INTERACT: a Tool for Unit Test Based Integration of Component-based Software Systems

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

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

***Abstract*—Testing complex component-based software systems is hard. Unit tests are focused but are not effective in exposing integration faults. However, integration test cases are difficult to develop and maintain. This paper presents a tool that uses unit tests to expose integration faults in component-based software systems. This is done by observing the component's unit test cases to derive the component's expectations of its interactions with other components. These expectations are validated using newly generated component integration test cases. Because the approach requires no new tests to be written, we consider it economical and effective.**

Keywords – component-based software; integration testing.

I. INTRODUCTION

Testing aims to expose faults and to assess that customer requirements are fulfilled. Many approaches have been developed to test software systems [1]. Testing isolated components of a software system - called *unit testing* - is an industry best practice. However, exposing certain types of faults at the unit level is impossible. Thus, tests on the integration level are needed that test the interaction of a component with other components - its environment [2]. However, creating and maintaining these integration tests is tedious. Architectural changes of the system and changes of the components require changes in the unit and integration tests [3]. We aim to automate this process for certain types of integration tests. To overcome some challenges of integration testing, we present a tool-supported approach that relies on existing unit tests. The knowledge encoded therein is used to determine how components expect to interact with their environment and to manipulate the unit tests such that integration aspects are tested. The thereby generated component integration test cases are related to each other such that they are equivalent to traditional integration tests that test those expectations.

The proposed approach and the tool were developed with the following research questions in mind:

- How can interaction expectations of components be extracted from unit test cases?
- How can tests, checking the interaction between a component and its environment, be derived from the unit test cases of the participating components?
- How to determine if a system can be integrated considering those component integration tests to continuously check the integration of a system as it evolves?

This paper is structured as follows: First, challenges of integration testing are presented in Section II. Section III introduces the **U**nit **T**est **B**ased **I**ntegration (UTBI) model which is the conceptual core of the approach. In Section IV we describe how UTBI models are used to derive the expectations components have regarding their environment and how these can be verified. Section V presents the tool INTERACT, implementing the proposed approach. Related work is discussed in Section VI. The planned next steps and future work conclude this paper in Section VII.

II. CHALLENGES OF INTEGRATION TESTING

Whenever a system is changed, tests need to be re-executed, and new tests are needed for new and changed features. In addition to knowing when a component is ready to be integrated, testers need to know how each component expects to interact with other components. Dedicated test specifications or any other form of documentation specify these interactions. However, creating integration tests is difficult, and studies show that documented specifications start to diverge from the implemented system over time [4] [5].

Furthermore, integration tests must be adapted when the underlying components change. Given an arbitrary number of interactions, there are a huge number of possible integration tests. Each interaction between two components can be tested separately by component integration tests as well as each possible subpath of interactions contained in the interaction path of all components that realize a customer feature. Creating and maintaining these tests is generally not feasible. Because of this, often only the most important integration tests are automated [1], [2], [6]. This contradicts the principle of testing as early as possible because these tests require all components to be ready for integration.

An automatable approach to keep the specification, which is used to test the integration, up-to-date with the system's implementation is needed to ensure that each component is integrated with all components as expected.

III. THE UNIT TEST BASED INTEGRATION META-MODEL

In the following, we introduce the UTBI meta-model (see Figure 1) which defines elements and relationships to model structural as well as behavioral information needed to test the

 ISBN: 978-1-68558-098-8

integration of components based on existing unit tests. A more detailed definition of the model and its foundations is already published. [7]

Components are core elements of the model. To abstract from various types of communication protocols, any interaction between components is treated as an activation of a component by a *Message* through an *Interface* that is *provided by* the component. A distinction is made between an *Incoming Interface* and an *Outgoing Interface*. Through an incoming interface, a message is *received by* a component, whereas messages are *sent by* 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 communication protocol. The protocol data is an attribute of the interface, e.g., for the Advanced Message Queueing Protocol (AMQP), the respective bindings are defined depending on the exchange type and queue bindings, while URLs and methods are used for (Representational State Transfer (REST).

For each component, the respective unit test cases are modeled. To this end, *Abstract Test Cases* for each Component Under Test (CUT) are included, which are templates without concrete input and expected values. A *Test Case* is *derived from* an abstract test case by providing concrete *values*.

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

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

Instances of the UTBI meta-model are called *UTBI component models*. They provide the core information for the automated integration testing process, introduced next.

IV. Integration testing based on UTBI models

In the following section, we describe the activities of the automated integration testing process based on such UTBI component models (see Figure 2).

Create UTBI component models (A1): To create all UTBI component models, the Unit Test Suites (UTS) of all components are executed. During execution, information regarding the unit test cases, the sent and received messages, and the used interfaces are extracted into a UTBI component model.

Derive interaction expectations (A2): To test the integration of all components, it is necessary to know how these components expect to interact with each other. These expectations towards their environment are implicitly defined by the messages triggered during UTS execution. Given a UTBI component model, every environment response that follows after a component response defines an expectation of that component towards the reaction of its environment. Resulting in a list of interaction expectations for each component.

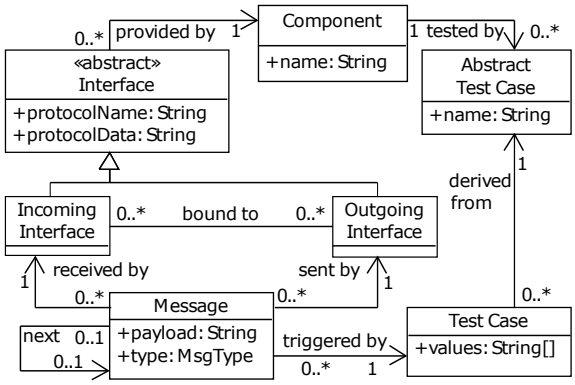

Figure 1. Unit Test Based Integration Meta-Model.

Create the UTBI system model (A3): To validate that all interaction expectations are fulfilled by the integrated system, all UTBI component models need to be merged into one *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 based on the protocol information attached to the interfaces. The resulting UTBI system model is complete concerning the provided UTSs.

Lookup possible interaction paths (A4): Each interaction expectation can be validated by searching for interaction paths from the interface via which the component response was sent by, to the interface that received the environment response. This might result in multiple paths, each spanning two or more components and interactions. However, which path is activated by the concrete component response depends on its content and the component's behavior.

Create interaction tests (A5): To validate the interaction expectations, the determined paths can be executed by exchanging the original stimulus message in a unit test case with the respective component response. We call these generated component integration test cases *Interaction Test Cases* (ITC) because they test exactly one interaction on an interaction path.

Run interaction tests (A6): When an ITC is successful, a new component response can be observed that replaces the originally mocked environment response in an additional interaction test case that is derived from the original UTC. If all ITCs on an interaction path are successful the interaction expectation is validated.

V. InterACt - an integration testing tool

The presented concept is implemented in the InterACt tool (code: https://github.com/NilsWild/InterACt, video: https://owncloud.swc.rwth-aachen.de/s/NtbMqMSByUoxv0q). InterACt is designed to support different protocols and their implementations. As shown in Figure 3 *Interface Observer*s collect the messages that are sent by and sent to the CUT. The collected data is stored in the *UTBI model store* to create the UTBI component models and the UTBI system

 ISBN: 978-1-68558-098-8

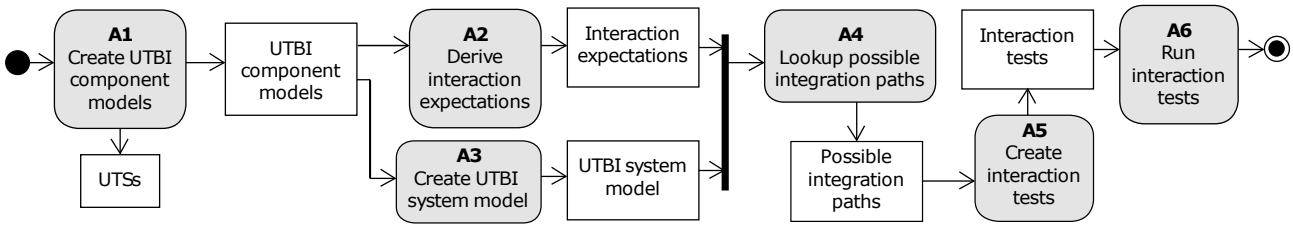

Figure 2. The integration testing process based on UTBI models.

model. INTERACT implements an extension mechanism to specify how the interfaces contained in the UTBI component models are bound via so-called *Interface Binder*s to create the UTBI system model. Based on the UTBI system model, the *Interaction Test Harness* retrieves integration data from the *ITC generator* and provides alternative parameters to the abstract test cases according to the derived interaction expectations and the corresponding interaction paths that need to be tested. INTERACT needs to be re-executed until no more interaction paths are untested or all interaction expectations are validated successfully.

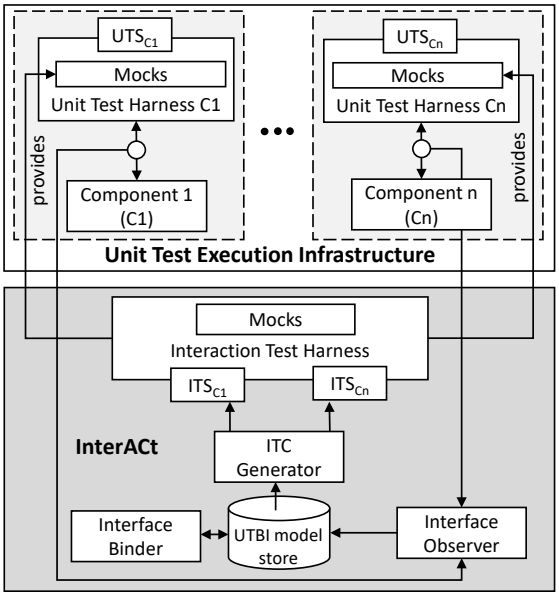

Figure 3. Embedding INTERACT in a unit test execution infrastructure.

A. An example application

To explain INTERACT'S integration testing process we present an example project (available on GitHub https://github.com/NilsWild/InteractionTestExample). This simple banking project consists of three components implemented as microservices using Spring Boot:

- `MoneyTransfer` (MT): Transfers money if the target IBAN is valid and the user's balance is sufficient.
- `IBANValidator` (IV): It checks if a given IBAN is valid.
- `BlacklistChecker` (BLC): It checks if a given IBAN is on the bank's blacklist.

Triggered by a transfer request, these components collaborate as follows (see Figure 5): First, the `MoneyTransfer` component asks the `IBANValidator` to validate the IBAN. To do so, the `IBANValidator` checks the IBANs format and requests the `BlacklistChecker` to check if the receiving IBAN is on the blacklist before it returns the validation result to the `MoneyTransfer` component. For each component, a unit test suite was created as well as mocks needed to test the components as shown in Figure 4.

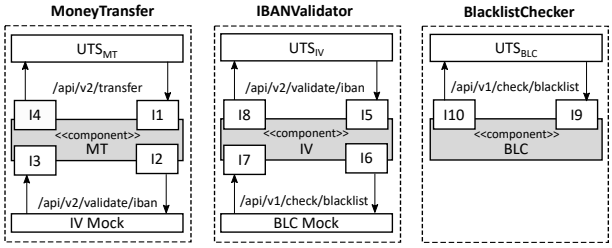

Figure 4. Components, unit test suits, and mocks of the example application.

Using this example, the integration process activities A1 to A6 are explained below.

Create UTBI component models (A1): To create the UTBI component models, INTERACT, its REST interface binder, and the UTBI model store (a neo4j database) are started. Then the unit test suites of all components are executed. The unit test cases are implemented similarly to JUnit parameterized tests. The same argument sources can be used but the tests are annotated with `InterACtTest` instead of `ParameterizedTest`. The parameters of each test case are the stimulus and environment response messages the CUT receives during test execution plus additional expected values for validation. For the `MoneyTransfer` component, three unit tests exist:

- MT-UT1: The `IBANValidator` mock returns that the IBAN is not valid. Thus, the transfer should fail.
- MT-UT2: The `IBANValidator` mock returns that the IBAN is valid but the test sets the state of the `MoneyTransfer` component such that the balance is insufficient. Thus, the transfer should fail.
- MT-UT3: The `IBANValidator` mock returns that the IBAN is valid and the test sets the state of the

 ISBN: 978-1-68558-098-8

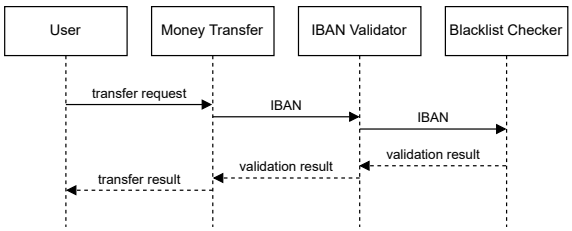

Figure 5. Target sequence of messages triggered by a transfer request.

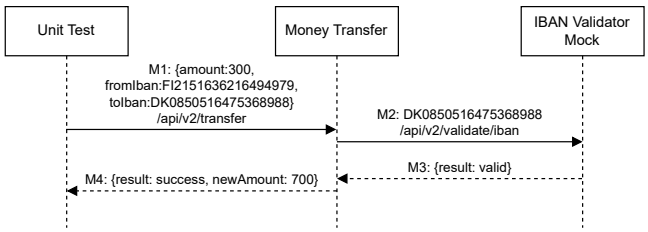

Figure 6. Sequence diagram showing the execution of unit test MT-UT3.

`MoneyTransfer` component such that the balance is sufficient. Thus, the transfer should succeed.

When they are executed, the UTBI component model for the `MoneyTransfer` component is created. It contains the triggered interfaces and the three collected message sequences. This model is stored in the UTBI model store. This is done for each component.

For the `IBANValidator` three unit tests exist:

- IV-UT1: The IBAN is valid but the `BlacklistChecker` mock returns that it is on the blacklist. Thus, the IBAN should be invalid.
- IV-UT2: The IBAN is valid, the `BlacklistChecker` mock returns that it is not on the blacklist. Thus, the IBAN should be valid.
- IV-UT3: An invalid IBAN is provided. Thus, the `BlacklistChecker` is not requested and the IBAN should be invalid.

For the `BlacklistChecker` two unit tests exist:

- BLC-UT1: The test provisions a blacklist that contains the given IBAN. Thus, the `BlacklistChecker` should respond with a match message.
- BLC-UT2: The test provisions a blacklist that does not contain the given IBAN. Thus, the `BlacklistChecker` should respond with a no-match message.

After all UTSs have been executed, all three UTBI component models are in the UTBI model store.

Derive interaction expectations (A2): Whenever new data is added to a UTBI component model INTERACT checks if an interaction expectation is contained in the new data. For the test case MT-UT3 (Figure 6), an interaction expectation from `M2` to `M3` is derived. For the other two behaviors of the `MoneyTransfer` component and the behaviors of the other components, interaction expectations are derived accordingly.

Create the UTBI system model (A3): After the UTBI component models are stored, INTERACT tries to bind the incoming and outgoing interfaces of each component using the provided interface binders. In our example, the REST interface binder handles the captured interfaces shown in Figure 4. They are bound such that the sequence of messages shown in Figure 5 is represented by the resulting UTBI system model. Specifically, the following interface pairs are bound together: (`I2` and `I5`), (`I6`, `I9`), (`I10`, `I7`), (`I8`, `I3`).

Lookup possible interaction paths (A4): Given the interaction expectations and the interface bindings, INTERACT tries to find path candidates to validate the expectations. For the interaction expectation from `M2` to `M3`, it tries to find a path from the outgoing interface `I2` with the URL `/api/v2/validate/iban` of the `MoneyTransfer` component to the incoming interface `I3` where `M3` was received. This is done with a breadth-first path expansion algorithm.

First, messages `M2` and `M3` are mapped to the interfaces `I2` and `I3` the messages were sent to, respectively received from. Next, all outgoing interfaces that are bound to the incoming interface `I3` that `M3` was received from, are collected. In this case `I8`. These are the interfaces that terminate the following path expansion. Starting with `I2` all incoming interfaces that are bound to it – in this case `I5` – and could thus receive `M2` are looked up. For all found interfaces, test cases that triggered messages on them are retrieved (IV-UT1, IV-UT2, IV-UT3). To keep the intention of the test cases, the found test cases are filtered such that only those that triggered `stimulus` messages on that interface are considered. This is true for all three of them. Next, all outgoing interfaces that are triggered as a reaction to a message on the respective incoming interface during these tests are collected. For IV-UT1 and IV-UT2 this is `I6`. For IV-UT3 this is `I8`, as the `IBANValidator` responds directly and the `BlacklistChecker` is not requested via I6. If one of those interfaces is a terminal interface the path is added to the list of possible interaction paths (IP). In the following, the IPs are represented by the unit test sequences that trigger the corresponding interfaces.

IP1 : MT-UT3→IV-UT3→MT-UT3

Then the remaining paths are further expanded starting with the found outgoing interface (`I6`) instead of `I2`. In our example the only interface bound to `I6` is `I9`. The test cases that triggered messages on `I9` are BLC-UT1 and BLC-UT2. `I10` is the outgoing interface that reacts to messages on `I9`. It is the start of the next expansion step as it is no terminal interface. The only interface bound to `I10` is `I7`. IV-UT1 and IV-UT2 triggered messages on `I7`. However, those messages were *environment response* and no *stimulus* messages. But as the tests have already been visited with the *stimulus* message and they are the next incoming message triggered during test execution, the path expansion is continued. This ensures that the integration path remains consistent with the unit test's intention. In both cases (IV-UT1, IV-UT2), a message on `I8` is triggered as a reaction to the message on `I7`. `I8` is a terminal interface. Thus, these paths are added to the list of possible interaction paths:

 ISBN: 978-1-68558-098-8

IP2 : MT-UT3→IV-UT1→BLC-UT1→IV-UT1→MT-UT3
IP3 : MT-UT3→IV-UT1→BLC-UT2→IV-UT1→MT-UT3
IP4 : MT-UT3→IV-UT2→BLC-UT1→IV-UT2→MT-UT3
IP5 : MT-UT3→IV-UT2→BLC-UT2→IV-UT2→MT-UT3

When no further expansion is possible, all interaction paths are found. Each found path is transformed into a *test execution plan*. Such a plan consists of the list of test cases in conjunction with the required information to replace the *stimulus* and *environment response* message with those triggered by the preceding test cases. All five paths and the resulting test execution plans are stored as candidates to validate the interaction expectation.

Create interaction tests (A5): When the test suites are re-executed the INTERACT JUnit extension requests the test execution plans for the CUT from INTERACT. Based on these plans, INTERACT determines parameter sets for the abstract test cases of the CUT based on the messages that were triggered by the components on that path so far. These parameter sets are sent to the JUnit test templates that represent the abstract test cases, resulting in new interaction test cases.

Run interaction tests (A6): When an interaction test gets executed and fails, the corresponding interaction path is not able to validate the interaction expectation. If every interaction test of an interaction path succeeds, the path validates the interaction expectation.

In our example, the process looks like this:

- IP1: Test case IV-UT3 which originally used an invalid IBAN and thus responded with a validation-failed message is parameterized with the IBAN contained in `M2` (DK0850516475368988) that was sent in MT-UT3 by `I2`, resulting in a new interaction test. With the now provided valid IBAN on `I5`, the test fails as the behavior that was tested by IV-UT3 was about receiving an invalid IBAN. The interaction path is not further considered.
- IP2-IP5: Based on IV-UT1 and IV-UT2 respectively, new interaction tests are generated that use the IBAN sent in MT-UT3 as well. As the IBAN format is valid like the one used in the unit test cases, both tests succeed. The paths get evaluated further. For IP3 and IP5, a new interaction test based on BLC-UT2 is generated. Therein the blacklist check received by `I9` contains the IBAN that originated from the unit test case of the `MoneyTransfer` component. As the test gets this message as a parameter, it sets the state such that the IBAN is not on the blacklist. The `BlacklistChecker` responds with a no-match message and the test succeeds. The response is sent via `I10` and is fed back to the `IBANValidator` test cases IV-UT1 and IV-UT2 as expected. In the two resulting interaction tests the `IBANValidator` receives the IBAN (DK0850516475368988), sends the blacklist check to the mock and the mock responds with the no-match response that was just observed in the preceding interaction test. The interaction test based on IV-UT1 fails, as the unit test case covered and asserted the behavior when the `BlacklistChecker` found a match. As the test failed, the path candidate IP3 was skipped for further evaluation. The one based on IV-UT2 succeeds accordingly. IP5 is evaluated further and the response triggered on `I8` is used as the mock response on `I3` in another interaction test based on test case MT-UT3. It succeeds and thus IP5 contains the unit tests that check the components' behaviors that are needed to satisfy the interaction expectation derived in A2. The expectation is validated by IP5. Note, that the response does not need to be equal to the mocked response `M3` in the unit test but leads to a validation of the defined assertions.

B. Detecting integration faults

Leung and White [3] presented a taxonomy for integration faults. Accordingly, integration faults are the result of misinterpretations of the documented specification on the providing or consuming side of a service as components are always developed based on an interpretation of their documented specifications.

INTERACT captures these interpretations by observing the sent and received messages by the executed UTSs and utilizes that information to validate that the consumer component and provider component interact compatibly with respect to their expectations. By analyzing the UTBI system model certain interaction fault types can be detected.

Mismatching interface definitions are detected as the replaced messages in the interaction test cases cannot be deserialized by the receiving component if the interface contract is broken. Furthermore, the assertions implemented in the UTS fail if the replaced environment responses or triggered component responses do not match the specified expectations. *Wrong function faults* are detected similarly.

In addition, *extra function faults* and *missing function faults* are detected, by querying the UTBI system model for unbound incoming and outgoing interfaces. If these are not public APIs they are either an indicator for such faults or an indicator for test gaps.

VI. RELATED WORK

Instead of testing the implementation, specification-based approaches like *protobuff* ensure the structural consistency of APIs by generating the actual implementation from specified documents. – These approaches lack behavioral information [8]. Thus, only interface faults can be prevented.

Approaches like consumer-driven contracts were developed to test early. However, these require additional tests and do not replace integration tests [9]. – In contrast to our approach, consumer-driven contracts cannot be used to check pass-through APIs, which are common in choreography-based architectures [10].

To test message-oriented systems, Santos et al. [11] propose a testing technique, that requires specifying the behavior of a system in advance. It is closely related to other specification-based testing approaches that use Linear Temporal Logic (LTL) to test such systems [12] [13]. – This is only possible if the specification is kept up-to-date with the actual specification of the system under test, which is rarely the case.

 ISBN: 978-1-68558-098-8

Benz [14] presents 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. [15] present an approach, called *differential unit testing,* that contrasts with our approach. Instead of using isolated unit test cases to derive integration test cases, they use 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.

Gälli et al. [16] present the *EG-meta-model* to create composable test cases. Since tests contain examples of how to use the units, these examples are extracted to composite new more complex tests. – The idea of composing unit test cases that serve as examples for the use of a component is also the basis of the presented approach. However, *InterACt* considers different kinds of communication protocols and extracts expectations towards other components from those examples to generate tests automatically.

Schätz and Pfaller [17] propose an approach to validate a component after it is embedded 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, their approach aims to verify the functionality of a component through system tests.

VII. Conclusion & future work

InterACt allows testing component-based systems using the implicitly specified interaction expectations encoded in the unit test cases. However, it is currently limited to those expectations encoded within the unit test cases and requires looking into the UTBI model store to verify that all interaction expectations are validated. To overcome these limitations and to widen the applicability of our approach, the following improvements are planned:

- Creating a report generator that provides an overview regarding the fulfillment of all interaction expectations.
- Extend InterACt to validate state expectations and extend the UTBI model by higher order interaction expectations. For example, once an "add IBAN to blacklist" request is sent and a 200 response code was received a transfer with that IBAN fails. This would allow testing that both parties interpret "IBAN was added to the blacklist" in the same way.
- Extending the approach to support asynchronous interfaces, where a request should result in some action but the result is not observed by the component that issued the request. Such expectations are not part of unit test cases and would require a separate specification approach. However, the validation process of InterACt could be reused.

InterACt as it is right now requires to parameterize the UTSs by the messages the CUT receives. However, unlike traditional integration testing which requires a resource-intensive integration environment, the interaction tests require the same resources as the UTSs. Furthermore, InterACt is capable to detect certain types of interface faults, missing function faults, and wrong function faults. Last but not least the interaction test cases adapt to architectural changes, as they are generated based on the provided interfaces and resulting interaction paths. We expect that our approach decreases the burden for integration testers, by reducing the amount of integration tests that need to be written manually. We plan to evaluate our approach and InterACt in a larger industry case study to not only show the concepts effectiveness but also evaluate its effectiveness and efficiency in a larger project.

References

[1] 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 Int. Publishing, 2017, pp. 88–107.

[2] 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.

[3] 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.

[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.

[5] 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.

[6] A. Mann, A. Brown, M. Stahnke, and N. Kersten, "State of devops report," Puppet, Circle CI, Splunk, Tech. Rep., 2019.

[7] N. Wild and H. Lichter, "Unit test based component integration testing (to be published)," in *30th Asia-Pacific Software Engineering Conference (APSEC 2023)*. IEEE Computer Society, 2023, [retrieved: Oct, 2023]. [Online]. Available: https://swc.rwth-aachen.de/docs/2023_APSEC__Wild_Preprint.pdf

[8] Google, "Protocol buffers," http://code.google.com/apis/protocolbuffers/, [retrieved: Oct, 2023].

[9] 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.

[10] C. K. Rudrabhatla, "Comparison of event choreography and orchestration techniques in microservice architecture," *Int. Journal of Advanced Computer Science and Applications*, vol. 9, no. 8, pp. 18–22, 2018.

[11] 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.

[12] A. Michlmayr, P. Fenkam, and S. Dustdar, "Specification-based unit testing of publish/subscribe applications," in *26th IEEE Int. Conference on Distributed Computing Systems Workshops (ICDCSW'06)*, 2006, pp. 34–34.

[13] L. Tan, O. Sokolsky, and I. Lee, "Specification-based testing with linear temporal logic," in *2004 IEEE International Conference on Information Reuse and Integration, IRI 2004*, 2004, pp. 493–498.

[14] S. Benz, "Combining test case generation for component and integration testing," in *3rd International Workshop on Advances in Model-Based Testing*, ser. A-MOST '07. New York, USA: ACM, 2007, p. 23–33.

[15] 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.

[16] M. Gälli, R. Wampfler, and O. Nierstrasz, "Composing tests from examples." *Journal of Object Technology*, vol. 6, pp. 71–86, 2007.

[17] B. Schätz 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).

 ISBN: 978-1-68558-098-8