Built-in Data-flow Integration Testing in Large-Scale Component-Based Systems *

Éric Piel, Alberto Gonzalez-Sanchez, and Hans-Gerhard Gross

Software Technology Department, Delft University of Technology, The Netherlands
{e.a.b.piel, a.gonzalezsanchez, h.g.gross}@tudelft.nl

Abstract. Modern large-scale component-based applications and service ecosystems are built following a number of different component models and architectural styles, such as the data-flow architectural style. In this style, each building block receives data from a previous one in the flow and sends output data to other components. This organisation expresses information flows adequately, and also favours decoupling between the components, leading to easier maintenance and quicker evolution of the system. Integration testing is a major means to ensure the quality of large systems. Their size and complexity, together with the fact that they are developed and maintained by several stake holders, make Built-In Testing (BIT) an attractive approach to manage their integration testing. However, so far no technique has been proposed that combines BIT and data-flow integration testing. We have introduced the notion of a virtual component in order to realize such a combination. It permits to define the behaviour of several components assembled to process a flow of data, using BIT. Test-cases are defined in a way that they are simple to write and flexible to adapt. We present two implementations of our proposed virtual component integration testing technique, and we extend our previous proposal to detect and handle errors in the definition by the user. The evaluation of the virtual component testing approach suggests that more issues can be detected in systems with data-flows than through other integration testing approaches.

1 Introduction

The component paradigm and the service paradigm advocate to the rapid construction of large-scale systems-of-systems. Both help facilitate the integration of third-party building blocks through fostering loose coupling, and ameliorating system maintenance to the extent that it can be carried out online. Many large-scale systems-of-systems, such as situational awareness systems, support-systems of all kinds, swarm robotics, and distributed sensor and actuator networks, employ powerful data sharing and event processing techniques and middleware platforms. Such event- and data-driven systems or parts thereof are more naturally

* 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.

expressed through event processing, data-flow processing, or message-driven architectural styles, implemented on top of, or being part of the component and service platforms. The size and complexity of these large systems, together with the fact they are developed and maintained by multiple parties, make the quality assurance activities focus not only on unit testing but also on the validation of the adaptation and integration process [6, 8].

Built-In Testing (BIT) [9, 20, 18] is a powerful method for validating the adaptation and integration of systems-of-systems of such dynamic and hybrid nature. BIT prescribes components to be equipped with the ability to check their execution environment, and the ability of being checked by their execution environment [10], before or during runtime. BIT also aims at a better maintainability of testing aspects surrounding each component. The responsibility in validating the components' environment is distributed and assigned to the components themselves which makes this method viable to assessing the integration and also the evolution of dynamic systems-of-systems.

The objective of integration testing is to uncover errors in the interaction between components or services, and their execution environment, i.e., other components and services, or the underlying middleware platform. The integration of a system must be assessed in its final context, because typically, such systems are extremely difficult to duplicate for testing. The integration must also be re-validated along with every reconfiguration, when services are replaced and reconfigured, or the system's topology is changed in any kind, in order to address evolving requirements.

Various techniques to support integration testing of systems following the event- or data-processing architectural styles exist, but in this paper we concentrate on testing data-flows as units. In earlier work, we have introduced the concept of a *virtual component* [16] that combines integration testing of data-flow-type systems with the advantages that built-in testing offers. In this article, we extend this work with the following contributions:

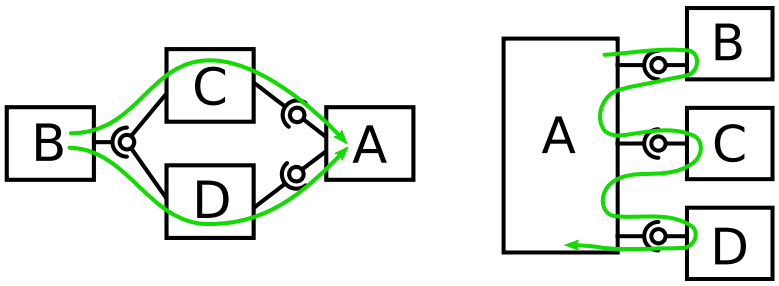
- We extend the component enumeration algorithm with additional functions to detect ill-formed flow definitions, allowing efficient development and maintenance of the tests.
- We present approaches for realizing the virtual component testing technique, and demonstrate how this should be done in two different platform styles, namely, client-server and publish-subscribe styles.
- We evaluate our proposed method using part of a concrete industry-scale surveillance system-of-systems [5, 19].

The paper is structured as follows. In Section 2, some background and related work is presented, including the concept of a virtual component. Section 3 introduces the new algorithms and methods to extend the concept of virtual component. A description of the implementation of the concept for two different component frameworks is outlined in Section 4, and the assessment of the effectiveness of virtual components for integration testing of typical data-flow-based systems is presented in Section 5. Finally, Section 6 concludes this article and presents future work.

2 Background and Related Work

Integration testing. In order to validate complex systems, one primordial step consists in performing unit testing on different levels of granularity of the system, such as *module*-level, *class*-level, *component*-level, etc. Even if every unit respects its own specification and has been successfully unit tested, there is the chance of residual system defects through component coordination issues and adaptation [1]. A common approach to ensure component integration is *integration testing* [6, 8], that validates the interactions between sets of black-box components [22]. In contrast to full system testing, it concentrates on subsets of the system to be assessed in combination, allowing early checking, e.g., before all components are available.

Data-flow and call-reply architectural styles. Architectural styles [17] determine the assembly and "wiring" of components in a system, and have consequences for integration testing. Our systems of interest are aimed at high-volume data processing, and organised following the *data-flow* paradigm, also termed as "message-driven architectures", "data push technology", or "publish-subscribe architectures". They all have in common that components receive input data, process it, and generate output data for other, subsequent components in a so-called "flow". Typically processing is performed asynchronously. Another



(a) Data-flow (b) Call-reply

Fig. 1: Examples of two typical architectural styles.

architectural style is the "call-reply" style typically found in service-oriented architectures. Here, client components are "aware" of their servers. Data is passed to the server, processed, and passed back to the client, mostly in a synchronous fashion. In data-flow styles, components do not have such "mutual awareness," which is significant for testing. Figure 1 illustrates these two styles. Both are suitable for the same application, although the implementation of their components would be different. For instance component A in the call-reply organisation must be "aware" of the component B, of the data it receives, the returned information, and even what to do next with this result.

The main advantage of the data-flow organisation is that the system architecture follows closely the data processing organisation of a physical system.

This helps the designer to translate a data-flow into an implementation. It also facilitates concurrent execution of components without blocking and waiting, and components are loosely coupled, since they have no mutual behavioural expectation (contract), but only a defined data type.

Built-In Testing. Built-In Testing (BIT) is a useful paradigm in order to simplify the testing of dynamic component-based systems [18, 20] and to improve the maintainability of the tests. BIT has two facets. First, components can be equipped with special ports supporting their testability, e.g., in order to control or observe internal state, or for separating testing and nominal operations [18] (test awareness). Second, the direct association of test-cases with the component [7] facilitates maintainability and traceability of test operations by keeping the test-cases and the test material closely linked together with the component [2]. This associates tests with the component throughout its life-cycle and supports re-assessment in various operation contexts. It also permits distribution of the test responsibility to the components themselves, maintaining independence of components and fostering their loose coupling, by decentralising both the definition of the tests and the testing.

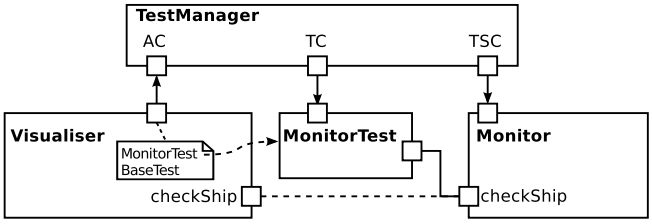


Fig. 2: Built-in Integration Testing – Example.

Several approaches have been proposed to use BIT also for integration testing. In most approaches, each component carries out all or part of its integration testing itself [4, 7]. The component's requirements on its execution environment (implicit execution context), and on other associated components (explicit execution context) can be validated using test-cases contained in the component, or delivered as test component in its own right [9]. We call this pattern *provider integration testing*. A typical example of such integration testing is depicted schematically in Figure 2. The Visualiser component tests its own integration with the Monitor component on which it depends. The Visualiser contacts the TestManager (one of the BIT facilities) through the Acceptance port (AC). Test Manager "knows", through the TC port, where to find the MonitorTest (a test service assessing the Monitor), and it notifies the Monitor that it will be tested, through TSC. Test results are reported back to the Visualiser. Because testing is performed from a component's perspective, it is only possible to validate the underlying platform and the components on which it depends directly. For the call-reply architectural style this kind of testing is sufficient. However, in a data-flow organisation, integration testing would be very limited.

Data-flow integration testing using virtual components. Earlier approaches were proposed for integration testing in data-flow architectural styles. Bertolino *et al.* have presented a method [3] to determine in a data-flow-based system which flows are the most relevant for assessing component integration. Their study deals with the combinations of concurrent executions of components, which is useful to determine a test goal. Unfortunately, they do not elaborate upon how this could be done.

Paul [15] describes "end-to-end" testing, a technique for assessment of system behaviour with respect to inputs and corresponding expected outputs. This is not an integration testing technique, but a system-level testing technique focusing on system use cases which does not address distribution of tests in large systems. Similarly, Jorgensen [13] describes "thread testing" as a series of inputs and expected outputs to validate the functional behaviour at the system level. It only focuses on the theoretical design of the test cases, and does not treat their implementation on any specific platform.

Another technique for integration testing relies on the generation of a large number of random input-data sequences to probe the various possible combinations of executions [21]. Each sequence is considered one test-case. Drawbacks are the sheer number of test-cases necessary to execute this technique, and that the oracle must fit any randomly generated sequence. Therefore the oracle only detects generic fault behaviour such as non-handled Java exceptions. Alternatively, the oracle can be based on a "golden" implementation, but this exists only in mature projects. There is no possibility to explicitly test typical component protocol interactions.

In our earlier work [16] to test data-flow-based component systems and services, we introduced the notion of *virtual component*, which permits the determination of one data-flow (or a part of a flow) through a system, and its representation in terms of a component in its own right, i.e. the virtual component. The data-flow may involve several physical components and represents them and their data exchange as one single "unit of high cohesion and low coupling with contractually specified ports for external communication". Testing this representation is then equivalent to integration testing the data-flow and its components associated. Here, BIT can be used to maintain tests for each flow, by associating them with the corresponding virtual component, in the same way as outlined above, in Figure 2. Consequently, tests can be managed independently in separate parts of the system. The component framework can set up the tests associated with the virtual components, execute the tester components, process and report any issues found, and maintain a history of testing along the evolution of the system.

Compared to a composite component, which is made of sub-components and is present in hierarchical component models [14], a virtual component does not influence the topology of the system. It is only a logical entity used for testing. Each virtual component is independent from any other one, so they can overlap, corresponding to the presence of several data-flows involving the same physical components. For instance, in Figure 3, two virtual components $VC1$ and $VC2$

are defined to represent the two data-flows B, C, E and B, D, whose integration can be tested independently.

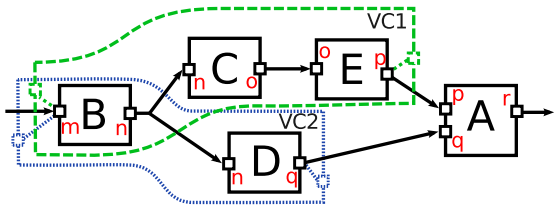


Fig. 3: Example of two virtual components.

Another difference between a virtual component and a composite component is the way it is defined. The latter is defined by the set of enclosed components, and the connections between its interfaces and the interfaces of the sub-components. A virtual component is only defined by the connections between its interfaces and the interfaces of the sub-components (situated on the edges of the flow). The set of components it encloses is computed dynamically according to a specific algorithm. This permits to adapt easily to the evolution of the system: when a component is added or removed, or when a connection is modified, if the modifications are not on the edges of the flow, the virtual component adapts to the new data-flow automatically. This is a strong advantage in the context of large systems where the virtual components are created by the testers while the architecture is modified by the developers. For example in Figure 3, the virtual component $VC1$ is defined to represent the data-flow going from component B to E. It is defined only by the input port m of B, and the output port p of E.

The algorithm to compute the set of components enclosed in a virtual component has been defined [16] so that in the typical cases its result is "intuitive". The set C of components contained in a virtual component specified by its sets of inputs P_i and outputs P_o is computed as follow:

1. The set C_p is computed by iteratively adding all the components predecessor to P_o. For each output port, the component owner of this port is added to the set. For each newly added component, 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. The set C_s is computed similarly by iteratively adding all the components successor to P_i.
3. C is defined as $C_p \cap C_s$.

For example, the set of components in the virtual component $VC1$ of Figure 3, defined with $P_i = \{m_B\}$ (the port m of B) and $P_o = \{p_E\}$ (the port p of E), is computed by finding the sets C_s, which is $\{B, C, E, D, A\}$, and C_p, which is $\{E, C, B\}$. The intersection of these two sets is $\{B, C, E\}$, the components in the data-flow.

As we will see in the next section, in practice, this algorithm is not sufficient to detect and correctly report errors in the definition of a virtual component,

which can be caused either by a mistake during the definition or by a modification of the system architecture.

3 Realizing Virtual Components in Component Models

Additional concepts and techniques are required in order to practically use virtual components in real component models. This section gives an overview of the properties of virtual components, and outlines additional requirements in order to realize the concepts of virtual components in two concrete component execution frameworks.

3.1 Detecting Ill-Formed Virtual Components

Virtual components are defined solely via their inputs and outputs. The algorithm to determine which components are part of a flow, and hence a virtual component, was presented in [16]. In practice, only a limited combination of inputs and outputs will lead to a meaningful data-flow. Incorrect combinations can be due to user errors in flow definition, or due to changes in the system architecture. Such combination can lead to tests validating component interactions not representative of the interactions in the complete system, prevent the test component to connect to the virtual component, or reveal directly that the implementation does not conform to the specification. In order to provide a user-friendly integration testing environment, ill-formed virtual components must be detected and reported with enough information, so that it is easy to correct or accept them. In the following, we discuss algorithms to handle most of the issues in flow definition.

Weak flows. A virtual component should always correspond to a complete set of component interactions, i.e. incorporate all components that contribute to the considered flow. However, some components in a flow might receive inputs from components which have not been defined as being part of that particular flow. We refer to these flows as *weak flows*. For example, the flow in Fig. 4, does not incorporate the input to s_C as part of the virtual component. In real systems, the combined behaviour of the components in the flow will also likely depend on these inputs unidentified in the virtual component. This might depend on the particular context of the running system and cannot be determined from the topology of the system alone. An integration test may fail because of a poorly defined virtual component, and not because of a fault. Inversely, the test could pass while in the actual system, with all the inputs, the implementation behaves wrongly. Generally, weak flows are signs of an oversight from the tester. Weak flows must be detected and indicated, so that the integration tester may determine the full test flow. Let us note that this is different for the symmetrical case with outputs, because not taking into account an output in the test cannot change the behaviour of the components.

The following algorithm can be used to verify the completeness of virtual components. P_i is the set of input ports, P_o is the set of output ports, P_w is empty initially, and C is the set of components in the virtual component:

1. For each input of each component in C, add it to the set P_w.
2. For each output of each component, for each of the input ports to which they are connected, remove the input port from P_w.
3. Remove all inputs of P_i from P_w.

If the P_w is not empty, the flow is weak, and the inputs contained in this set are the ones causing the weak flow.

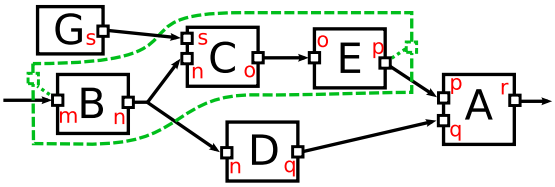


Fig. 4: Example of a *weak flow*.

Empty flows. Another problem of a virtual component is that of an *empty flow*, as illustrated in Fig. 5: there is no flow from input n_D to output p_E. Such ill-formedness appears if an input or an output explicitly part of the virtual component is not used in component interaction. In Fig. 5, the error is either the absence of port t_F in the virtual component (an error in the test definition), or the need for component D to also transmit its output to E (an error in the implementation). Such topologies should not be accepted as virtual component, and should be reported to the user as an error. The following algorithm permits

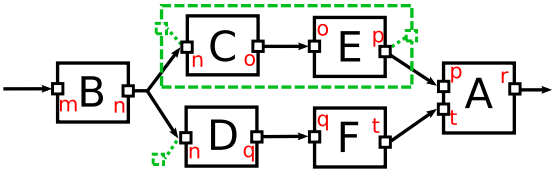


Fig. 5: Example of an *empty flow*.

to verify such condition:

1. For each input in P_i, add the component owning it to C_m.
2. For each output in P_o, add the component owning it to C_m.
3. Remove from C_m all the components in C.

In case the set C_m is not empty, there is one or more empty flow. Each empty flow starts or ends with one of the component in C_m.

Parallel flows. The third peculiar topology of a virtual component is *parallel flows*, illustrated in Fig. 6, in which the virtual component has been defined to correspond to two independent flows C, E and D, F. This is likely a sign that inputs and outputs which were related in the specification are not related to each other in the implementation. In such a case, the implementation is probably incorrect. At least, this ill-formedness is an indicator that one large-scale virtual component could be redefined as several smaller-scale virtual components, which would be easier to test and to maintain. However, this is not necessarily an issue caused by the tester, it could be intentional, for example, to validate the timing between the two flows. This is why only a warning should be displayed in such a case. Detecting parallel flows is equivalent to the connected component problem

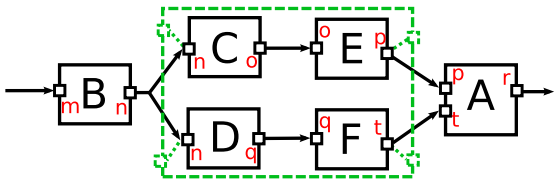


Fig. 6: Example of a *parallel flow*.

in graph theory, which may be addressed through techniques described in [11]:

1. Select i, one of the inputs in P_i, and remove from P_i.
2. Initialize C_c as an empty set.
3. Starting from the component owning i, recursively add all the successors and predecessors to C_c. For each of these components remove all their input ports from P_i.
4. Add C_c to the set of sets SC_c.
5. Repeat until P_i is empty.

If SC_c contains more than one set, then the virtual component contains parallel flows. Each of the flows corresponds to one of the sets in SC_c.

3.2 Extending the Component Model

For a component framework to support virtual components, its API and implementation must be extended. Here, we describe the amendments in general terms, as most component models provide comparable concepts supporting these modifications. Two concrete implementations are described in Section 4.

Typically, an API provides functions to start and stop components, to bind and unbind their interfaces – unless connections are implicit and components are automatically linked when using the same data type – and to add components to and remove components from the framework. The most essential change to be introduced in the component model is the concept of a "virtual" component. This adds to the concepts of composite and primitive types of most component models. In hierarchical frameworks, we propose to associate each virtual component with the composite component that is parent of all components comprising a

data-flow. In other words, each composite component has a set of virtual components to test several interactions between its sub-components. This organisation permits to follow the component and BIT paradigms naturally. That is, components are black boxes, and their development and deployment are separate. This allows for a scalable virtual component approach. Every component will have its own integration testing facility, which is also managed independently. Later in the development and testing process, additional virtual components can be associated with larger composite components to validate the global composites of these building blocks.

A new interface is added to the composite components for adding/removing virtual components. Note that composite components have already an interface to add/remove components but it is better not to use it, in order to avoid mixing testing functionality with nominal functionality. Using a separate interface to manage the virtual components means that only the parts of the framework involved with testing must be updated, and it ensures that they are completely transparent for the normal existing components.

The new component type "virtual" shares many of the typical component interfaces, but also has its own characteristics. The BIT interface used to associate and run test-cases can be realized exactly like in the other component types. The BIT interface used to notify a component of the fact that it is tested is also identical to the normal interface. It notifies all components contained in the flow. Similarly, the interface to request the start and stop of a component passes the requests to the components contained in the flow. This is used to initialise and end the components during a test. The interface found in composite components used to add and remove sub-components is not necessary for the virtual components, as components are automatically enclosed. Nevertheless, the functionality to list the sub-components inside a component is replicated in order to retrieve the information about which components are contained in a flow.

In composite components, the bind and unbind interface allow to associate the external interfaces with the sub-component interfaces. This API can be exploited to specify the inputs and outputs of the virtual component, with the idiosyncrasy that this specifies the actual shape of the virtual component. It should be noted that if multiple modifications to the connections are required and successively applied, the topology of the virtual component might be temporary incorrect (as specified in Subsection 3.1). Therefore, unless the framework supports to group modifications in an atomic way, the construct verification cannot be done directly after a change. The verification should be done either whenever the set of contained components is queried, or just before the test-cases associated with the flow are executed.

4 Implementation

In the context of this research, the concepts of virtual components have been implemented in two different component models. One adaptation was performed

```
<virtual-composite name="flowRawData2FilteredData">
  <interface name="in" role="server" signature="AISin"/>
  <interface name="out" role="client" signature="AISin"/>
  <binding client="this.in" server="ais-listener.ais-in"/>
  <binding client="filter.ais-out0" server="this.out"/>
  <test provider="JUnitProviderFlow" name="RawProcess" definition="RawProcess"/>
</virtual-composite>
```

Listing 1.1: Definition of a virtual component with a test-case in an ADL file.

for the OpenSplice[1] framework, which is a non-hierarchical publish-subscribe platform in which components are not explicitly connected, but are automatically assembled when they share common "topics". This adaptation is not freely available due to confidentiality restrictions. The other adaptation was performed for our Atlas component framework, which is based on the Fractal component model[2]. This framework supports hierarchical structure, has explicit connections, and permits reflective view on the components. It is freely available from our website[3], including the virtual component extension.

The Atlas extension introduces a new component type to represent virtual components, as well as the interfaces discussed above. Because the framework is fully aware of the virtual components, tests can be executed automatically during initialisation of system. Moreover, if the system is modified, the notification of changes are passed to the virtual component infrastructure, which will automatically reset the test status for the data-flows affected. The instantiation of the test component and its binding and unbinding are also handled automatically by the framework. In case a test-case fails, the failure is displayed on the framework console, and the system cannot be started until this is fixed.

The Architecture Description Language (ADL) used by Atlas for describing the system has also been extended to support these new concepts. Listing 1.1 presents an example of a virtual component expressed in this ADL. This flow is taken from the example system described in Section 5. The `interface` tags define the interface of the component. The `binding` tags define the beginnings and ends of the flow, by identifying the specific components at its edges. Finally, the `test` tag denotes the test component containing the test-cases for this flow. The `JUnitProviderFlow` is a component belonging to the BIT infrastructure of Atlas, which is in charge of handling the execution of test components.

Test-cases for data-flow integration testing are defined in terms of a JUnit class, provided that this class also implements the complementary interfaces of the flow. An excerpt of such a class is displayed in Listing 1.2. This corresponds to the flow mentioned in the previous listing, with one test-case `oneMessage` which validates the correct transmission of a message through the flow. The method `init()` is executed just before the test-cases are executed (and after the component has been created and bound to the flow). The method `AISin()` corresponds to the input interface of the testing component.

[1] `http://www.opensplice.org`
[2] `http://fractal.ow2.org`
[3] `http://swerl.tudelft.nl/bin/view/Main/Atlas`

```
public class RawProcess  implements AISin, BindingController {
  private AISin myServer;
  private List<AISMessage> AISrcv;

  @Before
  public void init() {
    AISrcv = new ArrayList<AISMessage>();
  }

  @Test
  public void oneMessage() throws NoSuchInterfaceException {
    AISMessage mes = new DynamicAISMessage("32w@HUP038O@O`s@1T1PO6", 5925L);
    myServer.AISin(mes);
    assertEquals("Message not received.", AISrcv.get(0), mes);
  }

  public void AISin(AISMessage m) {
    AISrcv.add(m);
  }
  ...
}
```

Listing 1.2: Definition of a test-case for validating the flow.

The OpenSplice implementation follows a different approach, mainly due to the lack of component hierarchy, and of centralised management functions (components can start and stop completely independently from the rest of the system). A special program was created to handle all the virtual components of a given system. Following the definition of the flows given in separate files, and the information about which components have been modified (to be provided by the user, because automatic notification of changes is not available), the program establishes the flows to be tested. Each flow is associated with an OpenSplice component written as a JUnit class. This contains all the test-cases for testing the integration of all components associated with the flow. The framework automatically connects components with compatible topics, so the test-cases, integrated as yet another component, are directly attached with the right flow. It is important to note that we use existing operations of the component model in order to implement these mechanisms.

In the next section, we use the Atlas implementation to evaluate with a real system to which extent the advantages of the virtual component approach pay off during integration testing.

5 Evaluation

In addition to performing a feasibility study, the goal of this evaluation is the assessment of whether the virtual component approach improves the failure detection rate during integration testing compared to the provider testing approach presented in Section 2. In order to being able to compare the performance of the two approaches in terms of failure detection, mutations are introduced in the system and for each of them, integration tests are executed, according to the principles of the two approaches. The evaluation is performed on a part of a maritime surveillance system used as case study.

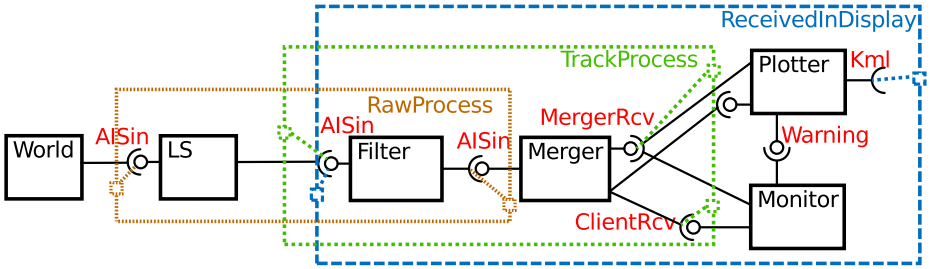


Fig. 7: Architecture of the surveillance system used, with 3 virtual components.

Study Subject. Before going into more details on the evaluation, we briefly present the case study system. The surveillance system receives information broadcasts from ships, called *AIS messages* [12], and it processes them in order to form a situational picture of the coastal waters. The (simplified) architecture of this system is displayed in Fig. 7. The `World` component simulates the ships transmitting data, by replaying AIS messages recorded from reality. The `LS` component receives all AIS data from the antennas physically spread along the coast. The `Filter` component suppresses duplicate messages, because some receivers cover overlapping areas. The `Merger` acts as a temporary database of AIS messages, and client components can consult it to receive tracking information of a ship. The clients must send typical database query requests for retrieving the ship tracks. They are connected following a call-reply architectural style. However, at a high level of abstraction, they are organised according to a data-flow architectural style. The `Monitor` and the `Plotter` are both clients of the `Merger`. The former detects discrepancies in the data, while the latter displays the ship tracks on the screen of the command and control centre (by sending vector drawings to the actual display system). The components are implemented as Atlas components in Java.

Test-cases. Three virtual components were defined, i.e., `RawProcess`, `TrackProcess`, and `ReceivedInDisplay` (Fig. 7). These correspond to three data-flows each of which has a defined expected behaviour. Every flow (i.e., each virtual component) is associated with several test-cases used to validate the defined behaviour. For example, one of the test-cases of `ReceivedInDisplay` sends AIS messages from two ships and verifies that instructions to display both ships are sent. For the provider integration testing, the components are also equipped with test-cases for assessing the correct responses of the components on which they depend. As an example, the LS component has a test-case which transmits some AIS messages and validates that no exceptions happened. The Plotter component comes with test-cases validating the interpretation of the database protocol by the Merger component.

Component Mutation Testing. Mutation testing is a technique in which faulty programs, i.e., the mutants, are generated in order to check the efficiency

of a test method to uncover failures. A mutant is a semantic modification in the implementation of a component introducing a fault.

We had these mutants generated through the μJava[4] tool for the Merger and Filter component. Each of the mutations was applied separately, providing a different version of the system, for which both integration testing methods were executed. "Equivalent" mutants, i.e., modifications that cannot lead to a fault because the system performs nominally as if it was the original version, were sorted out manually. Out of the 181 generated mutants, 94 were deemed as non-equivalent and included in the study. When a test does not find any errors, i.e., the mutated system is considered to operate fine, the result is termed "positive". When an error is reported, the result is termed "negative". "False positives" are the mutants which are said to be working fine, although it was manually verified that they behave outside of the specification. "False negative" represent cases for which a correct system is classified as having an error.

All tests pass when applied to the original (non-mutated) system. Table 1 summarizes the integration testing results obtained when using the provider and virtual component testing approaches. None of the tests applied has produced false negatives. This had been expected because all the tests passed on the original system. The provider integration testing approach is only able to trigger a few failures, i.e., 6% of the faulty mutants detected. In contrast, the virtual component integration testing approach is able to detect a much larger population of the faulty mutants, i.e., 49%. All the failures triggered by the provider testing approach are also identified by the virtual component testing approach.

	Source	True positive	False positive	True negative	False negative	Total
Provider testing	Filter	36	26	2	0	64
	Merger	51	62	4	0	117
Virtual component	Filter	36	7	21	0	64
	Merger	51	41	25	0	117

Table 1: Mutation test results.

Architecture mutation testing. To evaluate the detection of faults in the architecture, we seeded faults in the case study system by changing or removing connections between components. All the 5 mutated configurations had significant incorrect behaviour. The provider testing method detected 2 of the 5 faults, while the virtual component method detected all the 5 faults. More precisely, 3 of the faults were directly detected by the well-formedness checks, therefore not even requiring the execution of the test-cases.

Discussion. First, these results confirm our initial supposition that the virtual component integration testing approach is successful in detecting failures in the specific context of a data-flow architectural style. Second, the results suggests

[4] `http://cs.gmu.edu/~offutt/mujava/`

a much better capacity of detecting problems compared with a more "traditional" or "typical" integration testing approach. It should be noted that the tests were written without knowledge of the mutants. The creation of additional tests specifically crafted to detect the mutants, would have probably increased the true negatives.

Unit tests for the two mutated components were able to detect 75% of the bugs introduced, and 100% would probably be achievable with more elaborate test suites. However this would not be a fair comparison. Integration testing can only detect bugs in program sections which are executed, and therefore cannot detect all the mutations. Moreover, the integration testing also targets faults that are due to different interpretations of a same specification, or due to mistakes in the architecture of the system. This cannot be simulated by mutation testing alone, and unit testing cannot detect such issues, as highlighted by the architecture mutations, which none of the unit tests would have revealed.

6 Conclusions and Future Work

We have presented the implementation and usage of *virtual components* to facilitate the integration testing of component systems organised following a data-flow architectural style. First, three algorithms have been introduced to enforce well-formedness of the virtual components. They are key to a user-friendly realization of this new concept in a component middleware platform. A guideline to extend the typical component interfaces for the manipulation of virtual components was presented. Second, we introduced two implementations of virtual component testing for two types of component middleware platforms, demonstrating the applicability of the approach in practice. Finally, the evaluation of this integration testing approach using mutation testing on a system from our industrial partner showed the effectiveness in detecting errors in systems organised following a data-flow schema. We could show that on this system half of the component mutants were detected by this approach, in contrast to 6% detected by the traditional provider integration testing approach. All 5 architecture mutants were also detected instead of the 2 detected using provider testing.

In future work, we will study ways to minimize the number of test-cases executed during regression integration testing. For example, a test might be repeated only if it assesses non-functional properties, or it is repeated depending on a modification performed.

References

1. Abdullah, K., Kimble, J., White, L.: Correcting for unreliable regression integration testing. In: ICSM'95: Proceedings of the International Conference on Software Maintenance. p. 232. IEEE Computer Society, Washington, DC, USA (1995)
2. Beer, A., Heindl, M.: 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. pp. 1093–1100. IEEE Computer Society, Washington, DC, USA (2007)

3. Bertolino, A., Inverardi, P., Muccini, H., Rosetti, A.: 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. p. 77. IEEE Computer Society, Washington, DC, USA (1997)
4. Brenner, D., Atkinson, C., Malaka, R., Merdes, M., Paech, B., Suliman, D.: Reducing verification effort in component-based software engineering through built-in testing. Information Systems Frontiers 9(2-3), 151–162 (2007)
5. E. U. Commission, Maritime Affairs: An integrated maritime policy for the european union (Oct 2007)
6. Gao, J.Z., Tsao, H.S.J., Wu, Y.: Testing and Quality Assurance for Component-Based Software. Artech House (2003)
7. González, A., Piel, É., Gross, H.G.: 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). pp. 41–48. IEEE Computer Society, L'Aquila, Italy (Sep 2008)
8. Green Hat Software: Lessons from testing service oriented architectures white paper (2008)
9. Gross, H.G.: Component-Based Software Testing with UML. Springer, Heidelberg (2005)
10. Gross, H.G., Mayer, N.: Built-in contract testing in component integration testing. Electronic Notes in Theoretical Computer Science 82(6), 22–32 (2004)
11. Hopcroft, J., Tarjan, R.: Algorithm 447: efficient algorithms for graph manipulation. Commun. ACM 16(6), 372–378 (1973)
12. International Telecommunication Union: Recommendation ITU-R M.1371-1 (2001)
13. Jorgensen, P.: Software Testing: A Craftman's Approach. CRC Press, Inc., Boca Raton, FL, USA (2001)
14. Object Management Group: UML 2 Infrastructure (Final Adopted Specification). `http://www.omg.org/cgi-bin/doc?ptc/2003-09-15` (Sep 2003)
15. Paul, R.: End-to-end integration testing. In: APAQS'01: Proceedings of the Second Asia-Pacific Conference on Quality Software. p. 211. IEEE Computer Society, Washington, DC, USA (2001)
16. Piel, É., Gonzalez-Sanchez, A.: Data-flow integration testing adapted to runtime evolution in component-based systems. In: Workshop Software Integration and Evolution @ Runtime. ACM, Amsterdam, The Netherlands (Aug 2009)
17. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an emerging discipline. Prentice Hall (1996)
18. Suliman, D., Paech, B., Borner, L., Atkinson, C., Brenner, D., Merdes, M., Malaka, R.: The MORABIT approach to runtime component testing. In: 30th Annual International Computer Software and Applications Conference. pp. 171–176 (Sep 2006)
19. Thales Group: Maritime safety and security (2007), `http://www.thalesgroup.com/Portfolio/Security/D3S_Maritime_Safety_and_security/`
20. Vincent, J., King, G., Lay, P., Kinghorn, J.: Principles of built-in-test for run-time-testability in component-based software systems. Software Quality Journal 10(2), 115–133 (2002)
21. Yuan, X., Memon, A.M.: Generating event sequence-based test cases using GUI run-time state feedback. IEEE Transactions on Software Engineering 36(1) (2010)
22. Zhu, H., He, X.: Testing Commercial-off-the-Shelf Components and Systems, chap. A Methodology of Component Integration Testing, pp. 239–269. IEEE Computer Society (2005)