Data-flow Integration Testing Adapted to Runtime Evolution in Component-Based Systems *

Éric Piel
Software Engineering Research Group
Delft University of Technology
Mekelweg 4, 2628CD Delft, The Netherlands
e.a.b.piel@tudelft.nl

Alberto Gonzalez-Sanchez
Software Engineering Research Group
Delft University of Technology
Mekelweg 4, 2628CD Delft, The Netherlands
a.gonzalezsanchez@tudelft.nl

ABSTRACT

Systems of Systems are large-scale information centric component-based systems. Because they can be more easily expressed as an information flow, they are built following the data-flow paradigm. These systems present high availability requirements that make their runtime evolution necessary. This means that integration and system testing will have to be performed at runtime as well. Already existing techniques for runtime integration and testing are usually focused on component-based systems which follow the client-server paradigm, and are not well suited for data-flow systems. In this paper we present *virtual components*, a way of defining units of data-flow behaviour that greatly simplifies the definition and maintenance of integration tests when the system evolves at runtime. We present and discuss an example of how to use virtual components for this purpose.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: testing and debugging

General Terms

Design

1. INTRODUCTION

Systems of Systems (SoS) are large-scale component-based systems in which the sub-components are elaborate and complex systems in their own right. An example of such systems are Maritime Safety and Security (MSS) SoS, whose primary tasks are sensing issues at sea, analysing these issues forming a situational awareness picture, and initiating appropriate actions, in case of serious issues [8, 14, 18]. Systems of Systems will have to be adapted more often than a normal system, for example, an MSS system must be able to evolve at the same time as the situation at sea. The evolution of such a system must be done at runtime, as downtime is never acceptable for such critical missions.

The organisation and workflow of these systems is too dynamic and too complex to be modelled after traditional monolithic methods. The component paradigm offers more flexibility to modify the system at runtime, thanks to the loose coupling between each of the parts forming the system. Moreover, as the main purpose of a SoS is usually data sharing and processing, they are more naturally expressed by following the data-flow paradigm. Therefore, large parts of these systems are built following this paradigm, where each component reacts to new data inputs by processing them and generating data outputs that the next component in the flow will then process. Of course, systems which can not afford to be stopped and which are organised in a component-based data-flow are not restricted to the MSS world. They all have in common that every instance of these systems will eventually have to be updated, either to correct some detected malfunctions, or to adapt to the new needs of the surrounding world.

The objective of integration testing is to uncover errors in the interaction between components and their environment (other components or the platform). The integration of a system must be assessed on the final platform, before starting the system and every time the system is modified. As described in the next section, various techniques for integration testing of a data-flow system exist, but in this paper we concentrate on testing data-flows as units. Closely related to functional testing, this technique relies on the fact that the system specification defines precise expectations on how the data is processed between two points in the data-flow.

The contributions presented in this paper are the following:

- The formalisation of data-flow testing as unit for integration testing of data-flow systems.
- The adaptation of this technique to Built-In Testing, allowing better maintainability and joint usage with other testing techniques.
- A way to specify test-cases so that a minimal amount of effort is needed to adapt them when the system architecture evolves.
- A method to identify which are the test-cases required for regression testing after the system architecture evolves.

*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.

SINTER'09, August 25, 2009, Amsterdam, The Netherlands.

The paper is structured as follows. In Section 2, a brief description of related work and the methods on which our contributions rely will be presented. Section 3 introduces the notion of *virtual components*, whose purpose is to express the functional expectations of a data-flow in a way to make their management easy and compatible with the rest of the testing framework. The usage of these *virtual components* in the context of runtime evolution and runtime testing is described in Section 4. An example of how to use the presented contributions in practice is given in Section 5. Finally, Section 6 concludes this article and presents future works.

2. BACKGROUND

2.1 Integration Testing

In order to validate complex systems, one primordial step consists in performing unit testing on each "part" of the system. Depending on the paradigm on which the system is built, a "part" of the system can be a *module*, a *class*, a *component*, etc. Even if each part of the system respects ideally their specification and has been successfully tested in isolation, the entire system might still not be free from errors: errors can be introduced when integrating the parts together. The main types of errors have been well described by Abdullah *et al.* [1]:

- Interface errors: when a component uses another one with a wrong usage of the interface.
- Interpretation errors: when the functionality provided by a component B is not the one required by the component A.
- Pass through parameters: when a component is expected to just pass data without modification, but actually processes it.

In addition, even if all the components are correctly assembled with each other, there might be errors due to missing functionality in the system, or because the global behaviour does not fit to the requirements.

In order to ensure the quality of the integration, not only the interactions of the components have 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. First, because it requires both the model of the framework, and the models of the components themselves. Further, the increased complexity of the entire system often results in exponentially increasing complexity of the proof. This is one of the reasons why we are currently focusing only on validation, and more specifically testing. Nevertheless, it is worth mentioning the work by Garlan *et al.* [9] on modelling a generic publish/subscribe architecture for verifying systems based on this type of architecture using LTL properties. Extending this work, Baresi *et al.* [2] have proposed language support for modelling and verifying the system efficiently by reducing the number of states to explore. In this paper, we concentrate on the usage of testing for assessing the integration, and how this can be supported by the component-framework so that this type of testing fits with the rest of the testing process.

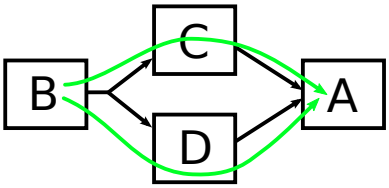

Figure 1: Example of a "data-flow" organisation.

2.2 Data-flow Integration Testing

The systems we are concerned with are mainly targeted for data processing. For this reason, they are often organised following the *data-flow* paradigm. This paradigm, which can also be found referred to as "message-driven architecture" or as "data push technology", is based on the notion of components receiving input data, processing it, and generating output data that typically forms either part of the input data of another component, or the output of the entire system. The series of components through which the data is processed correspond to a "flow". Figure 1 presents such an organisation, with two data-flows. This is to be contrasted with the usual "call-reply" organisation, which the system in Figure 2 follows.

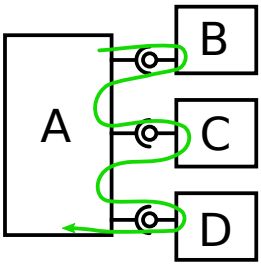

Figure 2: Example of a "call-reply" organisation.

One of the main advantages of this organisation is that the system architecture follows closely the data processing organisation of the system, helping the designer to easily translate the specification into an implementation. The data-flow paradigm also simplifies the concurrent execution of components, as this is managed automatically by the framework. Further, components are loosely coupled: when a component generates data received by another, there is no expectation of behaviour from any of them, they only expect the correct data type. This is very different from the traditional "call-reply" organisation, where caller components always have precise requirements on the response. This loose coupling eases runtime reconfiguration of the system.

Before going further with respect to past work, let us note that the "data-flow" paradigm we use should not be confused with the "definition-use dependencies" sometimes called similarly [13]. White and Leung [20] have presented a method to implement regression integration testing in systems following the "call-reply" organisation. Each module `A` which calls module `B` can have a set of test cases to validate the correct behaviour of module `B` with respect to its needs. When the system is modified, e.g., a component has been changed, only the interactions directly involved with the modified component have to be re-validated. This means it is necessary to run only the test cases which either are used by the changed

component to validate other components, or which are used by other components to validate the changed component.

Unfortunately, the fact that components are loosely coupled also means that integration is harder to test: it is not possible to directly validate a component implementation against the requirements of the components to which it is connected, simply because there are no requirements.

In [4], Bertolino *et al.* present a method to determine in a data-flow based system which sequence of component execution is most worthy to test for assessing the component integration. Their proposal deals mostly with concurrent execution of the components, and finding the smallest execution sequences which are worthy to be validated. They opt for focusing on executions corresponding either to the introduction of *one* given input data, or corresponding to a flow where a given component is executed only once. This can be used to guide what a test case should assess, but not how it could be done.

In [16], Paul describes "end-to-end" testing, which consists in assessing the behaviour of a system with respect to inputs and the corresponding expected outputs. It focuses on functional testing and therefore does not deal with testing sub-parts of the system, only the behaviour noticeable from the user's point of view. Nevertheless, by using the notion of "thin-thread", which is close to the one of a data flow, some test selection is performed. This test selection aims at minimising the amount of test cases needed to be executed after a reconfiguration of the system. Each thin-thread has a *risk* associated, corresponding to the probability an error happens in it and to the damage this error would cause on the functional behaviour of the entire system. Depending on which thin threads are exercised by a test case, it is possible to select the minimal amount of test cases necessary to pass below a given risk level.

Another technique for testing the integration of a system has been proposed by Yuan and Memon [21]. The principal idea is that a large number of random input-data sequences are generated in order to test the various possible combinations of executions. Each of the sequences is a test case. The oracle for the test cases must be adapted to fit any randomly generated sequence, so they use an oracle which detects only generic wrong behaviour (such as non-handled Java exceptions). In their study, the authors only consider testing graphical user interfaces, and therefore the generation of the input data is straightforward (it is only an event). In a more generic case, specific techniques to generate valid input data should be used. This method has the advantage that it can execute a very large number of test cases. However, oracles have to be generic, and therefore this method cannot detect whether the functional behaviour of the integration is incorrect, i.e. whether for each specific input sequence the correct outputs are generated. It should be possible to also test some typical interactions between the components, with a precise expected output. We discuss such a method, dedicated to test some typical interactions between the components, in the next section.

2.3 Built-In Testing

Built-In Testing (BIT) is a useful paradigm in order to test a dynamic component-based system [17, 19]. 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 [12, 11], before or during runtime.It aims at a better maintainability of testing aspects surrounding each component.

BIT has two facets. The first is concerned with the testability of the component. Components can be equipped with special ports in order to facilitate testing. For instance it can be the ability to control its 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 when that testing is happening [17].

The second facet of BIT is concerned with the association of test cases with the component [10]. The goal is to facilitate maintainability and traceability of the testing by keeping the test cases and the test material closely linked together with the component, which, as observed in [3] are very important properties when dealing with complex projects. For instance it permits to keep the unit tests that the developer used for validating the component all along the lifecycle of the component. The associated information can also be used for integration testing of two components together, as described in [15], or in [7] by using contracts. Conserving the test definition with the component all along the lifecycle helps to keep the tests synchronised in case of multiple versions of the components. For instance, when a component is updated to support additional functionality the tests to assess the functionality must be updated simultaneously. Associating tests to 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.

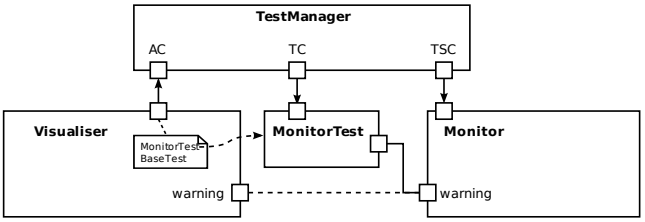

Figure 3: Built-in Testing in Action.

A possibility directly resulting from this second facet of BIT, is that components can carry out all or part of the integration testing by themselves [5]. The requirements of the component on its execution environment (e.g., the platform it is running on, the physical components it is linked with, the components it relies on for providing specific services) can be validated using test cases contained in the component. This distribution of the responsibility in validating the components' 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 to each component. The usage of BIT specifically for performing integration testing between components during runtime evolution has been presented in [10].

Figure 3 presents schematically a validation process when using a BIT infrastructure. The Visualiser component needs to test the Monitor component on which it depends. Via the Acceptance port (AC), one of the BIT facilities, the Visu-

aliser contacts the TestManager component, providing the MonitorTest test. TestManager takes care of the connection between the test and the Monitor component, letting the Monitor component know that a test is taking place through the Test Control (TSC) port (another of the BIT facilities), and finally reporting the result of the test to the Visualiser.

Unfortunately, as we have seen previously, in the case of a data-flow organisation 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. Only some tests might be possible to be done in this way: those directly validating the platform's compatibility and the hardware. In the following section, an integration testing method is described so that the specific behaviour of a flow can be tested, and so that it can be managed using the BIT methodology.

3. VIRTUAL COMPONENTS

In this section we introduce the notion of *virtual component* in order to perform and manage integration testing of a component-based system organised in data-flows. The testing method on which the virtual components are based is fairly straightforward, and although we have not been able to find any academic reference, we have found examples of it use in some companies.

3.1 Specifying a Virtual Component

The basic idea is that every data-flow to be tested corresponds to one component, i.e., a "unit of high cohesion and low coupling with contractually specified ports for external communication". The inputs (resp. outputs) of this component are the inputs (resp. outputs) of the data-flow. Unit testing the virtual component is then equivalent to integration testing the data-flow. Composite components, present in some component models, are not adequate for this purpose, because if one needs to test two overlapping data-flows, this will conflict with the main requirement of non-overlapping composite components. For example in Figure 4, it would not be possible to simultaneously define components for testing the flows B, C, E and B, D. Similarly, it is not possible to use a composite component to define a flow if the inputs and outputs are in different levels of hierarchy. Obviously, it would be counter-productive to have to modify the architecture of the system only to accommodate the validation of a part of it.

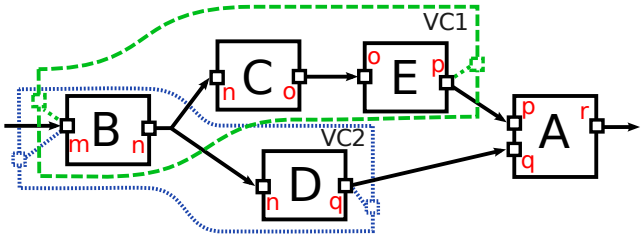

Figure 4: Example of two virtual components.

In order to avoid such limitations, we introduce the notion of *virtual component*. With respect to most of its properties, a virtual component can be seen as a composite component: it has a set of input and output ports, and it delegates its inputs and outputs to interconnected sub-components. However, in contrast to composite components, which are specified by defining which their sub-components are, a virtual component is only defined by its input ports and its output ports. More precisely, the inputs (resp. outputs) of the virtual component are specified by the inputs (resp. outputs) of the sub-component to which it is delegating. Moreover, the sub-components of a virtual component might have input- or output-ports connected with components outside the virtual component without being part of the boundaries. These inputs or outputs will not be used to assess the behaviour of the data-flow. For example in Figure 4, the virtual component $VC1$ is defined to represent the data-flow going from component B to E. It is specified only by the input port m of B, and the output port p of E. The actual set of sub-components is derived from this information. Although the output of B is also used by D, this output is not a port of the virtual component. Another unusual property is that a virtual component can overlap another virtual component without problem: they are completely independent from each other. Thanks to this specification, a virtual component does not interfere with the architecture of the system. The component framework never starts the virtual components of a system for normal execution, they are used solely for the testing phase.

Let us call P_i and P_o the sets of respectively input- and output-ports delimiting the virtual component. In order to determine the components contained in a virtual component, the following steps have to be performed:

1. Find the set C_p of components predecessor to P_o: all the components which could be required to generate the outputs. This set is computed repetitively. For each output port, the component owner of this port is added to the set. For each of the newly added components in the set, the input ports which are not in P_i are followed, and the component generating input for this port is again added to the set C_p. This is repeated until the set has not been extended.

2. Find the set C_s of components successor to the input ports: all the components which could be called when an input is generated. It is computed similarly to C_p, but by following the output ports not contained in P_o.

3. Compute the set C which is the intersection of C_s and C_p: this set corresponds to the components in the data-flow.

A virtual component is considered valid if and only if all the components which own the input- and output-ports are in C. Indeed, if this is not the case, it means there is no continuous flow between some of the inputs and the outputs. This is a sign that either the system is not correctly integrated, or that the specification of the data-flow is wrong.

For example, to compute the set of components in the virtual component of Figure 4, defined with $P_i = \{m_B\}$ (the port m of B) and $P_o = \{p_E\}$ (the port p of E), one starts by computing the flattened architecture. In this case, the architecture stays unchanged, as all the components are already primitive. The set C_s is $\{B, C, E, D, A\}$, and the set C_p is $\{E, C, B\}$. The intersection of these two sets is $\{B, C, E\}$, the components in the data-flow. As both B and E are in this set, the data-flow is correct.

3.1.1 Additional Remarks

In order to keep the definition of virtual components as concise as possible, a number of details have been ignored, that will be discussed in this paragraph.

Firstly, in order to increase the independence of the test definition from the system implementation, a virtual component could be allowed to be specified by ports placed at different levels of composition, or contained in different composite components. In such case, the first step for finding what components belong to a virtual component must be flattening the system architecture by recursively replacing composite components by their contained sub-components. As the connections in composite components are only delegations, it is easy to verify that the flattened architecture is equivalent to the original architecture. From a software engineering point of view, the fact that a data-flow defined in the system specification is not implemented within just one level of hierarchy is often a sign that the implementation should be refactored. However, this is not the task of the test engineers, and this freedom in the test definition prevents them from being limited by implementation decisions.

Secondly, in all systems there exist "boundary" components which, from the point of view of the framework, do not have any input (sources) or any output (sinks). As they lack either input or output ports, they will never form part of any flow. In practice, these components provide only a minimal conversion functionality so they can be excluded from the data-flows and tested separately. Therefore, for the testing of data-flows, we will abstract from this type of components.

3.2 Test Execution and Management

Following our definition of a virtual component, the process of creating and executing tests is relatively simple because, as we will see, it relies on the well known unit testing method. In order to integration test a set of components organised in a data-flow, one should compare the behaviour of this data-flow with respect to the specification. The validation of the data-flow consists in assessing that, for the given inputs, the generated outputs at the end of the data-flow conform to the specification. As the inputs of the virtual components correspond to the inputs of the data-flow and its outputs correspond to the outputs of the data-flow, integration testing the data-flow is equivalent to unit testing the virtual component.

Tests for a data-flow are therefore written by the test engineer as unit tests for the virtual component. This is by itself an advantage because this technique is well known and there exist a large number of tools and frameworks to develop and execute unit tests. Typically, this approach allows defining a handful of complex interactions in the system which must be tested extensively. The test cases can be sequences of events of an arbitrary length, and the oracle can precisely verify whether the output data conform to the specifications. It is complementary to the other testing methods seen in Section 2.

Moreover, as a virtual component is still a *component*, managing the integration tests for the data-flows can be greatly simplified if they are managed by a Built-in Test infrastructure. Test engineers only need to define a virtual component, add it to the system, and associate tests to it via the test management interface provided by BIT. The component framework then automatically decides when tests should be executed, sets up the test environment for them, reports their results and keeps a history of testing along the evolution of the system.

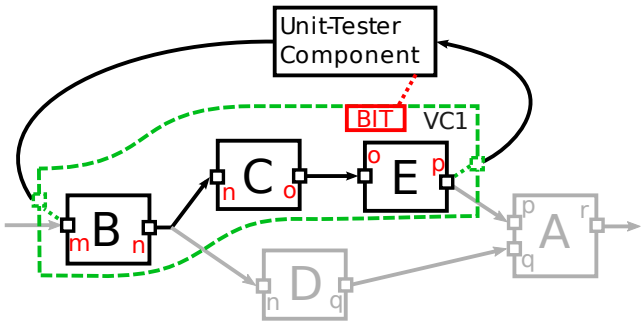

Figure 5: Testing a virtual component using a unit test.

Figure 5 depicts the configuration of the system during the testing phase. The components outside of the virtual component under test are isolated. The unit tester component, provided by the BIT information of the virtual component, is then connected to the inputs and outputs and the tests executed. Once the test cases have been executed, the results are passed to the component framework via the BIT facility and are then used as the results of the integration testing of the components.

4. EVOLUTION OF THE SYSTEM

Systems which must be modified at runtime are often component-based systems, because, as they are loosely coupled, the modifications are easier to apply. When a modification happens, the new configuration of the system must be re-validated with, ideally, the same quality as the original configuration. However, the runtime evolution should be applied relatively quickly, so the re-validation process should take as little time as possible. With respect to the testing method, this need for quick re-validation has two facets. First, the changes to the system must also be forwarded as changes to the tests. The effort to update the tests should be minimized. Second, to reduce the testing time, only the test cases which potentially have their result affected should be re-executed. Therefore, it is important to be able to detect which of the test cases are affected by the modification. In this section, these two points will be addressed.

4.1 Following the Evolution

The specification of virtual components has been especially designed to minimize the updates needed when a modification of the system occurs. The main idea is that data-flows are specified only via the input- and output-ports. Thus, if a modification takes place on one of the inner components of the data-flow, the members of the virtual component are automatically recomputed correctly. Moreover, the inputs or outputs which cross the border of a virtual component but are not part of its boundary are not specified. Therefore, if the type of a port is modified, or if ports are added or removed from a component on the edge of the virtual component, but they are not involved in the behaviour of the data-flow, no update of the virtual component is required.

Let us now review in more detail when a virtual component must be modified to accompany a modification of the

system. First of all, of course, if a data-flow *specification* is modified, i.e. its expected behaviour is changed, the test cases associated to the virtual component have to be updated. In case of modifications in the *implementation*, there can be six different possibilities:

1. A component is modified inside the virtual component, and it does not own one of the flow's ports.
2. A component is modified in the boundary of the flow, i.e., it owns one of the flow's ports.
3. A component is modified outside the virtual component.
4. A connection is removed or added between two components inside the virtual component.
5. A connection is removed or added between a component inside and one outside the virtual component.
6. A connection is removed or added between two components outside the virtual component.

Of these six cases, the two cases affecting only components outside the virtual component (3 and 6) can easily be discarded as not requiring a change of the virtual component definition. If a component is modified within the virtual component (1), or a connection is changed between two components inside the virtual component (4), the virtual component definition does not need to be changed. Indeed, the specification of the input- and output-ports are still valid, and the behaviour of the data-flow must follow the same specification as before. When a connection between a component inside and a component outside of the flow is changed (5), the set of components included in the virtual component might change. Nevertheless, for similar reasons than in previous cases, neither the specification of the ports nor the test cases have to be updated. Finally, if a component on the boundary of the virtual component is modified (2), an update of the virtual component specification is needed only if one of the ports on which the virtual component relies is modified. In such a condition, the data-flow cannot have the same behaviour, as one of the input or output is changed. The test cases will have to be updated as well, to fit the new port.

If a composite component was used to define a data-flow, it would have to be updated under the same conditions of case 2, but would also have to be updated in cases 1, 4 and 5 as its specification relies on all its sub-components and bindings.

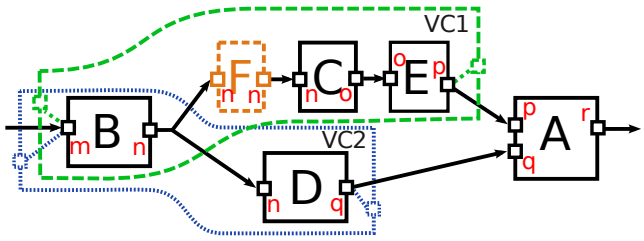

Figure 6: Adjustment of a virtual component after modifications.

Figure 6 illustrates the cases 2 and 5. Component F was inserted in the flow between B and C. The behaviour of the flow should still follow the same specification (typically, F was introduced to fix a bug or to improve the performances). As F has no ports on the boundary of the virtual component $VC1$, $VC1$ is automatically adjusted. Similarly, although the binding between B and C has changed, $VC2$ did not need any update.

To summarise, only when the associated test cases must be updated the specification of a virtual component has to be updated as well, e.g., when the functional specification of the data-flow evolves, or when one of the input- or output-ports is changed. Whenever the test cases can be kept as they are, the virtual component adapts to the new configuration automatically.

4.2 Regression Testing

Testing can be costly, mainly in terms of time but if it is done at runtime, also in terms of resource usage. When a modification of a system is applied, it affects generally just a small part of its complete behaviour. Testing the new configuration of a system when the previous configuration has already been tested is known as *regression testing*. The main goal is to verify that the behaviour of the system has not been altered as a result of the modification, while minimizing the number of test cases that have to be executed again. For this purpose, one needs to determine the test cases which do not need to be re-executed because their result would be for sure the same as during the previous run. This means that only the test cases whose outcome might be affected by the modification will be run to validate the new system configuration. White and Leung [20] have defined a method to determine which test cases must be re-executed for regression integration testing with a "call-reply" organisation. Their test selection method identifies the tests to be re-executed by looking for the ones validating interactions involving one (or two) modified component.

In the context of data-flow, the interactions tested are longer than just a caller and a callee, so it is not possible to re-test only the components directly linked to the modification. The entire data-flow must be re-tested, or, in other words, all the tests associated to the virtual component corresponding to the flow must be re-executed. The number of virtual components affected by a modification depends on the type of modification:

- A component is modified: all the virtual components which contain the modified component. In case of addition or removal of a component, this means all the virtual components which have a different set of components before and after the modification.
- A connection is added or removed: all the virtual components which have a different set of components before and after the modification, and all the virtual components which contain *both* components related to the connection.

In particular, this selection method discards the modifications affecting inputs previous to the data-flow, because they will never have any influence when running the unit tests of the virtual component, and the modifications affecting components using the outputs of the data-flow, because they would not have influence on the result of the unit tests anyway.

5. EXAMPLE

At the moment of writing this paper, the implementation of virtual components on our component framework is not yet completed. Nevertheless, it is possible to give an overview of how the entire process of integration testing data-flows will be done in practice in our environment. We present the usage of the method on a 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.

The architecture of this system is outlined in Figure 7. Components are implemented in Java, with additional information contained in a separate Architectural Description File to describe their interfaces (similar to ports), bindings between components, and tests associated to the components. They are executed in our component framework, which is based on the Fractal [6] framework.

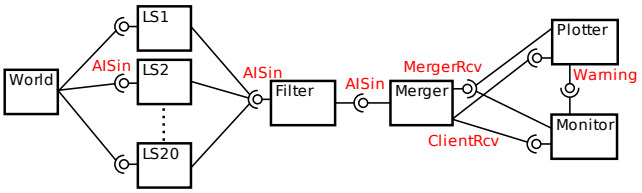

Figure 7: Architecture of the surveillance system used as example.

This system receives positioning information broadcast by ships and processes them in order to form a situational picture of the coastal waters. The *World* component generates the same data as ships in reality. Obviously, it is not be present in a real system, but it acts as a simulator for the ships' transmitters. The *LS* (Local Station) components simulate the individual AIS receivers by transmitting only messages broadcast in a coverage zone. The *Filter* component receives the data from the several *LS* components and removes the duplicate messages (which occur when a boat is in a zone covered by several receivers). The *Merger* component stores the messages into a database containing up to date information about every ship. This information can be accessed via a specific protocol. The *Monitor* component implements this protocol in order to passively observe all the data and detect inconsistencies in the information sent by ships. In case a problem is detected, a `Warning` is generated. The *Plotter* component displays on a map the ships, and the warnings.

Before the system is started, and after every component has passed the unit testing phase, the validation of the component integration takes place. Virtual components are used to test some specific interactions. For instance a virtual component can be set between `AISin` of `Filter` and `Warning` of `Monitor`. The specification is shown in Listing 1. The two `interface` definitions specify the external interfaces of the component, while the two `binding` definitions specify the inputs and outputs of the flow. The `test` definition corresponds to one of the BIT facilities, and associates test cases with the component. The provider `JUnitProviderFlow` indicates a special component used to run this type of test for virtual components. The first test, `TestWarning`, verifies

```
<virtual-composite name="flowCore">
     <interface name="AISin" role="server"
                          signature="AISin"/>
     <interface name="warning" role="client"
                          signature="Warning"/>
     <binding client="this.AISin" server="filter.AISin"/>
     <binding client="mon.warning03" server="this.warning"/>
     <test provider="JUnitProviderFlow" name="Warn"
                        definition="TestWarning"/>
     <test provider="JUnitProviderFlow" name="Dup"
                        definition="TestDuplicate"/>
</virtual-composite>
```

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

```
public class TestDuplicate extends JUnitFlowTest
                           implements Warning {
  AISin l;
  @Before
  public void setUp() {
    l = (AISin) bindings.get("AISin");
  }

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

    for (int i=0; i < 5; i++) {
        l.AISin(in[i]);
    }
    result = waitForMessages(null, 1.0f);
    Assert.assertTrue("Duplicated message not discarded.",
            result.length > 1);
  }
  ...
}
```

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

that warnings are correctly generated in case of inconsistencies in the messages received by the core of the system. A sequence of AIS messages is sent at a high frequency (which is not authorised in the AIS protocol) and the test case verifies that the correct warning is generated. The second test, `TestDuplicate`, verifies that only one warning is generated even if a message containing inconsistencies is received several times (which is often the case as the coverage of many local stations overlap).

A shortened version of this second test is presented in Listing 2. The Java class extends the helper class `JUnitFlowTest` which automatically connects the component to the component under test. The rest of the class is a typical JUnit test class. 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 5 times an identical AIS message, and reads the warning received during one second. In case the flows generates more than one warning, the test fails.

Only once all the tests specified in the system have been successfully passed, the system can be started. Our component framework allows to modify the configuration of the system at runtime. If either the *Filter*, the *Merger*, or the *Monitor* are updated, or a component is inserted in between them, then the test results of the virtual component *flowCore* are automatically invalidated. The data-flow has to

be re-validated with the new version of the components before the new configuration can be applied. The execution of the test cases is done following the normal runtime testing technique of the framework.

6. CONCLUSIONS AND FUTURE WORKS

In this paper, we have introduced the notion of *virtual component*. This notion facilitates the integration testing of component systems organised in a data-flow style. First, the specification of this type of components, which relies solely on defining the input- and output-ports of the data-flow, have been formally defined to determine the components included. As data-flows can be seen as components, it is possible to use the BIT facilities to manage the test cases in large and complex systems, all along their life-time. In addition, we have seen how this method interacts with the evolution of the system. As virtual components are defined only by their boundaries, in most cases they adapt to the new configuration automatically. A test selection method when doing regression testing was formulated so that only test-cases involving a modified data-flow are re-executed. Finally, an example using a small component-based system illustrated how to apply those contributions in practice.

In the short term, the implementation of the virtual component technique will be finalized in our component framework. We would like also to see how the technique can be applied to publish/subscribe frameworks, where the components are not explicit bound to others but only publish or subscribe to a type of data. In the long term, other integration testing methods should be investigated, especially ones that can take into account the non-functional properties of the components.

7. REFERENCES

[1] K. Abdullah, J. Kimble, and L. White. Correcting for unreliable regression integration testing. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 232, Washington, DC, USA, 1995. IEEE Computer Society.

[2] L. Baresi, C. Ghezzi, and L. Mottola. On accurate automatic verification of publish-subscribe architectures. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 199–208, Washington, DC, USA, 2007. IEEE Computer Society.

[3] A. Beer and M. Heindl. Issues in testing dependable event-based systems at a systems integration company. In *ARES '07: Proceedings of the The Second International Conference on Availability, Reliability and Security*, pages 1093–1100, Washington, DC, USA, 2007. IEEE Computer Society.

[4] A. Bertolino, P. Inverardi, H. Muccini, and A. Rosetti. An approach to integration testing based on architectural descriptions. In *ICECCS '97: Proceedings of the Third IEEE International Conference on Engineering of Complex Computer Systems*, page 77, Washington, DC, USA, 1997. IEEE Computer Society.

[5] D. Brenner, C. Atkinson, R. Malaka, M. Merdes, B. Paech, and D. Suliman. Reducing verification effort in component-based software engineering through built-in testing. *Information Systems Frontiers*, 9(2-3):151–162, 2007.

[6] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

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

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

[9] D. Garlan, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. In *International SPIN Workshop on Model Checking of Software*, pages 166–180, Portland, Oregon, US, May 2003.

[10] A. González, É. Piel, and H.-G. Gross. 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)*, pages 41–48, L'Aquila, Italy, Sept. 2008. IEEE Computer Society.

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

[12] 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.

[13] M. J. Harrold and M. L. Soffa. Interprocedual data flow testing. In *TAV3: Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, pages 158–167, New York, NY, USA, 1989. ACM.

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

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

[16] R. Paul. End-to-end integration testing. In *APAQS '01: Proceedings of the Second Asia-Pacific Conference on Quality Software*, page 211, Washington, DC, USA, 2001. IEEE Computer Society.

[17] 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*, pages 171–176, Sept. 2006.

[18] Thales Group. Maritime safety and security. http://shield.thalesgroup.com/offering/port_maritime.php, 2007.

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

[20] L. J. White and H. K. N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Conference Software Maintenance*, pages 262–271, 1992.

[21] X. Yuan and A. M. Memon. Alternating GUI test generation and execution. In *Testing: Academic and Industry Conference - Practice And Research Techniques (TAIC PART'08)*, pages 23–32, Windsor, United Kingdom, Aug. 2008. IEEE Computer Society.