This preprint is a preliminary version. The final version will be published at APSEC 2023.

Unit Test Based Component Integration Testing

Nils Wild
Research Group Software Construction
RWTH Aachen University
Aachen, Germany
wild@swc.rwth-aachen.de

Horst Lichter
Research Group Software Construction
RWTH Aachen University
Aachen, Germany
lichter@swc.rwth-aachen.de

***Abstract*—Effective and efficient testing of complex component based software systems is hard. Isolated test cases that test isolated components are focused and efficient but are not effective in detecting integration faults. Integration test cases on the other hand are hard to develop and maintain. In this paper we present an unit test based integration meta-model and testing approach, to test the integrateability of component based systems based on structural and behavioral information derived from test executions of the respective components test cases. The meta-model is formalized using the property graph model and integration criteria are presented to detect certain types of integration faults early on. Last but not least we propose an approach to derive integration tests from the data contained in the model.**

***Index Terms*—software testing, test automation, software quality assurance, integration testing**

I. Introduction

Component-based software architectures emphasize separation of concerns with respect to the wide-ranging functionality available throughout a given software system. Such architectures have proven to be beneficial to cope with team organization and rapidly changing requirements. They also allow for the composition of components to create tailored systems for the needs of individual customers. Customer requirements are usually fulfilled by multiple services that interact with each other through a well-defined API [1].

Testing aims to assess that such customer requirements are fulfilled. Many approaches have been developed to test different quality attributes of software systems and achieve a quality assessment effectively and efficiently. Testing isolated components of a software system - referred to as unit testing - is relatively cheap and fast. However, it's impossible to detect certain types of faults on the unit level. Thus tests on the integration level are needed, that test the interaction of a component with other components - its environment.

To mitigate integration faults, API definitions have to be consistent across consumers and producers of services. API changes need to be integrated into all affected components [2]. Technical and organizational measures must be taken to communicate such changes to all integrating parties [3]. Furthermore, the documentation has to be kept up-to-date [4].

Developers can start to integrate long before the components or even functions of a component are completely developed. In this paper, we will present a new approach to integration testing, which is economic and can be applied at an early stage.

Before presenting this approach to integration testing, we briefly describe the challenges of early integration and provide an overview of fault types that can be detected on the integration level.

II. Challenges of early integration

One of the core principles of agile approaches and DevOps is to shift left [5]. This means shifting the identification of problems to the left of the development life cycle, to increase the quality and decrease the costs of error correction in later phases. There are different approaches to shift-left. One is to test sub-artifacts right after they are produced. This can be a design model, a code unit, or any other artifact that is produced as a result of some activity. Another one is to develop incrementally. Instead of developing and testing a whole feature at once, intermediate results are tested. Assessing the quality has to be done continuously as artifacts evolve. To this end, economic automation is required. This is why DevOps includes test automation as well as continuous testing. From an economic perspective, the latter requires the former, but the former does not automatically imply the latter [6], [7].

Whenever a system is changed, tests need to be re-executed (regression testing) and new tests need to be developed for new and changed features. It is crucial to know when a component is ready to be integrated and how this component is expected to work in collaboration with others. Dedicated integration tests, models, or any other form of documentation specify that behavior. However, creating integration tests is difficult and studies show that available models and documentations start to diverge from the actual specification - the implemented system - over time [4], [8]. This often makes them unusable as a basis for integration testing.

An approach to keep the test specification up-to-date with the actual specification of the system is needed to make sure that each component is integrated with all components in the way they are expected to interact. Given an arbitrary amount of components that need to interact, there is a huge number of possible integration tests. Each interaction between two components can be tested separately - we will call these tests *interaction tests* - as well as each possible sub-path of interactions contained in the interaction path of all components that realize the customer feature.

Creating and maintaining all these test cases is not feasible. Because of that only a small portion of the total set of integration tests is actually developed and automated [9]–[11]. Often, only the most important tests are automated leading to an hourglass or ice cream cone shape of the testing pyramid [12]. This contradicts the principle of testing as early as possible since these important tests require all components to be ready to integrate.

III. Integration faults

Every integration test aims to detect faults, that could not be detected on the unit level of the individual components. Leung and White [13] presented a taxonomy to categorize faults. They distinguish three specifications for a function:

- The *documented specification* is created before the component is implemented.
- The *actual specification* is given by the implementation of a component based on some interpretation of the documented specification.
- The *expected specification* is based on an interpretation of the documented specification by those components that integrate with a component providing the function.

However, a documented specification does not always exist in a form that the components can be tested against or is incomplete. In that case, a way to test the compliance of the actual and expected specifications of the integrating components is needed, e.g. an integration environment. Based on their taxonomy, we consider the following types of integration faults.

A. Interpretation Faults

Interpretation faults are about a misunderstanding between the provider and consumer of some functionality.

- *Wrong function fault:* The provided function does not comply with the documented specification.
- *Missing function fault:* Some function that is required by the consumer component is not provided. This type of fault is similar to the wrong function type but in this case, the provider component does not offer the functionality at all. Furthermore, a missing function might not be part of the documented specification but just part of the expected specification.
- *Extra function fault:* This is the inverse type of the missing function fault type. Here a function is provided that is never used.

B. Miscoded call faults

Miscoded call faults are caused by calling some function when it should not be called, or not calling the function when it should be called. This fault type can be further distinguished:

- *Extra call fault:* A function is called on a path where it should not be called at all.
- *Wrong placed call fault:* A function is called at the wrong place on the path.
- *Missing call fault:* A function is not called on a path where it should be called.

C. Interface faults

An interface fault occurs whenever the interface between two collaborating components is violated, e.g. through a data type mismatch or a violated parameter rule.

IV. Problem statement and Goals

To overcome some challenges of integration testing, in this paper, we will propose and present a new approach to determine *interaction paths* of any length between components that are ready for integration testing. These interaction paths are to be used to derive integration tests in order to enable early integration testing. In addition, the approach should require little additional effort, i.e., it must be economical, and it should be automatable. Both goals are a prerequisite for application in the industry.

To achieve the economical goal, already existing knowledge regarding the interaction of components must be reused. As in every project unit test cases for all components are created and maintained, the knowledge encoded therein should be reused to determine how components expect to interact with their environment.

The proposed new approach to integration testing was developed with the following questions in mind:

RQ1 Can indicators of integration failure detection be determined from information extracted from unit test cases?

RQ2 How can interaction tests, checking the interaction between two components, be derived from the unit test cases of both components?

RQ3 How can integration tests be derived from those interaction tests to continuously check the integration of a system as it evolves?

The remainder of this paper is structured as follows: Section V introduces the unit test based integration meta-model which is the conceptual core of our approach. In Section VI we describe how instances of this meta-model are used to derive indicators for the integrability of component-based systems. Section VII presents the iterative integration testing process and its activities. In Section VIII we discuss how interaction test results can be interpreted to locate faults and discuss the advantages and limitations of the presented approach. We discuss differences from other approaches that ensure the integration of component-based systems in Section IX. The planned next steps and future work conclude this paper in X.

V. The unit test based integration meta model

In the following, we present the Unit Test Based Integration (UTBI) meta-model. It defines elements and relationships to model structural as well as behavioral information needed to test the integration of components with their environment based on information gathered by existing unit test suites.

The meta-model is depicted in Figure 1. *Components* (Comp) are core elements of the model. To abstract from various types of communication mechanisms and protocols any interaction between components is treated as an activation of a component by a *Message* (Msg) through an *Interface* that is *provided by* (provBy) the component, similar to the

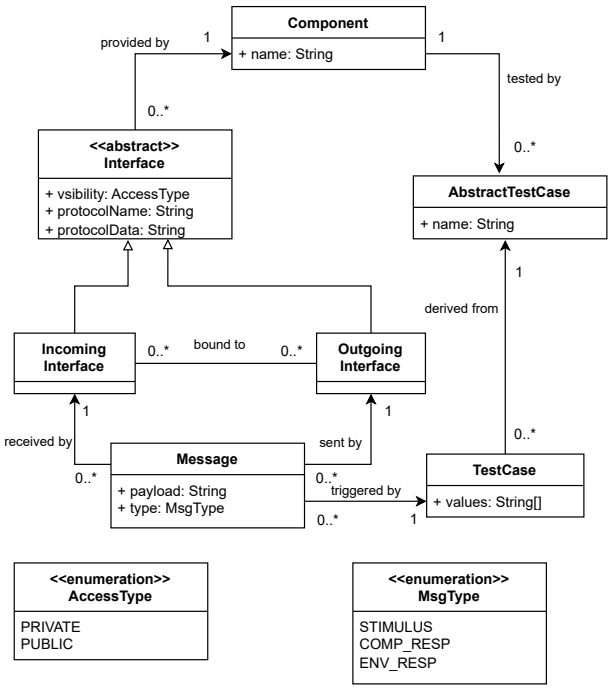

Fig. 1. Unit Test Based Integration Meta-Model

object activation concept used in Smalltalk [14]. Access to an interface can either be *public* - used to integrate with third-party components - or *private*. Furthermore, a distinction is made between an *IncomingInterface* (InIF) and an *OutgoingInterface* (OutIF). By means of an incoming interface, a message is *received by* (recBy) a component while messages are *sent by* (sentBy) an outgoing interface. An incoming interface is *bound to* an arbitrary number of outgoing interfaces and vice versa. Which interfaces are bound to each other depends on the concrete protocol that is used for communication. The protocol data is an attribute of the interface, e.g. in the case of AMQP the respective bindings are defined depending on the exchange type and queue bindings whilst URLs are used for REST.

For each component, the respective unit test cases are modeled as well. Following the ISTQB terminology [15], a component under test (CUT) is *tested by* an *AbstractTestCase* (ATC) which is a test case template without concrete values for input data and expected results. A *TestCase* (TC, also called concrete test case) is *derived from* (derFrom) an ATC by providing concrete *parameter values*.

Once a test case gets executed a sequence of messages is *triggered by* the test case. We distinguish three types of messages:

- A *stimulus message* (STIMULUS) is a message received by the CUT from the test case.
- A *component response message* (COMP_RESP) is a message sent by the CUT back to the test case or to other components (those components are called the CUTs *environment*).
- A *environment response message* (ENV_RESP) is a message sent by a component of the CUTs environment back to the CUT as a reaction to a received component response message.

A. A formal representation of the UTBI meta-model

To analyze UTBI models, i.e. models containing components, their test cases, and all messages triggered by the executions of these test cases, an appropriate formal representation is needed. Since the models are basically graphs the *property graph model* is used [16]. A property graph is a directed labeled graph where each node or edge could maintain a set of property-value pairs. Given finite sets of labels $\mathcal{L}$ and property keys $\mathcal{K}$ and an infinite set of property values $\mathcal{V}$, a property graph $\mathcal{G}$ over $(\mathcal{L}, \mathcal{K}, \mathcal{V})$ is a structure $(\mathcal{N}, E, \rho, \lambda, \nu)$, such that

- $\mathcal{N}$ and E are finite sets of node and edge identifiers,
- $\rho : E \to \mathcal{N} \times \mathcal{N}$ is a total function that associates a pair of node identifiers to each edge identifier,
- $\lambda : \mathcal{N} \cup E \to \mathcal{L}$ is a total function that associates each node and edge with a label from $\mathcal{L}$ and
- $\nu : (\mathcal{N} \cup E) \times \mathcal{K} \to \mathcal{V}$ is a partial function that associates nodes and edges a value for each property key.

To represent UTBI models the label set $\mathcal{L}$ is the union of the node labels $\mathcal{L}_\mathcal{N} = \{Comp, InIF, OutIF, ATC, TC, Msg\}$ and the edge labels $\mathcal{L}_\mathcal{E} = \{l_{testedBy}, l_{derFrom}, l_{trigBy}, l_{next}, l_{sentBy}, l_{recBy}, l_{provBy}, l_{boundTo}\}$. The properties defined for the UTBI meta-model elements are mapped by ν accordingly.

B. Graph Queries

Given the property graph representation of UTBI models, a means to query those models is needed. One way to express queries among a graph are Regular-Path Queries (RPQ) [17]. RPQs allow to retrieve pairs of nodes that are connected by a path conforming to a given regular expression. RPQs are basic building blocks of query languages (e.g. Cypher) for graph databases. An RPQ is an expression of the form $e(t_1, t_2)$, where e is a regular expression over the vocabulary of edge labels and t_1 and t_2 are terms (nodes or variables). Regular expressions over a set of edge labels $\mathcal{L}_\mathcal{E}$ are defined as $e ::= \epsilon \mid l$, with $l \in \mathcal{L}_\mathcal{E} \mid e + e \mid e \circ e \mid e^*$.

A Two-way RPQ (2RPQ) extends the vocabulary of a given RPQ by the inverse label (denoted as l^-) for each edge label in order to inversely traverse edges. 2RPQs are defined as follows [18]: $2RPQ = \{e(t1, t2) | e \in L(\mathcal{L}_\mathcal{E} \bigcup \{l^- | l \in \mathcal{L}_\mathcal{E}\})\}$.

Let $TRPQ_L$ be the set of all 2RPQs over the edge label set L. The function $match : TRPQ_L \to Boolean$ with $match(trpq) = (trpq = \emptyset)$ checks whether a 2RPQ evaluates to a path in the graph.

C. An example system

The application of the UTBI meta-model and its property graph representation is best illustrated with an example. Figure 2 shows the property graph of a system consisting of the two components *OrderService* (OS) and *WarehouseService* (WS) and their incoming and outgoing interfaces. For each component, only one unit test case is included. The execution of

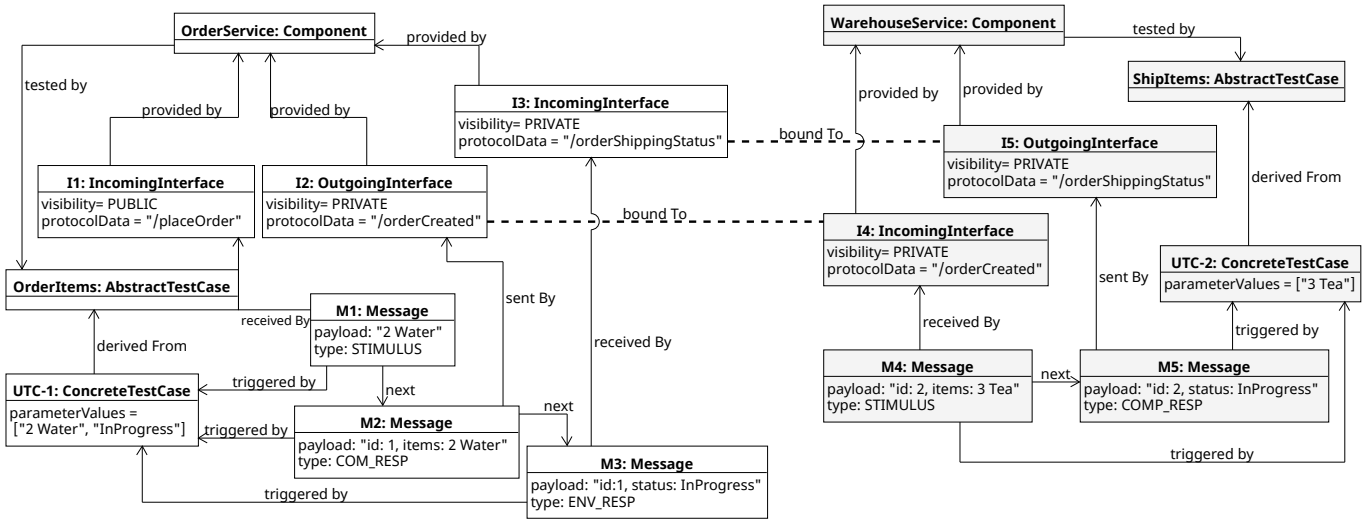

Fig. 2. A UTBI example model showing two interacting components with their unit test cases

these test cases triggers sequences of messages, sent between the test case, the CUT, and the CUT's environment.

Now RPQs and 2RPQs can be defined to meet information needs with respect to the example system. These are examples of conceivable information needs (x denotes a variable).

- Which interfaces does component OS provide?
  RPQ: $l_{provBy}(x, OS)$ or 2RPQ: $l^-_{provBy}(OS, x)$
- Which test cases exist for testing component OS?
  2RPQ: $l_{testedBy} \circ l^-_{derFrom}(OS, x)$
- Which components interact with component OS directly?
  2RPQ: $l_{provBy} \circ l_{boundTo} \circ l^-_{provBy}(OS, x)$
- Does component OS interact with component WS directly?
  $match(l_{provBy} \circ l_{boundTo} \circ l^-_{provBy}(OS, WS)) = true$

VI. Examining the model for integration faults

Given a property graph of a component-based system - like the one depicted in Figure 2 - the model can be analyzed for integration faults.

In order to do this, we first need to define some sets and predicates based on the property UTBI graph model presented in the previous section. Let

- CBS: set of all component-based systems and $s \in CBS$
- $C_s \subseteq \{c \in \mathcal{N} | \lambda(c) = Comp\}$: all components of s
- $M = \{m \in \mathcal{N} | \lambda(m) = Msg\}$: all messages
- $M_c = \{m \in M | match((l_{sentBy} + l_{recBy}) \circ l_{provBy}(m, c))\}$: all messages of $c \in C_s$
- $M_s = \bigcup_{c \in C_s} M_c$: all messages of s
- $IIF_c = \{i \in \mathcal{N} | \lambda(i) = InIF \wedge match(l_{provBy}(i, c))\}$: all incoming interfaces of $c \in C_s$
- $IIF_s = \bigcup_{c \in C_s} IIF_c$: all incoming interfaces of s
- $PrIIF_c = \{i \in IIF_c | \nu(i, type) = PRIVATE\}$: all private incoming interfaces of $c \in C_s$
- $OIF_c = \{i \in \mathcal{N} | \lambda(i) = OutIF \wedge match(l_{provBy}(i, c))\}$: all outgoing interfaces of $c \in C_s$
- $OIF_s = \bigcup_{c \in C_s} OIF_c$: all outgoing interfaces of s
- $PrOIF_c = \{i \in OIF_c | \nu(i, type) = PRIVATE\}$: all private outgoing interfaces of $c \in C_s$

The predicates in and out determine if a message $m \in M$ is received by an incoming interface or sent by an outgoing interface:

- $in : M \rightarrow Boolean$ with
  $in(m) = \exists i \in IIF | match(l_{recBy}(m, i))$
- $out : M \rightarrow Boolean$ with
  $out(m) = \exists i \in OIF | match(l_{sentBy}(m, i))$

In the following, we explain how to examine a proper model of a component-based system for several integration fault types (see Section III).

A. Examine for missing functions and extra functions faults

The basic unit that can be tested for integration is a single component. Assuming that every private interface should be bound to at least one other private interface, one can already check for missing function faults and extra function faults. Otherwise, there is at least one component providing some function that is not used by any other component. Or there is at least one component requiring some function that is not provided by any other component. Let $c1, c2 \in C_s \wedge c1 \neq c2$.

- Missing functions faults are indicated by incoming interfaces not bound to at least one outgoing interface:
  $\forall i \in PrIIF_{c1} \exists o \in PrOIF_{c2} : match(l_{boundTo}(i, o))$
- Extra functions faults are indicated by outgoing interfaces not bound to at least one incoming interface:
  $\forall o \in PrOIF_{c1} \exists i \in PrIIF_{c2} : match(l_{boundTo}(i, o))$

In addition, missing interface bindings indicate possible test gaps as no tests by other components are provided that use interfaces that are not bound.

B. Examine for wrong placed and missing call faults

The information captured in the models from the execution of the unit test cases can be used to define more elaborate integration checks to indicate certain miscoded call faults.

This can be best explained by an example. In the model depicted in Figure 2 the unit test case *UTC-1* of component *OrderService* expects an environment response message. The message sequence could be as follows: *UTC-1* stimulates component *OrderService* with message *M1* on interface *I1* (/placeOrder). The *OrderService* component responds with the message *M2* via interface *I2* (/orderCreated). Then, the environment - another component or a mock - responds with message *M3* on interface *I3* (/orderShippingStatus) to that message.

We call such an expectation regarding the environment's response to a request an *interaction expectation*. These need to be fulfilled by the components contained in the model as these represent the environment. This can be generally formulated as follows. Let $c_1, c_2 \in C_s$ with $c1 \neq c2$. Component c_1 expects an environment response message on incoming interface $inIF_{c1}$ as response to a message sent on its outgoing interface $outIF_{c1}$. This interaction expectation can only be met if there is a path from $outIF_{c1}$ to $inIF_{c1}$. We call this the *interface path expectation* that corresponds to the interaction expectation. This expectation is met if a component c_2 is stimulated by another unit test case with a message on an interface that is bound to $outIF_{c1}$ and sends a message via an interface that is bound to $inIF_{c1}$ in response or there is a recursive path over components $c_i..c_n$ that satisfies this condition.

This can be formalized: Let $m_1, m_2 \in M_s \wedge m_1 \neq m_2$. An *interaction expectation* is a tuple (m_1, m_2) where m_1 is sent to the environment and m_2 is a response of the environment to that message. In our example, the tuple (M_2, M_3) is an interaction expectation of the component *OrderService*. The set of all interaction expectations (IE) of a component c is given by:

$$
\begin{aligned}
IE_c = \{(m_1, m_2) | & \nu(m_1, type) = COMP\_RESP \wedge \\
& \nu(m_2, type) = ENV\_RESP \wedge \\
& match(l_{next} * (m_1, m_2))\} \wedge \\
& match(l_{sentBy} \circ l_{provBy}(m_1, c)) \wedge \\
& match(l_{recBy} \circ l_{provBy}(m_2, c))
\end{aligned}
$$

The set of all interaction expectations for a system s is given by union: $IE_s = \bigcup_{c \in C_s} IE_c$

As each message is sent to or received by an interface an interaction expectation can be mapped to a corresponding interface path expectation. Let $c \in C_s$, $i_1 \in OIF_c$ and $i_2 \in IIF_c$. An interface path expectation is a tuple (i_1, i_2) where i_1 is the starting (outgoing) interface and i_2 is the final (incoming) interface of an interaction path. An interface path expectation for an interaction expectation ie is obtained by mapping the first message of ie to the interface it is sent by and the second message to the interface it is received from.

Let $IPE_c \subseteq \{(i_1, i_2) | i_1 \in OIF_c \wedge i_2 \in IIF_c\}$ be the set of interface path expectations. The function $ipe : IE \rightarrow IPE_c$ returns the interface path expectation for a given interaction expectations of component c. It is defined as follows:

$$
\begin{aligned}
ipe((m_1, m_2)) = (i_1, i_2) | & match(l_{sentBy}(m_1, i_1)) \wedge \\
& match(l_{recBy}(m_2, i_2))
\end{aligned}
$$

In our example, the tuple (*I2*,*I3*) is the interface path expectation corresponding to the interaction expectation (M_2, M_3).

To determine if such a path can be triggered with a message on the starting interface, the interfaces that are bound to the starting interface need to be found. In our example, the incoming interface *I4* (/orderCreated) of component *WarehouseService* is bound to the outgoing interface *I2* (/orderCreated) of component *OrderService*. At least one component that provides such an interface needs to *react* to messages on that interface with a response on the final interface or there has to be a recursive path over multiple components that do so. In the example, the unit test case *UTC-2* of component *WarehouseService* specifies that component *WarehouseService* can react to a message at the incoming interface *I4* (/orderCreated) with a response on the outgoing interface *I5* (/orderShippingStatus). This interface is bound to the incoming interface *I3* (/orderShippingStatus) of component *OrderService* and thus complies to the interface path expectation that results from the execution of the unit test case *UTC-1* of component *OrderService*.

We denote a message m_2 on an outgoing interface a *reaction* to a message m_1 on an incoming interface if m_1 is the predecessor of m_2 or the predecessor message of m_2 was a reaction to m_1. This can be formalized by a reaction function as follows:

$$
\begin{aligned}
& reaction : (M_c \times M_c) \rightarrow Boolean \text{ with} \\
& reaction(m_1, m_2) = in(m_1) \wedge out(m_2) \wedge \\
& \quad \exists l_{next}(m_x, m_2) | \\
& \quad (m_x = m_1 \vee (out(m_x) \wedge reaction(m_1, m_x)))
\end{aligned}
$$

For interfaces we define a *reacts* function accordingly:

$$
\begin{aligned}
& reacts : (IIF_c \times OIF_c) \rightarrow Boolean \text{ with} \\
& reacts(i_1, i_2) = \exists m_1, m_2 \in M | match(l_{recBy}(m_1, i_1)) \wedge \\
& \quad match(l_{sentBy}(m_2, i_2)) \wedge reaction(m_1, m_2)
\end{aligned}
$$

To determine interaction paths across components, interface bindings need to be considered. The function $interactionPath_s : OIF_s \times IIF_s \rightarrow Boolean$ determines if there is an interaction path between an outgoing interface and an incoming interface in a system s:

$$
\begin{aligned}
& interactionPath_s(i, j) = \\
& \quad \exists i_1 \in IIF_s \exists i_2 \in OIF_s | match(l_{bountTo}(i, i_1)) \wedge \\
& \quad reacts(i_1, i_2) \wedge \\
& \quad (match(l_{bountTo}(i_2, j)) \vee interactionPath_s(i_2, j))
\end{aligned}
$$

The function $isSuitable : CBS \rightarrow Boolean$ determines if all interfaces of all components of a system $s \in CBS$ are bound in a suitable way such that they could fulfill all interface path expectations towards them. It is defined as follows:

$$isSuitable(s) = \forall c \in C_s \forall (i,j) \in IPE_c | \\ interactionPath_s(i,j)$$

It determines for any interface path expectation given by the component's unit test cases that it is matched by the behavior of other components based on their unit test cases, such that a path of interface invocations can be found as a reaction to a message on outgoing interface i such that a respecting incoming interface j might be invoked in response. This indicator can be used to check for certain missing and wrong placed call faults. Nevertheless, it is just an indicator, as the existence of those paths is necessary but not sufficient to ensure integrability, as the actual bodies of the messages have not been considered yet. However, the body of a message has an influence on whether the components actually interact in that way. The interface path expectation is an abstraction from interaction expectations. Different messages on the same starting interface can lead to different interactions. Testing these interaction paths requires proper integration testing using the messages that are actually triggered by the components on that path. The same applies to the detection of wrong function call faults and interface faults. As already explained in Section III wrong function calls are usually detected on the unit level. Nevertheless, there is a chance that the documented specification is misinterpreted when creating the tests, so the unit tests are successful, but the implementation is still faulty. Proper integration testing can detect such faults. The situation is similar for interface faults. E.g. if one component triggers messages that are too big to be consumed by their receivers, it would cause problems during integration testing. This is not within the scope of UTBI models. They can only provide indicators for integrability and support integration testing, as we will show in the next sections.

Extra call faults cannot be indicated with this approach because UTBI models only contain knowledge of what is expected and not what is not expected.

VII. The iterative UTBI testing process

After presenting the UTBI meta-model and model analyses to determine integrability indicators, the process of iterative integration testing based on such UTBI models and their analyses is presented next. The process is depicted in Figure 3. In the following, we describe the activities of this process and the artifacts that are created by them.

A. Create UTBI component models

As already mentioned before, our approach and the UTBI models are based on the existing unit test suites (UTS).

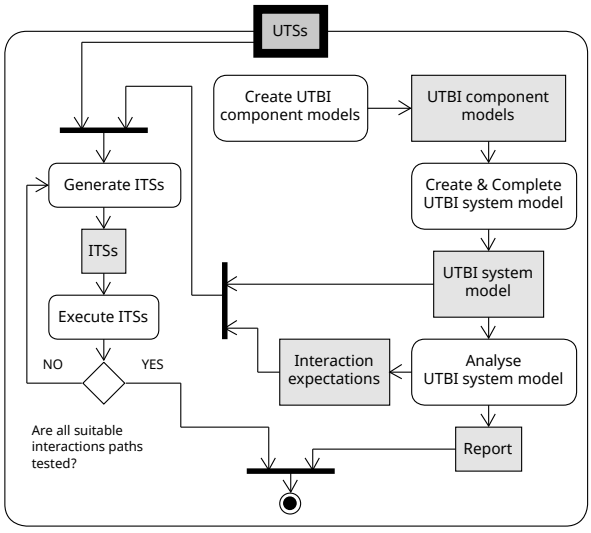

Fig. 3. The iterative integration testing process. Starting as soon as the UTSs are available. (UML activity diagram)

Therefore, to create a UTBI component model, the component's UTS is executed. During the execution of the UTS, the information regarding the contained unit test cases, the sent and received messages and used interfaces are extracted into a UTBI component model; the component itself is treated as a black box. The behavior captured in this way reflects both the actual specification of the component and the component's expected specification towards its environment.

B. Create & complete the UTBI system model

Once UTBI models for all components exist, these models need to be integrated into a global UTBI system model. For this purpose, incoming and outgoing interfaces that are bound to each other are determined. This is done using protocol-specific interface matching. The matching determines whether or not an incoming interface is bound to an outgoing interface, based on the protocol information of the interfaces stored in the UTBI component models. The resulting UTBI system model is complete with respect to the provided unit test suites.

C. Analyze the UTBI system model

The UTBI system model is analyzed using the integration fault indicators presented in Section VI. The result of the analysis is not only a report evaluating the indicators. In addition, the interaction expectations are discovered.

As stated before, the existence of interaction paths that comply with the interaction expectations is not sufficient to test for integration but is necessary and an indicator of integrability. Thus these interaction paths are used to generate so-called interaction test suites (ITS). Each *interaction test* tests an interaction of two components.

D. Generate & execute interaction test cases

Given a UTBI system model, each interaction path complying with an interaction expectation can be mapped to a

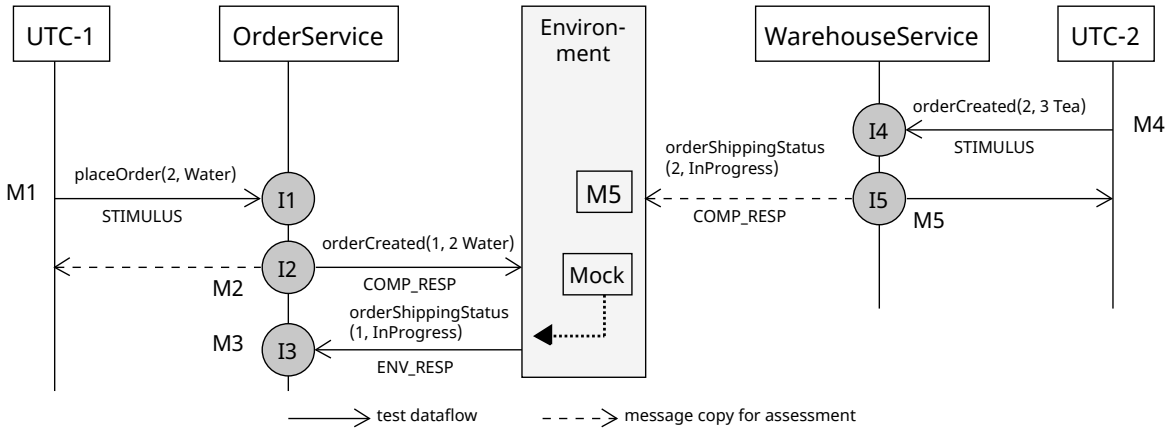

Fig. 4. Messages triggered by the unit test cases.

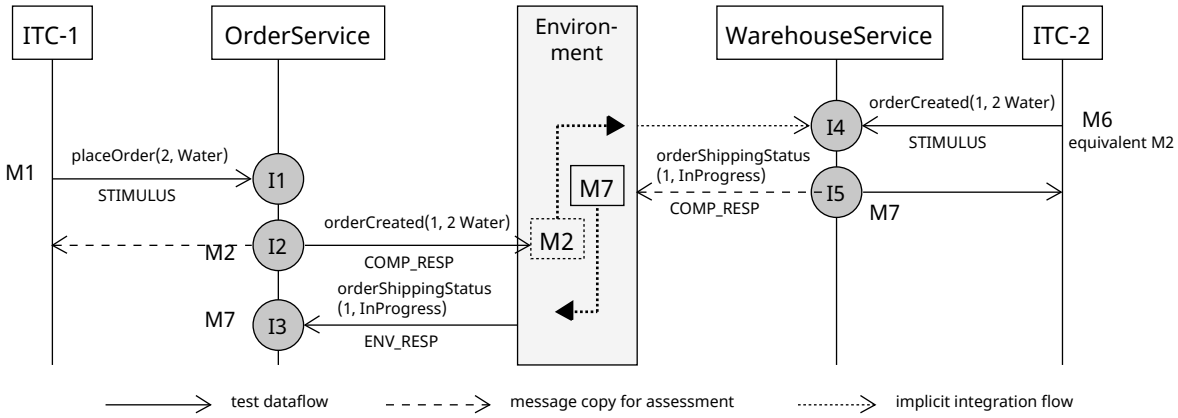

Fig. 5. Messages triggered by the generated interaction test cases.

sequence of unit test cases that led to that interaction path within the UTBI system model.

To describe the process of retrieving the test sequence in detail we introduce the $l_{reactionTo}$ relation between messages which describes that a component response message is a reaction to a stimulus or environment response message. In addition, the relation $l_{potentiallyTriggers}$ between interfaces is defined. It describes that a message sent to an outgoing interface of one component is potentially interacting with another component such that an outgoing interface of that other component is triggered. These relationships are defined as follows:

$l_{reactionTo} \subseteq M_s \times M_s$ with $M_s \times M_s :=$
$\{(m_1, m_2) | \exists m_3 \; match(l_{sentBy} l_{boundTo} l^-_{recBy}(m_2, m_3)) \wedge$
$reaction(m_3, m_1)\}$

$l_{potentiallyTriggers} \subseteq OIF_s \times OIF_s$ with $OIF_s \times OIF_s :=$
$\{(i_1, i_2) | l_{boundTo} l^-_{receivedBy} l^-_{reactionTo} l_{sentBy}(i_1, i_2)\}$

Utilizing the $l_{potentiallyTriggers}$ relation we can search for paths from the starting interface of an interaction expectation to the final interface. Each path that consists of a sequence of traversed $l_{potentiallyTriggers}$ relationships is mapped to corresponding test case sequences that contain these interface interactions. Using a breadth-first search algorithm the shortest sequence is returned first. For each unit test case of that sequence, a new interaction test case is generated. This is done iteratively, such that the original stimulus and environment response messages are exchanged by those component response messages triggered by the preceding test cases. This is repeated until a suitable interaction path for each interaction expectation is tested successfully or no more test sequences can be derived from the UTBI system model.

E. Example

The iterative integration testing process is explained using our example system again. Figure 4 depicts the messages with their interfaces, that are triggered by the execution of the unit test cases UTC-1 and UTC-2. During UTC-1 we observed

that M3 is expected as a response to M2. This results in the interaction expectation (M2, M3). This can be mapped to a corresponding interface path expectation (I2,I3), that describes that component *OrderService* expects an incoming message on interface I3 (/orderShippingStatus) as response to an outgoing message on interface I2 (/orderCreated). This interface path expectation is satisfied by the interaction path over the interfaces I2, I4, I5, and I3. It is derived from the behavior that was captured in the UTBI system model during the execution of UTC-1 and UTC-2.

The interation path is used to generate interaction test cases ITC-1 and ITC-2 as shown in Figure 5. For the ATC ShipItems that UTC-2 is derived from, a new interaction test case ITC-2 is generated where the triggered stimulus message is equivalent to the component response message M2 originally triggered by UTC-1. This message is used as a replacement for the stimulus message M4 in the original test case UTC-2. The interaction test case ITC-2 is captured similarly to the unit test cases before and stored in the UTBI system model. Thus the behavior of the *WarehouseService* in the context of the interaction with the *OrderService* as defined by UTC-1 is captured in the model. With the updated model, an interaction test case ITC-1 is generated based on the ATC OrderItems that UTC-1 is derived from. This time the environment response message M3 triggered by UTC-1 is replaced by the component response message M7 which is triggered by ITC-2. Thus, the interaction test case ITC-1 will no longer use the expected environment response, that was defined by the mock in the original unit test case UTC-1, but the actual response of the *WarehouseService* component in the context of that interaction.

Hence, the interaction expectation (M2, M3) is verified if all test cases on the interaction path over the interfaces I2, I4, I5, and I3 have been successful. These are the test cases UTC-1, ITC-2, and ITC-1. Note that M7 does not have to be the same as M3 but only equivalent with regards to the expectations of component *OrderService* towards the environment response message for the interaction test ITC-1 to be successful.

The UTBI system model evolves as the components and their unit test suites are getting developed. Using this model as a basis for integration test generation allows testing the interactions between components that are interacting according to the model at a very early stage. The whole process starts again if components and test suites change. The iterative approach ensures that the sequence of generated interaction tests is equivalent to an integration test that tests the interaction expectation. To automate the iterative integration testing process and demonstrate the soundness of the concept (Figure 3) a tool prototype, called InterACt [1], was developed together with a small running example project.

VIII. Discussion

In this section, we discuss how interaction test results can be interpreted and used to detect faults. We also discuss the advantages and limitations of the concept.

[1]InterACt can be accessed on Anonymous GitHub, URL: https://anonymous.4open.science/r/InterACt-4A21

A. Interpretation of interaction test results

With the presented approach, the discovered interaction expectations can be used to check whether the expectations of components towards their environment can be verified using the information extracted from the execution of all unit test cases. The following applies:

- An interaction expectation is verified if for at least one suitable interaction path all generated interaction test cases have been successful.
- An interaction expectation can not be verified if one generated interaction test case on each suitable interaction path fails.

In other words: if all interaction paths that satisfy the interface path expectation can not be successfully tested, either the components contain defects, the unit tests are too strict or the unit tests do not cover the interactions required to derive the correct path. Assuming the unit test cases cover the correct path but the interaction expectation can not be verified, at least one of the components contains a defect. This does not mean that the component whose test failed contains a defect, but requires further investigation by a developer.

B. Advantages and limitations

Runeson and Engstrom [19] describe the testing of software product lines as a three-dimensional problem across versions, variants and levels. In contrast to standard integration testing, our approach reuses the data, instructions, and oracles already provided by the unit test cases instead of specifying separate integration test case thus reusing test across levels. It also allows to test arbitrary compositions of components and versions, resulting in system variants, as the UTBI-System model is created based on a selection of UTBI-Component models. This leads to several advantages:

A1: This saves the effort of creating integration tests for the derived interaction expectations and allows to examine systems for wrong function faults and interface faults that can be traced back to incompatible actual and expected specifications. For example, if the output domain of one component is not a subset of the input domain of another component it is interacting with, the test cases of the latter are likely to fail when using the messages of the former as replacements for the messages used in the original unit tests.

A2: As the integration tests are based on an iterative execution of interaction test cases for single components the integration scope can be varied by testing single interactions up to complete paths, bridging the gap between testing individual components and full-fledged integration tests.

A3: Furthermore, the integration tests adapt themselves to changes in system design, component composition, and behavior changes.

A4: In addition, the iterative process requires only one component to be running at a time instead of large integration environments. This limits the maximum resource usage

to the maximum resource usage of one component rather than the entire system.

Our approach is meant to complement standard integration testing. It allows to detect integration problems even when little or no information about certain components is available. Furthermore, our approach allows to continuously search for integration faults during the development of components and unit test suites as they evolve. Last but not least, components affected by changes of another component can be detected, whenever they are part of one interaction path. However, the approach also has its limitations.

L1: A limitation of using the unit test cases as the single source of truth is that the integration tests can only be as complete as the coverage of the unit test suites of the components. Coverage in this case refers to the coverage of externally observable interaction paths through the activation of outgoing interfaces in response to received messages on an incoming interface. All relevant paths need to be captured for extensive integration testing using the proposed approach. Only those interaction expectations we can observe during the execution of the unit tests are tested. With that in mind, the generated interaction tests are not sensitive to all integration faults.

L2: Right now, the approach is limited to stateless components unless the test cases adapt the initial state of the component depending on the stimulus message. Otherwise, the state would not match the contents of the stimulus, so the test case of the component is meaningless and will probably fail no matter what. In our example given in Figure 2 the test cases for the *WarehouseService* have to adapt the warehouse inventory to contain water instead of tea when the stimulus message *M4* of the ATC ShippingTest is exchanged with the component response message *M2*.

IX. Related Work

In this section, we present related approaches to detect and mitigate integration faults and their differences from the one presented in this paper.

Instead of testing the implementation, specification-based approaches like *protobuff* ensure the structural consistency of APIs by generating the actual implementation from specified documents. – But these approches lack behavioral information [20]. Thus only interface faults can be prevented.

On the other hand, approaches like consumer-driven contracts were developed to test early, by decoupling parts of the integration test from the development of the interacting services through contracts that can be executed by all interacting parties. However, these are additional tests and do not replace integration tests [21]. – In contrast to the approach presented in this paper, consumer-driven contracts can not be used to check pass-through APIs, which are common in choreography-based architectures [22].

To test message-oriented systems, Santos et al. [23] propose a testing technique, that requires specifying the behavior of a system in advance, such that sequences of messages used to test the system can be derived from it. It is closely related to other specification-based testing approaches that use Linear Temporal Logic (LTL) to test such systems [24], [25]. – This is only possible if this specification is kept up-to-date with the actual specification of the system under test, which is rarely the case.

Benz [26] present an approach that requires existing models of components and systems to generate test cases that cover critical interaction scenarios. – Our approach reconstructs the models from the observation of the unit test cases and allows to execute the integration tests on a per-component basis.

Elbaum et al. [27] present an approach, called *differential unit testing,* that contrasts with the one we present in this paper. Instead of using isolated unit test cases to derive integration test cases, they used system test cases to derive unit test cases to test for differences in implementations of the same component in isolation. – This is only applicable if multiple implementations of the same component are developed, which is not realistic.

Gälli et al. [28] present the *EG-meta-model* that intends to develop test cases such that they can be composited. Since tests serve as a form of documentation they contain examples of how to use the units that are tested. These examples are extracted and can then be used to composite new more complex tests using the EG-Browser. – The idea of composing unit test cases that serve as examples of how to use a component is also the basis of the presented approach. However, we considered different kinds of communication protocols and extract expectations towards other components from those examples provided by the unit tests to automatically generate tests throughout the development and evolution of a system.

Schätz and Pfaller [29] propose an approach to validate the behavior of a component after its embedding into a system without instrumenting the component itself, treating it as a black-box test. – While our approach aims to assess the functionality of the system by reusing unit tests, this approach aims to verify the functionality of a component through system tests.

X. Conclusion & future work

The approach presented in this paper aims to support traditional integration testing by reusing already and always available artifacts, the unit test suites created for the developed components.

To answer the research question RQ1, we have developed the UTBI meta-model to get models containing structural and behavioral information of components and their interfaces based on unit test execution data. Using a property graph representation of UTBI models, we were able to define important integration failure indicators.

Analyzing the extracted information regarding the triggered messages and the involved interfaces, we developed a technique to generate so-called interaction tests to check the successful interaction between components. As interaction tests can be combined, they represent integration tests that

verify interaction expectations that were gathered from the UTBI model. This answers research questions RQ2 and RQ3.

In the future, a test engineering approach and framework will be developed to automatically generate interaction tests that adapt to changes in system design and component behavior for stateful components. For this purpose, the unit test cases of components need to be parameterized by incoming messages, so they can adapt the provisioned state before test execution to match the message contents, as already broached in Section VIII. In addition, a concept of contracts using LTL or similar logic could be introduced to be more expressive about the desired behavior of the environment of a component. Furthermore, interaction expectations that can not be derived implicitly from unit test data could be defined by such contracts. These can be evaluated in the same manner as the derived interaction expectations, using the data contained in the model as well as the test instructions and oracles contained in the unit test cases.

The tool prototype InterACt that implements the concept was already developed and evaluated on a small running example. The concept will be further evaluated regarding its ability to detect integration faults early and lower integration burden by reusing existing unit test cases in complex component-based systems, through an extension of the example project and an additional industry study to investigate the usability and scalability of the approach.

Complementary materials for reviewer:
InterACt: https://anonymous.4open.science/r/InterACt-4A21/README.md
Demonstration video (10 mins): https://youtu.be/0xobsjsO9rQ

References

[1] A. Sill, "The design and architecture of microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 76–80, 2016. [Online]. Available: http://doi.org/10.1109/MCC.2016.111

[2] M. Pezze and M. Young, *Software Testing and Analysis: Process, Principles and Techniques*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2008.

[3] S. Wang, I. Keivanloo, and Y. Zou, "How do developers react to restful api evolution?" in *Service-Oriented Computing*, X. Franch, A. K. Ghose, G. A. Lewis, and S. Bhiri, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 245–259. [Online]. Available: https://doi.org/10.1007/978-3-662-45391-9_17

[4] S. Mahmood and A. Khan, "An industrial study on the importance of software component documentation: A system integrators perspective," *Information Processing Letters*, vol. 111, no. 12, pp. 583–590, 2011. [Online]. Available: https://doi.org/10.1016/j.ipl.2011.03.012

[5] D. Firesmith, "Four types of shift left testing," Carnegie Mellon University, Software Engineering Institute's Insights (blog), Mar 2015, accessed: 2023-Jun-30. [Online]. Available: https://insights.sei.cmu.edu/blog/four-types-of-shift-left-testing/

[6] L. Riungu-Kalliosaari, S. Mäkinen, L. E. Lwakatare, J. Tiihonen, and T. Männistö, "Devops adoption benefits and challenges in practice: A case study," in *Product-Focused Software Process Improvement*, P. Abrahamsson, A. Jedlitschka, A. Nguyen Duc, M. Felderer, S. Amasaki, and T. Mikkonen, Eds. Cham: Springer International Publishing, 2016, pp. 590–597. [Online]. Available: https://doi.org/10.1007/978-3-319-49094-6_44

[7] M. Senapathi, J. Buchan, and H. Osman, "Devops capabilities, practices, and challenges: Insights from a case study," in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, ser. EASE'18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 57–67. [Online]. Available: https://doi.org/10.1145/3210459.3210465

[8] M. Nasution and H. Weistroffer, "Documentation in systems development: A significant criterion for project success," in *2009 42nd Hawaii International Conference on System Sciences*, 2009, pp. 1–9. [Online]. Available: https://doi.org/10.1109/HICSS.2009.167

[9] V. Garousi and T. Varma, "A replicated survey of software testing practices in the canadian province of alberta: What has changed from 2004 to 2009?" *Journal of Systems and Software*, vol. 83, no. 11, pp. 2251–2262, 2010. [Online]. Available: https://doi.org/10.1016/j.jss.2010.07.012

[10] A. Mann, A. Brown, M. Stahnke, and N. Kersten, "State of devops report," Puppet, Circle CI, Splunk, Tech. Rep., 2019. [Online]. Available: https://puppet.com/resources/whitepaper/state-of-devops-report

[11] B. Lima and J. P. Faria, "A survey on testing distributed and heterogeneous systems: The state of the practice," in *Software Technologies*, E. Cabello, J. Cardoso, A. Ludwig, L. A. Maciaszek, and M. van Sinderen, Eds. Cham: Springer International Publishing, 2017, pp. 88–107. [Online]. Available: https://doi.org/10.1007/978-3-319-62569-0_5

[12] "Google testing blog: Just say no to more end-to-end tests," https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html, 2015, (Accessed on 14/01/2023).

[13] H. K. N. Leung and L. J. White, "A study of integration testing and software regression at the integration level," *Proceedings. Conference on Software Maintenance 1990*, pp. 290–301, 1990. [Online]. Available: https://doi.org/10.1109/ICSM.1990.131377

[14] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*. USA: Addison-Wesley Longman Publishing Co., Inc., 1983.

[15] "ISTQB glossary — abstract test case," https://istqb-glossary.page/abstract-test-case/, (Accessed on 14/01/2023).

[16] R. Angles, "The property graph database model," in *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018*, ser. CEUR Workshop Proceedings, D. Olteanu and B. Poblete, Eds., vol. 2100. CEUR-WS.org, 2018. [Online]. Available: http://ceur-ws.org/Vol-2100/paper26.pdf

[17] I. F. Cruz, A. O. Mendelzon, and P. T. Wood, "A graphical query language supporting recursion," *SIGMOD Rec.*, vol. 16, no. 3, pp. 323–330, 1987. [Online]. Available: https://doi.org/10.1145/38714.38749

[18] M. Y. Vardi, "A theory of regular queries," in *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1–9. [Online]. Available: https://doi.org/10.1145/2902251.2902305

[19] P. Runeson and E. Engstrom, "Software product line testing – a 3d regression testing problem," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 742–746.

[20] Google, "Protocol buffers," http://code.google.com/apis/protocolbuffers/, (Accessed on 14/01/2023).

[21] C.-F. Wu, S.-P. Ma, A.-C. Shau, and H.-W. Yeh, "Testing for event-driven microservices based on consumer-driven contracts and state models," in *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, 2022, pp. 467–471.

[22] C. K. Rudrabhatla, "Comparison of event choreography and orchestration techniques in microservice architecture," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 8, 2018. [Online]. Available: http://doi.org/10.14569/IJACSA.2018.090804

[23] A. Santos., A. Cunha., and N. Macedo., "Schema-guided testing of message-oriented systems," in *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE,*, INSTICC. SciTePress, 2022, pp. 26–37. [Online]. Available: http://doi.org/10.5220/0010976100003176

[24] A. Michlmayr, P. Fenkam, and S. Dustdar, "Specification-based unit testing of publish/subscribe applications," in *26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06)*, 2006, pp. 34–34. [Online]. Available: https://doi.org/10.1109/ICDCSW.2006.103

[25] L. Tan, O. Sokolsky, and I. Lee, "Specification-based testing with linear temporal logic," in *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, IRI 2004*, 2004, pp. 493–498. [Online]. Available: https://doi.org/10.1109/IRI.2004.1431509

[26] S. Benz, "Combining test case generation for component and integration testing," in *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, ser. A-MOST '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 23–33. [Online]. Available: https://doi.org/10.1145/1291535.1291538

[27] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and replaying differential unit test cases from system test cases," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 29–45, 2009. [Online]. Available: https://doi.org/10.1109/TSE.2008.103

[28] M. Gälli, R. Wampfler, and O. Nierstrasz, "Composing tests from examples." *Journal of Object Technology*, vol. 6, pp. 71–86, 2007. [Online]. Available: https://doi.org/10.5381/JOT.2007.6.9.A4

[29] B. SchÃ¤tz and C. Pfaller, "Integrating component tests to system tests," *Electronic Notes in Theoretical Computer Science*, vol. 260, pp. 225–241, 2010, proceedings of the 5th International Workshop on Formal Aspects of Component Software (FACS 2008). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066109005222