

to be updated. When the system cannot be duplicated, it is possible to rely on *runtime testing* [27].

There are two main difficulties with using a component at the same time in the *contract* of the production and in the context of testing. Firstly, 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* [24].

Secondly, when testing components 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 instance, 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 not executing at all the test case. Nevertheless, whichever approach is taken, it is first necessary to be able to define this *test sensitivity* [24] of the component. The platform must provide a way for the component to “tell the difference” between test and non-test data or event, this permit the components to be *test-aware*.

Handling test isolation and test sensitivity is described in Section 4.

### 3 Integration testing in event-based systems

~~As seen previously,~~ event-based platforms allows to decouple the system into small parts which handle a specific type of event (or data) but in order to verify the correct behaviour of the total system, testing each part independently is not sufficient. It is also necessary to verify that the components interact correctly with each other. This is the goal of integration testing.

#### 3.1 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.

Firstly, we consider that the system is component-based [25]. That is, made of separate software units which can only interact with each other via a pre-defined interface. 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, in which case it cannot not contain logic by its own. Components need not be pure: they can have state and interact with the context. They can also be black, or “grey” boxes, i.e. their specification being known but their implementation being unknown.