# Automating Integration Testing of Large-Scale Publish/Subscribe Systems

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

## KEYWORDS

Publish/Subscribe, Integration testing, Component, Unit-testing, Runtime testing, Data-flow, Built-in testing

## ABSTRACT

Publish/subscribe systems are event-based systems separated into several components which publish and subscribe events that correspond to data types. Testing each component individually is not sufficient for testing the whole system; it also requires testing the integration of those components together.

In this chapter, first we identify the specificities and difficulties of integration testing of publish/subscribe systems. Afterwards, two different and complementary techniques to test the integration are presented. One is based on the random generation of a high number of event sequences and on generic oracles, in order to find a malfunctioning state of the system. The second one uses a limited number of predefined data-flows which must respect a precise behaviour, implementable with the same mechanism as unit-testing. As event-based systems are well fitted for runtime modification, the particularities of runtime testing are also introduced, and the usage in the context of integration testing is detailed. A case study presents an example of integration testing on a small system inspired by the systems used in the maritime safety and security domain.

## INTRODUCTION

Our research focuses on the fast integration of systems-of-systems (SoS). In collaboration with industrial partners, we ensure that the results can be applied for Maritime Safety and Security (MSS) systems, which are typical event-based systems and are often built on top of a publish/subscribe architecture. An important aspect of this research covers the validation through checking the correct integration of the system in order to ensure their reliability.

MSS systems are large-scale distributed SoS in which the sub-components are elaborate and complex systems in their own right. The primary tasks of MSS SoS are sensing issues at sea, analysing these issues, thus, forming a situational awareness, and initiating appropriate action, in case of serious issues (EU Commission, 2007; Thales Group, 2007; Lockheed Martin, 2008; Embedded Systems Institute, 2007). Provision of situational awareness requires analysis and synthesis of huge volumes of data coming from the various types of sensor components such as vessel-tracking (AIS) systems, satellite monitoring, radar systems, or sonar systems. A

distinguishing characteristic of MSS systems-of-systems is their data-centric, distributed and event-based nature (Muhl, Fiege, & Pietzuch, 2006), which means that the publish/subscribe paradigm is well fitted, and it is readily being used as underlying system infrastructure.

In their landmark paper on "The Many Faces of Publish/Subscribe", Eugster et al. (2003) present an array of advantages of the publish/subscribe paradigm as a cure for developing highly dynamic large-scale systems. However, the advantages of fully decoupling the communicating entities in publish/subscribe platforms in terms of time, space and synchronization, can also be seen as a curse when it comes to runtime evolution and ensuring a system's integrity after dynamic updates. In this chapter, we will discuss the issues of integration and acceptance testing arising from the loose coupling of communicating entities advocated by event-based systems. Because we are dealing with highly dynamic systems that have to provide constant operational readiness, we will concentrate on the challenges system engineers are facing when it comes to testing and accepting dynamic system reconfigurations.

First in section *Properties of Publish/Subscribe Platforms*, we will present the testing requirements in publish/subscribe systems and highlight the specific challenges of this context. In Section *Integration Testing in Event-based Systems*

, we will describe the usage of testing methods in order to implement integration testing in such dynamic and decoupled environment. The Section *Runtime Testing for Component-based Platforms* provides details on the specificities of handling testing at runtime. An example of usage of the methods previously defined will then be presented in Section *Runtime Integration Testing: A Controlled Experiment*, using a simplified version of an MSS system. An overview of the future research to come in the domain of testing and runtime evolution will be given in Section *Future Trends*. Finally, Section *Summary and Conclusions* will summarize and conclude this chapter.

## PROPERTIES OF PUBLISH/SUBSCRIBE PLATFORMS

The quality of a software system is a compromise between the cost of an error happening and the cost of improving the quality (cost being used with a large meaning such as time, money or physical damage). In order to improve the quality of a software system, the most commonly used technique is *software testing*. Due to the nature of their applications, systems based on event-driven platforms, and publish/subscribe platforms in particular, tend to have high needs for software quality and testing. Systems designed on event-driven platforms can have all kinds of applications, but some typical classes of applications are:

- Embedded software, which controls physical devices such as cars, music players, factory robots, etc. Sensors generate input data and events that are treated by software components which, in turn, send output data to controller devices.
- Web applications, which react to the requests from web users, process them and send the results.
- Graphical User Interfaces (GUI), where events such as clicks or key presses trigger specific code which modifies the state of the application and eventually leads to producing outputs.

The first two application classes often have strong *reliability requirements*: incorrect behaviour or lack of reaction could have very fatal consequences, either in terms of cost or safety.

Moreover, the event-driven approach permits to separate the code in several small entities dedicated to a specific task. A formal way to distinguish those entities is the notion of a *component*. This notion allows to handle the complexity of the software system as it is possible to

concentrate only on a small part of the functionality at a time. However, each component will still interact with the other components, either directly by sending or receiving data, or indirectly by affecting the state of a third component of the system. Testing each of the component separately is not sufficient (Gao, Tsao & Wu, 2003), integration problems only occur when the entities interact with each other, due to missing/wrong interactions, or because one component does not behave as another one expects it to. It is, therefore, necessary to perform *integration testing*.

The decomposition into components provides flexibility in the system architecture. Publish/subscribe architectures permit to keep components very loosely coupled as, in contrast to other architectures (client/server, peer-to-peer) the components are not directly bound to each other, but only bound to data or event types. This eases the *runtime reconfiguration* of a system, as components can be added or removed independently. Runtime reconfiguration is useful in the context of high availability systems, especially for the large-scale and complex ones which cannot be duplicated. Often duplication is not possible because of the cost of duplicating the hardware (such as having a entire boat available just for testing purpose), but also due to license cost for both a production and a test system, or in case of cooperation between different actors not willing to share full control of their components between each other. With respect to reliability, reconfiguration has to be treated carefully, as the original behaviour might be affected. Therefore, integration testing must also be done during reconfiguration. It is important to pay special attention to this case, where testing happens at runtime.

In the following, we highlight the issues encountered during integration testing when dealing with a publish/subscribe architecture, which are not so problematic in other runtime architectures such as client/server.

## Explicit vs. implicit dependencies

Most service-centric architectural models are based on binding required to provided interfaces. Bindings make dependencies and communication between components explicit (Oreizy, Medvidovic, & Taylor, 1998), i.e., components are constrained to interact with a known 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 identifying the modified components and bindings, and following the dependency graph derived from the bindings. Then, the test cases that exercised the modified parts have to be re-run (Orso, Do, Rothermel, Harrold, & Rosenblum, 2007).

In contrast, Publish/subscribe systems are characterized by loose coupling of their components. Each component listens to data messages of specific types and generates other data messages. There are no explicit dependencies between the components themselves. Dependencies are rather implicit in the data types 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 inhibits the testing after a reconfiguration of the system.

Ways to handle the dependencies is the various cases are presented in Section *Integration Testing in Event-based Systems*

## Rendez-vous vs. persistent messages

In architectures where components are linked directly, such as Remote Procedure Call (RPC) based systems, the communication between components happens only when they are simultaneously present (rendez-vous). Either the interaction between the components is

immediate, or it does not happen. On the other hand, in publish/subscribe platforms, communication between components can be time-decoupled. When a component emits data, no assumption is done about the presence of components that may receive them. The data might be used immediately, they might never be used, or they might be used later when a component interested by the data appears, even if the emitting component has, since then, been removed. The data might even be first used immediately and also at later time by another component. This *persistence* is provided by the platform.

Persistence of data or events leads to useful features such as better reliability, or complete independence between the components, but it also brings difficulties in testing: when a new component is added to the system, not only interactions with the current components have to be tested but also the effect of the components previously removed. As we will see in Section *Integration Testing in Event-based Systems*

, this can be considered as special case of component dependencies.

## Synchronous vs. asynchronous interaction

In some architectures, when a component requests the service of another component, it is blocked until the second component finishes. This ensures that the operation has been completed before the first component resumes. In other architectures, and often in event-driven platforms, component interactions are asynchronous, meaning that a component is still active while the service is 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 (Michlmayr, Fenkam, & Dustdar, 2006b). Because the reply event can occur at various times during the execution of the first component, additional care must be taken to ensure that all interaction combinations have been adequately assessed. In other words, it introduces an observability issue while testing: comparing the execution trace with the expected result would lead to an inconclusive test result.

Several proposals have already been introduced to handle asynchrony in requirement specification and in test cases. For instance TTCN-3 (Schieferdecker, 2007), a test case description language, permits to specify separately the messages sent, the values returned, and also the minimum and maximum time that the result may take to be answered.

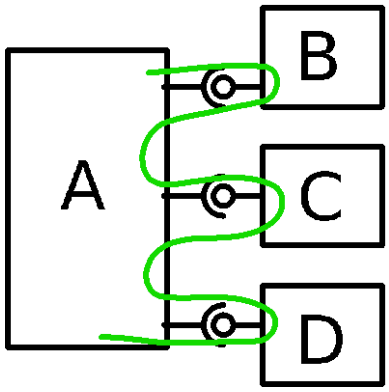


*Figure 1: Example of system organised following the call-reply model.*

## Call-reply vs. data-flow

In a service-centric architecture, an activity originates from a component which "pulls" a result from another component by requesting the processing of some data. The system follows what could be called a *call-reply* 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. This organisation is presented in Figure 1. The curved line represents the information flow. Component *A* calls component *B*, which replies. After that, component *A* calls component *C*, etc. This scheme can be recursive: a server can be client of other components. Writing integration test cases for such architecture can be done at the development time of each component, without requiring knowledge of the system in which it will take part. There is a direct causality between what a component sends and what it receives. From the client side, it is possible to express both the input and the expected output of the server, because the component (and the developer) has an expectation towards the service that can be translated into test cases.

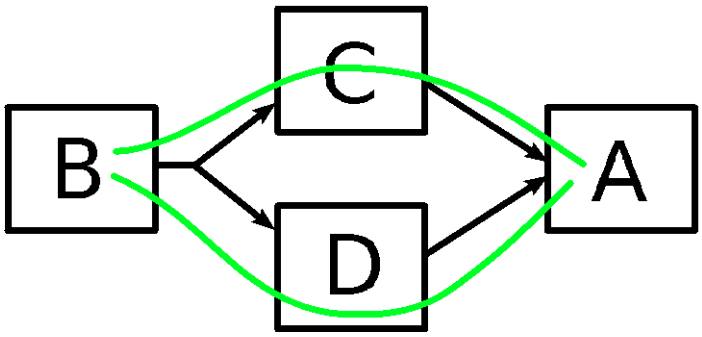


*Figure 2: Example of system organised following the data-flow model.*

In contrast, in a data-centric architecture, the data is "pushed" from one component to another. 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. An example of this type of organisation is shown in Figure 2. 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.

Therefore, integration tests are not associated with pairs of components but with either complete or partial *data-flows*. The integration test suite has to verify that the data is correctly processed. In Section *Integration Testing in Event-based Systems*

a method is presented to implement such testing which can also work in case of reconfiguration.

## Runtime testing

In case of runtime reconfiguration, testing of the new configuration has to be performed while the system is in operation (with the previous configuration). Ideally, this can be done by duplicating the entire system. Unfortunately the harsh reality often has limitation on the physical resources. The larger the system, the more likely are there limitations. For instance, it is not feasible to duplicate an entire ship whenever one of the software components of a radar must be updated. When the system cannot be duplicated, it is possible to rely on *runtime testing* (Vincent, King, Lay, & Kinghorn, 2002).

There are two main difficulties with using a component at the same time in the context of normal operation (production) and in the context of testing. First, testing involves interactions with the System Under Test (SUT), i.e. sending stimuli to verify that the SUT responds as expected. Runtime Testing bears the danger that testing interaction with the component will affect its other users. Test operations and data must be ensured to stay in the testing realm and not affect the other clients of a component or sub-system. This characteristic is known as *test isolation* (Suliman et al., 2006). Second, when components are tested which have effect outside of the system (i.e. they produce the output of the system) some operations might not be safe to test at runtime. For instance, it might not be possible to write in a database containing bank account information, or controlling the actuators in a robot while they are already being controlled following a different control algorithm.

Similarly, it can also happen that an input resource is unique. For example, in some frameworks, if a web service listens on one TCP/IP port, it is not possible to run another web service on the same port simultaneously. There are several possible solutions, such as setting up a simulator of the outside world, sharing the resource between the component under test and the component in production, or refraining from executing the test cases. Nevertheless, whichever approach is taken, first it is necessary to be able to define the *test sensitivity* of the component. The platform must provide a way for the component to "tell the difference" between test and non-test data or event, which permits the components to be what we call *test-aware*.

Handling test isolation and test sensitivity is described in Section *Runtime Testing for Component-based Platforms*.

## INTEGRATION TESTING IN EVENT-BASED SYSTEMS

Event-based platforms allow decoupling of components that are handling specific operations on data. Unit-testing of the components is not sufficient in order to validate the correctness of the entire system made of such components. In addition, it is necessary to check that the components interact correctly with each other. This validation is the goal of integration testing.

### Background

#### Assumptions on the system

Before detailing further some approaches to test the integration of an event-based system, it is important to define some basic assumptions and expectations on the system under consideration.

First, we consider that the system is component-based (Szyperski, 1998), meaning it is made of separate software units which can only interact with each other via predefined interfaces. In the context of publish/subscribe platforms, an interface is defined by the data types which will be received or sent (a simple *event* being represented as a data type which contains no data). Components may be hierarchically defined: a component can be composed out of several other sub-components. Components need not be *pure*[1]: they can have state and interact with the context. They can also be black, or "grey" boxes, i.e. their specification is known but their implementation is not.

Second, each component is considered to have already passed the unit-tests, and, therefore, implements its own specification correctly. The test cases should be available. This can be done either through providing inputs and an oracle, or by providing a specification of the component

[1] Pure is used here with the same meaning as in functional programming: no side effects are created as a result of an operation.

and a component capable of generating test cases out of it. For example Michlmayr et al. have proposed a framework (2006a; 2006b) to specifically unit-test publish/subscribe applications using Linear Temporal Logic (LTL): depending on the input data received it can define the type and order of the output data.

In most systems, there exist components which, from the point of view of the publish/subscribe framework, do not have inputs or do not have outputs. They correspond to *sensors* (obtain data from a hardware component, read data from a file, etc.) and *actuators* (display data on a screen, write data to a file, etc.). We will not include those special components in the rest of the discussion on integration testing as the information about which events they publish or subscribe to is not sufficient to test them. In practice, test cases should be adapted specifically to fit a component, and they should be able to handle the input and output correctly. Care should be taken to prohibit the sensor components from generating inputs which could interfere with the inputs provided by the test cases. They should either be stopped or their data should separated from the rest of the data using the same mechanisms which will be presented for runtime testing in Section *Runtime Testing for Component-based Platforms*.

## Verification and validation

Verification and validation are the two major processes to ensure the quality of the system. Those processes are complementary. Verification consists of ensuring that every properties and behaviour defined at a given level of abstraction still holds true at a lower level of abstraction. For instance, it might consist of manually comparing one requirement defined in English in the project agreement with the UML model of the system. *Formal* verification checks that a certain property is true using formal methods. For example, it can consist in automatically analysing a source code to verify that the transitions between the states can only follow the transitions defined in a model of the system in a Finite State Machine. The validation process is concerned by evaluating the implementation with respect to the expected behaviour of the system. It can be for instance done using test cases which contains the description of the inputs sent to the implementation and the expected resulting behaviour of the system.

For ensuring the quality of the integration of the system, not only the interaction of the components together has to be taken into account but also the framework on which the components will be executed. While both verification and validation are useful for this task, verification tends to be difficult because it requires also the model of the framework in addition to the models of the components themselves, and because the increase of complexity of the entire system often resolves in exponentially increasing complexity of the proof. This is one of the reasons we are focusing only on validation in this chapter. Nevertheless, it is worth mentioning the work by Garlan et al. (2003) on modelling a generic publish/subscribe architecture for verifying via LTL properties systems based on this type of architecture. Extending this work, Baresi et al. (2007) have proposed special language support for modelling and verifying the system efficiently thanks to the reduction of the number of states to explore. Zhang et al. (2006) have defined a modelling language dedicated at event-based systems which, using source code transformations, can both generate code used for the verification (using LTL properties associated with the original model) and execution traces of the implementation which can be used for the validation.

## Built-In Testing and beyond

Built-In Testing (BIT) is a useful paradigm in order to test a dynamic component-based system (Vincent et al., 2002; Suliman et al., 2006). 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 (Gross & Mayer, 2004), before or during runtime. It aims at a better maintainability of the testing aspect surrounding each component.

BIT has two facets. The first is concerned with the testability of the component. Components can be equipped with special interfaces in order to facilitate the testing. For instance, this can be the ability to control the component's internal state in order to set up quickly the context of a test case. It can also be the ability to let the component become aware of the fact that testing is happening, in case it is test-sensitive. This aspect will be treated more in depth in the next section, concerning runtime testing.

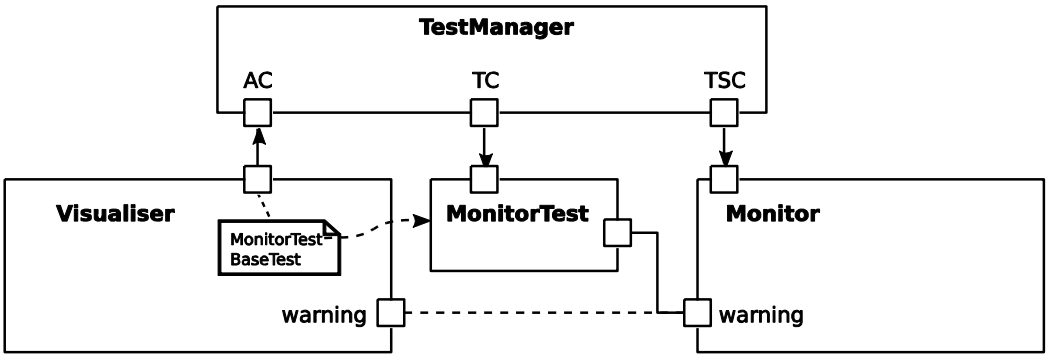


*Figure 3: Built-in Testing in Action*

The second facet of BIT is concerned with the association of test cases with the component, whose original goal was to allow the component to carry out all or part of the integration testing by itself. The requirements of the component on its execution environment (i.e.: the platform on which it is running, the physical components with which it is linked, the components on which it relies for providing specific services) can be validated using test cases contained in the component. That way, the components can perform much of the required system validation effort automatically and by "themselves" (Gross, 2005; Brenner et al., 2007). Figure 3 presents schematically the validation using the BIT infrastructure. The *Visualiser* component needs to test the *Monitor* component on which it depends. Via the AC interface, one of the BIT interfaces, the *Visualiser* contacts the *TestManager* component, providing the *MonitorTest* test. *TestManager* takes care of the connection between the test and the *Monitor* component, let the *Monitor* component know that a test is taking place via the TSC interface (another of the BIT interfaces), and finally report the result of the test to the *Visualiser*.

In the case of a data-flow organisation, only some tests might be possibly done in this way (those directly validating the platform compatibility and the hardware). As seen in the previous section, it is not possible to provide tests to ensure that the requirements on the other components are fulfilled, simply because components do not have any expectations on the other components. Nevertheless, we are going to present approaches that rely on the basic idea of embedding test cases with components and see how this may facilitate testing and maintenance of the system.

This distribution of the responsibility of validating the component's environment to the components themselves is very interesting for large-scale dynamic systems. It can help to maintain the independence of each of the participating components which are likely developed by

different teams as tests can be decentralised and associated with each component. Another interesting property of distributing the test definition is that it helps to keep the tests synchronised with the respective version of the components. For instance, when a component is updated to support additional functionality, the tests to validate the functionality must be updated simultaneously. Associating the tests with components also means that the testing infrastructure can automatically benefit from the dynamicity of the component infrastructure. Updating the testing information alone can be useful because test cases themselves can be erroneous, or for extending the test coverage while keeping the system running.

## Finding dependencies between components

As previously mentioned in Section *Properties of Publish/Subscribe Platforms*, and as we will see in more details later on, one important piece of information needed to perform integration testing efficiently is represented by the dependencies between the components. Usually in publish/subscribe systems, and opposed to other component-based architectures, the connections between components are not expressed explicitly, although one way to handle this lack of information would be to be very pessimistic and consider that every component interacts with every other component, the combinational explosion of the interactions to test would render this approach unusable when applied to large-scale system. In other words, the goal of finding dependencies between the components is to allow a better *test selection*. The dependency information can also be useful when the system is modified, so that the testing happens only on the parts which could be affected by the modification, and not on the whole system.

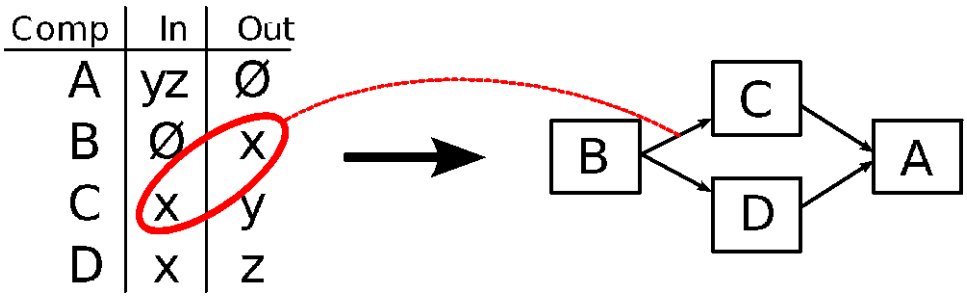


*Figure 4: Example of dependency graph computation*

In order to define a component dependency graph of the whole system, for each component one has to determine the components that might be triggered (executed) after the component has been triggered. In publish/subscribe platforms, components' interfaces are identified by the type of data which will be passed. An output interface corresponds to the generation of a given type of data. An input interface corresponds to a type of data which will trigger the component's execution. Therefore, constructing the dependency graph translates into listing every component of the system, for each of them list the types of data they listen to, and consider them dependent on every component generating events of this type of data. Figure 4 shows an example of the dependency graph computation. The table on the left-hand side represents the information on the components obtained from the architectural description. As component *B* generates data of type *x*, both components *C* and *D* are dependant on this component.

In some platforms, the input might be more precisely defined, for instance by the use of rules which filter the input data also according to their content. This is the case in the Data Distribution Service (DDS) standard (Object Management Group, 2007) in which a *topic* can be refined by

using *topic_expressions* (using a syntax derived from SQL). In this case, it might happen that a component produces a type of data which interests a second component but, due to the content of the data, this second component is never triggered. Unless this case can be detected statically, it is necessary to follow a pessimistic approach which considers that there is a dependency between the two components.

When reconfiguring the system at runtime by introducing a new component, it can happen that some components which are no longer part of the system have left data affecting this component. It is important to detect this situation during the acceptance process of the new configuration. Nevertheless, these components should not be included in the dependency graph because once the persistent data is received the actual behaviour (no more data) would be different from the behaviour of the component being active. The way to handle this persistent data will not be different from the rest of the persistent data during runtime testing, which is described in Section *Runtime Testing for Component-based Platforms*.

One of our assumptions on the system is that it is possible to statically know the full interface of the components. Studies have also been done on dependency computation of even-driven systems where the interfaces are implicit. This happens when the components are not explicitly defined, for instance when event filtering is done within the component via manually-written code, or when the events generated are not declared beforehand. Holzmann and Smith have proposed a method to compute the dependencies statically (1999), given that the source code is annotated (using the *@-format*). Memon et al. have introduced a tool (2003) to obtain this information by running the system and actively triggering every possible event (the tool is focused on GUI application based on Java).

We will now discuss two different methods for testing the integration of an event-based system.

## Testing random event sequences

Unit-testing checks each of the components in isolation. However, the integration of the components brings intrinsic problems which have to be detected. Once the system is entirely constituted, an event (or a data) can affect several components simultaneously, in which case they might be executed in any arbitrary order, which might cause unexpected behaviour. Components may not support data with specific values, as generated from another component. Moreover, components having a state (i.e.: non pure components) can be affected by the order in which the data is received. In order to detect such cases, a method that can be used is generating a random sequence of data and verifying that the system reacts correctly to it.

Defining random sequences of events is straightforward. Much more difficult is it to know how to generate the data associated with the event and to know whether the system reacted correctly to them. In case the input data is just events (without containing more information), it is possible to generate them automatically, but if the input contains complex data, this cannot be created randomly in an easy way. In order to generate the input data, a solution consists of relying on the unit-tests of each component. If the unit-tests are specified via a high abstraction model, it might be possible to use the specification to directly generate the input data. When this approach is not possible to obtain the input data it is necessary to execute the test cases entirely. Once the input data has been generated, the components triggered by these inputs generate outputs, which will, in turn, trigger other components. Eventually, from the generated inputs, all the components are triggered.

For every instance of generated input, the sequence of events increases, and every component which has received input must be validated. An oracle compatible with the component and adapted to the sequence of events must be available. There are several complementary ways to obtain oracles:

- The unit-test of a component might have an oracle valid for any input data. Typically, the oracle asserts some invariant properties specific to the component. This type of validation might also be found as *runtime monitors*, which are used to estimate the system health. If such monitoring components are present they can be also used are oracle.
- Some oracles are generic by focusing on common properties that hold true for any component. For instance an oracle might verify that the function terminates within a bounded time, that there is no memory leak or, as used in (Yuan & Memon, 2008), that the component does not crash (a non-handled Java exception).
- In the context of runtime testing it is possible to compare the outputs of the production configuration with those of the configuration under test. Different outputs indicate a potential failure. Whether this failure is in the production configuration or in the one under test has to be decided by the system integrator.

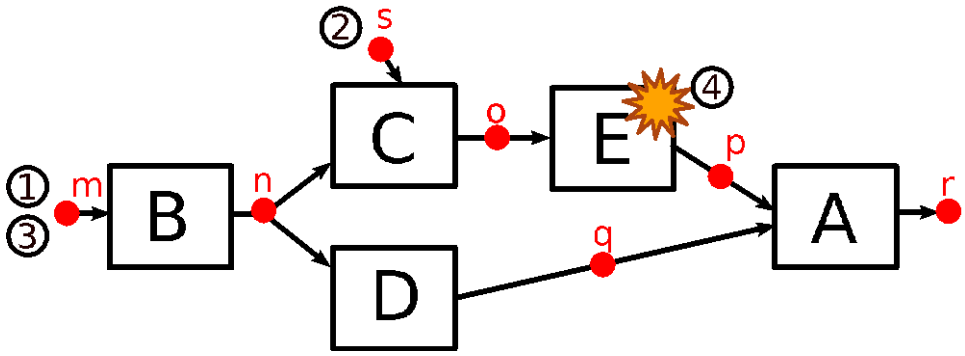


*Figure 5: Example of random sequence testing. 3 inputs are generated, which leads to a crash in component E. The crash is detected as a failure to pass the test case by the oracle.*

Figure 5 presents an example of one of the test case generated by the method of random event sequences. Three input data are inserted, in the order *m*, *s*, *m*. After each input, the data flow along the components as in the final system. The third insertion of input leads to a crash in component *E*. This crash is detected by a generic oracle, which, therefore, detects a failure in the test case.

A difficulty with the generation of random sequences of input data is the high number of integration tests created: it is infinite[2]. Depending on the context, it might be possible to simply decide that enough tests has been performed after a certain time span as passed, but it is still desirable to first execute the tests which are more likely to expose errors. A selection on the sequences can be done by limiting the sequence size to a maximum number of events. Moreover, as the integration testing focuses on the behaviour when multiple components are involved, it is useful to restrain the number of times a type of data is generated (for instance one or two times), so that more interaction happens and the number of possible combinations is reduced. Going

[2] Because each input event could be repeated any number of times.

further, Yuan and Memon have proposed a test selection technique (2008) which takes into account the interdependencies between events. They propose to concatenate only events which involve different components but for which the components have either a direct interaction or an indirect interaction (defined by *predicates*, which informally correspond to interactions with a common third component). This method has the advantage that if sufficiently many unit-tests are available, it can execute a very large number of test cases. However, as the oracles have to be broad enough to support any order of inputs to the components, this method does not allow to detect if the behaviour of the integration is incorrect, i.e. that for each specific input sequence the correct outputs were generated. This is why it is interesting to combine this method with a second method dedicated to test some typical interactions between the components. We will discuss such a method in the following sub-section.

### Testing specific data-flows

In some cases the integration of components might be incorrect without leading to strong misbehaviour of each component. For instance, two components which are supposed to communicate together might actually be publishing and subscribing to different types of data, leading to the subscriber not being triggered at all. It can also happen that due to a misunderstanding during the specification phase, the data types are not processed in the same way (e.g.: X and Y co-ordinates inverted). In order to detect these errors, specific test cases validating the behaviour of the integrated system must be executed. When an error on a running system is found and fixed, similarly, one would need to be able to specify a specific sequence of events with a specific oracle for regression testing.

If the system is entirely based on the call-reply model, it is possible to define a test case for the client component (from which the call is originated), so that the complete interaction sequence may be tested: every client component has explicit requirements on its servers, so it can have test cases validating those requirements on its own. As can be seen in Section *Properties of Publish/Subscribe Platforms*, in the case of data-flow model, defining such test cases requires knowledge of the integration by the system tester. Moreover, it is important that the test cases written can be reused as much as possible, even if the system is updated or reconfigured.

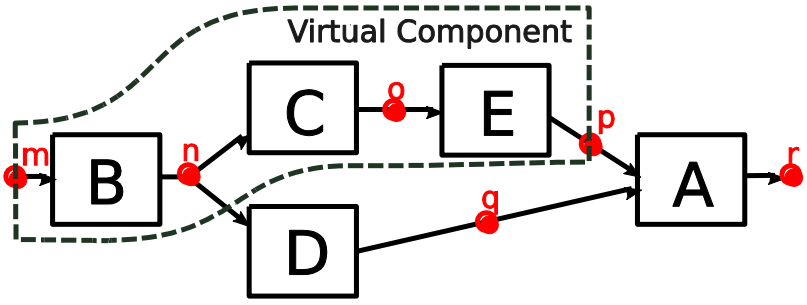


*Figure 6: Example of a virtual component. The virtual component defined solely by its inputs (m) and outputs(p) and automatically encompasses the behaviour of the components B, C, and E.*

Our approach consists of defining *virtual components* which correspond to a specific data-flow. The virtual component is only delimited by a set of inputs (data types) and a set of outputs (other data types). In contrast to the usual components, there might be inputs or outputs going into or out of the virtual component without being part of its interfaces. For instance, in Figure 6 the type *n* goes from the inner component *B* to the outer component *D*. This essentially means that

some inputs or outputs will never be used to validate the behaviour of this specific flow. As seen in Figure 6, the virtual component inherently encompasses all the components which are used in the flow of data between the input and the output events. Compare to normal composite component, this permits a high flexibility in case of modification around the virtual component, i.e. if components are added or removed within the data-flow the set of inner components will evolve automatically. Referring to the example displayed in Figure 6, if the components *C* and *E* were replaced by three components *X*, *Y*, and *Z* having the same functionality, the virtual component would still be valid and pass the tests. In case the component *C* is mistakenly removed, the virtual component would still be valid, but some test cases would likely not pass, as *E* would never receive any data.

Another advantage of using virtual components over normal composite components is that it is possible to define several overlapping flows. In Figure 6, it would, for example, be possible to define a flow between *m* and *q*, although this would signify that component B is contained simultaneously in several components, which is a forbidden property in component-based systems.

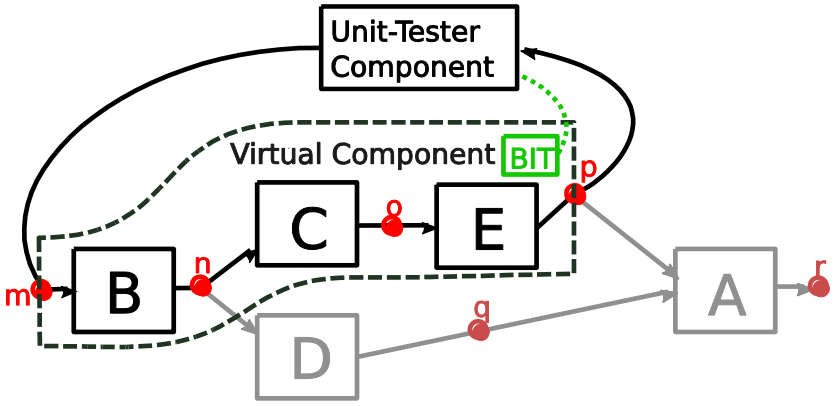


*Figure 7: Testing a virtual component consists of executing unit-tests associated with the component via the testing interface provided by BIT*

The virtual component does not have any content, it is used only has a placeholder to represent a specific functionality that the testers would like to test, and to associate with the flow a list of test cases. This association can be done by using a BIT infrastructure, as the virtual component is just another component, after all. Each test case is written as a unit-test for the virtual component, which can be easily handled since it is similar to the unit-tests for the normal components, as shown in Figure 7. This is a very interesting effect of our approach: integration testing becomes in practice very similar to unit-testing. This means a large part of the infrastructure used to run unit-tests (associating test cases with components, executing the test cases, etc.) can be reused. Even more important, this signifies that the engineer in charge of the testing can apply his knowledge of the usual unit-testing to integration testing.

Effectively, the unit-tests are executed on the sequence of components encompassed by the virtual component. The execution takes place in the integrated system, resulting in the real components *B*, *C*, and *E* being tested. The computation to determine which component is part of the virtual component is not necessary for running the test cases, this is done implicitly by the flow of data going from the inputs until the expected outputs.

Typically, this approach allows defining a handful of complex interactions in the system which must be tested extensively. The test cases can sequences of events of arbitrary length, and the oracle can precisely verify the output data conform to the specifications. It is *complementary* to the previous method which generates random sequences of events.

After having reviewed techniques to handle the integration testing in publish/subscribe architectures, we will concentrate on the specific difficulties coming from runtime testing.

## RUNTIME TESTING FOR COMPONENT-BASED PLATFORMS

Publish/subscribe systems are based on components, which facilitate the modification of just one part of the system. In addition, those components are loosely coupled, so it is technically simple to reconfigure the system at runtime. The reconfiguration itself is straightforward: the components are simply added to, or removed from, the pool of components running in the system. As the component connections are not explicit, but implicit in the event types that are published and subscribed to, there is no need to let the other components know about the modification. This is one of the reasons why publish/subscribe runtime architectures are readily used in systems with high availability requirements. Such systems cannot be stopped only because of a correction being introduced, a new feature being added, or an existing function being removed because the hardware supporting it is about to be removed. Therefore runtime reconfiguration plays a big role in the context of high availability.

However, the fact that a new configuration will work seamlessly, is not guaranteed a priori. Every re-configuration activity should follow the same verification and validation processes, in order to provide the same confidence in the new configuration as it was the case for the old system. Moreover, due to resource limitation, it is not always possible to duplicate the system for testing the new configuration while the previous one is used in production. This is a very likely situation with large-scale systems. To test the new configuration it is then possible to use runtime testing: execute test cases on some components while the whole production system keeps running normally.

### Background

Some studies on the topic of *regression testing* can be related to this problem. The main question of this topic concerns the validation of a system after some modifications. There are alternatives to runtime testing. As one of our assumptions is that components can be black or grey boxes, because it is likely that the source code of some components will not be available, approaches that require access to the source code cannot be applied in our context.

Although model-based approaches can be a solution (Orso et al., 2007; Muccini, Dias, & Richardson, 2005), 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 derive state models dynamically from usage traces (Mariani, Papagiannakis, & Pezze, 2007). As an advantage, these models will be smaller, as they will be restricted to the way the component has been used earlier in the system, thus, leaving out all the features of the component not relevant in the current context. There exist also some methods (Stuckenholz & Zwintzscher, 2004), that address this problem from a formal point of view, providing a way to measure system updates and finding conflict-free configurations.

### Runtime testability

As seen in Section *Properties of Publish/Subscribe Platforms*, in order to be able to implement runtime testing in a reliable fashion, the middleware needs to provide test-isolation and test-awareness. Let us have a deeper look at the difficulties coming along with runtime testing.

There are two orthogonal dimensions that lead to problems during runtime testing:

- **Component state.** If the component has states, or in other words its outputs depend not only on the current input but also on the previous ones, the testing data might interfere with the production data. For instance in a simple component providing as output a number corresponding to the sum of all the inputs it has received, a test case running concurrently to the normal usage will likely lead to wrong outputs both for the test case and for the normal usage.
- **Component side effects.** If a component's behaviour does not only involve its inputs and outputs, but also has side effects on the ``outside world'', it might be unsafe to perform testing at runtime. For instance, it is not possible to test at runtime a component driving the arm of a robot while the robot has to continue moving normally. As another example, the side effect could be the communication with a database in an external system. It would be a rather bad idea to execute test cases modifying the database of real bank accounts!

These two dimensions corresponds to the notion of *purity* in functional programming. A component which has either states or side effects is considered *test-sensitive*. Unfortunately, often components in real systems are test-sensitive. We could even go further and say that a system which has no component with side effect is useless, as it does not communicate with the outside world.

To overcome the limitations brought up by the test-sensitiveness of the components, the notion of test-awareness is introduced. Test-awareness allows being sure that a component can distinguish the events corresponding to the testing process from the events of the production process. In case of state, components can store a second state in order to handle the testing events separately. In case of side effect, the components' developers can choose either not to perform the function requested at all, or to use a simulation of the real world.

Still, not all components might be able to provide appropriate test-awareness functionality (e.g. because of lack of development time or difficulty of the implementation), or some functions might be considered too dangerous to try even if the test-awareness is implemented (because, after all, some things might go wrong during the test). In order to allow the testing framework to handle those cases, a test-control interface is needed for each component, that gives the component the possibility to restrain the testing to only some test cases at runtime, and allows more when the testing is done off-line in a controlled environment.

From this information, the *runtime testability ratio* metric can be derived. It is the ratio between the percentage of the system being testable at development time versus the percentage of the system being testable at runtime. The actual value depends on the way the testing coverage is measured, such as code coverage, model coverage, or coverage of usage scenarios. Nevertheless the runtime testability measurement gives an indication on how much is lost on the acceptance quality when testing has to be done at runtime.

## Avoiding interferences between production and testing domains

In order to maximise the runtime testability ratio, one has to maximise the number of test cases that can be performed during runtime. This can be done through minimizing the interferences between the running system and the tests being executed concurrently during runtime.

One way of minimizing interference is to add a special flag to the testing data in order to be able to discriminate it from production data. We call this approach *tagging*. During testing, components subscribe to events with this specific flag, and *every* output event generated when handling this data must also be tagged with the test flag. The principal advantage of this method is that one component can receive production as well as testing data and process outputs accordingly. In case of testing, a different component state can be used, or some side effect actions omitted. Components which are test insensitive can be used unmodified with this scheme if subscription to the test data and setting the test flag on the outputs is managed automatically by the framework or via a simple wrapper. For the components with more complex functionality, it is not possible to verify that the output data depend exclusively on input data from the production world or exclusively on input data from the test domain. It is up to the component to separate the handling correctly. Nevertheless, the framework can do some sanity checking by verifying that the output data is from the same domain as the input data that triggered the function call.

Figure 8 shows the usage of tags for data isolation on a system for which the reconfiguration consist of updating the component *E* by *E'*. Components with full black lines process the production data. Components with dotted outlines process the testing data. Components which are not modified process data for both production and testing. Components which are going to be removed, only process the data from the old configuration, the production data. Components which will be added, only process data from the future configuration, the testing data.

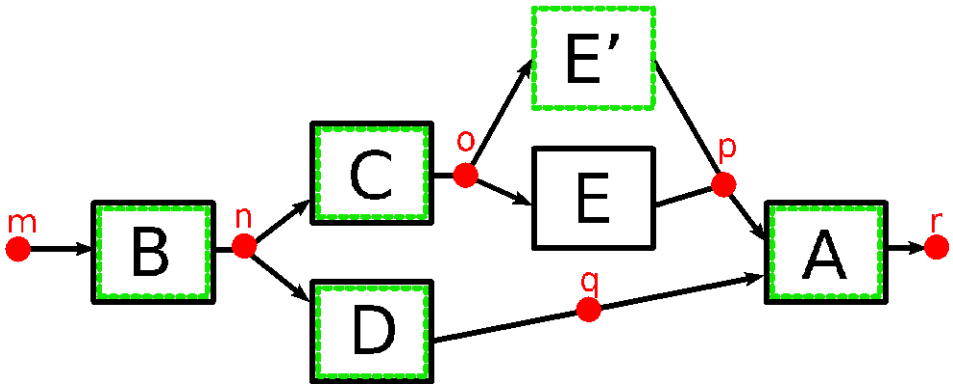


*Figure 8: Example of data isolation for runtime testing.*

In case data persistence exists in the publish/subscribe architecture, the testing could also be done by using the data left by previous components. This is persistent data that has the same type as the inputs expected for the testing, and which has been produced by components during runtime until the beginning of the validation process. As this data will potentially be read by the new components when they will be started in the production mode, starting them during testing with the same data available approximates their real behaviour better. So, before starting the test, the persistent data which is used as input of one of the components under test should be copied along with the test tag.

Another approach consists of a technique to allow the component to be aware of the testing (Suliman et al., 2006; González, Piel, & Gross, 2008). Obviously, this awareness has to be used with parsimony because it adds more burden to the component developer to handle the various cases, and also with care because the component's behaviour, when under test, must differ as little as possible from the normal behaviour. The basic information sent to the component concerns the beginning of a test and its end. This enables the component to duplicate its state or to set up a

simulator for its outputs. The provided information should also detail whether the component is going to be operating exclusively for the test, or whether it will also have to respond to production events. In particular, when updating a component, during the test the previous version of the component must only handle the production data, while the newest version must handle only the testing data.

It is important to mention that interference between production operation and testing can also occur at the non-functional properties level, i.e. there might be shortage of available processing power, memory usage, communication bandwidth, etc. Due to lack of space it is not possible to treat in detail this broad subject. Nevertheless, this is also an aspect that must be taken into account during runtime testing. The techniques such as Quality of Service or system health monitoring can be used in order to ensure that the test case execution never prevents the production operation to be executed normally and to schedule the testing at moments when the system's load is low.

## Reducing the amount of test cases

The testing process can be costly in terms of time and resources. When a modification of the system configuration is requested, it is best to reduce the delay to a minimum before it takes place. A typical modification affects only a very constrained part of the whole system, in terms of components as well as in terms of behaviour. For example, updating a component comprises only removing the component and replacing it by another component which has a similar interface and a close behaviour. Moreover, the previous configuration has passed the testing process. Therefore, it seems rather logical that not the whole testing process has to be repeated. Only the tests which might have a different outcome should be re-executed (and the newly introduced tests).

The general idea is to re-test only the parts which are affected by the changes. In integration testing, this means every interaction that involves a component which is affected by the changes.

When using the *random event sequence* technique, only sequences which involve an event affected by the modification have to be re-tested. An event (or data type) is affected when one of the subscribers has changed and, therefore, the reaction of the system to it might change. The fact that a producer has been modified must not necessarily be taken into account, as either the event is explicitly generated during the test using the test cases (so the modified component is not involved) or the event is generated through the modified component, in which case the sequence would be tested anyway, as the component has subscribed to the previous event.

With the validations based on *virtual components*, similarly, every virtual component containing a component affected by the modification will have to be revalidated, because the output data might be affected by the removed or added component. If a virtual component either contains a component that will be removed, or will contain a new component, it will have to be retested. It is important to mention that if the virtual component contains only components with dependencies on a modified component, it is not necessary to retest the data-flow as the interaction with the modified component would not be triggered anyway. Of course, new virtual components should be validated, and similarly, if a virtual component has new test cases, they should be executed before accepting the new configuration.

In case the architecture is hierarchical, in other words, there are components which are composed by a set of sub-component, the re-testing has to be done recursively. Each composite component containing a modified component has to be considered as modified as well. The testing technique is applied starting from the modified component, going through each parent

component, until the topmost level is reached. Theoretically, there is no order in which the levels of hierarchy have to be re-tested, because if a test case can detect a wrong behaviour it will be eventually run, whichever is the execution order. Nevertheless, a problem at a low level of component hierarchy can lead to test cases failing at this level but also at all the higher levels. Therefore, starting from the lowest level (the level at which the modification took place) allows to pinpoint more directly to where the error comes from.

## RUNTIME INTEGRATION TESTING: A CONTROLLED EXPERIMENT

In order to illustrate the methods outlined so far, as well as to summarize the whole process of validating the integration, we present the usage of those methods on a simplified sub-system of the MSS domain. This example is an experiment of reduced scale in which we apply and assess the key concepts of the presented methods. This scenario involves validating the integration of the assembled system, and keeping the quality at the same level after an update of the system has taken place.

### AIS: Automatic Identification System

The Automatic Identification System (AIS) is a worldwide adopted standard used for vessel identification (International Telecommunication Union, 2001). Ships must broadcast over radio their static voyage data (cargo type, destination, etc.) as well as their dynamic positioning data with a variable report rate that depends on the ship's speed and rate of turn. These messages are received by other ships and by the coast authorities, who can use the data for traffic control, collision avoidance, and assistance. In order to provide a full coverage of the Dutch coastal waters, many AIS base stations are distributed along the coast of The Netherlands as well as in buoys at sea. The messages received by these stations are then relayed to the central coastguard centre, and displayed on the screen of the control room.

The amount of information received cannot be grasped by humans. At any given time, more than 800 AIS identified ships can be sailing in Dutch waters, sending signals every so many seconds. Therefore, automated processing is essential. One particular automatic 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. The detected errors are shown as warnings to the operator next to the ship's icon on the display to indicate that the information is uncertain. Moreover, because AIS messages are broadcasted and can be received by many different base stations in range at the same time, multiple occurrences of the same message will be received in the central data centre with some seconds of delay in between, requiring some initial filtering and cleaning before the data is displayed.

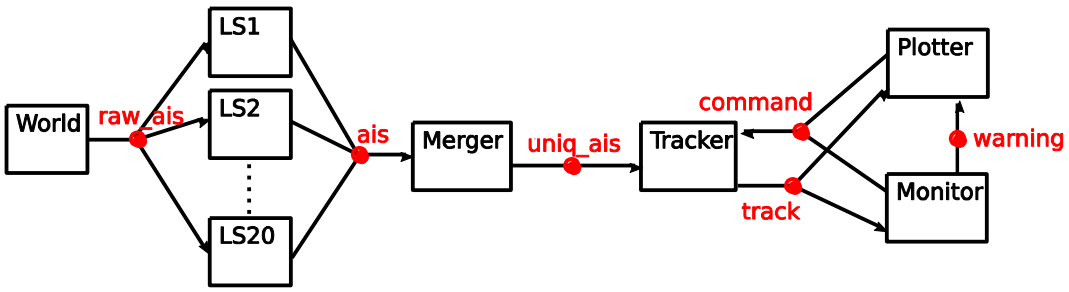


*Figure 9: Global view of the AIS processing system.*

## The initial system

The global organisation of the system is shown in Figure 9. The *World* component generates the same data as boats would in reality. It would obviously not be present in a real system, but would be replaced by the ships' AIS emitters. It actually replays a record of the AIS messages received by the Dutch coastguards during a week of normal operation. The *LS* (Local Station) components simulate the individual AIS receivers by transmitting only messages happening in the coverage zone specified as a parameter. The *Merger* component receives the data from the several *LS* components and removes the duplicated messages (which happen when a boat is in a zone covered by several receivers). The *Tracker* component converts the messages into a database containing the information about each ship. This information can be accessed via a specific protocol based on receiving commands and answering with `tracks`. The *Monitor* component passively observes all the data and detects inconsistencies in the information sent by the ships. In case a problem is detected, a `warning` is generated. The *Plotter* component displays on a map the ships, their associated information, and the warnings. The publish/subscribe framework (OpenSplice[3]) is based on the DDS standard. The components are implemented in Java. They are also described using a specific Architecture Description Language (ADL), in order to define explicitly the inputs and outputs, and to associate them with test cases.

## First integration acceptance

Before the system is started, and after every component has passed the unit-testing phase, the validation of the integration of the components must take place. During this validation the *World* component is not executed. Virtual components can be used to test some specific interactions. For instance a virtual component can be set between `uniq_ais` and `warning` to verify that *Monitor* and *Tracker* are compatible at the protocol level. A sequence of `uniq_ais` is sent at a high frequency (which is not authorised in the AIS protocol). The oracle verifies that the correct warning is generated.

It is also possible to test the core of the flow: *Merger* followed by *Tracker*. To do so, a virtual component is set up with `ais` and `command` as inputs, and `track` as output. Several test cases can be associated. The first test case has as input a series of identical messages and a command requesting the information for the ship specified in the messages. The oracle checks that only one data is returned. A second test case has as input several different messages, corresponding to several ships. The command input is more complex than in the first test case: it requests information about several ships, including several times the same ship and a non-present ship. The oracle validates whether the information in the `track` is adequate with respect to the requests.

```
<virtual-composite name="flowCore">
  <interface name="AISIn" role="subscribe" signature="ais" />
  <interface name="CmdIn" role="subscribe" signature="command" />
  <interface name="TrackOut" role="publish" signature="track" />
  <test provider="JUnitProviderFlow" name="Dup"
                                     definition="TestDuplicate" />
  <test provider="JUnitProviderFlow" name="MS"
                                     definition="TestMultiShip" />
</virtual-composite>
```

[3] http://www.prismtechnologies.com/opensplice-dds

*Listing 1: Definition of a virtual component with two test cases in an ADL file.*

As an example, Listing 1 is an abridged version of the ADL file that can be used with our framework for specifying the second virtual component. The three `interface` definitions specify the inputs and outputs of the component. The `test` definition corresponds to one of the BIT interfaces, and associates test cases with the component. The provider `JUnitProviderFlow` indicates a class used to run this kind of test (it is generic to all unit-tests for virtual components). The `definition` argument indicates a JUnit[4] class in charge of the test case. As we can see, the notion of virtual component conveniently allows us to reuse the usual infrastructure for unit-testing.

```
public class TestDuplicate extends JUnitFlowTest
                           implements TrackListener {
  private GenericDataWriter<AM_ais_msgDataWriter> aisW;
  ...
  @Before
  public void setUp() {
    aisW = writers.get("ais");
    readers.get("track").bind(this);
    ...
  }

  @Test
  public void noDuplicateMessageInRow() {
    AISMessage[] result;
    AISMessage in = new AISMessage("11OGQ0?0EpeVNE2:Hjakf0");

    for (int i=0; i < 10; i++) {
 int ret = aisW.w.write(in, DDS.HANDLE_NIL.value);
 Assert.assertTrue("Error while writing message.",
                           ret == RETCODE_OK.value);
    }

    result = getReceivedTracks(1.0f);
    Assert.assertTrue("Duplicated message not detected.",
            warnings.length == 1);
  }
  ...
}
```

*Listing 2: Definition of a test case for validating the flow.*

An extract of the unit-test `TestDuplicate` referenced in the ADL file is presented in Listing 2. The Java class extends the helper class `JUnitFlowTest` which automatically sets up the component in the system with the topics specified in the ADL file. The `setUp` method is executed at the beginning of the test case to complete the initialisation. The method `noDuplicateMessageInRow` corresponds to one test case. It emits 10 times a identical AIS message, and reads the tracks received during one second. The reception of the tracks is handled

[4] http://www.junit.org

by implementing the `TrackListener` interface (the methods corresponding to this interface are not shown here for brevity).

### Update of the system

Once the system is running, comes a moment when it is reconfigured. On the system, we update the *Monitor* component in order to detect more type of inconsistencies in the AIS data sent. To do so, within the "acceptance perspective", the *Monitor* component is removed, and a *Monitor2* component is added, with the same inputs and outputs. Before the new version of the system can be transferred to the "production perspective", it must pass the validation phase. All the virtual components which are affected by the modification are tested. Here, only the first virtual component, between `uniq_ais` and `warning`, is affected.

The *Tracker* component is instructed to use both the testing data and normal data via its testing interface. The *Monitor2* component is instantiated, instructed to use testing data only, and started. The other components, such as the *Monitor* or the *Plotter* do not receive any specific instructions, so they continue running normally, using only the production data. The unit-test component is then executed. The input data is generated with the "test" tag (corresponding to a *domain* in OpenSplice), and similarly the component subscribes to the test data only for receiving the output of the data-flow. If one test case fails, an error is displayed to the system integrator, who will have to provide a new version of the component which passes the test. Once the test case is passed, the transfer to production can be executed: the new components are restarted, and the deleted components are stopped.

## FUTURE TRENDS

In this chapter we have presented approaches to validate the integration of publish/subscribe systems in an initial configuration and at runtime. One of the assumptions we have taken is that the test cases were either already available or would be provided by the system integrator. A trend in the testing area seems to emerge concerning *automatic test generation*. The goal is to reduce the burden of writing test cases manually by using high level information regarding the expected behaviour of the system (expressed as a model) and transforming it into test cases. Ideally, the behaviour can be captured in models which are useful also for the rest of the development process, such as the documentation in UML, or the implementation of a component with a Finite State Machine. This has been considered for instance in the works of Abdurazik and Offutt (2000), Chandler et al. (2005), Hartman et al. (2005). The development of model transformation techniques should help to go towards this direction.

As this chapter was focused on testing, the subject of *verification of the integration* has not been treated here. Complimentary to the validation, the verification is another effective way to improve the quality of the system. One of the major problems encountered in this domain is when dealing with complex systems the number of states explodes. This is especially noticeable when verifying the component interaction in publish/subscribe systems because of non-determinism due to the concurrent execution of components. The latest works in this domain, such as presented by Baresi et al. (2007) have focused on reducing this combinational explosion, so that it can become a more usable approach. Moreover, there has also recently been work to allow verification with runtime evolution (Deveaux & Collet, 2006).

Concerning the validation aspect, it might be worthy to explore the *synergy between runtime evolution and validation*. Based on approaches used for regression testing, we have seen that it is

possible to reduce the amount of test cases to execute, because it is known that the running configuration has been validated. It might be possible to reduce the number even further by observing the running system and discarding test cases validating non-used data types, or non-used components. It could also be possible to leverage the fact that the two configurations are running simultaneously to compare the behaviour of the configuration under test against the validated configuration. The configuration under test would receive real inputs, but the outputs would be only compared instead of being really transmitted.

Finally, it is important to mention that all these discussed approaches will be possible to be applied only once a *user interface* provides to the system integrator the functionalities. Following the status of each test case (i.e. failed or passed) for every component and interaction between the components has to be straightforward. We have started work in this direction for a client-server framework (González, Piel & Gross, 2008). Defining virtual components by selecting inputs and outputs and associating test cases with it should also be an easy operation. It should also facilitate the runtime evolution of the system by allowing the user to view the current configuration and in parallel deriving it into another valid configuration.

## SUMMARY AND CONCLUSIONS

A large number of event-based systems have high availability and high reliability requirements. Ensuring the high software quality of such system is a must, in its initial configuration and also whenever the configuration evolves. Validating each component individually is not sufficient for testing the whole system; it also requires validating the integration of those components together. In event-based systems the dependencies between the components are not explicit, and, when the processing model follows the typical data-flow model, components have no expectations on the behaviour of the other components. As components for event-based architectures are loosely coupled, this type of architecture is well adapted for runtime reconfiguration. Moreover, when the system evolves at runtime and if due to its high complexity it can not be duplicated, the testing must also take place at runtime. In this situation, the testing must not interfere with the normal operation.

In this chapter, we have presented two different and complementary methods to test the integration. One is based on the random generation of a high number of event sequences and on already available oracles, in order to find a state of system malfunctioning. The second one uses a limited number of predefined data-flows which must respect a precise behaviour. It introduces the notion of virtual components, which have the advantages over usual components that they can overlap each other, and the exact set of inner component automatically follows the system's evolution. The two presented methods rely on several known and proven techniques such as unit-testing or Built-In Testing.

The usage of Built-In Testing allows keeping with each component a particular set of test cases. Thanks to this, at any time during the life cycle of the system, the test cases of any component are available, even at runtime. In this chapter, the specific aspects of runtime testing have been presented. First, components require to be assessed by the developer in order to identify state-dependant behaviour and resources outside the component which should not be accessed during testing or cannot be shared. Test-sensitive components can then make use of a special test interface to be aware of the testing process. At reconfiguration time, before a modification is accepted, the parts of the system which might behave differently must be re-tested for integration using this set-up and the integration testing techniques previously introduced. This ensures the high quality of the system at any given time of its lifespan.

## ACKNOWLEDGEMENTS

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

## REFERENCES

Abdurazik, A., & Offutt, J. (2000). Using UML collaboration diagrams for static checking and test generation. In A. Evans, S. Kent, & B. Selic (Eds.), UML 2000 - the unifed modeling language. advancing the standard. third international conference, York, UK, October 2000, proceedings (Vol. 1939, pp. 383-395). Springer.

Baresi, L., Ghezzi, C., & Mottola, L. (2007). On accurate automatic verification of publish-subscribe architectures. In Icse '07: Proceedings of the 29th international conference on software engineering (pp. 199–208). Washington, DC, USA: IEEE Computer Society.

Brenner, D., Atkinson, C., Malaka, R., Merdes, M., Paech, B., & Suliman, D. (2007). Reducing verification effort in component-based software engineering through built-in testing. Information System Frontiers, 9 (2-3), 151-162.

Chandler, R., Lam, C. P., & Li, H. (2005). Ad2us: An automated approach to generating usage scenarios from uml activity diagrams. In Apsec '05: Proceedings of the 12th Asia-Pacific software engineering conference (apsec'05) (pp. 9-16). Washington, DC, USA: IEEE Computer Society.

Deveaux, D., & Collet, P. (2006). Specification of a contract based built-in test framework for fractal.

Embedded Systems Institute. (2007). The Poseidon project. http://www.esi.nl/poseidon.

EU Commission. (2007, October). An integrated maritime policy for the European Union. European Commission, Maritime Affairs.

Eugster, P. T., Felber, P. A., Guerraoui, R., & Kermarrec, A.-M. (2003). The many faces of publish/subscribe. ACM Computing Surveys, 35 (2), 114-131.

Gao, J. Z., Tsao, H. J., & Wu, Y. (2003). Testing and quality assurance for component-based software. Artech House.

Garlan, D., Khersonsky, S., & Kim, J. S. (2003, May). Model checking publish-subscribe systems. In International spin workshop on model checking of software (pp. 166–180). Portland, Oregon, US.

González, A., Piel, É., & Gross, H.-G. (2008, September). Architecture support for runtime integration and verification of component-based systems of systems. In 1st international workshop on automated engineering of autonomous and run-time evolving systems (ARAMIS 2008). L'Aquila, Italy: IEEE Computer Society.

Gross, H.-G. (2005). Component-based software testing with UML. Heidelberg: Springer.

Gross, H.-G., & Mayer, N. (2004). Built-in contract testing in component integration testing. Electronic Notes in Theoretical Computer Science, 82 (6), 22-32.

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

Holzmann, G. J., & Smith, M. H. (1999). A practical method for verifying event-driven software. In Icse '99: Proceedings of the 21st international conference on software engineering (pp. 597-607). Los Alamitos, CA, USA: IEEE Computer Society Press.

International Telecommunication Union. (2001). Recommendation ITU-R M.1371-1.

Lockheed Martin. (2008). Maritime safety, security & surveillance integrated systems for monitoring ports, waterways and coastlines. http://www.lockheedmartin.com.

Mariani, L., Papagiannakis, S., & Pezze, M. (2007). Compatibility and regression testing of COTS-Component-Based software. In Icse '07: Proceedings of the 29th international conference on software engineering (pp. 85-95). Washington, DC, USA: IEEE Computer Society.

Memon, A., Banerjee, I., & Nagarajan, A. (2003, November). Gui ripping: reverse engineering of graphical user interfaces for testing. In 10th working conference on reverse engineering (pp. 260-269). Piscataway, NJ, USA.

Michlmayr, A., Fenkam, P., & Dustdar, S. (2006a, September). Architecting a testing framework for publish/subscribe applications. In 30th annual international computer software and applications conference (compsac'06) (Vol. 1, pp. 467-474).

Michlmayr, A., Fenkam, P., & Dustdar, S. (2006b, July). Specification based unit testing of publish/subscribe applications. In 26th IEEE international conference on distributed computing systems workshops (icdcsw '06) (p. 34).

Muccini, H., Dias, M., & Richardson, D. J. (2005). Reasoning about software architecture-based regression testing through a case study. In Proceedings of the 29th annual international computer software and applications conference (Vol. 2, pp. 189-195). Washington, DC, USA: IEEE Computer Society.

Muhl, G., Fiege, L., & Pietzuch, P. (2006). Distributed event-based systems. Springer.

Object Management Group. (2007). Data distribution service for real-time systems, v1.2.

Oreizy, P., Medvidovic, N., & Taylor, R. N. (1998). Architecture-based runtime software evolution. In Icse '98: Proceedings of the 20th international conference on software engineering (pp. 177-186). Washington, DC, USA: IEEE Computer Society.

Orso, A., Do, H., Rothermel, G., Harrold, M. J., & Rosenblum, D. S. (2007). Using component metadata to regression test component-based software. Software Testing, Veri_cation and Reliability, 17 (2), 61-94.

Schieferdecker, I. (2007, February). The Testing and Test Control Notation version 3; core language (Tech. Rep.). European Telecommunications Standards Institute.

Stuckenholz, A., & Zwintzscher, O. (2004). Compatible component upgrades through smart component swapping. In R. H. Reussner, J. A. Stafford, & C. A. Szyperski (Eds.), Architecting systems with trustworthy components (Vol. 3938, p. 216-226). Springer.

Suliman, D., Paech, B., Borner, L., Atkinson, C., Brenner, D., Merdes, M., et al. (2006, September). The MORABIT approach to runtime component testing. In 30th annual international computer software and applications conference (Vol. 2, pp. 171-176).

Szyperski, C. (1998). Component software: Beyond object-oriented programming. New York, NY, USA: ACM Press and Addison-Wesley.

Thales Group. (2007). Maritime safety and security. http://shield.thalesgroup.com/offering/port_maritime.php.

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

Yuan, X., & Memon, A. M. (2008, August). Alternating GUI test generation and execution. In Testing: Academic and industry conference - practice and research techniques (taic part'08) (pp. 23{32). Windsor, United Kingdom: IEEE Computer Society.

Zhang, H., Bradbury, J. S., Cordy, J. R., & Dingel, J. (2006). Using source transformation to test and model check implicit-invocation systems. Science of Computer Programming, 62 (3), 209 - 227. (Special issue on Source code analysis and manipulation (SCAM 2005))

## KEY TERMS AND THEIR DEFINITIONS

**Component**: an abstract and unique compositional unit of high cohesion and low coupling with contractually specified interfaces for external communication and an execution context.

**Call-reply**: the model of event sequence when the communication between two components is based on a request and a response.

**Data-flow**: the model of event sequence when the communication between two components is based on the data transmission only in one direction.

**Validation**: development phase during which one ensures that an implementation does behave as defined in the requirements.

**Verification**: development phase during which one ensures that a model does conform to all the restrictions defined in the models of higher level of abstraction.

**Integration Testing**: testing of the behaviour of a set of components assembled in their final configuration.

**Test Sensitivity**: the property of a component that indicates to which extend the component can be tested without unwanted side-effects.

**Built-In Testing**: testing technique used for equipping components with the ability to check their execution environment, and their ability to be checked by their execution environment.