be encountered when integrating the SoS. Due to the fact that SoS are comprised of many different systems, the most likely situation is that the interaction between components will be hybrid, i.e. based on many different types of platforms.

We have identified three main aspects, stemming from the interconnection and interaction design of the MSS, that will influence the testing strategy.

Explicit vs. Implicit Dependencies. Most service-centric architectural models, like CORBA or EJB are based on binding required to provided interfaces. Bindings make dependencies and communication between components explicit, i.e. components are constrained to interact with a certain set of other components, defined by the system architect. When a part of the system is modified, obtaining the list of components affected by the reconfiguration is a matter of following the dependency graph derived from the bindings. Publish-subscribe systems, on the other hand, are characterized by loose coupling of their components. Each component reads data messages of specific types and generates other data messages. There are no explicit dependencies between components, but a many-to-many dependency implicit in the data model they publish, or subscribe to. This simplifies the integration process considerably, but on the other hand, it makes the task of finding dependencies between component instances more difficult, and, thus, regression testing after a reconfiguration of the SoS much more difficult.

Synchronous vs. Asynchronous Interaction. In some architectures, when a component requests the service of another component, it is blocked until the second component finishes. In other architectures, component interactions are asynchronous, meaning that a component is still active while the service is being processed. When the called component finishes the processing, it can alert the first component through some event. Asynchrony is advantageous in terms of performance and independence, but it also introduces complexity when testing. Because the return event can occur at various times during the execution of the first component, additional care must be taken to ensure that all situations have been assessed.

Call-Return vs. Data Flow. In a service-centric architecture, activities originate from a rich set of different events that components send to each other and respond to. The system follows what could be called a *call-return* model, the basis of a client-server architecture. A component needs a service (the client) and calls a component that provides this service (the server) with the specific data to be treated. Once the processing is finished, the server returns the result. Writing integration test-cases for such an architecture is relatively easy. From the client side, it is possible to express both the input and the expected output of the server, because the component has an expectation towards the service (model) that can be translated into test cases.

Conversely, in a data-centric architecture, there exists only one event that triggers activity: the reception of new data. The system is better understood as a series of *data flows*. A component receives data from the previous component in the flow, processes the data and sends the result to the next component in the flow. In this component model, integration test-cases cannot be associated with one specific component alone, because components do not have any requirements on their predecessors or successors; they only know about the data. Integration testing must, therefore, be associated to *a set of components*, and the integration test suite has to verify that the data input to this set is correctly processed. Defining such test-cases and determining how they should be applied during reconfiguration is much harder than with the call-return model. In the context of SoS, it is also challenging to define a testing infrastructure that incorporates concepts from both worlds.

5. Summary and Future Work

In this paper, we have presented the general software engineering challenges and the particular integration, testing, and acceptance challenges that we are facing in the development of highly dynamic maritime safety and security systems-of-systems. These encompass component autonomy, geographical distribution, and high dynamicity coupled with high reliability. These characteristics of MSS systems lead to the requirement that they should be able to be tested during runtime while they evolve. The vision of the Poseidon project is to provide (semi-)automatic integration and acceptance (testing) mechanisms for MSS systems. That way, subsystems could be joining or leaving autonomously without threatening the integrity of the overall MSS application.

Our current research work, in collaboration with Thales, aims at resolving the problems of runtime evolution of acceptance requirements and reduction of the costs of acceptance, as well as the definition of a runtime platform that enables us to demonstrate our propositions. Future work will concentrate on the testing challenges of publish and subscribe platforms, in particular the effects of their decoupling characteristics on fault identification, and on how MSS can be protected from side effects of runtime testing.

Acknowledgments. This project is partially supported by the Dutch Government under the BSIK03021 program.

References

[1] J. Allen, S. Barnum, R. Ellison, G. McGraw, and N. Mead. *Software Security Engineering: A Guide for Project Managers*. Addison-Wesley, 2008.

[2] P. Collet, R. Rousseau, T. Coupaye, and N. Rivierre. A contracting system for hierarchical components. In *Component-Based Software Engineering, 8th International Symposium (CBSE'2005)*, volume 3489 of *LNCS*, pages 187–202, St-Louis (Missouri), USA, May 2005. Springer Verlag.

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

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

[5] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

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

[7] H.-G. Gross and N. Mayer. Built-in contract testing in component integration testing. *Electronic Notes in Theoretical Computer Science*, 82(6):22–32, 2004.

[8] W. Humphrey. Systems of systems: Scaling up the development process. Technical Report ESC-TR-2006-017, Software Engineering Institute, 2006.

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

[10] S. McCamant and M. D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, pages 440–464, Oslo, Norway, June 16–18, 2004.

[11] G. Pardo-Castellote. OMG data-distribution service: Architectural overview. In *23rd Workshop on Distributed Computing Systems*, pages 200–206. IEEE, 2003.

[12] W. Reisig, J. Bretschneider, D. Fahland, N. Lohmann, P. Massuthe, and C. Stahl. Services as a paradigm of computation. In *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 521–538. Springer-Verlag, Sept. 2007.

[13] A. Stuckenholz and O. Zwintzscher. Compatible component upgrades through smart component swapping. In R. H. Reussner, J. A. Stafford, and C. A. Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 216–226. Springer, 2004.

[14] 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.

[15] 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.