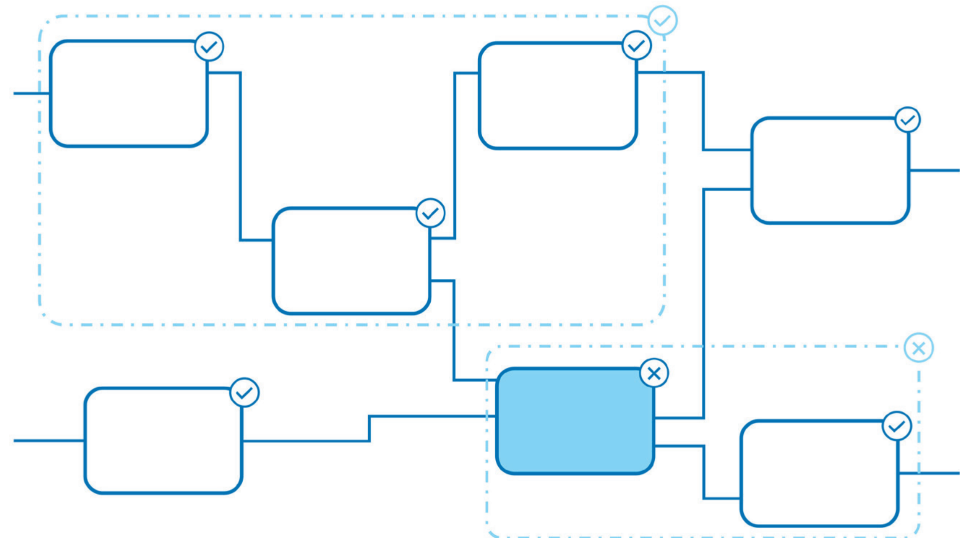
SE | RWTH AACHEN UNIVERSITY

Nils Wild

Interaction-based Integration Testing of Component-based Software Systems



Aachener Informatik-Berichte,
Software Engineering

Band 64

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe
Prof. Dr. rer. nat. Horst Lichter

Interaction-based Integration Testing of Component-based Software Systems

Von der Fakultät für Informatik der
RWTH Aachen University zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Nils Wild, M.Sc. RWTH
aus Düsseldorf

Berichter: Universitätsprofessor Dr. rer. nat. Horst Lichter
Professor Ian Gorton, Ph.D.

Tag der mündlichen Prüfung: 04. November 2025

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

Aachener Informatik-Berichte, Software Engineering

herausgegeben von
Prof. Dr. rer. nat. Bernhard Rumpe
Software Engineering
RWTH Aachen University

Band 64

Nils Wild
RWTH Aachen University

Interaction-based Integration Testing of Component-based Software Systems

Shaker Verlag
Düren 2026

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Zugl.: D 82 (Diss. RWTH Aachen University, 2025)

Copyright Shaker Verlag 2026
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the publishers.

Printed in Germany.

Print-ISBN 978-3-8191-0533-3
PDF-ISBN 978-3-8191-0485-5
ISSN 1869-9170
eISSN 2944-6910
https://doi.org/10.2370/9783819104855

Shaker Verlag GmbH • Am Langen Graben 15a • 52353 Düren
Phone: 0049/2421/99011-0 • Telefax: 0049/2421/99011-9
Internet: www.shaker.de • e-mail: info@shaker.de

Kurzfassung

Komponentenbasierte Softwaresysteme ermöglichen modulare Entwicklung und Wiederverwendung, stellen jedoch hohe Anforderungen an die Integrationstests. Integrationsfehler entstehen häufig durch das Zusammenspiel einzelner Komponenten und lassen sich mit isolierten Unit-Tests nur schwer erkennen. Bestehende Integrationsansätze erfordern meist einen hohen manuellen Aufwand sowie speziell konfigurierte Testumgebungen und Testfälle.

Diese Arbeit stellt den Interaction-based Integration (IBI) Testing Ansatz vor, der die automatisierte Durchführung von Integrationstests durch die Wiederverwendung vorhandener Unit-Tests ermöglicht. Grundlage des Ansatzes sind das IBI-Metamodell zur Beschreibung von Struktur-, Verhaltens- und Interaktionserwartungen sowie ein darauf aufbauender Testprozess. Aus bestehenden Unit-Tests werden Interaktionserwartungen abgeleitet und sogenannte Interaktionstestfälle generiert, indem Testergebnisse eines Unit-Tests als Eingaben für weitere Tests verwendet werden. Dadurch kann Systemverhalten simuliert werden, ohne dedizierte Integrationstests oder -umgebungen zu benötigen.

Der Ansatz wird formal mithilfe von Property-Graphen und regulären Pfadanfragen beschrieben, im Tool InterACt umgesetzt und anhand einer Fallstudie sowie einer industriellen Evaluation untersucht. Die Ergebnisse zeigen die grundsätzliche Machbarkeit des Ansatzes und identifizieren zugleich praktische Einschränkungen und offene Fragestellungen auf, die als Ausgangspunkt für eine gezielte Weiterentwicklung des Ansatzes dienen.

Abstract

Component-based software systems (CBSSs) enable modular design and reuse, but their integration poses substantial testing challenges. Integration faults often stem from interactions between components and are difficult to detect with isolated unit testing. Existing integration testing practices require significant manual effort, particularly in configuring test environments and maintaining test suites as systems evolve.

This dissertation introduces the Interaction-based Integration (IBI) Testing Approach, a novel approach to automate integration testing by reusing unit test cases. The IBI testing approach comprises the IBI Metamodel, which captures structural, behavioral, and interaction expectation aspects of CBSSs, and the IBI Testing Process, which operationalizes the IBI Metamodel. The approach leverages unit test cases to derive interaction expectations, and generate so called interaction test cases, by reusing the test outputs of one unit test case as inputs for another. It thereby simulates system behavior, enabling integration testing without the need for dedicated integration test cases or environments. In contrast to existing approaches, which require dedicated integration environments and handcrafted test cases, this work enables integration validation through existing unit test artifacts.

The IBI testing approach is formalized using property graphs and regular path queries. It is implemented in the InterACt tool, evaluated through a demonstration case study on a microservice system and an exploratory case study in collaboration with an industry partner. The results confirm the feasibility of the IBI Testing Approach while identifying practical limitations that guide future improvements.

This work contributes a reusable integration testing model, a formalized testing process, and a tool-supported implementation that provide a novel approach to integration testing of CBSSs. The findings highlight the potential of reusing unit test cases for integration testing, while also addressing the challenges and limitations of this approach in practice.

Contents

I. Introduction 1

1. Motivation and Research Design 3
1.1. Problem Statement . . . 3
1.2. Research Objectives . . . 5
1.3. Research Questions . . . 6
1.4. Solution Idea . . . 7
1.5. Research Approach . . . 8
1.6. Publications and Supervised Theses . . . 10
1.7. Dissertation Structure . . . 12

2. Foundations 15
2.1. Component-based Software Systems . . . 15
2.2. Testing of Component-based Software Systems . . . 18

3. Related Work 21
3.1. Built-in Testing Approach . . . 21
3.2. Component Metadata for Integration Testing . . . 22
3.3. System Specifications . . . 23
3.4. Test Reuse . . . 24
3.5. Test Coverage on the Integration Level . . . 25
3.6. Summary . . . 26

4. Running Example 29
4.1. System Architecture . . . 29
4.2. System Characteristics . . . 31

II. The Interaction-based Integration Testing Approach 33

5. The Interaction-based Integration Metamodel 35
5.1. Assumptions and Modeling Decisions . . . 35
5.2. Structure of the Interaction-based Integration Metamodel . . . 36
5.3. Component-based System Metamodel . . . 38
5.3.1. Component Structure . . . 39
5.3.2. Component Dynamics . . . 40
5.3.3. Composition of Components . . . 41

5.4. Test Metamodel 42
5.5. Interaction Expectations Metamodel 44
5.6. Interaction Expectation Verification Metamodel 47
5.7. Summary 48

6. The Interaction-based Integration Testing Process 49
6.1. The Interaction-based Integration Testing Process at a Glance 49
6.2. Formalizing Interaction-based Integration Models 52
6.2.1. Property Graphs 52
6.2.2. Regular Path Queries 53
6.2.3. Representing Interaction-based Integration Models as Property Graphs 54
6.3. The Interaction-based Integration Testing Process in Detail 59
6.3.1. Action 1: Capture Interaction-based Integration Component Models 60
6.3.2. Action 2: Create Interaction-based Integration System Model 61
6.3.3. Action 3 & 4: Generate Interaction Expectations 62
6.3.4. Action 5: Derive Potential Component Behavior Sequences . 67
6.3.5. Action 6 & 7: Generate and Execute Interaction Test Suites . 70
6.3.6. Action 8: Verify Interaction Expectations 73
6.3.7. Action 9: Collect Test Gaps 74
6.3.8. Action 10: Create Report 75

7. The Interaction-based Integration Testing Process in Action 77
7.1. Motivation & Overview of the Example Systems 77
7.2. Inputs to the Interaction-based Integration Testing Process 79
7.3. Capture Interaction-based Integration Component Models & Create Interaction-based Integration System Models 82
7.3.1. Transfer Gateway 82
7.3.2. Account Twin Service 84
7.3.3. Notification Service 85
7.3.4. Create the Interaction-based Integration System Models . . . 86
7.4. Collect Interaction Expectations 87
7.4.1. Unit Test-based Interaction Expectations 87
7.4.2. Derived Interaction Expectations 87
7.5. Derive Component Behavior Sequences 87
7.6. Generate & Execute Interaction Test Suites 89
7.7. Verify Interaction Expectations 91
7.8. Create Report 92

8. InterACt: A Tool for Automated Interaction-based Integration Testing 93
8.1. Implementation Goals 93
8.2. Architecture Overview 94

8.3. Realization . . . 96
8.3.1. JUnit 5 Test Framework Adapter . . . 96
8.3.2. Interface Observers . . . 98
8.3.3. Test Execution Manager . . . 100
8.3.4. Integration Controller . . . 101
8.3.5. User Interface . . . 104
8.4. Future Enhancements . . . 107

III. Evaluation and Conclusion 109

9. Evaluation Approach 111

10. Demonstration Case Study 113
10.1. Objectives . . . 113
10.2. Case Study Design . . . 114
10.2.1. Demonstration System . . . 114
10.2.2. Seeded Integration Faults . . . 115
10.3. Test Suite Execution . . . 115
10.4. Results . . . 116
10.5. Discussion . . . 118
10.6. Threats to Validity . . . 118

11. Exploratory Industrial Case Study 121
11.1. Objectives . . . 121
11.2. Case Study Design . . . 122
11.2.1. Component Selection . . . 122
11.2.2. Preparation and Instrumentation . . . 123
11.3. Identified Limitations . . . 125
11.3.1. Conceptual Limitations . . . 125
11.3.2. Technical Limitations . . . 127
11.4. Discussion . . . 128
11.5. Threats to Validity . . . 129

12. Conclusion 131
12.1. Discussion of the Research Questions . . . 132
12.2. Contributions . . . 133
12.3. Future Work . . . 135
12.4. Closing Remarks . . . 137

References 139

List of Figures 149

Acronyms 151

Symbols **153**

Identifiers **159**

Part I.

Introduction

Chapter 1.

Motivation and Research Design

Table of Contents

1.1. Problem Statement 3
1.2. Research Objectives 5
1.3. Research Questions 6
1.4. Solution Idea 7
1.5. Research Approach 8
1.6. Publications and Supervised Theses 10
1.7. Dissertation Structure 12

1.1. Problem Statement

Component-based Software Systems (CBSSs) promote the separation of concerns to modularize diverse functionalities within software systems. CBSSs have demonstrated advantages in managing team organization and accommodating rapidly changing requirements [Vit03]. By enabling the composition of reusable components, they facilitate the development of tailored software solutions [ABM00]. In such systems, multiple components interact via well-defined APIs to realize customer features [Sil16]. While modularizing functionality simplifies individual component development, it introduces significant challenges for ensuring the overall system's quality.

Testing plays a pivotal role in assuring quality, especially in highly composable and configurable systems [WLK20; LF17; PSL14]. A robust test strategy requires automation and multi-level testing to target different fault categories. As Mike Cohn, who introduced the test automation pyramid, observed [Coh]:

> "One reason teams found it difficult to write tests sooner was because they were automating at the wrong level."

This remark underscores the need to align test automation efforts with the types of faults that can be uncovered at each level of testing. While unit testing, which verifies the functionality of individual components, is relatively cheap, integration

testing, which examines interactions between components, demands significantly more effort [JJB+07]. Nevertheless, this effort is critical: faults such as interface mismatches, misinterpretations, and incorrectly coded interactions emerge only during integration testing, although both unit and integration tests may exercise similar code paths. These integration faults often arise from misunderstandings about the structure, functionality or behavior of interfaces between interacting components [LW90].

Despite its importance, recent studies indicate that automated testing at the integration level is less prevalent compared to unit-level test automation. A study by Winter et al. [WVS+21] found that 50% of participants reported full automation of their unit tests, whereas less than 21% achieved full automation at the integration level, and less than 9% automated all their system tests. Moreover, due to the vast number of possible interactions, only a small fraction of potential integration test cases are actually conducted [GV10; MBS+19; LF17].

Based on these known problems and difficulties, practitioners face several key challenges in integration testing:

C1 Collaboration: Testing the interactions between components, developed by different teams, requires team collaboration. Ideally, all possible interactions and changes to individual components and the system would be documented, and the impact of changes communicated to relevant teams. However, studies show that documented specifications often diverge from the implemented system over time, making them unreliable as a basis for test oracles [MK11; NW09].

C2 Test Case Selection: Identifying the necessary tests among the vast number of possible integration tests is daunting [BNdS22]. The complexity arises as each interaction between components can be checked by a multitude of integration tests. Each pairwise interaction can be tested in isolation, as well as part of longer interaction chains spanning multiple components.

C3 Test Case Maintenance: When a component changes, its related integration tests often require updates. This adds significant maintenance effort, especially when numerous interactions are affected [mInc22].

C4 Managing Test Environments: Setting up and maintaining test environments is often challenging. This is especially true for systems with many interacting components [BMK20].

Despite these challenges, integration testing is considered one of the most beneficial areas for investment in test automation [BNdS22]. The lack of automation is seen as the biggest pain point in testing overall [mInc22]. As tests must be re-executed and revised whenever a system changes or new features are added, solutions to these

issues are crucial. One of the core principles of agile methodologies and DevOps is to “shift left”, meaning to identify problems early in the development lifecycle to enhance quality and reduce error correction costs in later phases. Thus, test automation and continuous testing form the foundation of a successful test strategy. Economically, continuous testing requires automation, but automation alone does not guarantee continuous testing [RML+16; SBO18].

1.2. Research Objectives

This thesis introduces a novel approach, called the Interaction-based Integration (IBI) Testing Approach, consisting of the IBI Metamodel, IBI Testing Process, and IBI Tool InterACt. The IBI Testing Approach aims to automate integration testing by systematically reusing existing unit test artifacts and is designed to address the increasing complexity of CBSSs and the associated challenges of maintaining integration test coverage as CBSSs evolve. The research, presented in this thesis, is guided by the following objectives:

O1 Automate the Integration Testing Process: The primary objective is to automate the integration testing process for CBSSs. This includes capturing relevant information from individual components, generating test cases, and verifying the integration without requiring to maintain separate integration test suites manually.

O2 Reuse Existing Test Artifacts: To minimize redundant effort and leverage existing development practices, the IBI Testing Approach aims to reuse information from unit test cases. Unit test cases already encode assumptions about component behavior and interactions, which should be systematically extracted and transformed into integration-level verification criteria.

O3 Enable Early and Incremental Testing: A key goal is to enable early detection of integration issues by verifying integration assumptions as early as possible in the development process. This includes verifying that all components are properly connected, identifying unused or unreachable components, and progressively verifying system behaviors as they emerge from individual component behaviors and their interactions. By identifying integration issues sooner, teams can address problems before they propagate, leading to more stable and reliable software systems.

Two additional considerations refine these objectives:

Composition of Components: Knowledge about how individual components are composed into customer applications is critical. This requires analyzing component

interfaces, communication protocols, and data flows between components. These elements ensure that the integration tests accurately reflect system behavior and dependencies.

Maintenance Effort for Integration Environments: Maintaining integration environments can be resource-intensive. The proposed IBI Testing Approach should reduce this effort by automating setup and teardown. It should also ensure that tests remain resilient to changes in the system architecture.

While CBSSs, particularly distributed systems, face a wide range of challenges such as fault tolerance, latency, and scalability [Gor22], this thesis explicitly focuses on verifying functional properties of CBSSs. The proposed IBI Testing Approach aims to ensure that components are correctly connected and exhibit the expected behavior during interactions. Non-functional properties, including availability, performance, and resilience to partial failures, are considered out of scope for this work.

1.3. Research Questions

Unit test cases, developed alongside individual components, form the foundation of the proposed IBI Testing Approach. To enable isolated testing, unit test cases commonly use mocks to simulate interactions with the component's environment. These mocked interactions reflect how a component is expected to interact with others, and thereby implicitly encode interaction expectations. If such expectations can be systematically identified and formalized, they may serve as a basis for integration testing. Building on this, the question arises of whether interaction expectations, for example those implicitly defined through mocks in unit test cases, can be verified without requiring dedicated integration specifications or test environments.

To investigate this, the research focuses to answer the following research questions:

RQ 1: How can interaction expectations be systematically extracted from unit test cases?
This question is refined by the following two sub-questions:

- **RQ 1.1:** What types of interaction expectations are implicitly encoded in unit test cases?
- **RQ 1.2:** Which additional interaction expectations must be considered for comprehensive integration verification?

When interaction expectations are extracted from unit test cases, these expectations need to be formalized in a way that they can be used for automated integration verification. This leads to the second research question:

RQ 2: How can interaction expectations be formalized to support automated integration verification?

When formalized interaction expectations are available, they need to be verified against the actual behavior of the components that are to be integrated. This leads to the third research question:

RQ 3: How can integration testing be conducted based on formalized interaction expectations?

RQ 3.1: How can it be determined whether a given set of components can be successfully integrated?

RQ 3.2: How can interaction expectations be verified without developing dedicated integration tests and maintaining dedicated integration environments?

1.4. Solution Idea

The proposed IBI Testing Approach leverages existing unit test cases to systematically derive integration tests. The central idea is that mocks, developed for unit test purposes, implicitly define interaction expectations and any system behavior that fulfills an interaction expectation can be decomposed into a sequence of individual component behaviors and interactions. By verifying these interactions in the context given by the interaction expectation, and by ensuring that the inputs and outputs of subsequent component behaviors are consistent, the system can be verified without executing it as a whole. This is achieved by simulating the system behavior through the coordinated verification of individual component behaviors.

In the IBI Testing Approach, unit test data is reused to verify inter-component interactions. If the output of one unit test case falls within the input equivalence class of another, it can serve as a stimulus for generating an interaction test case for the receiving component. A sequence of such test cases in conjunction with an interaction expectation forms an integration test case that verifies the system behavior realized by those components.

The IBI Testing Approach operates iteratively by feeding the output of one unit test case into the input of another. This enables the automated construction of interaction test cases without requiring a dedicated integration environment. As a result, regression testing can focus on the segments of system behavior that have changed, reducing integration effort. The IBI Testing Approach directly addresses the core challenges of integration testing in CBSSs by:

- reducing collaboration overhead across teams (C1) by reusing the information provided by unit test cases,
- adapting to evolving component behaviors and system architecture (C2, C3) as the generated interaction test cases change when the unit tests change, and

- eliminating the need for costly test environments (C4), since each interaction test case that forms part of an integration test case is executed as a unit test case.

The IBI Testing Approach is realized through three key steps:

Extract Component Behavior and Interactions: Unit test cases serve as a source of information about individual component behavior and their interactions with surrounding components. By capturing messages exchanged during unit test execution, a precise model of how each component behaves and interacts is created.

Centralize and Analyze Information: The extracted behavioral and interaction data is centrally collected and systematically analyzed. This analysis identifies interaction expectations and component dependencies that require verification at the integration level, including those not explicitly covered by individual unit tests.

Generate and Execute New Interaction Tests: Based on the analysis, interaction test cases are automatically generated. These tests verify the correctness of the interaction flows derived from combining existing unit test cases, ensuring that components cooperate as expected. The entire process of test generation and execution is automated to minimize manual effort and ensure up-to-date integration tests in response to evolving system behavior. This automation enables continuous integration verification without manual coordination between teams or reliance on explicit integration environments.

1.5. Research Approach

To address the outlined objectives, this thesis adopts the Design Science Research (DSR) approach. DSR is a well-established research methodology in technical disciplines that supports the development of artifacts—such as models, methods, or tools—to solve real-world problems [HMP+04; Cro93; Goe14].

Design Science Research Framework

The DSR framework by Hevner et al. [HMP+04] is grounded in two foundational contexts: the *environment* and the *knowledge base*, shown in Figure 1.1.

The *environment* consists of people, organizations, and technology. It defines the problem space by capturing goals, tasks, constraints, and opportunities perceived within real-world contexts. Business needs are shaped by organizational structure, strategies, processes, and available technologies. These needs define the relevance of research activities.

The *knowledge base* includes foundations (e.g., theories, models, and prior research), methodologies (e.g., empirical techniques), and experience. It provides

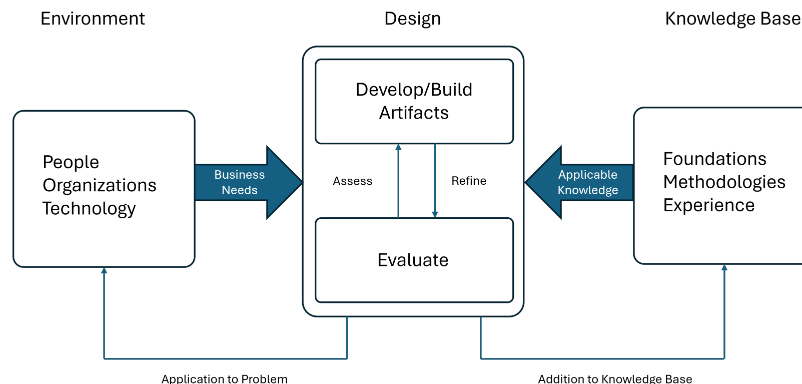


Figure 1.1.: Design Science Research framework adapted from Hevner et al. [HMP+04]

the scientific grounding used to develop artifacts and evaluate their utility.

Within this framework, DSR advances through two interlinked activities:

- The *develop/build* activity uses applicable knowledge to create artifacts that address problems derived from the environment.
- The *evaluate* activity assesses the artifacts' utility and effectiveness, contributing insights back to the knowledge base and informing future refinement.

DSR thus operates as a bridge between the problem-driven needs of the environment and the scientifically grounded insights of the knowledge base. As Hevner et al. [HMP+04] argue, good design science research should combine *utility*, which refers to solving relevant problems, with *truth*, which refers to contributing to scientific knowledge. "Truth informs design and utility informs theory." [HMP+04] A practically useful artifact may reveal previously unknown theoretical relationships, just as validated theories may guide the creation of more effective artifacts.

Design Science Methodology

Peffers et al. [PTR+07] propose a structured methodology for conducting DSR, guiding the development and evaluation of artifacts. This methodology consists of six key steps:

1. **Problem Identification and Motivation:** Clearly define the problem to be addressed and justify the relevance of finding a solution.
2. **Define Solution Objectives:** Specify what a successful solution should achieve, based on the problem context.

3. **Design and Development:** Construct the artifact, including models, methods, or systems, that are intended to solve the identified problem.

4. **Demonstration:** Show how the artifact can be applied to solve instances of the problem in a relevant context.

5. **Evaluation:** Assess how well the artifact performs with respect to the problem and objectives using appropriate evaluation methods.

6. **Communication:** Document and disseminate the results to relevant academic and practitioner audiences.

These steps ensure a systematic and rigorous research process that balances practical relevance and scientific contribution.

1.6. Publications and Supervised Theses

In the course of this thesis, several scientific contributions were made and published in peer-reviewed venues. These publications present different aspects of the IBI Testing Approach. In addition, the research results presented in this thesis have inspired and guided a variety of Bachelor's and Master's theses. These theses were conducted under my supervision and contributed valuable practical evaluations, tool extensions, and case studies related to the core research.

Publications

The following peer-reviewed publications resulted directly from the research conducted in this thesis. I am the main author of all listed papers and responsible for the underlying research ideas, the development of the concepts, and the preparation of the papers.

[WLM24] Nils Wild, Horst Lichter, and Constantin Mensendiek. Expectation-based integration testing of unidirectional interactions in component-based software systems. In *19th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2024)*, pages 202–213, Angers, France. SCITEPRESS – Science and Technology Publications, Lda, April 2024

This paper builds on the findings of Constantin Mensendiek's Master's thesis. I further refined the concepts and led the writing of the publication. The work contributed to the development of the IBI Metamodel, which is presented in Chapter 5.

[WL23b] Nils Wild and Horst Lichter. Unit test based component integration testing. In *30th Asia-Pacific Software Engineering Conference (APSEC 2023)*,

pages 1–10, Seoul, South Korea. IEEE Computer Society, December 2023

This paper introduces the foundational ideas of the IBI Testing Process. I developed the contributions and led the writing. The concepts presented form the core of Chapter 6.

[WL23a] Nils Wild and Horst Lichter. InterACt: a tool for unit test based integration of component-based software systems. In *18th International Conference on Software Engineering Advances (ICSEA 2023)*, pages 58–63, Valencia, Spain. IARIA, November 2023

This paper presents the initial proof of concept for the InterACt tool. I am the main contributor and led the writing. The implementation details discussed are integrated into Chapter 8.

[WLK20] Nils Wild, Horst Lichter, and Peter Kehren. Test automation challenges for application landscape frameworks. In *International Conference on Software Testing, Verification and Validation, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020

This paper outlines key challenges in integration testing for CBSSs, providing the motivation for this thesis. I am the main contributor and led the writing. The results are reflected in Section 1.1 of the introduction.

Supervised Bachelor's and Master's Theses

Several parts of this thesis are informed by work conducted in the context of Bachelor's and Master's theses I supervised. For each thesis, I defined the thesis topic and research objectives, and provided the overarching architectural and evaluation structure. The students independently developed the detailed approach, carried out the implementation and evaluation, and authored the written thesis. I provided continuous guidance and feedback throughout the process and reviewed the final thesis. For transparency, this section notes where the outcomes of individual theses are reflected in specific chapters of the thesis.

- Gregorius Ian Halisantoso. Evaluation of InterACt in an Industrial Software Landscape. Master's Thesis. 2024.
  The content is mainly incorporated into Chapter 11.

- Robert Lange. Development and Evaluation of a User Interface for InterACt. Bachelor's Thesis. 2024.
  The resulting prototype is the foundation of InterACt's user interface presented in Chapter 8.

- Konstantin Nhat-Thanh Dao. Design and Development of an Extensible Analysis Tool for InterACt. Bachelor's Thesis. 2024.
  The results influenced the tool InterACt presented in Chapter 8.

- Constantin Mensendiek. Unit Test based Verification of Unidirectional Interactions in Component-based Systems. Master's Thesis. 2023.
  The finding are mainly incorporated into Chapters 5 and 8.

- Sandro Schulte. Unit Test Based Verification of Related Interactions in Stateful Component-Based Software Systems. Master's Thesis. 2023.
  The findings are mainly incorporated into Chapters 5 and 6.

- Bora Avcu. An Integrability Model for Component Based Systems. Bachelor's Thesis. 2023.
  The findings are mainly incorporated into Chapter 5.

- Florian Braun. Test Based Architecture Reconstruction and Documentation. Bachelor's Thesis. 2023.
  The findings are mainly incorporated into Chapter 5.

- Caspar Zecha. Test Selection Model for Highly Configurable Systems. Master's Thesis. 2022.
  The findings influenced the design of the IBI Metamodel presented in Chapter 5.

- Arosan Annaligam. Using Mock Expectations to Derive Integration Test Cases. Master's Thesis. 2022.
  The findings contributed to the development of the IBI Metamodel presented in Chapter 5 and the IBI Testing Process presented in Chapter 6.

- Viktor Schneider. Behaviour Modelling for Consumer Driven Contracts. Master's Thesis. 2022.
  The findings contributed to the development of the IBI Metamodel presented in Chapter 5 and the IBI Testing Process presented in Chapter 6.

- Md. Monis Khan. Modeling Event Driven Systems for Integration Testing. Master's Thesis. 2021.
  The findings contributed to the development of the IBI Metamodel presented in Chapter 5.

1.7. Dissertation Structure

This thesis is organized in accordance with the six steps of the DSR methodology presented in Section 1.5. Each chapter of the thesis contributes to one or more of these steps, ensuring both scientific rigor and practical relevance.

- **Problem Identification and Motivation** is addressed in Chapter 1, specifically in Section 1.1. This section outlines the key challenges in integration testing of CBSS and motivates the need for a new approach.
- **Define the Objectives for a Solution** is covered in Section 1.2. This section formulates the goals of the research based on the identified challenges and sets the direction for the design of the artifact.
- **Design and Development** forms the core of Part II. Chapter 5 introduces the IBI Metamodel. Chapter 6 presents and formalizes the IBI Testing Process. Chapter 8 describes the implementation of InterACt, the tool that implements the IBI Metamodel, and IBI Testing Process.
- **Demonstration** is addressed in Chapter 7. This chapter illustrates how the IBI Testing Process is applied to a CBSS. Each step of the IBI Testing Process is demonstrated using specific system configurations and test cases and interaction expectations.
- **Evaluation** is covered in Part III. Chapter 9 explains the evaluation methodology. Chapter 10 presents a demonstration case study using the Piggy Bank system. Chapter 11 provides insights from an industrial exploratory case study. This case study identifies practical limitations and validates the applicability of the proposed IBI Testing Approach in an industrial setting.
- **Communication** is achieved through this thesis and through several peer-reviewed publications that disseminate the results. An overview of these publications and supervised theses is provided in Section 1.6. Chapter 12 concludes the thesis by summarizing the contributions and outlining directions for future research.

This structure ensures that all steps of the DSR process are addressed systematically. The thesis proceeds from problem identification through artifact development and evaluation, concluding with dissemination of the results.

Chapter 2.

Foundations

Table of Contents

2.1. Component-based Software Systems 15
2.2. Testing of Component-based Software Systems 18

This chapter introduces the fundamental concepts and terminology relevant to the integration testing of CBSSs, that is used throughout this thesis.

2.1. Component-based Software Systems

Developing complex and high-quality software systems has long posed significant challenges. In the late 1960s, concerns over the growing gap between software complexity and development capabilities led to what became known as the *software crisis*. These concerns prompted the organization of the first NATO Software Engineering Conference in 1968 [Pet68], which laid the groundwork for systematic software development and already included a vision paper on CBSSs and their development [McI68]. In the years that followed, a variety of architectural paradigms were introduced to improve modularity and manageability. These ranged from procedural and object-oriented programming [Weg90] to component-based software development [Szy02] and, more recently, microservices [Fow]. While these approaches reduced the complexity of individual components, they introduced new challenges at the system level. In particular, the overall complexity of the system increased as more functionality became distributed across independently developed components. In such systems, user-facing features are typically realized through the interaction of multiple components via APIs [Sil16]. This thesis focuses on such CBSSs.

In general, a system can be considered a “combination of interacting elements organized to achieve one or more stated purposes”, according to the International Standard - Systems and Software Engineering Vocabulary [IEE17].

An interaction with an element of the system is always started by a trigger, which in turn triggers a behavior of this element that realizes a functionality that satisfies the purpose of this element. The resulting behavior result can in turn trigger further interactions. An interaction is thus defined as follows:

Interaction An interaction I is a tuple $I = (T, B, R)$ with T being the behavior trigger, B being the triggered behavior, and R being the behavior result.

As CBSSs are considered in this thesis, the interacting elements are components. There is no general definition of what a component is. However, many definitions for different contexts have been proposed. For example, in the context of CBSSs, Szyperski defines a software component as “a unit of composition with contractually specified interfaces and explicit context dependencies” [Szy02]. Similarly, the ISO/IEC/IEEE 24765-2017, Systems and software engineering–Vocabulary defines a component as “one of the parts that make up a system”[IEE17]. Although these definitions offer useful perspectives, none of them fully capture the specific role components play in the context of this thesis. Therefore, a set of tailored definitions is introduced to precisely reflect the assumptions and terminology used throughout this thesis:

Component A component is a modular software unit that encapsulates specific functionality, that is exposed through a set of well-defined interfaces. It serves as a building block for assembling component-based systems.

Interface An interface is a shared boundary that connects two or more components [IEE17].

Component-based Software System A component-based software system is a set of interacting components organized to achieve one or more stated purposes. The simplest system is a system that consists of just one component. A system can be a component of another system.

Subsystem A subsystem is a system contained in another system.

This thesis assumes that any CBSS can be regarded a message-based system. This means that the components of a CBSS exchange messages with each other via interfaces in order to utilize the functionalities of the other components of the system.

Message A message is communication sent from one component to another via an interface. A message may carry a request or an informational payload. [IEE17].

The set of components a given component exchanges messages with, as well as those that affect or are affected by its behavior through transitive message exchanges or state changes, collectively shape the component's environment.

> **Environment** The environment of a component c consists of all other components in the system whose behavior is directly or indirectly influenced by, or influences, interactions with the component c, including components affected through state changes or transitive effects.

Interactions are referred to either as component interactions, system interactions or environment interactions, depending on the context. This also applies to behavior triggers, behaviors and behavior results. A system and environment interaction therefore represent a collection of component interactions that are triggered transitively in response to the system and environment behavior trigger respectively.

The behavior of an entity, such as a component, its environment, or a system, depends not only on the messages received, but potentially also on its state. The behavior is defined as follows:

> **Behavior** A behavior B is a tuple $B = (T, R)$, where T is the behavior trigger and R is the resulting behavior result. A behavior defines how an entity reacts to a specific behavior trigger by producing a corresponding result.

> **Behavior Trigger** A behavior trigger T is a tuple $T = (\{M_1, \cdots, M_n\}, S)$ with M_i being a message that is received by an entity - also called stimulus message or stimulus for short - and S being the entity state.

> **Stimulus** A stimulus is a message received by one entity (the receiver) that was sent by another entity (the sender), stimulating the receiver, causing a response [IEE17].

> **State** The state of an entity is defined by the values assumed at a given instant by the variables that define the characteristics of an entity at a given instant in time [IEE17].

As the triggered behavior might change the entity state and cause messages to be sent to the environment the behavior result is defined as:

> **Behavior Result** A behavior result R is a tuple $R = (S, \{M_1, \cdots, M_n\})$ with S being the resulting entity state and M_i being a message that is sent in reaction to the stimulus, also called reaction message or reaction for short.

> **Reaction** A reaction is a message sent by one entity (the sender) that will be received by other entities (the receivers). A reaction is the response to a stimulus.

2.2. Testing of Component-based Software Systems

Testing is most efficient when conducted early in software development, as it allows defects to be identified and resolved before they cause additional effort in later stages. In CBSSs, testing begins with the verification of individual components using unit tests to ensure compliance with their specified requirements. Once the components are tested, the interactions between these components need to be checked with integration tests, if they satisfy the overall system requirements. This section introduces the concepts associated with this testing procedure, emphasizing the transition from component-level verification to interaction-based system testing.

> **Test Case** A test case is a sequence of test steps designed to verify specific aspects of a software component or system. Each step describes an interaction between an actor and the unit under test, consisting of an input, an action, and an expected result. When the unit under test is a single component it is called a unit test if a system is the unit under test it's called an integration test [ISTQB].

To avoid repetition, concrete test cases are derived from abstract test cases that define general test steps without specifying concrete data. An abstract test case therefore defines an expected input domain for which the specified expectations apply and the inputs of the derived test cases adhere to. The results that satisfy the expectation are the expected output domain.

> **Abstract Test Case** An abstract test case is a set of abstract test steps, defining an input domain and specifying the expected output domain for a given input [ISTQB].

The test steps contain the state and the messages exchanged with the unit under test. Those messages are distinguished, depending on their role within an interaction. As already stated, a message that is sent to a unit, stimulates the unit and is thus called a stimulus. The receiving unit handles the stimulus depending on its current state, causing a behavior to be triggered that results in reaction messages. These reactions cause environment interactions. If a reaction is sent with the intention of eliciting a response from the environment, the reaction is considered a request message. The message received in response to that request is an environment response message which is the result of that environment interaction.

> **Request** A request is a message sent from one component (the sender) to another component (the receiver), stimulating and directing the receiver to fulfill one of its responsibilities [IEE17].

> **Environment Interaction** An environment interaction is an exchange of messages between a component and its environment. It consists of requests and environment responses.

> **Environment Response** An environment response is a message sent from the environment to a component, in response to a request. The environment response is the result of the environment interaction and a reaction of another component in the environment of the component that sent the request.

Given the general definition of components and systems introduced, integration tests specify how other components expect to interact with the System Under Test (SUT). An integration test thus verifies the interaction expectation of those components by verifying that the corresponding system behavior satisfies those interaction expectations.

> **Interaction Expectation** An interaction expectation IE is a tuple $IE = (T, R)$. With T being a behavior trigger and R being a behavior result expected in response to T.

For unit tests the environment interactions for the Component Under Test (CUT) are mocked. Thus, they implicitly define such interaction expectations without the need to specify them separately. However, the expectation is not that the actual response is equal to the response specified by the mock, but to be a replacement for that mocked response such that the test case is successful.

> **Mock** A mock is a test double that simulates the expected behavior of the environment during testing [ISTQB].

Chapter 3.

Related Work

Table of Contents

3.1. Built-in Testing Approach 21
3.2. Component Metadata for Integration Testing 22
3.3. System Specifications 23
3.4. Test Reuse 24
3.5. Test Coverage on the Integration Level 25
3.6. Summary 26

This chapter provides an overview of existing research on integration testing for CBSSs. In accordance with the DSR approach introduced in Section 1.5, this chapter serves to position the proposed approach within the *knowledge base* context. The knowledge base comprises prior theories, models, and methods that inform the design and development of new artifacts. By reviewing related work, this chapter highlights the benefits and drawbacks of existing approaches and identifies key limitations that are addressed by the IBI Testing Approach proposed in this thesis.

3.1. Built-in Testing Approach

Built-in Testing (BIT) integrates testing capabilities directly into each component. This strategy originated in object-oriented programming and the design philosophy known as *Design by Contract* [JM97], which embeds test logic within component implementations [BG03].

Wang et al. [WKW99] introduced a BIT technique to improve the maintainability of CBSSs. Test logic is implemented as additional methods, and components operate in two modes: normal mode (test methods inactive) and maintenance mode (test methods executed). However, this approach increases the component's size and may introduce unnecessary dependencies or even security risks in the production artifact.

Edwards [Edw01] proposed the use of runtime-attachable wrappers for checking preconditions, postconditions, and invariants, offering more flexibility. Similarly, Atkinson and Gross [AG02] introduced Component+ BIT (C+ BIT), which separates

test cases from component implementations but still embeds BIT capabilities to allow mutual verification of interface contracts.

Momotko and Zalewska [MZ04] extended this idea by supporting runtime testing of contracts and Quality of Service (QoS) properties. Martins et al. [MTY01], building on Beizer's Transaction Flow Model (TFM) [Bei95], developed Concat, a tool that integrates testing resources into components to automatically generate interface tests. However, their approach depends on accurate and up-to-date use-case specifications and requires the consumer to specify test inputs explicitly.

With the emergence of service-oriented and microservice architectures, these ideas evolved into consumer-driven contracts [Rob06], where consumer expectations are communicated to and verified by providers. Tools like Pact [Pac] support such contract definitions in code. However, the approach is limited to pairwise interactions and requires explicit consumer-provider relationships, making it less suitable for verifying multi-component integration in evolving systems [LMM19].

BIT techniques bring testing closer to components but do so at the cost of increased effort and tighter coupling between test logic and production code. As a result, they risk introducing new faults, increase security concerns, and require continuous synchronization with the evolving system. Furthermore, most approaches rely on explicitly defined interface contracts, which must be maintained manually and often become outdated or incomplete.

In contrast, the IBI Testing Approach avoids such intrusion altogether. Instead of modifying components, it passively observes their externally visible behavior during unit test execution. Ideally, interface contracts are not specified manually but are inferred automatically from observed component behavior and test outcomes. Integration across any number of components is simulated by manipulating and composing unit tests, enabling scalable, automated integration testing without altering production artifacts.

3.2. Component Metadata for Integration Testing

Just as consumer-driven contracts are based on explicitly defined specifications, several integration testing approaches rely on artifacts that model, analyze, or document component interactions. These artifacts help define interface contracts, guide test selection, and provide structure for test case generation [ODR+07]. Commonly used forms include UML models, data flow graphs, and execution histories that capture runtime behavior.

Orso et al. [ODR+07] proposed attaching control-flow and data-flow graphs to components. These graphs are analyzed to detect changes affecting consumers and to support test selection. Wu et al. [WCO03] extended this idea by incorporating UML models as component metadata. These models define context-sensitive relationships between components and support integration test selection, especially for multi-component interactions.

Other approaches rely on graphical or behavioral models: Belli and Budnik [BB05] use GUI and state models to generate automated test cases, while Silva et al. [SAM09] propose a CASE tool that defines workflows for producers and consumers to collaboratively generate and execute integration tests. Naseer et al. [NuRH10] adopt reverse engineering to extract structural data from compiled components, helping users select test inputs for black-box testing.

However, these techniques typically depend on manually created or updated models, which are prone to divergence from the implementation over time, as discussed by Mahmood and Khan. [MK11] as well as Nasution and Weistroffer [NW09]. Even when reverse engineering is used to synchronize models, test inputs still need to be manually provided, requiring detailed knowledge about how to set up and integrate all involved components.

In contrast, the IBI Testing Approach avoids these limitations. Rather than solely relying on explicitly defined models or user-provided metadata, interface contracts are automatically derived by observing unit test executions and component behaviors. Furthermore, integration test data does not need to be specified manually. By feeding the output messages from one component into the unit tests of another, the system automatically simulates integration scenarios across components. This eliminates the need for upfront knowledge about how components interact, enabling scalable and fully automated integration testing without maintaining external specifications or crafting dedicated integration test data.

3.3. System Specifications

In addition to approaches where individual components are annotated with metadata, many integration testing strategies are based on system-level specifications. These specifications aim to capture expected interactions or global behavior and are often expressed using formal models such as first-order logic, temporal logic, or various state-based notations. Such models enable precise reasoning and can be used to derive integration tests automatically.

Wu et al. [WPC01] introduced the Component Interaction Graph (CIG) to define test elements and coverage criteria for integration testing. CIGs express dependencies between components and allow the derivation of test cases based on adequacy criteria. Santos et al. [SCM22] proposed an approach that generates sequences of messages to stimulate and verify the system, relying on an up-to-date formal specification of the system's behavior.

Many other works leverage formal methods to model system behavior and drive test generation, including temporal logics such as linear-time logic [MFD06; TSL04], Petri nets [Xu11], and state machines [FL00; SMB08; CK93], as well as other model-based techniques [SMM10; SM07].

While these approaches provide strong formal guarantees, they rely heavily on manually defined models that must be kept consistent with the evolving

implementation. Furthermore, they assume that testers have global knowledge of the system, including which components are involved in fulfilling a particular requirement. In complex or rapidly changing systems, such assumptions can severely limit the applicability and scalability of these methods.

In contrast, the IBI Testing Approach does not rely exclusively on manually defined models. Instead, it infers interaction expectations automatically from observed unit test behavior. Only interaction expectations that cannot be derived from these observations are defined manually. Even these are treated in a black-box fashion: rather than requiring the testers to identify the components involved, the system analyzes observed behaviors and searches for interaction flows that lead from the given input to the expected output. This reduces manual effort, eliminates the need for global system knowledge, and allows the testing approach to scale with evolving systems.

3.4. Test Reuse

As CBSSs emphasize the reuse of components, research has also explored various strategies for reusing test cases to improve testing efficiency and reduce duplication. Test reuse can occur in many forms, ranging from copying and adaptation to approaches that abstract and generalize test behavior.

Basic techniques involve copying and adapting existing test cases, often through parameterization [TS05] or behavior-driven testing, which defines reusable test steps [Nor06]. Gälli et al. [GWN07] introduced the EG-metamodel, which captures example usages of components that can be composed into new test cases. Other works propose methods for recommending reusable tests based on system connections [WKB+21], or matching similar GUI events across applications [MMP+21].

In some cases, reuse extends across abstraction levels. Schätz and Pfaller [SP10] reuse system tests to verify components indirectly, treating the system as a black box. Elbaum et al. [ECD+09] apply differential unit testing to reuse system tests for component comparison. For configurable and product-line systems, Fischer et al. [FRL+19] and others [NdCM+11; ER11; OG09; LLS+12; OZL+11] investigate reuse across configurations using model-based and regression testing techniques.

While these approaches improve efficiency, they often require manual adaptation and remain limited to reusing tests within the same testing level. They rarely support reusing unit-level behavior to derive or verify integration-level expectations. Furthermore, reuse is typically syntactic or structural—focused on test scaffolding rather than the semantic meaning of test interactions.

In contrast, the IBI Testing Approach introduces a novel reuse strategy. It reuses unit tests not only as sources of information but also as building blocks for integration tests. Expectations about component interactions are inferred by analyzing the behaviors exhibited during unit test execution. These same unit tests are then reused

and manipulated to simulate integration flows by chaining their observed inputs and outputs. This represents a semantic and automated form of test reuse, enabling the construction of integration tests without requiring additional integration-specific test logic or data.

3.5. Test Coverage on the Integration Level

Test coverage measures the extent to which a system is exercised by a given test suite. By quantifying coverage, software engineers can identify untested areas, guide test selection, and increase confidence in the system's reliability, as discussed by Zhu et al. [ZHM97], Hemmati [Hem15], and Ivankovi et al. [IPJ+19]. On the unit level, white-box criteria such as statement and branch coverage are commonly used, alongside black-box criteria such as functional and requirements coverage [ZHM97; WRH+06].

For integration testing, specialized coverage criteria have been proposed to reflect the interaction-centric nature of component-based systems. These include:

1. **Interface Coverage**: Ensures that all interfaces exposed by components are exercised [DMM01; SOP07].

2. **Interaction Coverage**: Ensures that all possible component interactions are tested [IPJ+19].

3. **Data Flow Coverage**: Focuses on the propagation of data between components, ensuring all usage paths are exercised [HSW19].

While these coverage models provide valuable metrics, they are usually evaluated independently of the expectations that define correct system behavior.

The IBI Testing Approach incorporates coverage considerations implicitly, through the verification of interaction expectations. If an interaction expectation cannot be verified, it indicates that some required interaction, interface usage, or data flow was not exercised by the available unit tests. Because integration tests are synthesized by chaining observed unit test behavior, untestable interaction expectations reveal that essential parts of the expected integration flow are not covered.

Although this approach cannot identify precisely which part of the flow is missing, the inability to construct a verifying interaction flow serves as a practical indicator of insufficient coverage. This helps reveal testing blind spots. In this way, coverage gaps are surfaced as part of the regular IBI Testing Approach, aligning test adequacy with system behavior rather than abstract structural metrics.

3.6. Summary

Software testing is essential for ensuring the reliability of software systems, as software faults remain a primary cause of failures in dependable systems [ALR+04]. With the rise of CBSSs, integration testing has grown increasingly complex due to the intricate interactions between components. One challenge is fault propagation, which complicates root-cause analysis and makes debugging more difficult [SCK04; PHZ24]. This added complexity often results in incomplete or delayed verification.

A further challenge stems from the reuse of components across different systems or contexts. A component that behaves correctly in one context may exhibit unexpected behavior in another, making it difficult to anticipate all interaction scenarios through manual testing alone [Wey98; MDB+07]. This leads to redundancy in test efforts and difficulty in isolating faults during system integration.

Consequently, integration testing of CBSSs has emerged as a critical area of research for many years, with numerous studies exploring ways to address these challenges [BG03; KG11; WLK20; WLM+20; LGM+24]. Existing solutions attempt to mitigate these issues using embedded tests, formal specifications, metadata, and test reuse techniques. However, these approaches often involve significant manual effort, require static specifications that become outdated, or fail to adapt to dynamic component interactions in distributed systems.

Moreover, most techniques require a dedicated integration environment, increasing the maintenance burden and limiting the scalability of automated testing. Traceability between unit tests and integration-level behaviors is rarely leveraged, leading to missed opportunities for automating test derivation and coverage validation.

This chapter reviewed a broad range of approaches to integration testing in CBSS, including BIT, contract-driven verification, model-based techniques, and test reuse strategies. While each of these approaches addresses important challenges, they also share common limitations.

Most existing techniques depend heavily on manually written artifacts, such as interface contracts, test inputs, or system-level specifications. These artifacts must be maintained alongside the implementation, making them prone to inconsistency, especially in dynamic or evolving systems. Furthermore, test reuse is often limited to structural adaptations or focused on single components, with little support for deriving or verifying multi-component behavior. Even coverage models designed for integration rarely align directly with functional expectations, requiring separate effort to ensure that tested interactions actually fulfill requirements.

In contrast, the IBI Testing Approach proposed in this thesis addresses these limitations by shifting the focus from manually constructed test environments and specifications to the systematic reuse of observable unit test behavior. Instead of relying on static models or embedded test logic, it captures component behavior during unit test execution to automatically construct component models. Integration is simulated by composing these behaviors based on protocol-aware interface

bindings, and system-level verification is driven by interaction expectations that can either be defined explicitly or derived from unit test observations.

Chapter 4.

Running Example

Table of Contents

4.1. System Architecture 29
4.2. System Characteristics 31

The *Piggy Bank system*[1] serves as the running example throughout this thesis. The system is used to illustrate key concepts and to demonstrate the capabilities of the proposed integration testing approach. The system supports typical personal finance tasks such as tracking income and expenses across multiple accounts, providing financial insights, defining savings goals, and sending notifications about relevant changes or achievements. The Piggy Bank system is implemented as a microservice architecture in Kotlin, with each component realized as an independently deployable Spring Boot application. The components communicate via a mix of REST and AMQP protocols, reflecting the heterogeneous communication patterns typical of modern distributed systems.

4.1. System Architecture

An overview of the architecture is shown in Figure 4.1. The Piggy Bank system consists of five components:

Transfer Gateway (TG): The Transfer Gateway receives all transfers via the *Transfer* REST interface from a simulated third-party service, representing bank APIs or other services like PayPal. It keeps track of accounts marked for monitoring, via the *Watched Account* REST interface, and splits each transfer into corresponding credit and debit transactions for the affected accounts. These transactions are then forwarded to the Account Twin Service via the *Transaction* REST interface. The transfers that affected a monitored account are also published via an AMQP exchange, represented by the *Transfer Info* interface, to be consumed by other services.

1. All artifacts of the Piggy Bank system are publicly available at `https://github.com/NilsWild/Piggy-Bank`

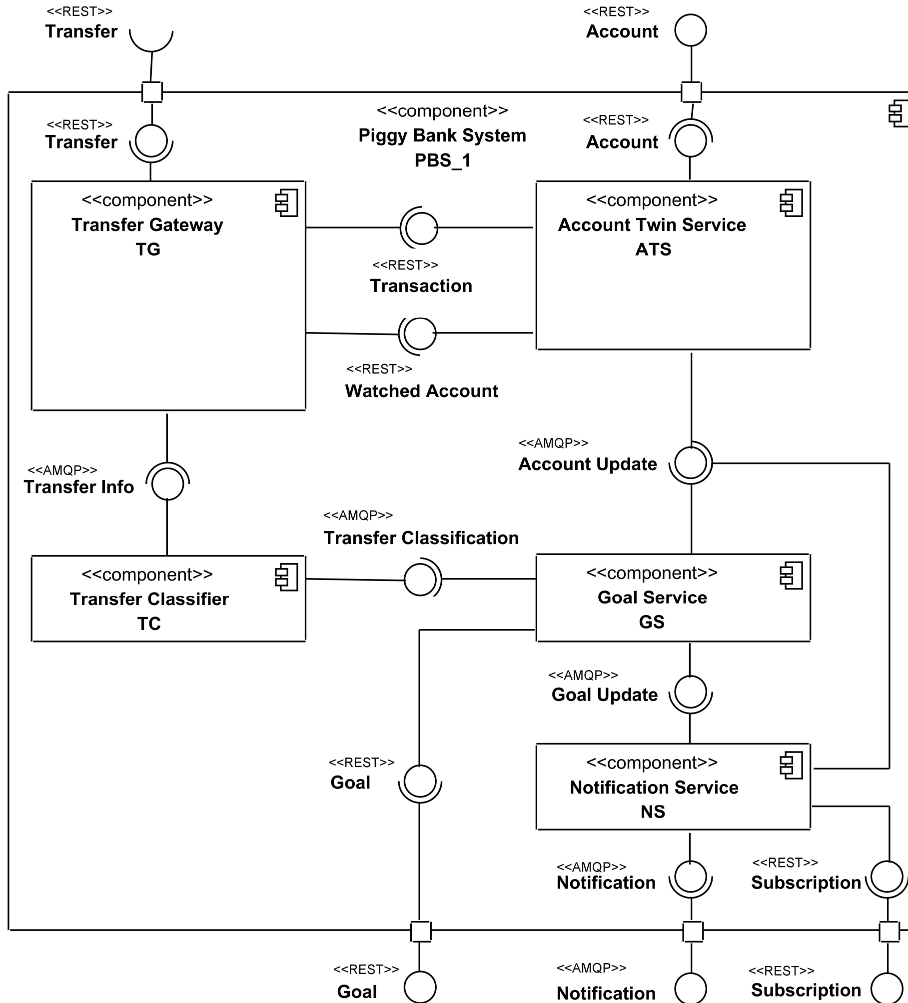


Figure 4.1.: Architecture of the Piggy Bank system.

Account Twin Service (ATS): The Account Twin Service maintains the current balance for all monitored accounts. New accounts can be added and deleted for monitoring via the *Account* REST interface. When an account is added or deleted the Transfer Gateway is informed via the *Watched Account* REST interface. Upon receiving a transaction from the Transfer Gateway via the *Transaction* REST interface, the Account Twin Service updates the corresponding account balance and emits an account update event via an AMQP exchange represented by the *Account Update* interface.

Powered By Visual Paradigm Community Edition

Transfer Classifier (TC): The Transfer Classifier classifies each transfer into predefined classes such as living expenses, holidays, or investments. The Transfer Classifier therefore uses an AMQP queue to subscribe to the *Transfer Info* exchange of the Transfer Gateway. The classifications are emitted to an AMQP exchange, represented by the *Transfer Classification* interface. The classifications enable downstream services to reason about the nature of transfers and transactions. In case of the Piggy Bank System the only downstream service is the Goal Service.

Goal Service (GS): The Goal Service allows users to define financial goals (e.g., saving a specific amount, lowering living expenses) using the *Goal* REST interface. It subscribes to account update events from the Account Twin Service, using an AMQP queue to subscribe to the *Account Update* exchange, and transfer classifications from the Transfer Classifier, using an AMQP queue to subscribe to the *Transfer Classification* exchange, to monitor goal progress and detect goal achievements. The Goal Service emits goal updates via an AMQP exchange, represented by the *Goal Update* interface.

Notification Service (NS): The Notification Service manages user subscriptions to different topics (e.g., account updates, goal achievements). The Notification Service subscribes to account updates from the Account Twin Service, using an AMQP queue to subscribe to the *Account Update* exchange, and goal updates from the Goal Service, using an AMQP queue to subscribe to the *Goal Update* exchange, and sends notifications via an AMQP exchange, represented by the *Notification* interface accordingly. The subscriptions are managed via the *Subscription* REST interface.

4.2. System Characteristics

Together, these components exhibit several characteristics typical for distributed service-based CBSSs:

1. **Heterogeneous communication protocols:** REST is used for synchronous interactions, while AMQP supports asynchronous event-driven messaging.
2. **Feature realization through collaboration:** User-facing features, such as notifications or goal tracking, are realized through the cooperation of multiple components.
3. **Request-response patterns:** Certain interactions (e.g., transfers from third-party services) follow synchronous request-response interactions over REST.
4. **Unidirectional messaging:** Events (e.g., goal updates) are emitted without awaiting acknowledgments, reflecting typical asynchronous integration patterns.

Part II.

The Interaction-based Integration Testing Approach

Chapter 5.

The IBI Metamodel

Table of Contents

5.1. Assumptions and Modeling Decisions 35
5.2. Structure of the Interaction-based Integration Metamodel 36
5.3. Component-based System Metamodel 38
5.4. Test Metamodel 42
5.5. Interaction Expectations Metamodel 44
5.6. Interaction Expectation Verification Metamodel 47
5.7. Summary 48

The integration of CBSSs requires a structured approach to effectively manage the complexity associated with interactions and dependencies among components. This chapter introduces the IBI Metamodel, which forms the foundation for the IBI Testing Approach. The IBI Metamodel defines all necessary elements and relationships to capture the components and interaction expectations imposed on a CBSS, thereby supporting systematic verification of component interactions.

To provide a comprehensive understanding, this chapter is structured as follows. Section 5.1 introduces the fundamental assumptions that guided the modeling process and outlines the modelling decisions. Section 5.2 gives an overview of the four submodels that make up the IBI Metamodel. The remaining sections describe each submodel in detail. The IBI Metamodel ultimately supports both the structural and behavioral analysis of CBSS, thereby addressing challenges C1 (Collaboration) and C2 (Test Case Selection).

5.1. Assumptions and Modeling Decisions

The IBI Metamodel is based on the core assumption that every CBSS can be abstracted as a message-based system. Regardless of the concrete communication mechanisms used, component interactions are uniformly represented as the exchange of messages through well-defined interfaces between components.

This abstraction enables a consistent and technology-agnostic representation of integration behavior, allowing diverse communication styles to be modeled under a common structure. As Gorton notes, architectural descriptions must suppress

unnecessary details to remain understandable and analyzable. This is typically achieved by treating components as black boxes and specifying only their externally visible properties [Gor11]. The IBI Metamodel follows the same principle to focus attention on the structural and behavioral aspects of components relevant for integration testing of CBSSs, while abstracting away implementation details that do not affect the interaction behavior.

As each metamodel, the IBI Metamodel does not describe a particular system but provides the schema for creating concrete IBI models, which reflect the structure and behavior of specific SUTs.

To support integration testing effectively, the IBI Metamodel is guided by three core requirements:

- **Synchrony with Implementation:** IBI models must reflect the current implementation of components and their composition into systems.
- **Testability:** IBI models must include information on both how components are tested and how they interact.
- **Traceability of Interactions:** It must be possible to trace interaction flows between components based on observed unit test cases.

To fulfill these requirements, unit test cases serve as the primary data source. They contain not only test inputs and expected outputs but also encode interaction patterns with the component's environment, typically represented by mocks. As such, the unit test cases provide the structural and behavioral information needed to create IBI models.

5.2. Structure of the IBI Metamodel

The structure of the IBI Metamodel is directly informed by the foundational elements of CBSSs introduced in Chapter 2. Specifically, the IBI Metamodel must be capable of representing components, interfaces, messages, states, behaviors, and their relationships. Additionally, it must reflect the test-related concepts such as test cases and interaction expectations.

The IBI Metamodel is composed of four interrelated submodels, each addressing a specific aspect of integration testing:

Component-based System Metamodel: It models the structure of the system, including components, their versions, interfaces, and associated properties.

Test Metamodel: It models how individual components have been tested at the unit level, including test cases and observed interactions.

Interaction Expectations Metamodel: It represents expectations regarding the interaction of one component with a system, independent of any specific implementation.

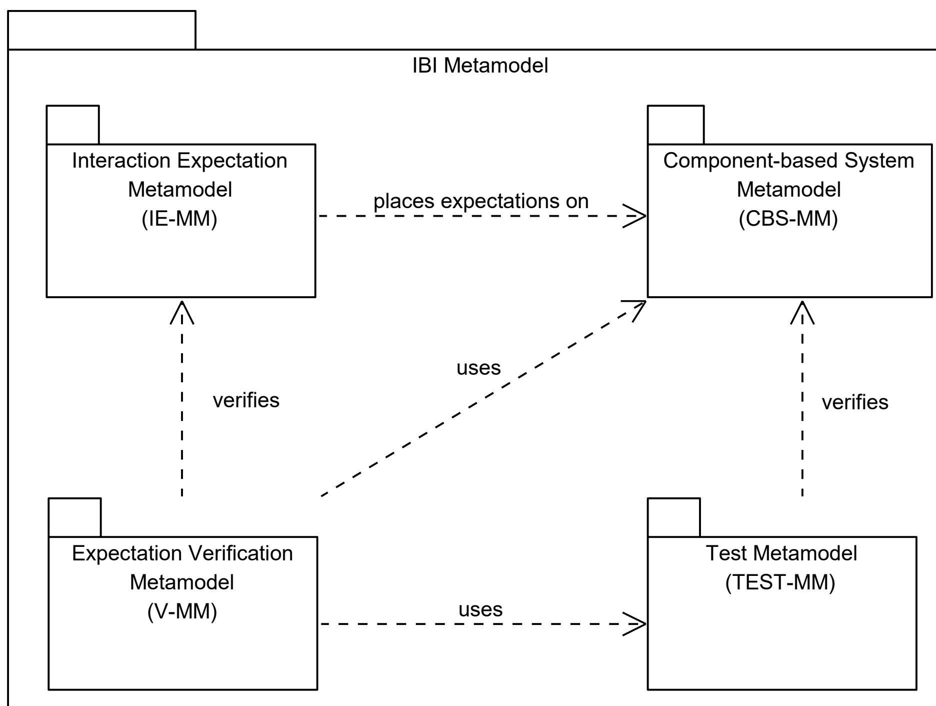


Figure 5.1.: Structure of the IBI Metamodel

Interaction Expectation Verification Metamodel: It models the relationship between interaction expectations and system behaviors that satisfy them.

These submodels offer all information needed to enable integration test generation, traceability of system behavior, and verification of interaction expectations.

In the following, all element and relationship names are printed in *italics* when referring to elements and associations in the model. In the figures accompanying each submodel, elements with a gray background are not part of the current submodel but originate from one of the other submodels. These gray-highlighted elements are included to illustrate relationships between the current submodel and elements defined in other submodels. The source submodel of each gray-highlighted element is indicated in parentheses beneath its name.

5.3. Component-based System Metamodel

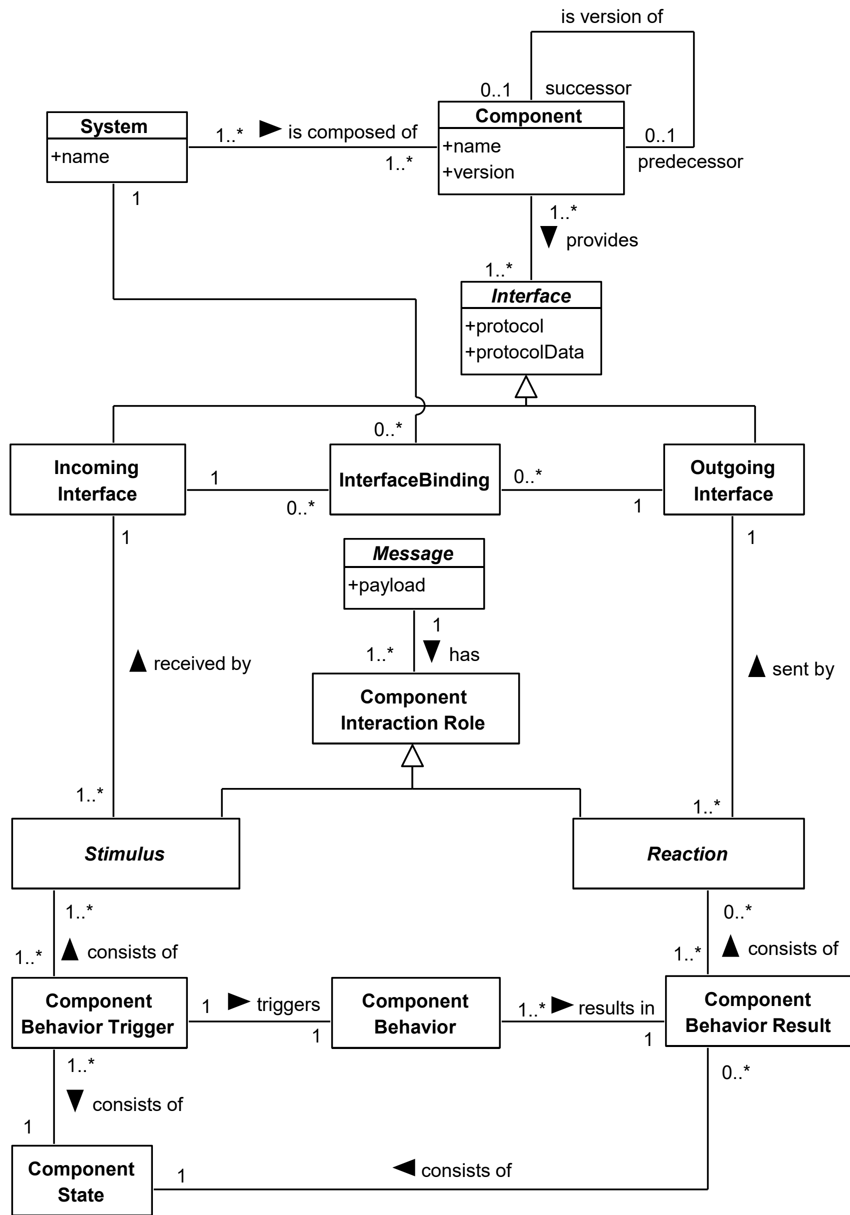


Figure 5.2.: The Component-based System Metamodel.

The Component-based System Metamodel (CBS-MM), depicted in Figure 5.2, forms the foundation of the IBI Metamodel. It represents CBSSs by capturing their components, the interfaces they provide, their relationships with other components, and their evolution across versions. This version-aware structure supports traceability of changes, such as added interfaces, modified interaction behavior, or replaced dependencies, by linking components to their respective versions and documenting. As a result, specific component versions can be analyzed to determine how changes may affect integration behavior, which is essential for targeted regression testing.

5.3.1. Component Structure

Each component has a purpose and provides features to fulfill that purpose. These features can be called by other components via interfaces. Likewise, a component can utilize the features of other components via such interfaces. This section details the elements that represent the structural properties of components relevant for integration testing.

Component

Each component is represented by a *Component* element. A *Component* is identified by its name and version. Each component can evolve over time, thus each *Component* is associated with *previous* and *subsequent* versions linked to capture the development trajectory.

In the following the term "component" is also used to refer to the set of all components represented by *Components* that share a common name, regardless of a specific version.

Interface

Interfaces are the access points, through which components interact with one another. An interface is represented by an *Interface* element that is characterized by a protocol (e.g., REST, SOAP) and associated protocol-specific data, such as URLs. An *Interface* is associated with the *Component* that *provides* it, supporting regression analysis and version comparison. *Interfaces* are categorized as either *Incoming Interface* or *Outgoing Interface* to clearly distinguish the direction of communication.

- ***Incoming Interface:*** An *Incoming Interface* defines an entry point for external inputs, specifying how a component can be stimulated by other components. *Incoming Interfaces* are essential for mapping out how messages enter components and how its features can be accessed.
- ***Outgoing Interface:*** An *Outgoing Interface* defines how a component publishes its outputs. *Outgoing Interfaces* represent the interfaces used by a component

to interact with its environment, allowing it to relay results or utilize features of other components.

5.3.2. Component Dynamics

The features that are accessible via the interfaces of a component are realized by component behaviors that are triggered by messages sent to a component. This section details the modeling of these dynamic aspects including messages, their roles in the interaction with a component and the thereby triggered component behaviors.

Message

A *Message* represents data sent to or received by a component through an interface. Each *Message* carries a payload representing the data content.

Component Interaction Roles

Messages can take two different roles in interactions between components:

- *Stimulus:* A *Message* takes on the *Component Interaction Role* of a *Stimulus* when it is *received by* an *Incoming Interface*. From the perspective of the component providing that interface, the *Message* represents an external input and is therefore referred to as a *Stimulus Message* or *Stimulus* for short.
- *Reaction:* A *Message* takes on the *Component Interaction Role* of a *Reaction* when it is *sent by* an *Outgoing Interface*. From the perspective of the component providing that interface, the *Message* represents an output that is perceivable by components in the component's environment and is therefore referred to as a *Reaction Message* or *Reaction* for short.

During integration, a stimulus received by a component is the reaction emitted by another component. Therefore, a single message may have different roles depending on the sender and receiver perspective. Thus, a *Message* may have multiple *Component Interaction Roles*.

Component State

A *Component State* represents the state of a component.

Component Behavior Trigger

A *Component Behavior Trigger consists of* one *Stimulus* or multiple *Stimuli* and a *Component State*.

Component Behavior

A *Component Behavior* is *triggered by* a *Component Behavior Trigger*.

Component Behavior Result

A *Component Behavior Result* is the result of a *Component Behavior* and consists of *Reactions* and an updated *Component State*.

5.3.3. Composition of Components

One of the central concepts of component-based development is the reuse of components by composing them into different systems. Rather than considering all potential component combinations for the integration test, it is more efficient to focus on predefined system configurations that are relevant to the business. Thus, the *System* element is introduced to the IBI Metamodel to select component compositions to avoid irrelevant integration tests.

System

A *System* represents a named composition and configuration of *Components*.

Interface Binding

In order for the components of a system to be integration tested, it is necessary to know how messages are transferred between outgoing and incoming interfaces of different components. This relationship is represented by *Interface Bindings*. An *Interface Binding* represents the relationship between *Outgoing* and *Incoming Interfaces* of *Components* in a *System*. An *Outgoing Interface* is *bound to* an *Incoming Interface* if their definitions align according to protocol requirements and system configuration. Therefore, different *Systems* can specify different *Interface Bindings*.

5.4. Test Metamodel

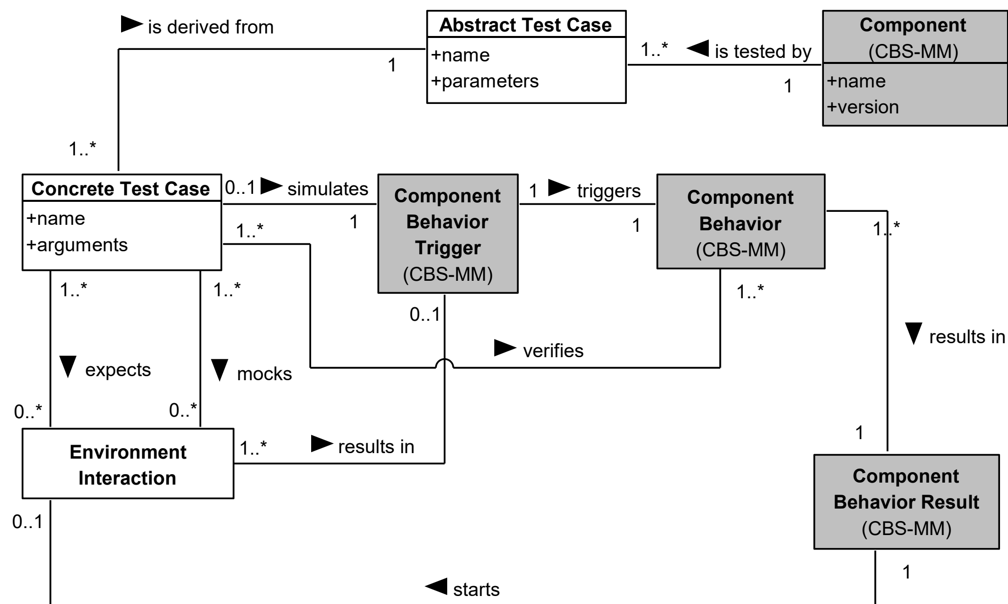


Figure 5.3.: The Test Metamodel.

While the Component-based System Metamodel specifies how components interact with each other and how messages are processed by a component, it lacks information on how these behaviors can be verified. Unit tests provide this information. They define test cases to verify specific component behaviors. During their execution, a test stimulus is sent to the CUT to verify the component's internal behavior as well as its interactions with its environment.

Concrete test cases use concrete arguments, thus verifying specific behaviors and interactions. In contrast, abstract test cases focus on the underlying behavior and interaction patterns. Typically, concrete test cases are derived from abstract test cases. This is also reflected by the Test Metamodel (Test-MM).

Abstract Test Case

An *Abstract Test Case* has a name, a list of parameters and *tests* a particular *Component*. *Abstract Test Cases* that share the same name and parameters but are associated with different *Components* might not be equal.

Concrete Test Case

A *Concrete Test Case* is *derived from* an *Abstract Test Case* by providing specific arguments for each parameter defined in the *Abstract Test Case*, making them executable. They *simulate* the conditions defined by a *Component Behavior Trigger* under which the component is expected to operate in a certain way.

Environment Interaction

An *Environment Interaction* represents the interaction of a component with its environment that is *expected* by the *Concrete Test Case* in response to the *triggered Component Behavior*. The *Environment Interaction* is *started* by the *Component Behavior Result* that the *Component Behavior results in*. They provide a means to understand what assumptions a component has towards its interaction with its environment.

5.5. Interaction Expectations Metamodel

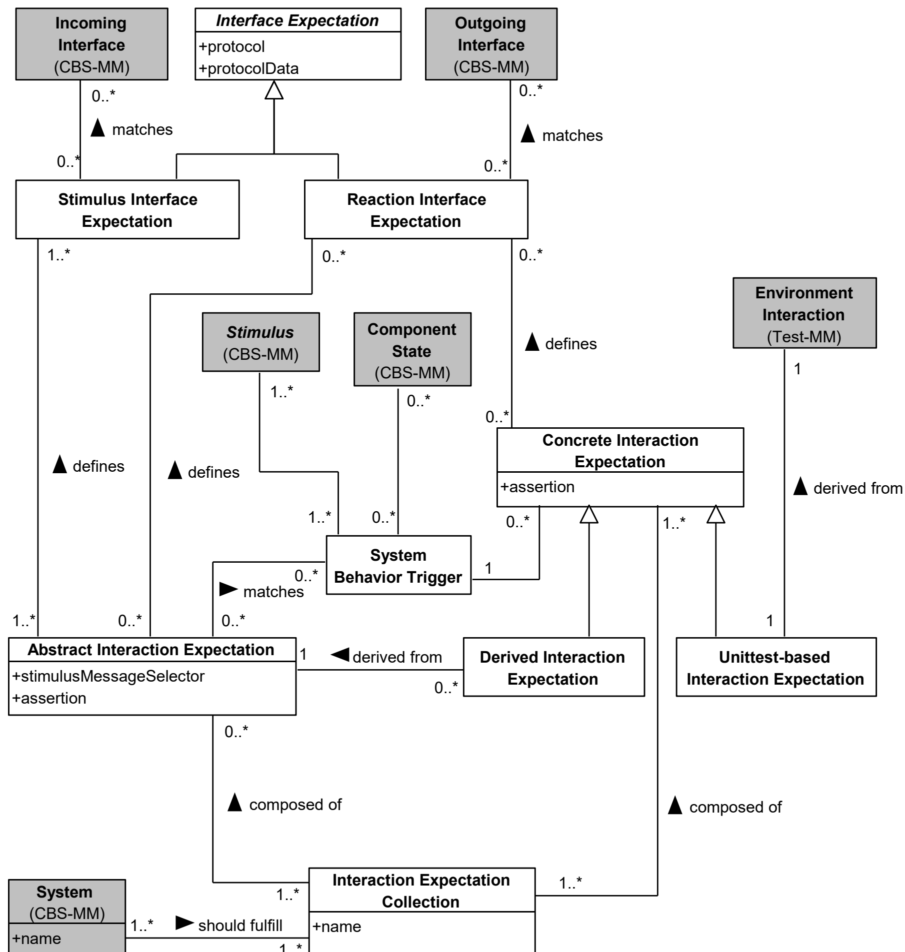


Figure 5.4.: The Interaction Expectation Metamodel.

The interaction expectations metamodel (IE-MM), depicted in Figure 5.4, serves as a crucial element of the IBI Metamodel, addressing the need to explicitly define and manage the expectations regarding the interactions with a system. The interaction expectations are the foundation for integration testing and serve as test oracles. As CBSSs evolve, their integration tests must adapt to changes in architecture and component behavior. The IBI Metamodel aims to facilitate this adaptability. Every system interaction is started by a *System Behavior Trigger*.

System Behavior Trigger

A *System Behavior Trigger* is defined by a set of *Stimuli* and the *Component States* of the components of the system, effectively defining the system state and system stimulus.

Abstract Interaction Expectation

An abstract interaction expectation is a reusable template for interaction expectations, which can be instantiated with different arguments. The concept of abstract interaction expectations is closely aligned with the concept of abstract test cases. Just as abstract test cases define the structure of a test without concrete test data, abstract interaction expectations describe the interaction independently of concrete inputs, enabling reuse and generalization across different system configurations. Abstract interaction expectations therefore provide a bridge between high-level design intentions and concrete interaction expectations. An *Abstract Interaction Expectation* element consists of four elements:

1. *Stimulus Interface Expectations:* Each *Stimulus Interface Expectation* specifies how a component intends to send a message to the system.

2. *Stimulus Message Selector:* The *Stimulus Message Selector* determines if a *System Behavior Trigger* is valid regarding the abstract interaction expectation. *System Behavior Triggers* are inputs to a system and therefore can be partitioned into different equivalence classes that lead to different output equivalence classes. The *Stimulus Message Selector* is responsible to specify for which input equivalence class the expectation towards the outputs should hold.

3. *Reaction Interface Expectations:* A *Reaction Interface Expectation* defined by an *Abstract Interaction Expectation* specifies where a *Message* is expected to be *received* in response to the *System Behavior Trigger*. Each *Reaction Interface Expectation* specifies an *Incoming Interface* that might not exist in the system but is used by a component that expects to receive a response to the *System Behavior Trigger*. As long as an *Outgoing Interface* of a component in the system aligns with this definition the integration can be tested. This makes the IBI Metamodel robust to changes in the system. If the *Outgoing Interface* of the component that sends the reaction is changed, or the component as a whole, the component that relies on the feature that is provided through that interface, might still integrate successfully if another interface or component takes over.

4. *Assertion:* An *Assertion* specifies the expectations towards the payload of the reaction messages in relation to the stimulus message.

Concrete Interaction Expectation

To define interaction expectations toward a system interaction, two elements are required: the trigger that initiates the behavior and the expected reaction to that trigger. These define the observable input and output conditions under which the system behavior is considered correct.

Unlike approaches that prescribe fixed communication sequences and involved components, the IBI Metamodel treats the system as a black box. It avoids specifying internal message flows or which components are responsible for producing or consuming specific messages. Instead, it focuses on observable behavior from the perspective of a component interacting with the system. This makes the interaction expectations more robust to changes in internal system structure or component responsibilities.

A *Concrete Interaction Expectation* consists of three elements:

1. *System Behavior Trigger*
2. *Reaction Interface Expectations:* This corresponds to the *Reaction Interface Expectations* of the *Abstract Interaction Expectation*.
3. *Assertion:* An *Assertion* that specifies the expectations towards the reactions. It defines the criteria to assess the correctness of the interaction. It provides the actual test oracle for integration tests that are based on the *Interaction Expectation*.

Two types of *Concrete Interaction Expectations* exist: *Unit test-based Interaction Expectations* and *Derived Interaction Expectations*.

Unit Test-based Interaction Expectation

A *Unit test-based Interaction Expectation* is directly associated with an *Environment Interaction*, present in the unit tests. Its assertion is given by: "The *Reactions* must *trigger Component Behaviors* that result in *Reactions*, such that these *Reactions* can replace the *Stimuli* of the *Component Behavior Trigger* that *results from* the *Environment Interaction* with the test still being successful."

Derived Interaction Expectation

A *Derived Interaction Expectation* is defined systematically by concretizing the *Abstract Interaction Expectation* it is *derived from* with specific data. The trigger of these *Derived Interaction Expectations* is determined by a *Component Behavior Trigger* whose *Stimulus* is directed at a *Component's Incoming Interface* that aligns with the *Stimulus Interface Expectation* of the *Abstract Interaction Expectation* it is *derived from* and a set of expected *Component States* making up the *System Behavior Trigger*.

Interaction Expectation Collection

The expectations imposed on a system depend on the composition of its components. Therefore, an *Interaction Expectation Collection* is used as an abstraction that groups together sets of *Abstract* and *Concrete Interaction Expectations*. The *Interaction Expectation Collections* can be formed using different criteria, such as feature sets or other characteristics, supporting their reuse across different system configurations. Thus, each *Interaction Expectation Collection* might be associated with multiple *Systems*.

5.6. Interaction Expectation Verification Metamodel

Given the interaction expectations, a means to determine if those expectations can be verified is needed. The Interaction Expectation Verification Metamodel (V-MM) links each concrete interaction expectation with the component behaviors that realize the system behavior that satisfies the interaction expectation.

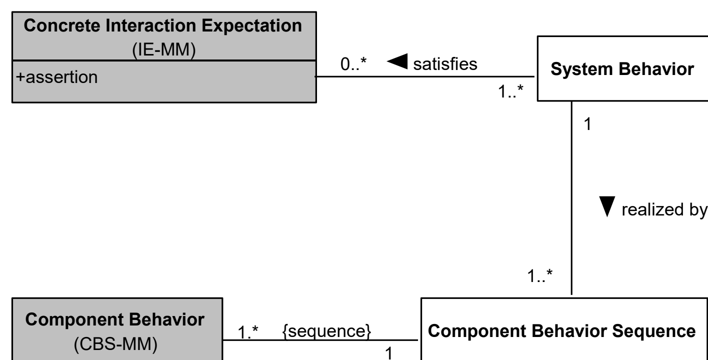


Figure 5.5.: The Interaction Expectation Verification Metamodel.

In principle, any system behavior can be represented by sequences of component behaviors that are triggered one after the other. The system behavior trigger of the interaction expectation leads to a component behavior being triggered resulting in some reaction message that becomes the stimulus of a component behavior trigger of another component and so on and so forth. If all those component behaviors have been verified by a unit test the interaction expectation is verified implicitly as well. This requires the unit test suites of different components to be compatible in the sense that each reaction message of one component is used as a stimulus message in the test case of those components that would receive that message.

In practice, one reaction message might become the stimulus of multiple component behaviors that are triggered in parallel, thus the representation as a sequence might seem inappropriate. However, there should be $n!$ verifying component behav-

ior sequences if n is the number of parallel processed messages by the component, treating each permutation as a separate instance of a system behavior, to ensure that the order of processing does not matter. When multiple interactions are triggered in sequence, the number of combinations grows exponentially. It can be argued that this can generally be disregarded when testing the functional integration of a system. Parallel processing of messages can only lead to errors if race conditions occur within a component. Uncovering this is not the aim of an integration test.

The Interaction Expectation Verification Metamodel, depicted in Figure 5.5, thus introduces only two elements: The *System Behavior* and *Component Behavior Sequence*.

System Behavior

A *System Behavior* is *realized by* one or multiple of a *Component Behavior Sequences* and *satisfies* one or multiple *Concrete Interaction Expectations*.

Component Behavior Sequence

A *Component Behavior Sequence* is a sequence of *Component Behaviors* that trigger each other.

5.7. Summary

The IBI Metamodel defines a representation of component behaviors and their relationships, which is essential for verifying the integration of CBSSs. It links unit test cases to the behaviors they verify, enabling traceability and ensuring that the model reflects the current implementation.

Interaction expectations capture assumptions made about a system's external behavior and allows reuse across different system configurations. Grouping them into collections enables logical grouping into feature sets. Instead of exhaustively testing all combinations, the system element enables precise specification of which components it is composed of, how they interact with each other, and what expectations apply.

As a result, the IBI Metamodel forms the foundation for identifying integration faults. Any mismatch between expected and actual component behavior may indicate architectural misalignment, incorrect assumptions, or faulty implementations. The next chapter shows how this foundation is used to define a process to detect such issues.

Chapter 6.

The IBI Testing Process

Table of Contents

6.1. The Interaction-based Integration Testing Process at a Glance 49
6.2. Formalizing Interaction-based Integration Models 52
6.3. The Interaction-based Integration Testing Process in Detail 59

The presented IBI Metamodel provides a concise representation of the SUT, the interaction expectations imposed on it, the components it is composed of, and the test cases that verify each component. This metamodel serves as the foundation for the integration testing approach introduced in this chapter.

To verify whether interaction expectations are satisfied, the Interaction Expectation Verification Metamodel requires that the reaction messages produced by one component must correspond to the stimulus messages received by others. While this condition enables implicit verification of system-level behavior based on individual unit tests, achieving and maintaining such compatibility of unit test suites manually is infeasible in industry-scale projects.

This chapter introduces the IBI Testing Process, which builds on the IBI Metamodel to systematically verify CBSS. First an overview of the IBI Testing Process is given followed by a formalization based on property graphs to represent IBI models, that are used to derive information to test the SUT. Property graphs are queried using Regular Path Query (RPQ) to identify dependencies and analyze component interactions, to drive the integration process. After introducing the formal foundations, the IBI Testing Process is explained step by step. To illustrate the discussed concepts, the chapter concludes with an example based on the Piggy Bank Project (see Chapter 4), demonstrating the application of the proposed process.

6.1. The IBI Testing Process at a Glance

The IBI Testing Process defines all actions necessary to verify the system behavior by individually verifying each component interaction in the context of the system behavior. A simplified version is visualized as a UML activity diagram in Figure 6.1, excluding input and output data. At its core, the IBI Testing Process leverages IBI models to systematically derive testable system behaviors and verify them by

iteratively generating so-called interaction test cases, that are effectively unit test cases with the special property that their inputs and outputs are coordinated with each other. Thereby simulating the integration.

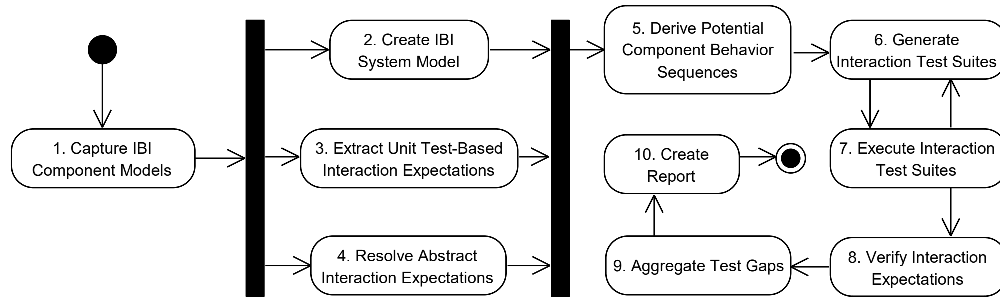


Figure 6.1.: The simplified IBI Testing Process.

The inputs to the IBI Testing Process are:

- The Unit Test Suites (UTSs) of all components the SUT consists of
- The interface bindings
- The abstract interaction expectations, that specify functional requirements imposed on the SUT

The IBI Testing Process consist of 10 actions:

1. **Capture IBI Component Models:** An IBI component model is created for each component based on its UTS. Each IBI component model contains the component, its interfaces, as well as the test cases along with the interactions, and messages that are observed during their execution.

2. **Create IBI System Model:** The IBI component models are combined into an IBI system model, forming an integrated representation of the SUT by considering the interface bindings.

3. **Extract Unit Test-Based Interaction Expectations:** To reuse as much information from the UTSs as possible, the expectations a component has towards its interactions with its environment are extracted from its interaction with mocks during unit testing.

4. **Resolve Abstract Interaction Expectations:** The abstract interaction expectations do not define concrete test data but stimulus message selectors, to select messages for which the expectation should hold. Therefore, the IBI component models are searched for appropriate messages to resolve abstract interaction expectations to derived interaction expectations.

5. **Derive Potential Component Behavior Sequences:** The IBI system model and interaction expectations are analyzed to derive potential component behavior sequences that might verify the interaction expectations. These component behavior sequences represent potential system behaviors that need to be tested to verify if they represent actual system behavior.
6. **Generate Interaction Test Suite:** The derived component behavior sequences are iteratively verified by generating Interaction Test Suites (ITSs) containing interaction tests. Each interaction test focuses on a specific interaction within a chain of interactions representing a system behavior, ensuring that expectations are met at each step. The component reactions observed during the execution of each interaction test determine the inputs of the subsequent interaction tests.
7. **Execute Interaction Test Suite:** The generated ITSs are executed. In some cases the same interaction test might be part of different ITSs. In that case they only need to be verified once. The component reactions observed during the execution of each interaction test are used for the inputs of the subsequent interaction tests being generated.
8. **Verify Interaction Expectations:** The interaction test results are analyzed to determine if the imposed interaction expectations could be verified or not.
9. **Aggregate Test Gaps:** In the previous steps, test gaps might be identified. E.g. if no potential component behavior sequence for an interaction expectation could be found it might indicate a test or implementation gap. These gaps are aggregated.
10. **Create Report:** Finally, the integration test results derived from the interaction test and test gaps are aggregated into a concise report.

The IBI Testing Process reuses the UTSs in multiple ways by extracting the expectations, component behavior sequences and test data used for integration testing, ensuring that the integration tests are always up-to-date with component development. Furthermore, the iteratively generated interaction test cases require no special integration environment. Last but not least it provides insight into potential integration issues on a per-interaction basis and indicates areas for improvement in the UTSs, facilitating early testing. It thereby addresses the challenges presented in the introduction of this work.

6.2. Formalizing IBI Models

The IBI Testing Process creates and utilizes IBI models. These models are analyzed to extract relevant information, such as interaction expectations, potential system behaviors and test gaps. To precisely describe the analysis of IBI models, a formal representation that can be queried is introduced.

Property graphs are a well-established formalism for representing structured data [Ang18] and are a natural fit to represent CBSS, where components interact with each other, forming interconnected relationships. In such property graphs, the IBI model elements are represented as nodes with associated properties and the associations between them as edges. As property graphs belong to the broader class of edge-labeled graphs, they are well suited for querying with RPQs. RPQs allow the traversal of graphs by defining patterns of relationships between nodes [CMW87; Var16]. This section formalizes the IBI Metamodel based on the property graph concept and thus also all IBI models, introduces the concept of RPQs, and demonstrates how they enable querying of IBI models. This formalization is the foundation for the detailed explanation of the IBI Testing Process in the next section.

6.2.1. Property Graphs

A property graph PG is a directed labeled graph where each node maintains a set of property-value pairs. Given finite sets of node labels L_N, edge labels L_E, property keys K, and an infinite set of property values V, a property graph PG over (L_N, L_E, K, V) is a structure $(N, E, edge, \lambda_N, \lambda_E, prop)$, where

- N is a finite set of nodes,
- E is a finite set of edges,
- $edge : E \rightarrow N \times N$ is a total function that associates a pair of nodes to each edge,
- $\lambda_N : N \rightarrow L_N$ is a total function that associates each node with a label from L_N,
- $\lambda_E : E \rightarrow L_E$ is a total function that associates each edge with a label from L_E, and
- $prop : N \times K \rightarrow V$ is a partial function that associates each node with a value for selected property keys.

A path in PG is defined as a sequence of nodes and directed edges, where edges can be traversed in both directions. A path is formally represented as:

$$p = \nu_0(e_1)^{\sigma_1}\nu_1 \dots (e_{len})^{\sigma_{len}}\nu_{len}$$

where:

- $len \geq 1$ is the length of the path,
- $\nu_i \in N$ are nodes,
- $e_i \in E$ are edges,
- $\sigma_i \in \{+, -\}$ determines the traversal direction of an edge e_i:

$$\sigma_i = + \quad \text{if } edge(e_i) = (\nu_{i-1}, \nu_i),$$
$$\sigma_i = - \quad \text{if } edge(e_i) = (\nu_i, \nu_{i-1}).$$

The sequence of edge labels for a path p is denoted as:

$$edges(p) = \lambda_E(e_1)^{\sigma_1} \ldots \lambda_E(e_{len})^{\sigma_{len}}$$

For notational simplicity, original edge labels are written without '$+$', while inverse edge labels are explicitly marked:

$$\lambda_E(e_i) \quad \coloneqq \lambda_E(e_i)^+ \quad \text{(original label)}$$
$$\lambda_E(e_i)^- \coloneqq \lambda_E(e_i)^- \quad \text{(inverse label)}$$

The set of all paths in a property graph PG is denoted as:

$$Paths(PG).$$

6.2.2. Regular Path Queries

RPQs allow retrieving pairs of nodes that are connected by a path whose edge labels conform to a given regular expression. A RPQ is an expression of the form:

$$(\nu_i, regex, \nu_j)$$

where:

- $\nu_i, \nu_j \in N$ are terms representing nodes.
- $regex$ is a regular expression over the set of original edge labels and their inverse labels.

The set of valid regular expressions $\mathcal{R}(L_{E^\pm})$ over the set of original and inverse edge labels $L_{E^\pm} = L_E \cup \{l^- \mid l \in L_E\}$ is defined recursively as follows:

- Any edge label or its inverse is a valid regular expression:

$$\forall l \in L_{E^\pm} : l \in \mathcal{R}(L_{E^\pm})$$

- If $regex_i, regex_j$ are valid expressions, their concatenation is also valid:

$$\forall regex_i, regex_j \in \mathcal{R}(L_{E^\pm}) : regex_i \cdot regex_j \in \mathcal{R}(L_{E^\pm})$$

- If $regex_i, regex_j$ are valid expressions, their union (alternation) is also valid:

  $$\forall regex_i, regex_j \in \mathcal{R}(L_{E^\pm}) : regex_i + regex_j \in \mathcal{R}(L_{E^\pm})$$

- Any valid expression can be repeated (Kleene Star) zero or more times:

  $$\forall regex \in \mathcal{R}(L_{E^\pm}) : regex^* \in \mathcal{R}(L_{E^\pm})$$

The function

$$match : \mathcal{R}(L_{E^\pm}) \times {L_{E^\pm}}^* \rightarrow Boolean$$

determines whether a word $w \in {L_{E^\pm}}^*$ (a sequence of edge labels, i.e., $edges(p)$) matches a regular expression $regex \in \mathcal{R}(L_{E^\pm})$. Formally:

$$match(regex, w) = \begin{cases} \text{true,} & \text{if } w \text{ belongs to the language defined by } regex, \\ \text{false,} & \text{otherwise.} \end{cases}$$

Given a regular expression $regex \in \mathcal{R}(L_{E^\pm})$, the RPQ $(\nu_i, regex, \nu_j)$ retrieves all pairs of nodes (ν_i, ν_j) that are connected by a path p such that:

$$match(regex, edges(p))$$

Since in this work the starting node of a path is always known, RPQs are treated as if they return only the variable term (i.e., the second node in the result). This simplifies the notation by reducing expressions to:

$$(\nu_i, regex, ?) = \{\nu_j \mid \exists p \in \#symPathsPG, match(regex, edges(p))\}$$

Furthermore, if no variable term is given, the RPQs are treated as a boolean function:

$$(\nu_i, regex, \nu_j) = \begin{cases} \text{true,} & \exists p \in Paths(PG), match(regex, edges(p)), \\ \text{false,} & \text{otherwise.} \end{cases}$$

6.2.3. Representing IBI Models as Property Graphs

To enable querying of IBI models, their structure is mapped to a property graph PG for both components and systems composed of these components. This mapping follows these rules:

- IBI model elements are mapped to nodes, which are labeled with the element name.
- IBI model associations between elements are mapped to edges, which are labeled with the association name.
- IBI model element properties are mapped by an appropriate function $prop$.

Based on these general mapping rules, the representation of IBI component models and IBI system models as property graphs is explained in detail.

IBI Component Model

Each component is represented by a property graph consisting of nodes with the following set of node labels, corresponding to element names in the metamodel:

- **Node Labels from the Component-Based System Metamodel:** *Component, IncomingInterface, OutgoingInterface, Message, Stimulus, Reaction, ComponentBehaviorTrigger, ComponentBehavior, ComponentBehaviorResult, ComponentState*

- **Node Labels from the Test Metamodel:** *AbstractTestCase, ConcreteTestCase, EnvironmentInteraction, EnvironmentRequestResponseInteraction*

The edge labels in the property graph correspond to the associations in the IBI Metamodel that connect the elements contained in the nodes. It should be noted that the abstract elements *Interface* and *ComponentInteractionRole* are not included in the node label set. The related associations are represented as follows:

- The abstract element *Interface* does not have to be represented, as IBI models contain instances of *IncomingInterface* and *OutgoingInterface*. Thus, components are directly connected with incoming and outgoing interfaces by edges labeled $provides$.

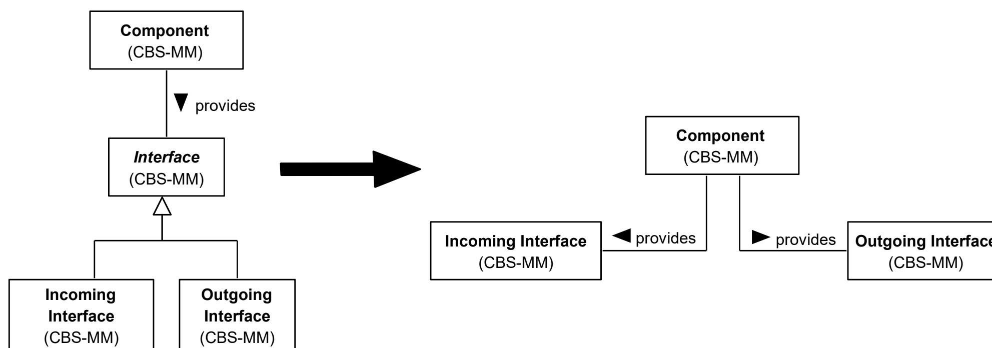


Figure 6.2.: The $provides$ association in the IBI Metamodel (left) and its representation in the property graph (right).

- The abstract element *ComponentInteractionRole* does not have to be represented either for the same reason. Messages are directly connected with their roles (*Stimulus, Reaction*) by edges labeled $hasRole$.

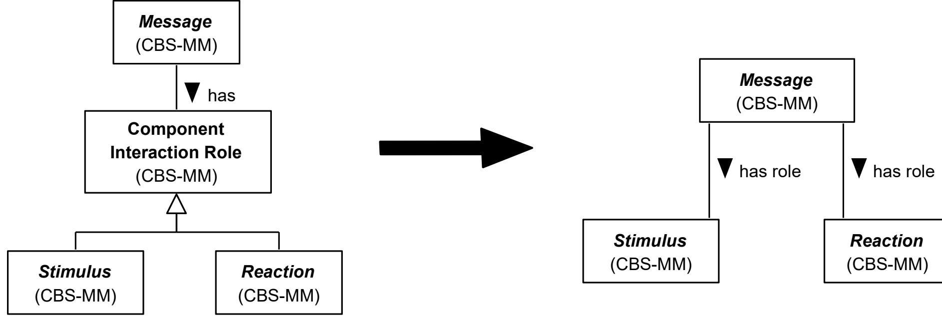


Figure 6.3.: The *Component Interaction Roles* in the IBI Metamodel (left) and their representation in the property graph (right).

By applying these simplifications, the graph structure is reduced while retaining the semantics. This streamlined representation also facilitates efficient querying using RPQs. The following definitions formalize key relationships between components, interfaces, component behaviors, and environment interactions in the IBI component model for component

The set of interfaces provided by the component c:

$$IF_c = (c, provides, ?)$$

The set of incoming interfaces of the component c:

$$IF_{\text{in},c} = \{if \mid if \in IF_c \wedge \lambda_N(if) = IncomingInterface\}$$

The set of outgoing interfaces of the component c:

$$IF_{\text{out},c} = \{if \mid if \in IF_c \wedge \lambda_N(if) = OutgoingInterface\}$$

All component behaviors CB_c of the component c:

$$CB_c = (c, provides \cdot receivedBy^{-} \cdot consistsOf^{-} \cdot triggers, ?)$$

A component behavior $cb_c \in CB_c$ is a tuple (cbt, cbr) with:

- $cbt \mid (cbt, triggers, cb_c)$ being the component behavior trigger of cb_c.
- $cbr \mid (cb_c, resultsIn, cbr)$ being the component behavior result of cb_c.

A component behavior trigger cbt of a component behavior cb_c is a tuple (S, q_{pre}) with:

- $S = \{s \mid (cb_c, triggers^{-} \cdot consistsOf, s) \wedge \lambda_N(s) = Stimulus\}$ being the stimuli and

- q_{pre} being the state of component c so that the stimuli S triggers cb_c.

A stimulus $s \in S$ of a component behavior cb_c is a tuple (m, if_{in}) with:

- $m \mid (m, hasRole, s)$ being the stimulus message and
- $if_{\text{in}} \mid (s, receivedBy, if_{\text{in}})$ being the incoming interface by which m is received by c.

The component behavior result $cbr \mid (cb_c, resultsIn, cbr)$ of a component behavior cb_c is a tuple (R, q_{post}) with:

- $R = \{r \mid (cb_c, triggers^{-} \cdot consistsOf, r) \wedge \lambda_N(r) = Reaction\}$ being the set of resulting reactions and
- q_{post} being the state after the component behavior cb_c.

A reaction $r \in R$ of a component behavior cb_c is a tuple (m, if_{out}) with:

- $m \mid (m, hasRole, r)$ being the reaction message and
- $if_{\text{out}} \mid (r, receivedBy, if_{\text{out}})$ being the outgoing interface by which m is sent by c.

This formalization of component behavior results enables chaining behavior across components: the reactions of one behavior become stimuli for the next. This relationship forms the basis for modeling system-wide interactions.

IBI System Model

While the previous section defined component behavior in isolation, systems consist of multiple components whose interactions must be modeled collectively. The IBI system model integrates individual IBI component models by introducing interface bindings. These interface bindings allow reasoning about communication across component boundaries.

A SUT is conceptually represented by the *System* element of the IBI Metamodel. Like components, a SUT is formalized as a property graph, referred to as an IBI system model. A IBI system model includes the IBI component models of all components that are part of the SUT. The essential relationship in an IBI system model is the interface binding, which defines how components interact through incoming and outgoing interfaces. In the IBI Metamodel, an interface binding is modeled as a ternary relationship between a *System*, an *Incoming Interface*, and an *Outgoing Interface*. However, since an IBI system model represents a single SUT, this ternary relationship is simplified in the property graph representation to a direct edge labeled $boundTo$ between an outgoing interface and an incoming interface, as illustrated in Figure 6.4.

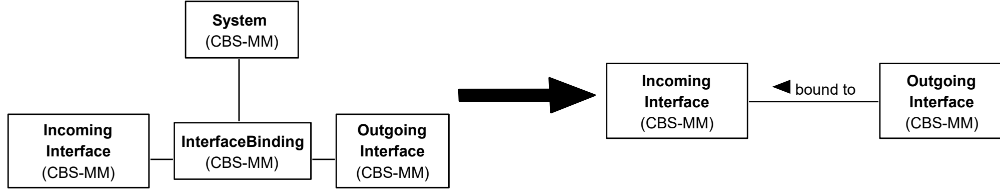


Figure 6.4.: The $boundTo$ association in the IBI Metamodel and its representation in the property graph.

Let

$$C_{\text{SUT}} = \{c_1, \dots, c_n\}$$

be the set of all components the SUT consists of.

The sets of all incoming and outgoing interfaces contained in the IBI system model of the SUT are defined as:

$$IF_{\text{in,SUT}} = \bigcup_{c \in C_{\text{SUT}}} IF_{\text{in},c}$$

$$IF_{\text{out,SUT}} = \bigcup_{c \in C_{\text{SUT}}} IF_{\text{out},c}$$

The environment Env_c of a component $c \in C_{\text{SUT}}$ consists of all other components in the system whose behavior might be directly or indirectly influenced by, or influences, interactions with component c—including components affected through state changes or transitive effects. These components are identified by traversing the interface bindings in the IBI system model:

$$Env_c = (c, (provides \cdot (boundTo + boundTo^{-}) \cdot provides^{-})^{*}, ?)$$

Usually a system has no disconnected components so that:

$$\forall c_i, c_j \in C_{\text{SUT}},\ c_j \in Env_{c_i}$$

The set of all component behaviors in a SUT is given by:

$$CB_{\text{SUT}} = \bigcup_{c \in C_{\text{SUT}}} CB_c$$

The set of all messages captured in a SUT is given by:

$$MSG = \{\nu \in N \mid \lambda_N(\nu) = Message\}$$

6.3. The Interaction-based Integration Testing Process in Detail

In this section, each action of the IBI Testing Process is explained in detail. While the actions were introduced superficially in Section 6.1, this section presents the complete control flow and the data produced and consumed by each action, as depicted in Figure 6.5. Each action is explained by specifying its inputs, outputs, and execution details. The previously introduced formalization is applied to precisely specify how the information contained in the IBI models is utilized for integration testing.

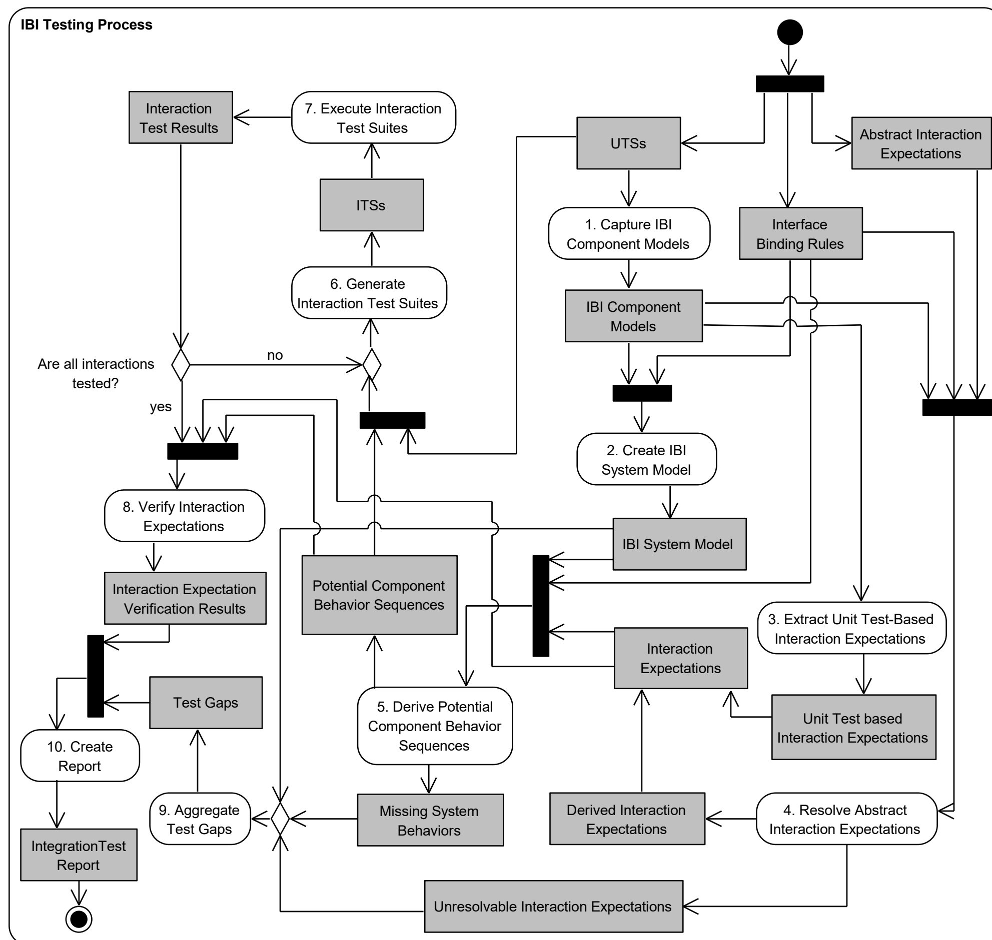


Figure 6.5.: IBI Testing Process

6.3.1. Action 1: Capture IBI Component Models

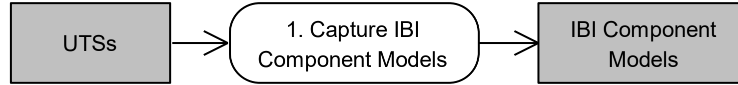


Figure 6.6.: Capture IBI component models

Input The UTSs of all components $c \in C_{\text{SUT}}$

Output An IBI component model for each component $c \in C_{\text{SUT}}$

Action For each component $c \in C_{\text{SUT}}$, the IBI component model is derived by observing the messages exchanged between component c and its environment (test harness and mocks) during execution of its UTS. These are used to extract component behaviors and environment interactions for each test case in an IBI component model. A concrete test case ctc of a component c is characterized by a sequence of alternating component behaviors and environment interactions:

$$ctc = (cb_1, ei_1, \ldots, ei_{n-1}, cb_n)$$

where:

- A component behavior cb_i is a tuple (cbt_i, cbr_i) with:
  - cbt_i being the component behavior trigger of component behavior cb_i, given as a tuple $(S_i, q_{\text{pre},i})$ and
  - cbr_i being the component behavior result of component behavior cb_i, given as a tuple $(R_i, q_{\text{post},i})$.
- An environment interaction ei_i is a tuple (cbr_i, cbt_{i+1}) with:
  - cbr_i being the component behavior result of the previous component behavior cb_i that starts the environment interaction ei_i and
  - cbt_{i+1} being the component behavior trigger of the subsequent component behavior cb_{i+1} that is triggered in consequence of environment interaction ei_i.
- The state $q_{\text{post},i}$ of each component behavior result cbr_i of a component behavior cb_i is equal to the state $q_{\text{pre},i+1}$ of the subsequent component behavior trigger cbt_{i+1} of component behavior cb_{i+1}.

The set of all concrete test cases of a component c is given by:

$$CTC_c = (c, testedBy \cdot derivedFrom^{-}, ?)$$

The set of all component behaviors of a concrete test case ctc is given by:

$$CB_{\text{ctc}} = (ctc, verifies, ?)$$

The position of a component behavior cb_i in its concrete test case ctc is given by:

$$pos(cb_i) = i$$

The created IBI component model provides a structured representation of:

- All unit tested component behaviors.
- All environment interactions that are part of the unit tests.

6.3.2. Action 2: Create IBI System Model

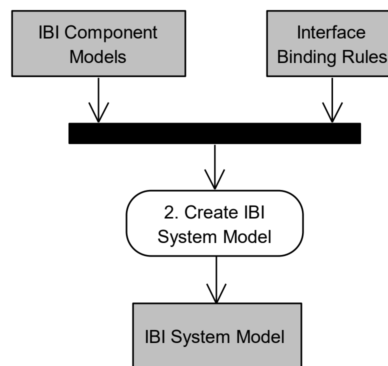


Figure 6.7.: Create IBI system model

Input

- IBI component models of all components $c \in C_{\text{SUT}}$
- Interface binding, given as a function $bound : IF_{\text{in}} \times IF_{\text{out}} \rightarrow Boolean$, where IF_{in} is the domain of incoming interfaces and IF_{out} is the domain of outgoing interfaces. These domains include all interfaces, including, but not limited to, those provided by the components in the SUT.

Output An IBI system model of the SUT

Action An IBI system model is created by establishing the interface bindings between incoming and outgoing interfaces contained in the component models according to the interface binding rules. The IBI system model is constructed by applying the following rule:

$$\forall if_{\text{out}} \in IF_{\text{out,SUT}}, \forall if_{\text{in}} \in IF_{\text{in,SUT}},\ (if_{\text{out}}, boundTo, if_{\text{in}}) \iff bound(if_{\text{out}}, if_{\text{in}})$$

6.3.3. Action 3 & 4: Generate Interaction Expectations

Interaction expectations define how a user or component intends to interact with a system. These expectations are crucial to the presented IBI Testing Process, as they serve as the foundation for verifying system behavior. However, interaction expectations are not defined explicitly, since doing so would tie them closely to the system's implementation—which may change over time—but are instead extracted and derived from unit test data. This section details how interaction expectation are obtained:

- **Unit Test-based Interaction Expectations:** These are extracted from the environment interactions contained in the IBI component models.
- **Derived Interaction Expectations:** These are inferred from the abstract interaction expectations imposed on the SUT.

As already introduced, each interaction expectation consists of:

- A system behavior trigger, defining the conditions under which the expectation applies.
- A set of interfaces at which reactions from the SUT are expected.
- An assertion, specifying the expected system behavior.

An interaction expectation ie is expressed by a tuple

$$ie = (sbt, IF_{\text{in,exp}}, assert)$$

with sbt being the system behavior trigger, $IF_{\text{in,exp}}$ the set of expected reaction interfaces, and $assert$ the assertion. A system behavior trigger sbt is a tuple (R, q_{SUT}) with:

- R is a set of component reactions that are sent to the SUT. These reactions act as stimuli for the SUT.
- q_{SUT} being the initial system state given by a set of initial component states $\{q_{c_1}, \ldots, q_{c_n}\}$ with $n = |C_{\text{SUT}}|$.

Action 3: Extract Unit Test-based Interaction Expectations

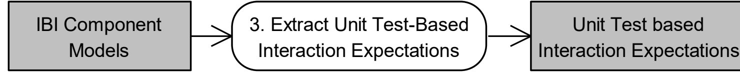


Figure 6.8.: Extract Unit Test-based Interaction Expectations

Input IBI component models of all components $c \in C_{\text{SUT}}$

Output Set of unit test-based interaction expectations IE_{utb}

Action Unit test-based interaction expectations capture the expectations a component $c \in C_{\text{SUT}}$ has regarding its interactions with its environment Env_c. For each environment interaction a unit test-based interaction expectation ie_{utb} is extracted:

$$ie_{utb} = (sbt, IF_{\text{in,exp}}, assert)$$

Each environment interaction $ei_i = (cbr_i, cbt_{i+1})$ is started by component behavior result $cbr_i = (R_i, q_{\text{post},i})$. Each reaction is represented as a tuple (m, if_{out}), where m is a message that is sent by the CUT c by an outgoing interface if_{out}. The reaction messages m are stimulus messages for yet to be determined components $?c \in Env_c$ that provide an incoming interface if_{in} that is bound to that outgoing interface if_{out}: $(if_{\text{out}}, \text{boundTo}, if_{\text{in}})$. Since the environment interaction does not specify the state of components in the environment Env_c the initial system state of the system behavior trigger consists of the initially unknown states $?q_{c_i}$ of all components $c_i \in Env_c$ as well as the known state $q_{\text{post},i}$ of the CUT c according to the component behavior result cbr_i that started the environment interaction ei_i. This results in a system behavior trigger:

$$sbt = (R_i, \{q_{\text{post}}, ?q_{c_1}, \dots, ?q_{c_n}\})$$

The assertion $assert$ of the unit test-based interaction expectation ensures that a subset of components in the environment Env_c of CUT c react to the reactions R_i such that their reactions are received at the same interfaces as the mocked stimuli S_i of the component behavior trigger $cbt_{i+1} = (S_{i+1}, q_{\text{pre},i+1})$ of the environment interaction ei_i, and that their reactions can replace the mocked stimuli S_{i+1} so that the subsequent component behavior cb_{i+1}, in the concrete test case ei_i is contained in, can still be verified. Using RPQs, the set of reaction interface expectations of the unit test-based interaction expectation is determined by the interfaces where the mocked reactions S_{i+1} of Env_c were received:

$$IF_{\text{in,exp}} = \bigcup_{s \in S_{i+1}} (s, receivedBy, ?)$$

Action 4: Resolve Abstract Interaction Expectations

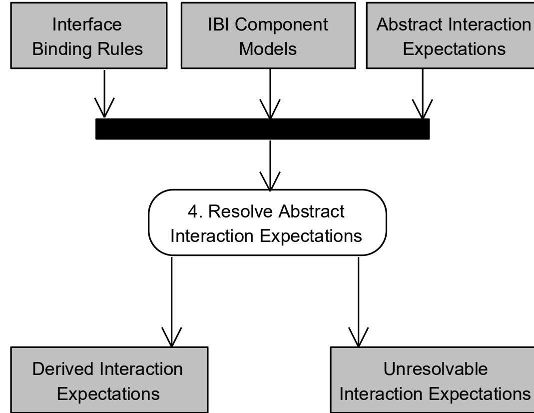


Figure 6.9.: Derivation of Interaction Expectations.

Input

- Abstract Interaction Expectations $iea \in IE_a$, each represented as a tuple:

  $(IF_{\text{out,exp}}, msgSelect, IF_{\text{in,exp}}, assert)$

  where

  - $IF_{\text{out,exp}}$ is the set of interfaces that messages are sent by to the SUT.
  - $msgSelect : \binom{MSG}{|IF_{\text{out,exp}}|} \to Boolean$ is a function that selects sets of messages sent by the interfaces in $IF_{\text{out,exp}}$ for which the expectation should hold.[1]
  - $IF_{\text{in,exp}}$ is the set of interfaces by which messages, from the SUT, are expected to be received in response.
  - $assert : \binom{MSG}{|IF_{\text{out,exp}}|} \times \binom{MSG}{|IF_{\text{in,exp}}|} \to Boolean$ is a function that asserts the expected system behavior based on the stimulus and reaction messages of the SUT.

- IBI component models of all components $c \in C_{\text{SUT}}$.
- Interface binding rules, given by a function $bound : IF_{\text{in}} \times IF_{\text{out}} \to Boolean$ as for the creation of IBI system models.

1. $\binom{A}{n}$ denotes the set of all subsets of size n of a set A.

Output

- Set of derived interaction expectations IE_d, specifying interaction expectations based on existing unit test cases
- Unresolvable interaction expectations, representing cases where no suitable unit test cases exist in the IBI component models

Action Derived interaction expectations are defined similarly to unit test-based interaction expectations, with one difference: derived interaction expectations are generated by resolving abstract interaction expectations using existing unit test data of the IBI component models instead of using an observed reaction messages from an environment interaction to define the system behavior trigger. To achieve this, the action consists of three steps:

1. **Resolve Stimulus Interface Expectation:** Identify interfaces in the IBI component models that align with the stimulus interface expectations $IF_{\text{out,exp}}$ of the abstract interaction expectation iea.
2. **Select Stimulus Messages:** Identify message sets received on those interfaces that satisfy the stimulus message selector $msgSelect$ of the abstract interaction expectation iea.
3. **Create Derived Interaction Expectations:** Generate a derived interaction expectation for each selected message set.

If no message sets are found, the abstract interaction expectation iea is unresolvable, indicating that either no matching interface exists in the IBI component models or that the equivalence class for the stimulus is not covered by any unit test.

These steps are formalized as follows:
A derived interaction expectation ie_d is a tuple

$$(sbt, IF_{\text{in,exp}}, assert)$$

with:

- sbt being a system behavior trigger that needs to be resolved.
- $IF_{\text{in,exp}}$ being the set of expected reaction interfaces of the abstract interaction expectation the derived interaction expectation is derived from.
- $assert$ being the assertion of the abstract interaction expectation iea the derived interaction expectation ie_d is derived from.

The stimulus interface expectations $IF_{\text{out,exp}}$, defined in the abstract interaction expectation iea, must be bound to an existing incoming interface in the IBI system model to resolve the system behavior trigger sbt:

$$\forall if_{\text{out}} \in IF_{\text{out,exp}}, \exists if_{\text{in}} \in IF_{\text{in,SUT}} : bound(if_{\text{out}}, if_{\text{in}})$$

The messages received on those interfaces are filtered using the $msgSelect$ function to determine sets of messages to stimulate the SUT for which the interaction expectation should hold. The set of valid stimuli sets is given as:

$$\begin{aligned} R_{selected} = \{R \subseteq MSG \times IF_{\text{out,exp}} \\ & (\forall (m, if_{\text{out}}) \in R, \exists if_{\text{in}} \in IF_{\text{in,SUT}} : bound(if_{\text{out}}, if_{\text{in}}) \wedge \\ & (m, receivedBy, if_{\text{in}})) \wedge \\ & (\forall if_{\text{out}} \in IF_{\text{out,exp}}, \exists m \in MSG : (m, if_{\text{out}}) \in R) \wedge \\ & msgSelect(\bigcup \{m \mid (m, if_{\text{out}}) \in R\})\} \end{aligned}$$

For each reaction set $R \in R_{selected}$, a derived interaction expectation is created. Its system behavior trigger is defined as:

$$sbt = (R, ?q_{c_1}, \ldots, ?q_{c_n}).$$

Both, the unit test-based and derived interaction expectations are collected in a set if interaction expectations IE.

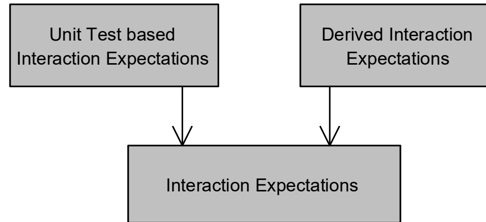


Figure 6.10.: Collect Interaction Expectations.

6.3.4. Action 5: Derive Potential Component Behavior Sequences

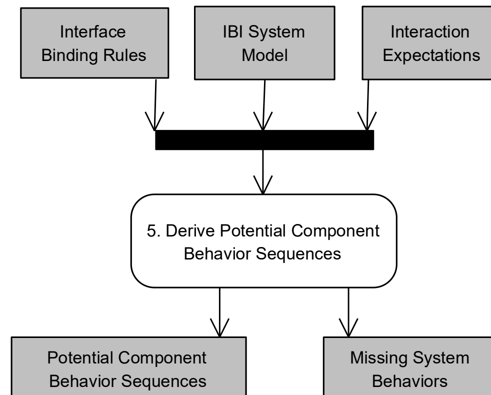


Figure 6.11.: Derive Potential Component Behavior Sequences

Input

- Interaction expectations $IE = IE_{utb} \cup IE_d$, containing the unit test-based and derived interaction expectations, given as tuples $(sbt, IF_{\text{in,exp}}, assert)$
- IBI system model of the SUT
- Interface binding rules, given by a function $bound : IF_{\text{in}} \times IF_{\text{out}} \to Boolean$ as for the creation of IBI system models

Output

- Set of potential component behavior sequences $CBSeq$ that represent potential system behaviors satisfying the interaction expectations.
- Set of missing system behaviors, indicating expectations for which no suitable component behavior sequence exists.

Action Since unit tests verify only isolated behaviors, integration testing must verify that these behaviors can be composed into valid end-to-end system behaviors that fulfill interaction expectations. Thus, once interaction expectations are collected, the next step is to identify potential system behaviors fulfilling these expectations. Each component behavior in the IBI system model corresponds to a specific component behavior tested in isolation using a unit test. However:

- A unit test verifies only a single representative of an equivalence class of inputs.

- The equivalence class itself is not explicitly defined.
- Instead, the abstract test case implicitly defines the equivalence class: if the test is executed with different test data and still succeeds, then the new data belongs to the same equivalence class.

Since there is no formal definition of the equivalence class, it remains unknown whether a message can replace the stimulus of a component behavior. Thus, it is also unknown whether the component behavior can be triggered by the reactions of other component behaviors.

To address this, potential behavior graphs are determined that represent potential system behaviors that satisfy the interaction expectations. These potential behavior graphs are derived by iteratively combining component behaviors that are triggered by the reactions of other component behaviors, according to the interface bindings. The potential behavior graphs are determined as follows:

1. Every interaction expectation specifies a system behavior trigger sbt that defines the initial set of reactions R that can trigger component behaviors.
2. The set of component behaviors that can be triggered by the set of reactions R is determined. A component behavior can be triggered if and only if for every stimulus s in the component behavior trigger, there is a reaction $r \in R$ sent by an outgoing interface if_{out} that is bound to the incoming interface if_{in} of the stimulus s. The set of component behaviors selected for a component does not share reactions, ensuring that each reaction is only received once by a component. If multiple sets of component behaviors can be triggered, separate graphs are created for each alternative execution path.
3. The set of all reactions of all selected component behaviors for each graph is added to the set of reactions R.
4. These steps are repeated for each graph until no more component behaviors can be triggered.

The resulting behavior graphs represent all possible message flows. However, not all of these potential system behaviors are valid. There has to be at least one that results in reactions that are received at the expected interfaces of the interaction expectation. Whether those reactions satisfy the assertion of the interaction expectation can not be determined by static analysis. This is left to the execution of the interaction tests. The process is formalized as follows:

Let an interaction expectation ie be a tuple $(sbt, IF_{\text{in,exp}}, assert)$ then a potential behavior graph $PBG = (CB, E)$ is a directed acyclic graph with:

- $CB \subset CB_{\text{SUT}}$ being a subset of component behaviors in the IBI system model that are part of a potential system behavior.
- $E \subseteq CB \times CB$ being the set of edges connecting component behaviors.

- A set of nodes $CB_{\text{source}} \subseteq CB$ that represent component behaviors triggered by the system behavior trigger sbt of an interaction expectation, that represent the source nodes of the potential behavior graph:

$$\forall((S, q_{\text{pre}}), cbr) \in CB_{\text{source}}, \forall(m, if_{\text{in}}) \in S, \exists(m, if_{\text{out}}) \in R\colon bound(if_{\text{out}}, if_{\text{in}})$$

- A set of nodes $CB_{\text{exp}} \subseteq CB$ that represent component behaviors that result in reactions by interfaces bound to the reaction interface expectations $IF_{\text{in,exp}}$ of the interaction expectation:

$$\forall if_{\text{in}} \in IF_{\text{in,exp}}, \exists(cbt, (R_{cb}, q_{\text{post}})) \in CB_{\text{exp}}, \exists(m, if_{\text{out}}) \in R_{cb}\colon bound(if_{\text{out}}, if_{\text{in}})$$

- Edges in PBG, written as $(cb', cb) \in E$, where $cb = ((S_{cb}, q_{\text{pre}}), (R_{cb}, q_{\text{post}}))$ and $cb' = ((S_{cb}', q_{\text{pre}}'), (R_{cb}', q_{\text{post}}'))$, exists such that for each stimulus a reaction of a previous component exists:

$$\forall(m, if_{\text{in}}) \in S_{cb}, \exists!(m', if_{\text{out}}) \in \bigcup_{cb' \in CB\colon (cb', cb) \in E} R_{cb}', (if_{\text{out}}, \text{boundTo}, if_{\text{in}})$$

  The set of all reactions (m', if_{out}) that replace the stimuli for a given component behavior cb is denoted as RS_{cb}.

- A reaction can not trigger multiple component behaviors of the same component:

$$\forall c \in C_{\text{SUT}}, \forall cb, cb' \in CB \cap CB_c, cb \neq cb'\colon \exists cb'' \in CB, (cb'', cb) \in E \wedge (cb'', cb') \in E \rightarrow RS_{cb} \cap RS_{cb'} = \emptyset$$

- There is a path from a node in CB_{source} to each node in CB_{exp}.

- Let the depth of a component behavior cb in the potential behavior graph be the length of the longest path from a node in CB_{source} to the node cb. This is expressed as:

$$depth(cb) = \max_{cb' \in CB_{\text{source}}} \{length(p) | p \text{ is a path from } cb' \text{ to } cb\}$$

  The component behaviors are in the same order as in the concrete test case:

$$\forall cb \in CB, \exists c \in C_{\text{SUT}}, \exists ctc \in CTC_c, cb \in CB_{ctc}, \forall cb' \in CB_{ctc}\colon depth(cb') < depth(cb) \iff pos(cb') < pos(cb)$$

  Otherwise the specification given by the concrete test case is violated.

A potential component behavior sequence is a topological ordering $(cb_1, \ldots, cb_n)$ of the nodes in PBG, such that for every edge $(cb_i, cb_j) \in E$, cb_i precedes cb_j in the sequence. A potential component behavior sequence represents a potential system behavior.

6.3.5. Action 6 & 7: Generate and Execute Interaction Test Suites

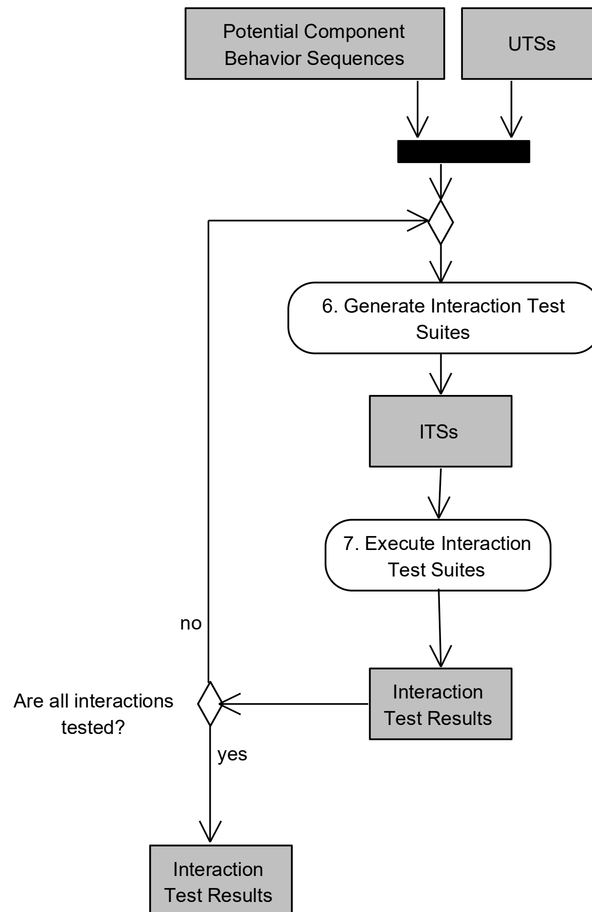


Figure 6.12.: Iterative Interaction Expectation verification

The potential component behavior sequences collected in the previous step represent potential system behaviors that could satisfy the interaction expectations. However, because these component behavior sequences are derived from isolated unit test executions, they must be verified in the context of the system behavior to determine whether they indeed represent actual system behaviors when components interact.

This verification is performed iteratively, using interaction test cases in which the original unit test stimuli are replaced with reactions from preceding component behaviors. If the modified test cases are verified, the potential component behavior sequence is confirmed to represent valid system behavior; if any test fails, the

potential component behavior sequence is discarded as an incorrect representation of system behavior.

Input

- UTSs of all components $c \in C_{\text{SUT}}$
- Potential Component Behavior Sequences $CBSeq$

Output Interaction test results that capture the result of each interaction test case

Actions The verification consists of two main steps:

- Generate interaction test suites.
- Execute interaction test suites.

Each of these steps is performed iteratively, with each ITS building upon the results of the previous ones.

Action 6: Generate Interaction Test Suites
For each potential component behavior in a sequence, an interaction test case is generated. The component behavior trigger is modified in two ways:

- **Stimulus Replacement:** The stimulus in the component behavior trigger is replaced with the reaction message from the preceding component behavior.
- **Component State Adjustment:** If a component appears multiple times in a component behavior sequence, its state is set to the last observed state. If it is the first occurrence of the component in the component behavior sequence, the state is initialized using the arbitrary state provided by the original test case. This resolves the state variables $?q_{c_i}$ of the SUT's state q_{SUT}.

Thus, for each cb_i in a potential component behavior sequence $(cb_1, \ldots, cb_n)$, let c denote the component associated with cb_i (i.e., $cb_i \in CB_c$). Let $cbt_{\text{orig}} = (S, q_{\text{pre}})$ be the component behavior trigger of component behavior cb_i, then the new component behavior trigger is generated as:

$$cbt_{\text{adapt}} = (S_{adapt}, q_{\text{pre,adapt}}).$$

where:

- R_{pre,cb_i} is the set of the reactions of all previous component behaviors cb_j with $j < i$.

- S_{adapt} is the set of stimuli that replace the original stimuli S in the component behavior trigger cbt_{orig}. It is defined as:

$$S_{\text{adapt}} = \{(m, if_{\text{in}}) \mid \exists (m, if_{\text{out}}) \in R_{\text{pre},cb_i}, \exists (m', if_{\text{in}}) \in S \colon (if_{\text{out}}, \text{boundTo}, if_{\text{in}})\}$$

- $q_{\text{pre,adapt}}$ is determined based on previous occurrence of a component behavior $cb \in CB_c$ of component c in the component behavior sequence. Let $cbr = (R, q_{\text{post}})$ be the component behavior result of the previous component behavior cb of component c, then $q_{\text{pre,adapt}} = q_{\text{post}}$. If there is no previous cb, $q_{\text{pre,adapt}}$ is assigned the arbitrary state defined by the abstract test case based on the new stimulus messages S_{adapt}.

The resulting interaction test cases that verify whether the component behavior sequence represents a feasible system behavior are modified unit test cases of the UTSs. The generated interaction tests cases are added to the interaction test suites ITS of the respective components.

Action 7: Execute Interaction Test Suites

Each interaction test case that is part of an ITS is executed and observed in the same manner as the unit test cases. The observed component behaviors — and the messages collected as a result — are then used as inputs for subsequent interaction test cases, ensuring that later interactions in the component behavior sequence build upon the actual outputs of earlier interactions. If an interaction test case is required by multiple component behavior sequences, it is executed only once. This is especially useful when multiple systems contain the same components.

For example, if one component behavior changes while the others remain the same, only the affected sections of the system behavior need to be retested. The execution of these interaction tests determines whether the assumptions made in the potential behavior graph hold true in the real system. Specifically:

- If the test case succeeds, the adapted set of stimulus messages S_{adapt} is a valid substitute for the original set of stimulus messages S of cb_i, confirming that both belong to the same equivalence class.
- If the test fails, the assumption that S_{adapt} and S are interchangeable is invalid, and the potential component behavior sequence is discarded.

By performing this iterative testing approach, the IBI testing process systematically verifies which potential behavior sequences correspond to actual system behaviors.

6.3.6. Action 8: Verify Interaction Expectations

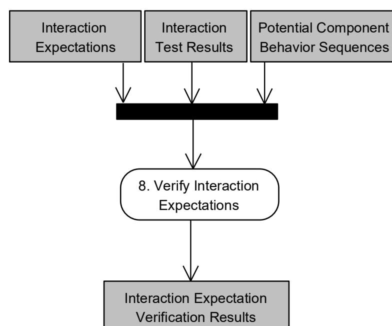


Figure 6.13.: Verify Interaction Expectations

Input

- Interaction Expectations IE
- Potential component behavior sequences $CBSeq$
- Interaction test results

Output Interaction Expectation Verification Results.

Action Once the interaction test results are obtained, the next step is to determine which potential component behavior sequences actually represent implemented system behaviors, thereby verifying the interaction expectations. Each interaction expectation is verified by checking whether at least one system behavior that satisfies the expectation is realized by a potential component behavior sequence.

A potential component behavior sequence

$$(cb_1, \ldots, cb_n)$$

is considered valid if:

- All interaction tests are successful (i.e., stimulus replacements do not break the test).
- The reaction messages satisfy the assertion specified by the interaction expectation.

If at least one component behavior sequence satisfies these conditions, the interaction expectation is verified. Otherwise, it is failed, indicating a gap in test coverage or a missing system behavior. An abstract interaction expectation is verified if all its derived interaction expectations are verified.

6.3.7. Action 9: Collect Test Gaps

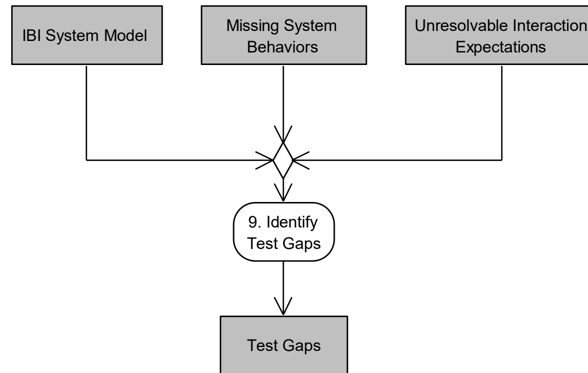


Figure 6.14.: Collect Test Gaps

Input

- IBI system model of the SUT
- Missing system behaviors
- Unresolvable interaction expectations

Output Test gaps

Action Analyzing the IBI system model of the SUT reveals areas where additional testing or system refinements may be necessary. This includes missing system behaviors and unresolved interaction expectations. The following scenarios indicate potential test gaps:

Disconnected Component: A component or set of components in the IBI system model that is disconnected from other component contained in the property graph indicates a potential test gap. Disconnected components suggest that no interactions involving these components are expected from other components. Formally, disconnected components exist if:

$$\exists\, c_i,\, c_j \in C_{\text{SUT}} : c_i \notin Env_{c_j}$$

Unmatched Interface Expectations: An interface expectation in an interaction expectation that does not match any interface in the IBI system model indicates a test gap.

No Matching Messages for Abstract Interaction Expectation: A test gap exists if no messages in the IBI system model match the stimulus message selector of an abstract interaction expectation. This indicates that no representative from the equivalence class of messages (as defined by the stimulus message selector) has been tested. Consequently, parts of the input space remain unexplored at the unit test level, leading to insufficient test coverage.

Absence of a Satisfying Component Behavior Sequence: Test gaps are also identified when an interaction expectation exists for which no satisfying component behavior sequence could be found. This indicates that either the corresponding system behavior has not been tested by unit test cases, or it does not exist in the system.

6.3.8. Action 10: Create Report

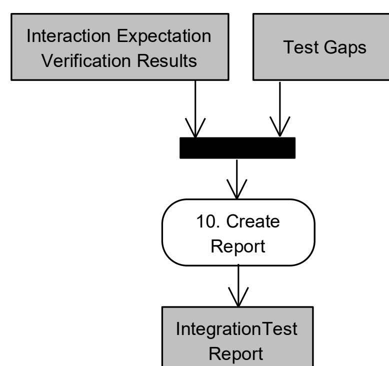


Figure 6.15.: Create Report

Input

- Interaction expectation verification results
- Test gaps

Output Integration Test Report

Action All interaction expectation verification results and identified test gaps are aggregated into a comprehensive integration test report. This report provides an overview of the integration test outcome, summarizing both verified and failed interaction expectations, highlighting test gaps, and enabling fault localization by tracing failures back to individual component behaviors.

Chapter 7.

The IBI Testing Process in Action

Table of Contents

7.1. Motivation & Overview of the Example Systems 77
7.2. Inputs to the Interaction-based Integration Testing Process 79
7.3. Capture Interaction-based Integration Component Models & Create Interaction-based Integration System Models 82
7.4. Collect Interaction Expectations 87
7.5. Derive Component Behavior Sequences 87
7.6. Generate & Execute Interaction Test Suites 89
7.7. Verify Interaction Expectations 91
7.8. Create Report 92

7.1. Motivation & Overview of the Example Systems

To illustrate the IBI Testing Process in practice, this chapter uses two example system configurations derived from the Piggy Bank system introduced in Chapter 4. These systems are intentionally simplified versions of the complete Piggy Bank system and are designed to highlight both the successful verification of interaction expectations and the detection of integration faults.

The following requirement applies to both systems:
“If a transfer is made to a maintained account and the user has subscribed to notifications, a notification shall be sent confirming that the account was credited with the transferred amount.”

The two systems differ in a single, intentionally harsh aspect: one includes the Notification Service, while the other does not. While the change may appear drastic, it is chosen for illustrative clarity. In practice, similar verification failures could also arise from more subtle changes, such as missing interfaces, missing interface bindings or changes to the Notification Service’s behavior, that would likewise prevent the requirement from being fulfilled. However, such scenarios introduce additional complexity and are omitted here to keep the example focused and

accessible. The two systems enable a concise and effective demonstration of the IBI Testing Process and its verification capabilities.

The example systems used throughout this chapter are:

- ***PBS_1* (System with Notification Service)**, shown in Figure 7.1, consists of the following components:
  - **Transfer Gateway**: Handles incoming transfer requests and initiates balance updates.
  - **Account Twin Service**: Maintains account states and processes credit/debit transactions.
  - **Notification Service**: Sends notifications for transactions if a user subscription exists.
- ***PBS_2* (System without Notification Service)**: Identical to *PBS_1* but with the Notification Service removed.

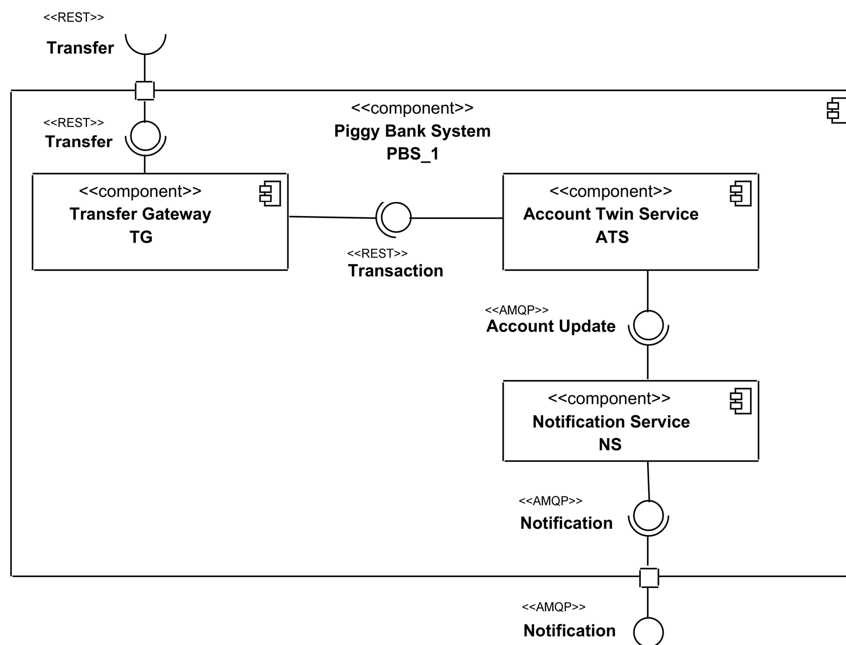


Figure 7.1.: Component diagram of the *PBS_1*.

The IBI Testing Process is applied to both systems in parallel throughout this section. The results of each step of the IBI Testing Process for each system are compared to illustrate how the absence of the Notification Service affects integration verification.

Powered By Visual Paradigm Community Edition

In the following Figures of the IBI component and system models, the central elements (*Components*, *Component Behaviors* and *Messages*) are highlighted in white for better comprehensibility.

7.2. Inputs to the IBI Testing Process

The IBI Testing Process, introduced in Chapter 6, defines the necessary steps to verify component interactions and thereby system behavior. To initiate the process, the following three types of input are required:

- The unit test suites (UTS) of all components in the SUT,
- The interface bindings,
- The set of abstract interaction expectations.

This section introduces the specific inputs used for the two example systems *PBS_1* and *PBS_2*.

Unit Test Suites

Each component in the Piggy Bank system is accompanied by a UTS each consisting of a set of unit test cases. The following reduced UTSs are considered for this example:

- **UTS for the Transfer Gateway**
  - Unit test case *utc1_TG*: Verifies that a transfer from one account to another is correctly processed.
- **UTS for the Account Twin Service**
  - Unit test case *utc1_ATS*: Verifies crediting an account.
  - Unit test case *utc2_ATS*: Verifies debiting an account.
- **UTS for the Notification Service**
  - Unit test case *utc1_NS*: Verifies that a notification is sent when a user is subscribed and an account update has occurred.

These unit test cases are explained in detail in the next section, where the corresponding IBI component models are constructed. For *PBS_1*, the unit test case *utc1_NS* is included, while for *PBS_2*, it is omitted since the Notification Service and its UTS is not part of the system.

Interface Bindings
To analyze interactions at the system level, the IBI component models are connected via interface bindings. These bindings define how outgoing messages from one component are routed to the corresponding incoming interfaces of another. Each conceptual interface in the component diagram in Figure 7.1 is mapped to one or more interface elements in the IBI component models, depending on the underlying communication protocol.

REST-based Communication: REST-based communication involves a request followed by a response. Therefore, each REST-based interaction is modeled using two interfaces per component:

- The request interface pair
  - An *outgoing* request *interface* for the calling component,
  - An *incoming* request *interface* for the called component.
- The response interface pair:
  - An *outgoing* response *interface* for the called component,
  - An *incoming* response *interface* for the calling component.

In this example:

- The Transfer REST interface is represented by:
  - *if_in1_TG* as the *incoming* request *interface*,
  - *if_out2_TG* as the *outgoing* response *interface*.
- The Transaction REST interface is represented by:
  - *if_out1_TG* as the *outgoing* request *interface* of the Transfer Gateway and *if_in1_ATS* as the *incoming* request *interface* of the Account Twin Service,
  - *if_out2_ATS* as the *outgoing* response *interface* of the Account Twin Service and *if_in2_TG* as the *incoming* response *interface* of the Transfer Gateway.

AMQP-based Communication: AMQP is an asynchronous messaging protocol and therefore does not distinguish between requests and responses in the same way. For AMQP-based communication, each conceptual interface is modeled as a single interface per component in the respective IBI component models:

- An *outgoing interface* for the sending component,
- An *incoming interface* for the receiving component.

In this example:

- The Account Update AMQP interface is represented by:

  - *if_out1_ATS* as the *outgoing interface* of the Account Twin Service,
  - *if_in1_NS* as the *incoming interface* of the Notification Service.

- The Notification AMQP interface is represented by:
  - *if_out1_NS* as the *outgoing interface* of the Notification Service.

For this example the interface bindings for the two Piggy Bank systems are summarized in Table 7.2. The table shows which *outgoing interfaces* are bound to which *incoming interfaces* in *PBS_1* and *PBS_2*.

PBS_1		*PBS_2*	
Outgoing Interface	Incoming Interface	Outgoing Interface	Incoming Interface
if_out1_TG	*if_in1_ATS*	*if_out1_TG*	*if_in1_ATS*
if_out1_ATS	*if_in1_NS*	*if_out2_ATS*	*if_in2_TG*
if_out2_ATS	*if_in2_TG*		

Table 7.2.: Interface Bindings for Piggy Bank systems *PBS_1* and *PBS_2*

Abstract Interaction Expectation
As already mentioned, the same requirement is applied to both systems:

“If a transfer is made to a maintained account and the user has subscribed to notifications, a notification shall be sent confirming that the account was credited with the transferred amount.”

This requirement is transformed into the abstract interaction expectation *aie1*, representing the interaction expectation independently of specific test data or system configuration.

The abstract interaction expectation *aie1* is composed of the following elements:

- **Stimulus Interface Expectation** *if_out_aie1*: Represents the outgoing interface through which a transfer request is sent to the system.
- **Stimulus Message Selector** $msgSelect$: Selects messages representing valid transfers as triggers for the expected system behavior.
- **Reaction Interface Expectation** *if_in_aie1*: Represents the incoming interface through which a notification is expected in response.
- **Assertion** *assert_aie1*: Asserts that the notification reflects the account was credited with the transferredamount, as specified in the transfer request.

7.3. Capture IBI Component Models & Create IBI System Models

To systematically analyze the integration of components, IBI component models are constructed by executing unit test cases for each component. Once the IBI component models are created, they are combined to form a IBI system model. The following subsections describe the unit test cases used to derive IBI component models and illustrate how the IBI system models are built from these component representations.

7.3.1. Transfer Gateway

The Transfer Gateway (TG) is responsible for handling transfers and initiate the update of relevant account twins. For this example one unit test case, referenced as *utc1_TG*, is considered to construct the IBI component model of the TG component (see Figure 7.3). First, the unit test harness initializes the TG component with the

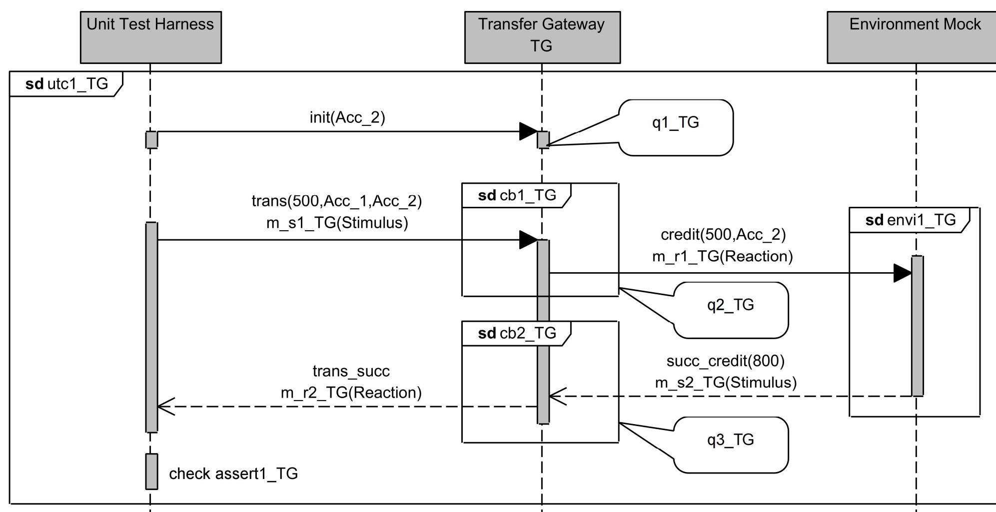


Figure 7.3.: Sequence diagram of the Transfer Gateway's unit test case *utc1_TG*.

state *q1_TG*, so that account *Acc_2* is included in the list of accounts maintained by TG. The unit test harness then sends the stimulus message *m_s1_TG* via TG's incoming interface *if_in1_TG*, indicating that 500€ have been transferred from account *Acc_1* to account *Acc_2*. This stimulus message triggers TG's component behavior *cb1_TG*, which results in the updated state *q2_TG*, and the reaction message *m_r1_TG* that is sent by TG's outgoing interface *if_out1_TG* to the environment mock.

Powered By: Visual Paradigm Community Edition

The environment mock is configured to mock the environment interaction *envi1_TG* so that this mock responds to the reaction message *m_r1_TG* with the stimulus message *m_s2_TG*. This stimulus message confirms an updated balance of 800€ for the account *Acc_2*, where 800€ is a magic number, specified by the unit test case *utc1_TG*. When TG receives the stimulus message *m_s2_TG* by TG's incoming interface *if_in2_TG*, TG's component behavior *cb2_TG* is triggered, resulting in the state *q3_TG*, and the reaction message *m_r2_TG* that is sent by TG's outgoing interface *if_out2_TG* to the unit test harness. The reaction message *m_r2_TG* is used in the assertion *assert1_TG*, of the unit test case *utc1_TG*, to ensure that the TG component handled the transfer successfully.

The resulting IBI component model for the Transfer Gateway is illustrated in Figure 7.4, with the component, component behaviors and messages highlighted in white.

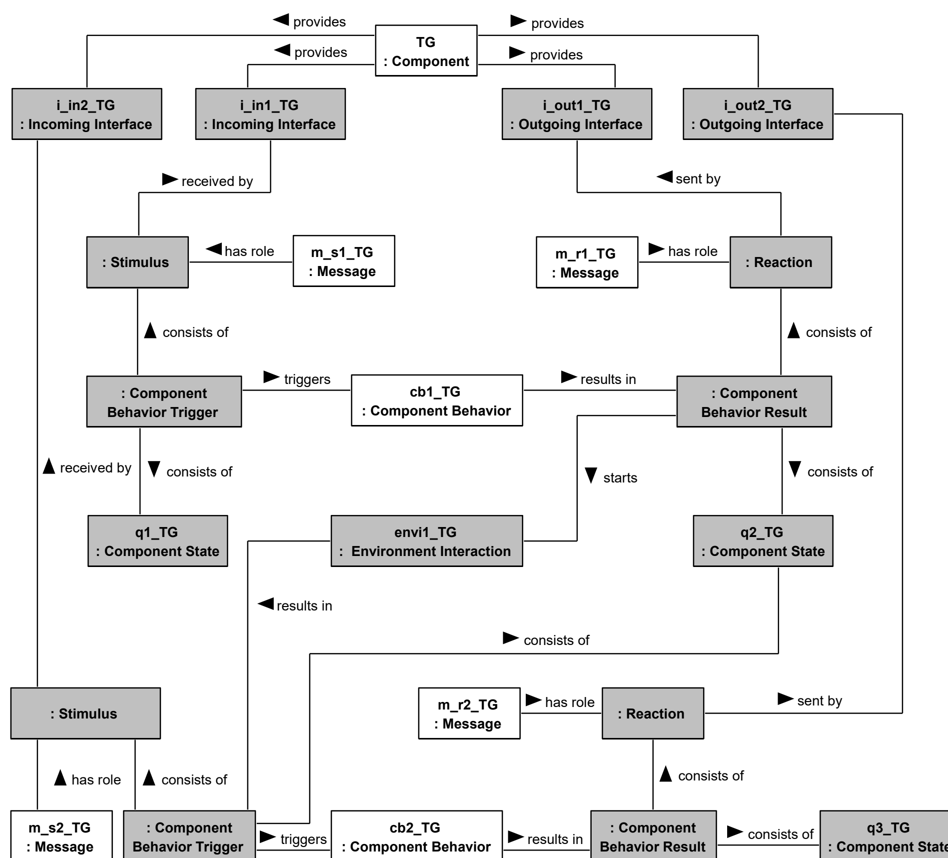


Figure 7.4.: IBI component model for the Transfer Gateway.

7.3.2. Account Twin Service

The Account Twin Service (ATS) is responsible for maintaining account balances and processing credit and debit transactions. The IBI component model of component ATS is constructed, considering two unit test cases: one to test a credit transaction (*utc1_ATS*) and another to test a debit transaction (*utc2_ATS*).

Unit Test Case *utc1_ATS*: Credit Transaction First, the unit test harness initializes the ATS component with the state *q1_ATS*, so that *Acc_3* exists with a balance of 100€ in the account set managed by ATS, where 100€ is a magic number specified by the unit test case *utc1_ATS*. The unit test harness then sends the stimulus message *m_s1_ATS* via ATS's incoming interface *if_in1_ATS*, indicating that 300€ have been credited to *Acc_3*. This stimulus message *m_s1_ATS* triggers ATS's component behavior *cb1_ATS*, resulting in the updated state *q2_ATS* and two reaction messages: *m_r1_ATS* sent by ATS's outgoing interface *if_out1_ATS* (notifying that *Acc_3* was credited 300€ with an updated balance of 400€) and *m_r2_ATS* sent by ATS's outgoing interface *if_out2_ATS* (confirming the credit and updated balance of 400€). The assertion *assert1_ATS*, of the unit test case *utc1_ATS*, ensures that the balance for *Acc_3* is updated by ATS correctly.

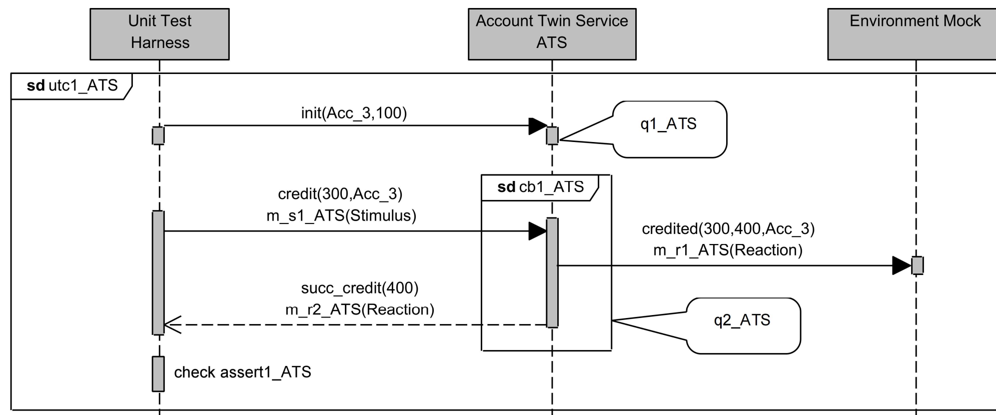


Figure 7.5.: Sequence diagram of the Account Twin Service's unit test case *utc1_ATS* (credit transaction).

Unit Test Case *utc2_ATS*: Debit Transaction First, the unit test harness initializes the ATS component with the state *q3_ATS*, so that *Acc_4* exists with a balance of 600€ in the account set maintained by ATS, where 600€ is a magic number specified by the unit test case *utc2_ATS*. The test harness then sends the stimulus message *m_s2_ATS* via ATS's incoming interface *if_in1_ATS*, indicating that 400€ should be debited from

Acc_4. This stimulus message *m_s2_ATS* triggers ATS's component behavior *cb2_ATS*, resulting in the updated state *q4_ATS* and two reaction messages: *m_r3_ATS* sent by ATS's outgoing interface *if_out1_ATS* (notifying that *Acc_4* was debited 400€ with an updated balance of 200€) and *m_r4_ATS* sent by ATS's outgoing interface *if_out2_ATS* (confirming the debit and updated balance of 200€). The assertion *assert2_ATS*, of the unit test case *utc2_ATS*, ensures that the balance for *Acc_4* is updated by ATS correctly.

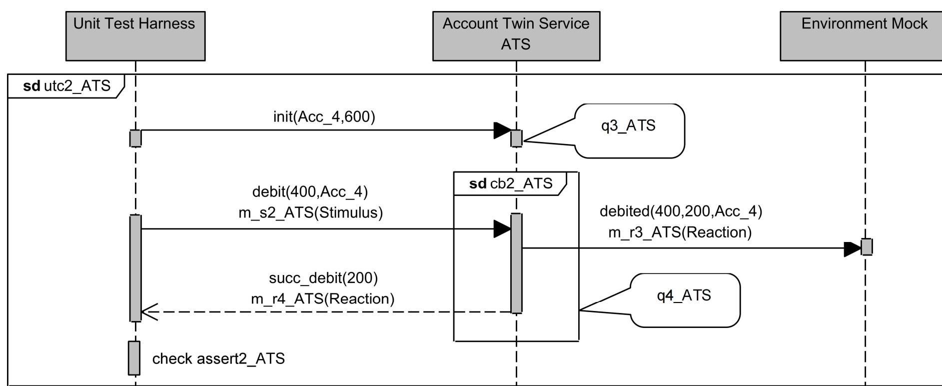


Figure 7.6.: Sequence diagram of the Account Twin Service's unit test case *utc2_ATS* (debit transaction).

7.3.3. Notification Service

Powered By Visual Paradigm Community Edition

The Notification Service (NS) is responsible for notifying users when an account transaction occurs, provided they have subscribed to receive such notifications. To construct its IBI component model, one unit test case, referenced as *utc1_NS*, is considered (see Figure 7.7).

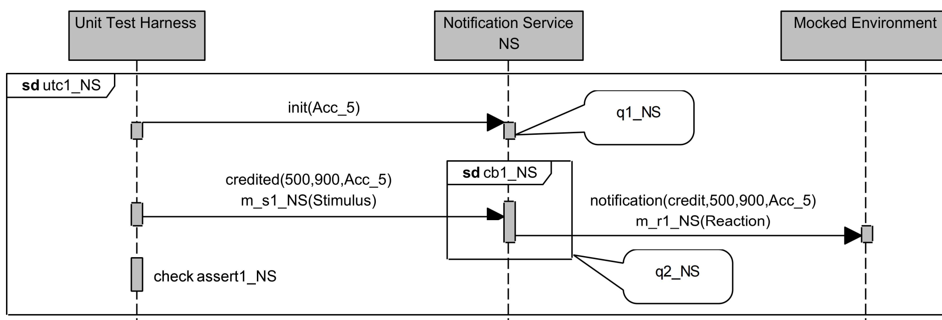


Figure 7.7.: Sequence diagram of the Notification Service's unit test *utc1_NS*.

Powered By Visual Paradigm Community Edition

First, the unit test harness initializes the NS component with the state *q1_NS*, so that notifications for transactions involving *Acc_5* are enabled. The test harness then sends the stimulus message *m_s1_NS* via NS's incoming interface *if_in1_NS*, indicating that 100€ have been credited to *Acc_5*, resulting in a new balance of 600€. This stimulus message *m_s1_NS* triggers the component behavior *cb1_NS*, resulting in the updated state *q2_NSS* and the reaction message *m_r1_NS*, which is sent by NS's outgoing interface *if_out1_NS*. The reaction message *m_r1_NS* is used in the assertion *assert1_NS*, of the unit test case *utc1_NS*, to ensure that *m_r1_NS* contains the transaction details (affected account, credited amount and updated balance).

The IBI component models created for the components ATS and NS look similar to the shown IBI component model for the TG component (see Figure 7.4).

7.3.4. Create the IBI System Models

The IBI system models for the Piggy Bank Systems (PBSs) *PBS_1* and *PBS_2* are constructed by establishing interface bindings between components.

The IBI system model bindings for *PBS_1* are visualized in Figure 7.8. For *PBS_2* it looks similar but the NS component as well as its incoming interface *if_in1_NS* and outgoing interface *if_out1_NS* are not included. Thus, the binding between ATS's outgoing interface *if_out1_ATS* and NS's incoming interface *if_in1_NS* is not contained either.

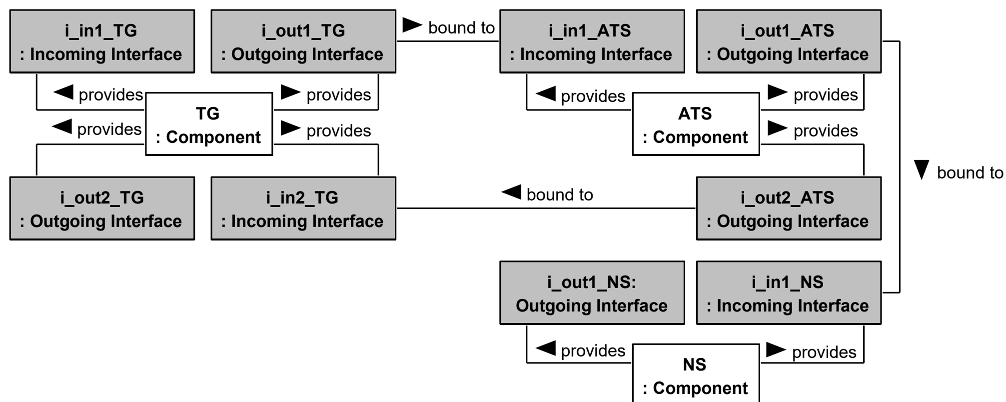


Figure 7.8.: Interface bindings for *PBS_1*.

The IBI system models for *PBS_1* and *PBS_2* form the foundation for deriving interaction expectations and generating interaction test cases in subsequent steps.

7.4. Collect Interaction Expectations

Next, unit test-based interaction expectations are extracted, and the abstract interaction expectations are resolved using the IBI component models. This is demonstrated using the environment interaction *envi1_TG* of TG's unit test case *utc1_TG* as an example.

7.4.1. Unit Test-based Interaction Expectations

The environment interaction *envi1_TG* in TG's unit test case *utc1_TG* is started by the component behavior result that consists of the reaction message *m_r1_TG* and results in the environment sending the stimulus message *m_s2_TG*. Since *m_s2_TG* is received by TG's incoming interface *if_in2_TG*, the unit test-based interaction expectation *utie1_TG* is extracted that states that components in the environment of the Transfer Gateway (Env_{TG}) need to react to *m_r1_TG* such that a reaction message is sent via an outgoing interface that is bound to TG's incoming interface *if_in2_TG*. This message needs to be a replacement for the original environment response *m_s2_TG* so that the assertion *assert1_TG* of TG's unit test case *utc1_TG* is still satisfied. The extracted unit test-based interaction expectation *utie1_TG* is given by the tuple:

$$\mathit{utie1_TG} = ((\underbrace{\underbrace{\{(\mathit{m_r1_TG}, \mathit{if_out1_TG})\}}_{R}, \underbrace{\{\mathit{q2_TG}, ?q_{SUT \setminus TG}\}}_{q_{\text{SUT}}})}_{sbt}, \underbrace{\{\mathit{if_in2_TG}\}}_{IF_{\text{in,exp}}}, \underbrace{\mathit{assert1_TG}}_{assert})$$

7.4.2. Derived Interaction Expectations

The abstract interaction expectation *aie1* is resolved by selecting appropriate stimulus messages in the IBI component models. Therefore, an incoming interface that is bound to the stimulus interface expectation needs to be identified. For this example TG's incoming interface *if_in1_TG* is the only matching interface. The stimulus message *m_s1_TG* that was received is a valid transfer and is thus selected by the stimulus message selector. This results in the derived interaction expectation *die1*:

$$\mathit{die1} = ((\underbrace{\underbrace{\{(\mathit{m_s1_TG}, \mathit{if_out_aie1})\}}_{R}, \underbrace{?q_{SUT}}_{q_{\text{SUT}}})}_{sbt}, \underbrace{\{\mathit{if_in_aie1}\}}_{IF_{\text{in,exp}}}, \underbrace{\mathit{assert_aie1}}_{assert})$$

7.5. Derive Component Behavior Sequences

Now that the interaction expectations are defined, potential component behavior sequences that satisfy these expectations are determined. To derive concrete system behaviors that may fulfill interaction expectations, potential behavior graphs are constructed based on the captured component behaviors for components in

PBS_1 and *PBS_2* respectively.

For *PBS_1*, the following interface bindings exist:

- The interface binding between *if_out1_TG* and *if_in1_ATS*.
- The interface binding between *if_out2_ATS* and *if_in2_TG*.
- The interface binding between *if_out1_ATS* and *if_in1_NS*.

Since two different behaviors for ATS can be triggered by a message that is received on incoming interface *if_in1_ATS*, two potential behavior graphs are constructed for *PBS_1* and the derived interaction expectation *die1*. The two potential behavior graphs are depicted in Figure 7.9 and Figure 7.10.

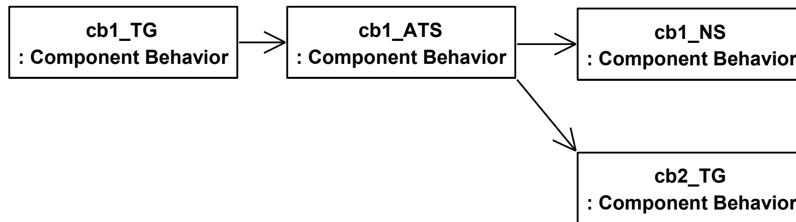


Figure 7.9.: First PBG for *die1* in *PBS_1*.

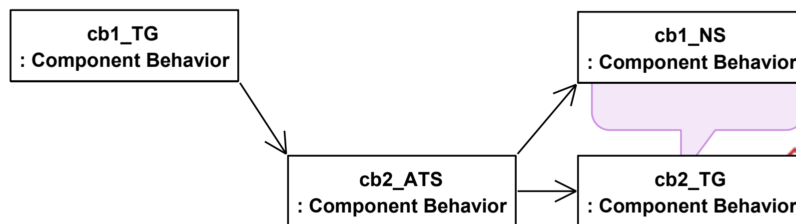


Figure 7.10.: Second PBG for *die1* in *PBS_1*.

Since *PBS_2* does not include the Notification Service no suitable graph can be constructed for the derived interaction expectation *die1*. For *utie1_TG* the potential behavior graphs are the same for both PBSs and consist of a single component behavior *cb1_ATS* and *cb2_ATS* respectively as shown in Figure 7.11 and Figure 7.12.

Figure 7.11.: First PBG for *utie1_TG*.

Figure 7.12.: Second PBG for *utie1_TG*.

The potential component behavior sequences to verify each interaction expectation are derived from those potential behavior graphs. Table 7.13 summarizes the potential component behavior sequences found in the IBI system models.

	PBS_1	*PBS_2*
utie1_TG	(*cb1_ATS*) (*cb2_ATS*)	(*cb1_ATS*) (*cb2_ATS*)
die1	(*cb1_TG*, *cb1_ATS*, *cb1_NS*) (*cb1_TG*, *cb2_ATS*, *cb1_NS*)	–

Table 7.13.: Potential component behavior sequences in *PBS_1* and *PBS_2* for interaction expectations *utie1_TG* and *die1*.

For the unit test-based interaction expectation *utie1_TG* the same potential component behavior sequences, $cbseq_1 = (cb1_ATS)$ and $cbseq_2 = (cb2_ATS)$, are identified for both PBSs. For *PBS_1*, two potential component behavior sequences, $cbseq_3 = (cb1_TG, cb1_ATS, cb1_NS)$ and $cbseq_4 = (cb1_TG, cb2_ATS, cb1_NS)$, are identified for the derived interaction expectation *die1*. No valid component behavior sequence is found for *PBS_2* because the Notification Service, required to fulfill the derived interaction expectation *die1*, is not part of the system.

7.6. Generate & Execute Interaction Test Suites

The identified potential component behavior sequences $cbseq_1 \dots cbseq_4$ are used to generate interaction test suites that are executed to verify these component behavior sequences.

For *PBS_1*, the component behavior sequences $cbseq_3$ and $cbseq_4$ correspond to the derived interaction expectation *die1*, while $cbseq_1$ and $cbseq_2$ correspond to the unit test-based interaction expectation *utie1_TG* for *PBS_1* and *PBS_2*. To avoid redundancy, the interaction test cases for *cb1_TG* in these component behavior sequences are omitted since the component behavior is already verified by TG's unit test case *utc1_TG*.

The interaction test cases for *cb1_ATS* in $cbseq_1$ and $cbseq_3$ are identical, as are the test cases for *cb2_ATS* in $cbseq_2$ and $cbseq_4$. Consequently, only two unique interaction test cases are generated:

- One based on the unit test case *utc1_ATS* for *cb1_ATS*, using *m_r1_TG* as the stimulus message.
- One based on the unit test case *utc2_ATS* for *cb2_ATS*, also using *m_r1_TG* as the stimulus message.

Component states *q1_ATS* and *q3_ATS* are adjusted to *q1'_ATS* and *q3'_ATS* such that *Acc2* is included in the set of accounts before executing the interaction test cases

based on *m_r1_TG*.

During execution, the interaction test case based on *cb2_ATS*, depicted in Figure 7.14, fails because *m_r1_TG* corresponds to a credit scenario rather than a debit scenario. As a result, $cbseq_2$ and $cbseq_4$ are discarded as invalid component behavior sequences.

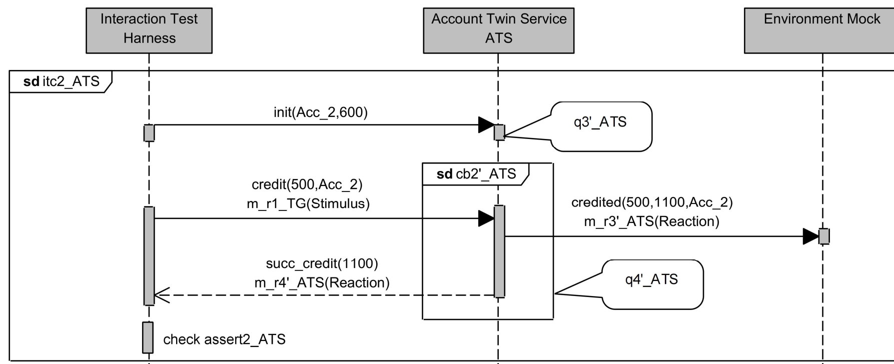


Figure 7.14.: Interaction test based on *cb2_ATS*.

Conversely, the interaction test case for *cb1_ATS* succeeds, resulting in component behavior *cb1'_ATS* resulting in two new reaction messages: *m_r1'_ATS* and *m_r2'_ATS*, as shown in Figure 7.15. This confirms $cbseq_1$ as a valid component behavior sequence.

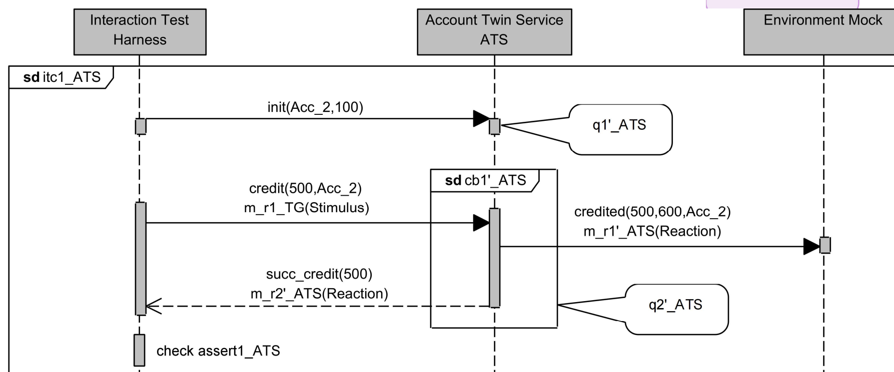


Figure 7.15.: Interaction test based on *cb1_ATS*.

For $cbseq_3$, to verify the derived interaction expectation *die1*, one additional interaction test case is generated, replacing the original stimulus message *m_s1_NS* with *m_r1'_ATS*. The successful execution of this test produces the reaction message *m_r1'_NS*, as illustrated in Figure 7.16.

Powered By Visual Paradigm Community Edition

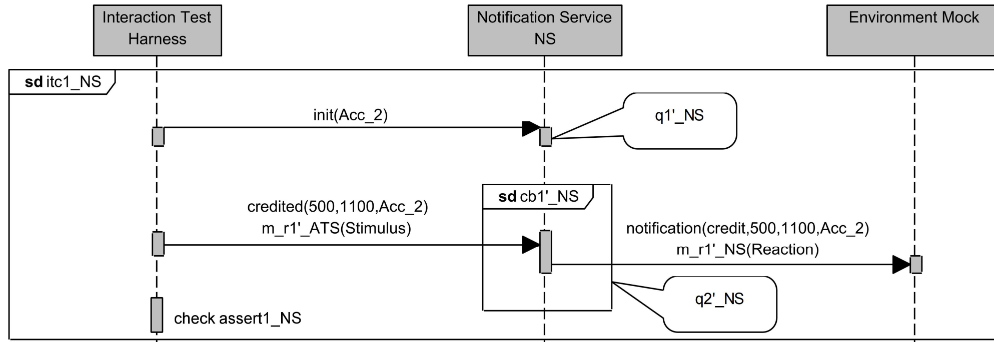


Figure 7.16.: Interaction test based on *cb1_NS*.

7.7. Verify Interaction Expectations

Once the interaction test cases have been executed, the final step is to verify the interaction expectations by assessing whether the observed system behavior aligns with the defined expectations.

Powered By Visual Paradigm Community Edition

For the unit test-based interaction expectation *utie1_TG*, the execution of component behavior sequence $cbseq_1$ has already been confirmed as successful, producing reaction messages *m_r1'_ATS* and *m_r2'_ATS*. To verify this expectation, a final interaction test is generated by replacing the mocked response *m_s2_TG* in the Transfer Gateway's unit test with the response message *m_r2'_ATS*. This interaction test executes successfully, confirming that the unit test-based interaction expectation holds for both *PBS_1* and *PBS_2*.

For the derived interaction expectation *die1*, verification is performed by asserting that the reaction message *m_r1'_NS* correctly reflects the original transfer request sent to the system. This assertion ensures that the amount and credited account in the notification match the details of the initial transfer. Since the interaction test case based on *cb1_NS* has been successfully executed, this expectation is also verified for *PBS_1*. However, in *PBS_2*, no such verification is possible because the interface *if_in1_NS* does not exist in this configuration. This indicates that the derived interaction expectation *die1* remains unfulfilled, highlighting a missing component or test coverage gap in *PBS_2*.

The results of these verifications are summarized in Table 7.17.

Expectation Type	**Verified for *PBS_1***	**Verified for *PBS_2***
Unit Test-based Interaction Expectation	Yes ($cbseq_1$)	Yes ($cbseq_1$)
Derived Interaction Interaction Expectation	Yes ($cbseq_3$)	No

Table 7.17.: Verification results for interaction expectations.

7.8. Create Report

After the verification of the interaction expectations, a report is generated to summarize the results of the IBI Testing Process. This report provides a structured overview of the verified interaction expectations, failed tests, and potential test gaps.

For *PBS_1*, the report confirms that both the unit test-based and derived interaction expectations were successfully verified. The system configuration meets the defined expectations, ensuring that the integration behaves as intended. The report includes:

- A list of verified expectations, confirming correct component interactions.
- The executed interaction test cases with observed results.
- The integration test that is simulated by those interaction test cases as depicted in Figure 7.18.

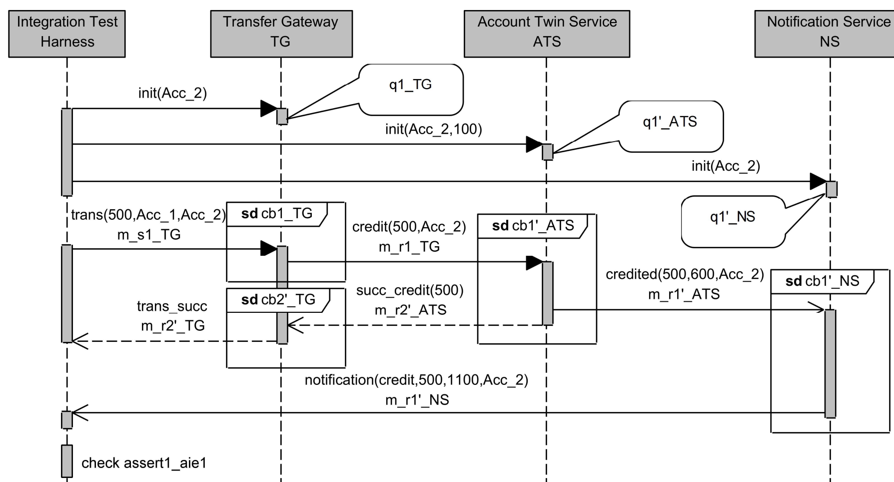


Figure 7.18.: The integration test case simulated by the generated interaction test cases.

For *PBS_2* the abstract interaction expectation *aie1* could not be verified, since no interface matching the reaction interface expectation could be found indicating that some component or test case that covers that interface is missing. In this case a developer could use the report of *PBS_1* to identify that the interaction expectation would require the Notification Service to be part of the system and either decide to drop the expectation or add the Notification Service to *PBS_2*.

Powered By: Visual Paradigm Community Edition

Chapter 8.

InterACt: A Tool for Automated Interaction-based Integration Testing

Table of Contents

8.1. Implementation Goals 93
8.2. Architecture Overview 94
8.3. Realization 96
8.4. Future Enhancements 107

The IBI Testing Process automates integration testing by generating interaction test suites from existing unit test suites. These interaction test suites verify component interactions and thereby system behavior. This chapter introduces InterACt[1], a Proof of Concept (PoC) implementation of the IBI Metamodel and IBI Testing Process, developed to demonstrate and evaluate their feasibility in line with the DSR methodology.

8.1. Implementation Goals

The implementation of InterACt was guided by the following goals and requirements:

- **Automated Model Construction:** IBI component and system models must be constructed automatically from executed unit test cases, minimizing manual intervention.
- **Interaction Expectation Validation:** The implementation must support both deriving interaction expectations from unit test cases and verifying interaction expectations against observed system behavior.
- **Non-intrusiveness:** The system must observe component behavior without modifying the production code or requiring the introduction of test-specific logic into the CUTs.
- **Extensibility:** InterACt must be extensible with respect to:

1. InterACt is publicly available at `https://github.com/NilsWild/InterACt`

  - Supported test frameworks (e.g., JUnit 5, PyTest),
  - Communication protocols (e.g., REST, AMQP).

- **Traceability:** There must be clear traceability between the unit test cases, the observed component behavior, and the derived interaction test cases to support verification, debugging, and coverage analysis.

8.2. Architecture Overview

InterACt follows a modular architecture depicted in Figure 8.1, where each component contributes to one or multiple actions of the IBI Testing Process:

- **Test Framework Adapter:** A Test Framework Adapter is responsible for integrating with the underlying unit test framework. The Test Framework Adapter receives interaction test cases from the Test Execution Manager and passes them to the Test Framework for execution. It also hooks into the test lifecycle to report test identifiers, input parameters, and results to the Test Execution manager.
- **Interface Observer:** An interface observer captures the interactions with the CUT during test execution. This includes messages sent and received by the CUT as well as metadata about the interfaces used. The observed interactions are reported to the Test Execution Manager.
- **Test Execution Manager:** The Test Execution Manager is the central coordination component during test execution. It requests interaction tests from the Integration Controller, tracks test case execution, correlates the observed interactions with unit and interaction test cases to derive component behavior, and constructs IBI component models that are sent to the Integration Controller after test completion. The Test Execution Manager also ensures that all interactions have been captured before allowing a unit or interaction test case to complete.
- **Integration Controller:** The Integration Controller is the central component that stores and analyzes the IBI models in the IBI Model Store. It constructs the IBI system model from submitted IBI component models, extracts interaction expectations, generates interaction test cases, and verifies the interaction expectations.
- **User Interface (UI):** The UI provides a visual frontend for insights into system behavior, verification results, and test coverage. It supports the user in analyzing interaction test results and identifying test gaps.

Each of these components contributes to a different step of the IBI Testing Process, from test execution and data capture to interaction expectation verification and reporting.

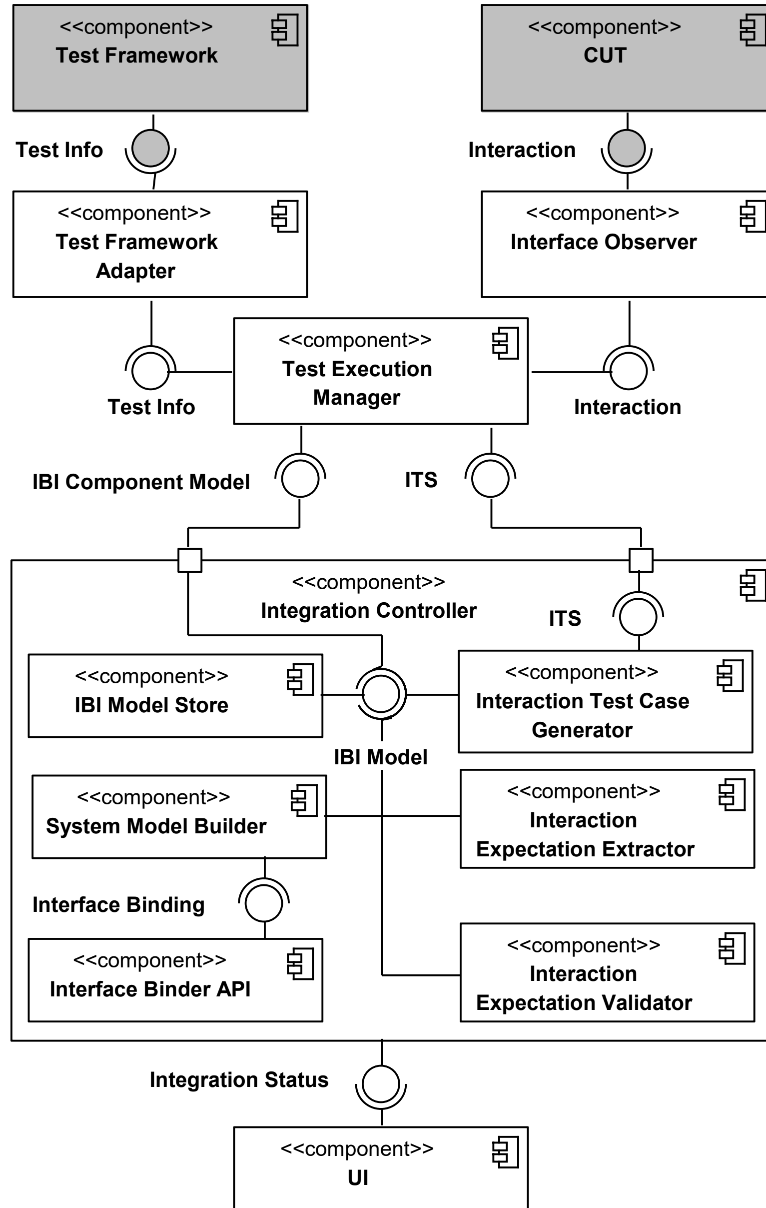


Figure 8.1.: Overview of InterACt's architecture.

8.3. Realization

While the architecture emphasizes modularity and extensibility, the current implementation scope of InterACt is limited to JUnit 5 and components realized using the Spring Framework. This scope reflects the goal of demonstrating the feasibility and practical application of the IBI Testing Process. Such an initial technology focus is typical for tools that need to integrate with heterogeneous systems implemented across different languages and architectures, as seen in the evolution of systems such as Kieker, Dynatrace, and SonarGraph. For instance, Kieker, a monitoring framework, documented the significant complexity of supporting diverse technologies and the incremental broadening of its scope over time [HvH15]. Similar observations have been made for tools like Dynatrace [Dyn] and SonarGraph [hGmb]. Initial implementations focused on specific technology ecosystems before expanding to broader support.

InterACt is primarily implemented in Kotlin. The Integration Controller is built using the Spring Framework and stores the IBI models in a Neo4j graph database, which realizes the IBI Model Store. Neo4j was chosen due to its native support for property graphs, which aligns with the structure of the IBI models. While Cypher, Neo4j's query language, cannot express all regular path queries (RPQs) due to theoretical limitations in pattern-matching semantics [GLP+25], its current feature set suffices for InterACt's integration analysis requirements. Neo4j is actively evolving Cypher toward GQL compliance - the new ISO standard for graph databases - ensuring future compatibility with advanced RPQ features while preserving existing functionality [Neo24]. The UI component is developed using *Angular*, providing a browser-based frontend for integration analysis. A section of the IBI system model of the Piggy Bank system, as stored in Neo4j, that includes the components, their interfaces, and interface bindings, is shown in Figure 8.2.
The current implementation also includes:

- A **JUnit 5 Test Framework Adapter** that integrates with JUnit's test lifecycle,
- Multiple **Interface Observers** tailored to Spring technologies, including WebMVC, WebClient, RestTemplate, and Spring AMQP,
- A **Test Execution Manager** implementation in Kotlin that coordinates test execution and IBI model capture.
- Interface Binders for REST and AMQP protocols that bind interfaces during IBI system model construction.

The following subsections describe the implementation of each component in detail.

8.3.1. JUnit 5 Test Framework Adapter

The JUnit 5 Test Framework Adapter is realized as a JUnit 5 extension integrating InterACt with JUnit 5. It builds upon JUnit's parameterized test capabilities, treating

Figure 8.2.: A section of the IBI system model of the Piggy Bank system, as stored in Neo4j.

each test template method as an abstract test case.

InterACt introduces the `@InterACtTest` annotation that inherits from JUnit's `@TestTemplate` annotation and registers the `InterACtTestsExtension` (see Listing 1), to mark a test method to be instrumented by InterACt.

Listing 1 The `@InterACtTest` annotation.

```
@Retention(AnnotationRetention.RUNTIME)
@Target(AnnotationTarget.FUNCTION,
 ↪  AnnotationTarget.ANNOTATION_CLASS)
@TestTemplate
@ExtendWith(InterACtTestsExtension::class)
annotation class InterACtTest(
  ...
)
```

The arguments of any InterACt test method are the stimulus messages followed by optional expected values as follows:

```
testmethod(stimulus_1,..., stimulus_n, expectedValue_1,...,
                     expectedValue_n)
```

For unit test cases, the arguments, including both stimulus messages and expected values, can be provided using standard argument sources such as *@MethodSource*

or *@CSVSource*. For interaction test cases, the `InterACtTestsExtension` requests the interaction test suite from the `TestExecutionManager`. Since interaction test cases are intended to verify the interactions between components and not the logic of individual components in detail, they do not include expected values, since they require domain knowledge and can not be generated for arbitrary inputs. Therefore, the JUnit 5 Test Framework Adapter assigns `null` to all expected value parameters when executing interaction test cases.

The Test Framework Adapter then creates `TestTemplateInvocationContext` instances for each test case. Each `TestTemplateInvocationContext` registers an `InterACtTestInvocationInterceptor`, which integrates with JUnit 5's lifecycle. The interceptor provides the `TestExecutionManager` with all relevant data needed to trace each test case, including the test class, test method name, test type (unit or interaction), input arguments, argument values, and lifecycle events.

To avoid unintended assertion failures due to the `null` values for expected values, the `InterACtTestInvocationInterceptor` handles assertions that rely on those expected values separately from those that do not. The JUnit 5 Test Framework Adapter therefore provides two lambda functions. Assertions wrapped in the lambda function `inherently` are checked during unit tests as well as interaction tests and should thus not rely on expected values. Assertions wrapped in the lambda function `forExample` are only checked during unit tests and can rely on expected values. For instance:

```
inherently –
  assertEquals(response.statusCode, 200)
”
forExample –
  assertEquals(account.balance, expectedBalance)
”
```

The test results are passed back to the `TestExecutionManager` for further processing.

Since the JUnit 5 Test Framework Adapter is built on top of JUnit 5's extension model, it can be easily integrated into existing JUnit 5 test suites.

8.3.2. Interface Observers

Interface observers are responsible for capturing the interactions with the CUT during test execution. These interactions include the messages sent and received by the CUT, along with metadata describing the interfaces involved. Each interface observer is tailored to a specific communication protocol and framework, ensuring that InterACt can accurately monitor interactions across different technologies and frameworks.

Captured data is immediately forwarded to the Test Execution Manager to be associated with the currently executing test case. Since InterACt currently does not

support parallel test execution, it assumes that all messages received during a test execution belong to the active test case as reported by the Test Framework Adapter.

The following observers are implemented as part of InterACt.

Spring WebMVC Interface Observer

The Spring WebMVC interface observer captures interactions between the CUT and the test harness in a Spring WebMVC environment. It hooks into the Spring WebMVC framework to capture the interaction with the CUT during test execution.

To capture incoming interfaces and the messages received thereby, the observer utilizes a `RequestBodyAdviceAdapter` when the messages contain a body. For requests without a body, a `HandlerInterceptor` is used instead. Additionally, a `OncePerRequestFilter` is employed to capture responses sent by the CUT.

The observer extracts and records interface metadata, including:

- HTTP verb (e.g., `GET`, `POST`, `PUT`).
- URL template and path variables.
- Whether the interface is a request or response interface. For REST, two interfaces are created for each request-response pair. One for the request and one for the response.

Spring AMQP Interface Observer

The Spring AMQP interface observer captures interactions between the CUT and its environment in a Spring AMQP context. Unlike the WebMVC interface observer, it does not rely solely on the Spring framework. Instead, it leverages the RabbitMQ Firehose Tracer plugin to capture all messages sent and received by the CUT and the relevant metadata about exchanges and queues.

To ensure that no messages are lost during test execution, the observer integrates with the Spring AMQP framework to track the number of sent and received messages. It registers a `MessagePostProcessor` for each `RabbitTemplate` bean in the application context, including the `RabbitTestTemplate`, which increments a counter for each message exchanged with the CUT.

In parallel, the Spring AMQP interface observer binds an observation queue to RabbitMQ's Firehose Tracer exchange, allowing it to collect the actual messages involving the CUT. Upon test completion, the observer verifies that the number of captured messages from the Firehose tracing queue matches the internal counter maintained by the `MessagePostProcessor`. Only after this verification succeeds the observer signals to the Test Execution Manager that message observation is complete.

Spring RestTemplate Interface Observer

The Spring RestTemplate interface observer captures the interactions between the CUT and components in its environment, which are typically represented by mocks, in a Spring RestTemplate environment. It hooks into the Spring RestTemplate framework to capture messages sent and received by the CUT during test execution.

To capture outgoing requests and their corresponding responses, the observer registers a `ClientHttpRequestInterceptor` for each `RestTemplate` bean in the application context. The `ClientHttpRequestInterceptor` intercepts HTTP requests and responses, extracts relevant interaction data, and forwards it to the Test Execution Manager for further processing.

Spring WebClient Interface Observer

The Spring WebClient interface observer captures the interactions between the CUT and components in its environment, which are typically represented by mocks, in a Spring WebClient environment. It hooks into the Spring WebClient framework to capture messages sent and received by the CUT during test execution.

To capture outgoing requests and their corresponding responses, the observer registers an `ExchangeFilterFunction` for each `WebClient` bean in the application context. The `ExchangeFilterFunction` intercepts and extracts relevant request and response information, which is then forwarded to the Test Execution Manager for further processing.

8.3.3. Test Execution Manager

The Test Execution Manager is the core runtime component that links test case execution with observed component behavior. It orchestrates communication between the Test Framework Adapter, interface observers, and the Integration Controller. Its responsibilities include:

- **Interaction Test Suite Retrieval:** Requests interaction test suites from the Integration Controller and forwards them to the Test Framework Adapter.
- **Test Tracking:** Tracks the execution lifecycle of each test case and associates metadata such as test identifiers and parameters.
- **Interaction Correlation:** Correlates the interactions observed by interface observers with the test case currently in execution and derived the component behavior from those interactions.
- **Capture Completion:** Verifies that all interactions have been captured before allowing a test case to complete.
- **IBI Model Construction:** After execution, constructs the corresponding IBI component model and sends it to the Integration Controller.

For the PoC implementation, the Test Execution Manager interacts with both the Test Framework Adapter and interface observers via direct method calls. It is implemented in Kotlin, as it is the same language used for the JUnit 5 Test Framework Adapter and interface observers.

Unit and Interaction Test Execution Process

The execution of unit and interaction test cases and the collection of IBI component models follow a structured process:

Step 1: Request Interaction Test Suite When a test is triggered, the Test Framework Adapter requests an interaction test suite. The Test Execution Manager forwards this request to the Integration Controller and supplies the retrieved interaction test suite to the Test Framework Adapter.

Step 2: Setup the Test Execution Context The Test Framework Adapter informs the Test Execution Manager when a test case starts, allowing it to prepare for capturing interactions.

Step 3: Derive Interaction Collection and Behavior Interface observers capture interactions with the CUT, including messages and interface metadata, during test case execution. The Test Execution Manager associates these interactions with the ongoing test case and derives the corresponding component behavior.

Step 4: Ensure Observers Finished Interaction Capturing Before test completion, the Test Framework Adapter queries the Test Execution Manager to confirm whether all interactions have been captured. The Test Execution Manager consults the interface observers to ensure that interaction capture is complete.

Step 5: Construct and Submit Component Model Once a test case finishes, the Test Execution Manager extends the IBI component model with the captured component behaviors. When all test cases finished the Test Execution Manager submits the IBI component model to the Integration Controller. The Integration Controller then processes the IBI component model, extracting interaction expectations and generating interaction test cases as needed.

8.3.4. Integration Controller

The Integration Controller is responsible for processing IBI component models submitted by the Test Execution Manager, maintaining the IBI system model, extracting and verifying interaction expectations, and generating appropriate interaction test cases. It follows a modular and event-driven architecture to support extensibility. The Integration Controller consists of the following subcomponents corresponding to the components shown in Figure 8.1:

- **IBI Model Store:** Stores the incoming IBI component models persistently. It emits Spring application events whenever a new or updated IBI component model is stored.
- **System Model Builder:** Listens for IBI component model events and constructs the IBI system model by resolving interface bindings between component interfaces. Interface bindings are determined via Interface Binders that implement the Interface Binder API.
- **Interaction Expectation Extractor:** Listens for IBI component model events and extracts interaction expectations from the updated IBI system model. When new interaction expectations are extracted, it emits an event to notify other components.
- **Interaction Expectation Validator:** Listens for both IBI component model and interaction expectation events. It verifies whether the system behavior fulfills the defined interaction expectations based on the current IBI system model.
- **Interaction Test Case Generator:** Listens for interaction expectation and IBI component model events. It generates interaction test cases that simulate system behaviors necessary to verify the extracted interaction expectations.

Communication between the subcomponents is realized using Spring's *ApplicationEvent* mechanism, ensuring loose coupling.

Interface Binder API

InterACt provides an Interface Binder API, to support arbitrary communication protocols. This API defines a contract for protocol-specific implementations that determine how component interfaces should be bound during IBI system model construction.

Interface binders are implemented as Spring components and are automatically discovered and invoked by the system model builder during the construction of IBI system models. This modular approach allows specialized binders to handle complex or domain-specific protocols.
Current Interface Binder API implementations include:

- **REST Interface Binder:** Binds interfaces based on HTTP methods, URL templates, and request/response directionality.
- **AMQP Interface Binder:** Binds interfaces using AMQP metadata such as exchange type, routing key, and queue bindings.

Realization of the Formalization in Cypher

The formalization of the IBI Testing Process defines several RPQs that are used to drive the model construction and verification process. These RPQs are realized in the Neo4j graph database using Cypher queries. The queries are designed to retrieve relevant information from the IBI component and system models stored in the database.

To illustrate how the RPQs are realized technically, we begin with a simple example. The formal definition of the set of incoming interfaces $IF_{\text{in},c}$ of a component c is given in Section 6.2.3 as:

$$IF_c = (c, \textit{provides}, ?)$$
$$IF_{\text{in},c} = \{if \mid if \in IF_c \wedge \lambda_N(if) = \textit{IncomingInterface}\}$$

The corresponding Cypher query retrieves all `IncomingInterfaces` that are provided by a component with id `$compId`:

Listing 2 Query to retrieve all incoming interfaces of component with id `$compId`.

```
MATCH (c:Component-id:$compId˝)-[:PROVIDES]-¿(if˙inc:IncomingInterface)
RETURN if˙inc
```

This directly reflects the formalization by filtering on both the `PROVIDES` relationship and the `IncomingInterface` label.

Another example is given by a more advanced query that operationalizes the concept of potential component behavior sequences, introduced in Section 6.3.4. Although InterACt's Neo4j-based implementation does not explicitly represent component behaviors as individual nodes, it encodes the same semantics through the relationships between unit tests, messages, interfaces, and interface bindings.

Specifically:

- A component behavior is represented by a unit test triggering a sequence of messages, modeled via `TRIGGERED_MESSAGES` relationships from a `UnitTest` node to `Message` nodes.
- The edges between component behaviors are derived by traversing from a message to the interface it was sent or received by (`SENT_BY` or `RECEIVED_BY`), following the `BOUND_TO` relationship to the corresponding receiving or sending interface, and then to the next triggered message.
- The system stimulus of an interaction expectation is linked to the interaction expectation via the `EXPECT_FROM` relationship, modeling the starting point for verifying a potential component behavior sequence.

Based on this graph structure, the following query illustrates how the formalization is realized in the implementation. Given a unit test identifier `$testId`, the following query identifies for which interaction expectations the potential behavior graphs might be impacted by the new component behaviors verified by the unit test. It does so by traversing the component behaviors and detecting whether they could be triggered by a system stimulus defined in an interaction expectation. The query is as follows:

Listing 3 Query to determine interaction expectations affected by a new unit test case.

```
MATCH (c:UnitTest -id: $testId˝)
CALL apoc.path.subgraphNodes(c, -
  relationshipFilter:
  ↪ "¿TRIGGERED˙MESSAGES–¿RECEIVED˙BY–¡BOUND˙TO–¡SENT˙BY–¡TRIGGERED˙MESSAGES"
˝) YIELD node
WITH [n IN nodes WHERE n:Message] AS messages
UNWIND messages AS message
MATCH (message)¡-[:EXPECT˙FROM]-(expectation)-[:DERIVED˙FROM˙TEST]-¿(s:UnitTest)
WHERE s.id ¡¿ $testId
RETURN expectation
```

The `apoc.path.subgraphNodes` function thereby generates the potential behavior graphs PBG by following the defined edge traversal pattern:

- From a triggered message, find the interface it was received by.
- Follow the `BOUND_TO` relationship to the corresponding outgoing interface the message could be sent by.
- Find the messages sent via this interface.
- Find the messages that triggered sending that message.
- Continue traversing until the system stimulus of an interaction expectation is reached. Every interaction expectation that is reached via this traversal has a system stimulus that could transitively trigger the component behavior verified by the new unit test case.

These queries illustrate how the formalized concepts are realized technically within InterACt, providing a concrete mapping from theoretical definitions to executable model operations.

8.3.5. User Interface

The User Interface (UI) provides an interactive, browser-based frontend for exploring the results of the IBI Testing Process. It is implemented in *Angular* and communicates with the backend via a *GraphQL API*.

The UI enables users to analyze interaction expectations, and assess the integration status of the SUT at a glance. Its goal is to make system behaviors, verification results, and test coverage transparent and easily accessible.

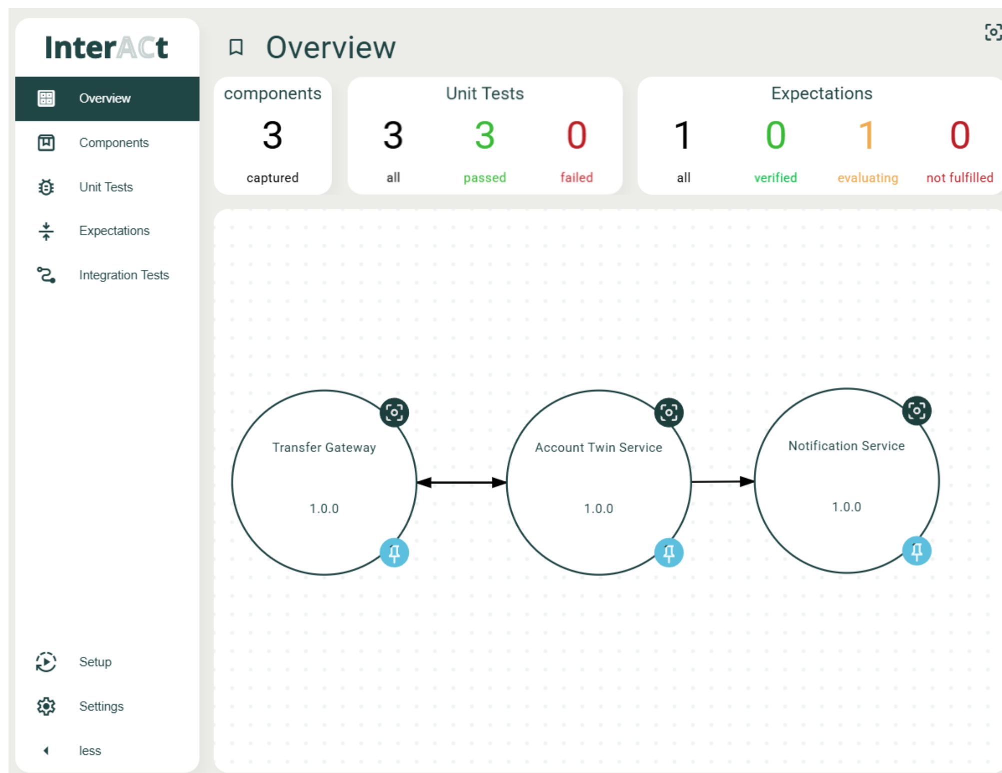


Figure 8.3.: System overview displaying detected components, test results, and expectation statuses.

System Overview View The system overview view provides a high-level representation of the system under test. As shown in Figure 8.3, it displays the following information:

- **Captured Components:** The UI lists all components that have been detected in the SUT.
- **Unit Test Results:** The UI summarizes the total number of executed unit tests along with their results (passed or failed).
- **Expectation Statuses:** The UI indicates the number of expectations that have been verified, are still under evaluation, or have not been fulfilled.

- **Graph-Based Visualization:** The detected components and their interactions are displayed as a directed graph, providing a representation of the system's structure.

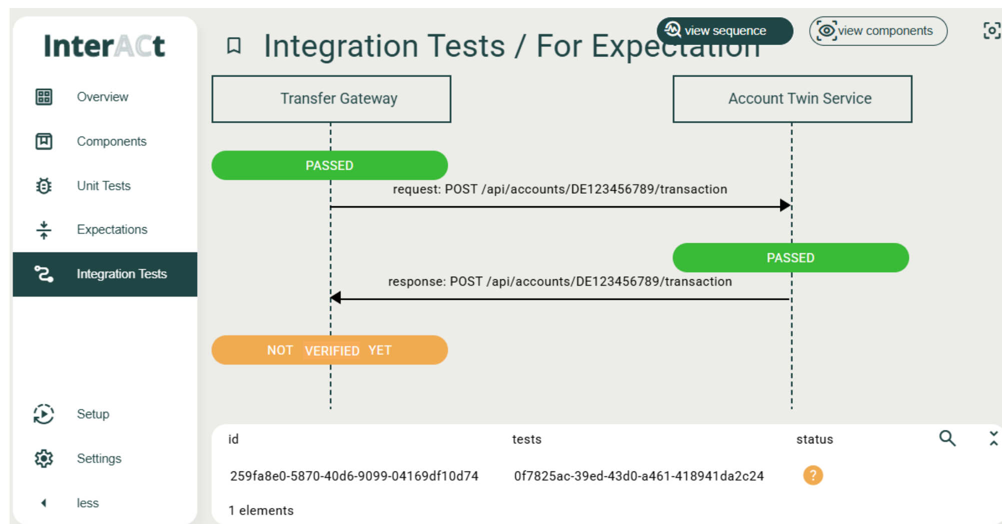


Figure 8.4.: Visualization of integration tests for an interaction expectation. Two interaction test cases have been passed already, and the third one is still pending verification.

Integration Test Visualization View To assess the correctness of component interactions, the UI visualizes interaction tests associated with interaction expectations. Figure 8.4 illustrates how integration tests are presented:

- **Expectation-Based Grouping:** Interaction tests are categorized based on the interaction expectations they verify.
- **Sequence Diagram Representation:** Each interaction test is visualized using a sequence diagram, showing the message flow between components.
- **Validation Status:** The UI provides feedback on whether an interaction test case has passed, failed, or is still pending verification.

The UI simplifies the process of analyzing integration results by providing a structured representation of both the system under test and its associated interaction expectations and test cases. While the current UI targets expert users, its design lays the groundwork for future enhancements such as test filtering, drill-down capabilities, and non-technical stakeholder reporting.

8.4. Future Enhancements

As a PoC implementation, InterACt focuses on demonstrating the IBI Testing Process. InterACt's application scope is currently limited to systems built with the Spring Framework and tested using JUnit 5. However, several enhancements can be made to extend the applicability, automation, and usability of InterACt in future work:

- **Support for Additional Test Frameworks:** The current implementation supports only JUnit 5. Additional Test Framework Adapters (e.g., for PyTest or TestNG) would allow InterACt to be used across a broader range of programming languages and ecosystems.
- **Enhanced Test Metadata Support:** The modular design of the Test Framework Adapter allows for extensions to capture and forward additional test metadata to the Test Execution Manager. For example, if frameworks like Instancio are used to generate randomized test data for unit test cases, the corresponding seed values could be provided.
- **Observer Parallelization and Tracing:** InterACt currently assumes that test cases are executed sequentially. Supporting parallel test execution requires robust correlation of interaction data with individual test cases. One potential solution is the integration of distributed tracing mechanisms, such as OpenTelemetry, to propagate trace identifiers through observed messages.
- **Expanded Protocol Support:** While the current implementation supports REST and AMQP through dedicated interface observers and binders, additional protocols (e.g., gRPC, Kafka, GraphQL) could be supported by implementing new interface observers and interface binders via the existing extension points.
- **Improved Interaction Test Selection Strategies:** The Integration Controller currently provides all generated interaction test cases to the Test Execution Manager. More selective test selection strategies could prioritize newly affected paths or paths that represented actual system behavior in previous versions of a system, improving feedback cycles.
- **Push-Based Execution Model:** The current architecture uses a pull-based mechanism, where the Test Framework Adapter requests interaction tests. A push-based alternative could be explored where the Integration Controller actively dispatches test cases or triggers test execution in CI pipelines.
- **UI Enhancements for Broader Usability:** While the current user interface targets expert users, its foundation enables extensions such as:
  - Meaningful naming of interaction tests,
  - Filtering and sorting of interaction tests,
  - Drill-down into test execution details,

- Exportable summaries and reports for non-technical stakeholders.

The GraphQL API allows to freely query the underlying data, enabling the development of custom dashboards or reports tailored to specific needs.

Part III.

Evaluation and Conclusion

Chapter 9.

Evaluation Approach

This part of the thesis evaluates the proposed IBI Testing Approach including the IBI Metamodel and IBI Testing Process, using its PoC implementation, InterACt.

The evaluation follows the principles of the DSR methodology [PTR+07], where the goal is to design, implement, and rigorously evaluate an artifact that addresses a relevant and practically motivated problem. Accordingly, the evaluation focuses on assessing the artifact — in this case, InterACt, the IBI Testing Process and IBI Metamodel — with respect to its conceptual soundness and practical applicability. To that end, the evaluation addresses the following objectives:

1. **Conceptual Validity:** Evaluate whether the IBI Testing Process, implemented by InterACt, can systematically derive and verify inter-component behavior from unit test data of component-based systems, captured in IBI models.

2. **Practical Applicability and Limitations:** Explore the applicability and limitations of the approach and its implementation in an industrial system.

To address these objectives, two complementary evaluation strategies were selected based on common Design Science evaluation methods [HMP+04] and the empirical study classification in software engineering proposed by Wohlin et al. [WRH+12]:

1. **Demonstration Case Study:** A demonstration case study using the microservice-based Piggy Bank system (introduced in Chapter 4) evaluates the fault detection capabilities of the IBI Testing Process under controlled conditions, including seeded integration faults. This provides evidence for conceptual validity and aligns with the demonstration activity defined in the DSR methodology.

2. **Exploratory Case Study:** An exploratory industrial case study investigates the applicability and limitations of InterACt and the underlying IBI Metamodel and IBI Testing Process when used in an existing system. This contributes to evaluating the practical utility in a real-world setting, consistent with the evaluation activities defined in the DSR methodology.

These two studies were chosen to provide both a controlled environment for evaluating the effectiveness of the IBI Testing Approach and a realistic context to identify

practical constraints and challenges. The following chapters present the evaluations in detail.

Chapter 10.

Demonstration Case Study

Table of Contents

10.1. Objectives 113
10.2. Case Study Design 114
10.3. Test Suite Execution 115
10.4. Results 116
10.5. Discussion 118
10.6. Threats to Validity 118

This chapter presents a demonstration case study that evaluates the IBI Testing Approach using the Piggy Bank system—a microservice-based system introduced in Chapter 4 as the basis for controlled evaluation. In contrast to the exploratory case study, the Piggy Bank system provides full control over the implementation and configuration of components, communication protocols, and expected behaviors. This controlled environment enables the targeted injection of integration faults and facilitates a systematic assessment of the conceptual soundness and fault detection capabilities of the IBI Testing Approach.

10.1. Objectives

The goal of this controlled evaluation is to systematically assess the conceptual soundness and fault detection capabilities of the IBI Testing Approach in a setting where all components, interactions, and faults are under complete control. The evaluation is based on predefined interaction expectations and deliberately injected integration faults in the Piggy Bank system.

The evaluation is guided by the following questions:

Q 1: Does the IBI Testing Approach correctly extract IBI component and system models, and the corresponding interaction expectations from InterACt unit test cases?

Q 2: Can the IBI Testing Approach detect integration faults that violate interaction expectations?

- Are the expected violations correctly detected across all faulty branches?
- Do the observed failures align with the seeded faults?

Q 3: Can the IBI Testing Approach verify correct integration behavior when all interaction expectations are fulfilled?

- Does the reference implementation lead to the successful verification of all interaction expectations?
- Are any false positives (i.e., incorrect failure detections) observed?

10.2. Case Study Design

To systematically evaluate the effectiveness of the IBI Testing Approach, a controlled setup was created using multiple branches of the Piggy Bank system. Each branch represents a distinct system configuration: one correctly functioning baseline and four variants containing deliberately seeded integration faults.

10.2.1. Demonstration System

The Piggy Bank demonstration system consists of five microservice components: the Transfer Gateway (TG), Account Twin Service (ATS), Transfer Classifier (TC), Goal Service (GS), and Notification Service (NS). Each component is implemented in Kotlin and interacts with others using HTTP (REST) or AMQP protocols.

To evaluate the IBI Testing Approach, each component is tested using InterACt unit test cases, which are JUnit 5 test cases annotated with the `@InterACt` annotation, following the conventions described in Chapter 8. These InterACt unit test cases cover both happy paths and unhappy paths, providing the component behaviors for extracting unit test-based interaction expectations. Table 10.1 summarizes the number of InterACt unit test cases and the corresponding interaction expectations expected to be extracted per component.

Table 10.1.: Overview of components, InterACt unit test cases, and unit test-based interaction expectations

Component	InterACt Unit Test Cases	Expected Unit Test-based Interaction Expectations
Transfer Gateway (TG)	18	4
Account Twin Service (ATS)	3	2
Transfer Classifier (TC)	4	0
Goal Service (GS)	5	0
Notification Service (NS)	16	0
Total	**46**	**6**

In addition to unit test-based interaction expectations, two *abstract interaction expectations* were defined to evaluate system-wide integration behavior that spans multiple components and communication protocols and can not be extracted from the InterACt unit test cases. These abstract interaction expectations were chosen because they share the same interfaces for both stimulus (Transfer Interface) and response (Notification Interface) but differ in the stimuli that are accepted by the message selector and the expected outcomes. As a result, verifying each abstract interaction expectation requires identifying distinct component behavior sequences, demonstrating the ability of the IBI Testing Approach to differentiate integration behaviors based on variations in system inputs.

aie1: If a transfer is made *to* a managed account, a notification should be sent confirming that the account was credited.

aie2: If a transfer is made *from* a managed account, a notification should be sent informing the account was debited.

These expectations span multiple components and communication protocols, making them suitable for evaluating the ability of the IBI Testing Approach to verify system-wide integration behavior.

10.2.2. Seeded Integration Faults

The `main` branch implements the fully functional system in which all integration paths conform to the specified interaction expectations. From this reference implementation, four additional branches were derived, each introducing a single integration fault. These faults follow the categories defined by Leung and White [LW90] and are designed to test different aspects of interface compatibility, communication correctness, and expectation verification. Table 10.2 summarizes the branches and their respective faults.

Each faulty branch introduces exactly one integration fault to ensure that the impact on verification results can be unambiguously attributed to that fault.

10.3. Test Suite Execution

To gather the results the test suites are executed uniformly across all branches using the IBI Testing Approach as implemented by InterACt. For each branch, the goal is to verify or discard all potential component behavior sequences for each interaction expectation and determine whether the seeded faults are correctly detected as violations of the respective interaction expectations.

The unit and interaction tests are executed using the following command:

```
mvn test -P interact-tests -fae
```

Table 10.2.: Seeded integration faults across branches.

Branch	Seeded Fault Description
`main`	Reference implementation with correct integration behavior. All abstract and unit test-based interaction expectations are fulfilled.
`interface-fault`	The Account Twin Service expects transfer IDs to be numeric, but the Transfer Gateway sends UUIDs instead, causing an interface mismatch.
`miscoded-call-fault`	The Account Twin Service calls an incorrect REST endpoint when registering managed accounts at the Transfer Gateway.
`missing-function-fault`	The Notification Service is excluded from test execution, i.e., it has no InterACt unit test cases. This simulates a missing component in the interaction flow.
`wrong-function-fault`	The abstract interaction expectations are altered to require that the notification message includes the account to which or from which the transfer was made. The Notification Service fails to meet this condition.

The `-fae` (fail-at-end) option ensures that test execution continues even if some interaction tests fail. This is necessary because a failed interaction test does not necessarily indicate a fault in the system. It may rather reflect a component behavior sequence that does not fulfill an interaction expectation. Other sequences may still lead to successful verification and should therefore be executed.

The command is run repeatedly until all component behavior sequences are either verified or discarded, and all interaction expectations are either verified or cannot be fulfilled. Depending on the test execution order, up to five executions may be required. This corresponds to the depth of the longest verification chain in the Piggy Bank system, which spans five steps: capturing the abstract interaction expectations, executing the Transfer Gateway behavior, the Account Twin Service behavior, the Notification Service behavior, and final verification of the abstract interaction expectation.

10.4. Results

The evaluation confirms that InterACt successfully captures and models component interactions and verifies interaction expectations across all system variants. For each component, a corresponding IBI component model is captured. These IBI component models contain all observed messages, interface information, and relationships required for integration analysis.

A total of 46 InterACt unit test cases are executed across the five components. From these, the Integration Controller extracts 6 unit test-based interaction expectations, matching the number of expected unit test-based interaction expectations. In addition, 2 AIEs are captured. To verify these expectations, 84 interaction test cases

are generated and executed on the reference (`main`) system. The results show that all interaction expectations are successfully verified, confirming that the system behaves as expected.

In contrast, interaction expectation verification fails in each faulty branch, consistent with the seeded integration fault. For the faulty branches, the number of interaction test cases executed is lower than in the reference branch. This is because the interaction test cases are iteratively generated. If a component behavior in a component behavior sequence fails to be verified by an interaction test case, the sequence is discarded, and no further interaction test cases are generated for that sequence. This leads to fewer interaction test cases being executed in the faulty branches compared to the reference branch. Table 10.3 provides an overview of the executed interaction test cases.

Table 10.3.: Verification results across branches.

Branch	**# Interaction Tests**	**# Verified Interaction Expectations**	**# Failed Interaction Expectations**
`main`	84	8	0
`interface-fault`	20	2	6
`miscoded-call-fault`	66	6	2
`missing-function-fault`	16	6	2
`wrong-function-fault`	84	6	2

In all faulty branches, the failed interaction expectations correspond directly to the integration fault introduced in the respective variant:

1. In the `interface-fault` branch, the Account Twin Service expects numeric transfer IDs, but the Transfer Gateway sends UUIDs. As a result, the interaction test cases generated for the Account Twin Service fail, since the provided messages can not be deserialized. Thus, the abstract interaction expectations *aie1* *aie2* and the unit test-based interaction expectations of the Transfer Gateway, can not be verified.

2. In the `miscoded-call-fault` branch, the Account Twin Service calls an incorrect REST endpoint when attempting to register managed accounts. As a result no interface binding is created in the IBI system model, so no potential component behavior sequence can be derived. Consequently, the Account Twin Service's unit test-based interaction expectations can not be verified.

3. In the `missing-function-fault` branch, the Notification Service is excluded from the test setup (i.e., it has no InterACt unit test cases). Thus, no component behavior sequence can be determined, where a notification is issued, resulting in both abstract interaction expectations *aie1* and *aie2*, not being verified.

4. In the `wrong-function-fault` branch, the abstract interaction expectations *aie1* and *aie2* are altered to expect the notification to include the account that sent or received the transfer to or from a managed account. Since, the functionality of the Notification Service is not altered, the notification message does not include the account. As a result, the abstract interaction expectations *aie1* and *aie2* are failed.

No false positives or false negatives are observed. Each failed verification is attributable to the seeded fault, demonstrating the effectiveness of the IBI Testing Approach in this controlled environment.

10.5. Discussion

The evaluation demonstrates that the IBI Testing Approach is capable of extracting IBI component and system models from InterACt unit test cases and generating correct interaction expectations. All interaction expectations are successfully verified in the fault-free reference system, and all seeded faults result in failed expectations that directly align with the injected integration faults.

These results confirm that the IBI Testing Approach functions as intended in a fully controlled environment and that interaction test generation and expectation verification work reliably when required test coverage is available and all system behaviors are observable. No false positives or false negatives were observed, indicating a high level of precision in expectation verification.

While these findings validate the concept of the IBI Testing Approach, they also highlight the limitations of demonstration case studies: they do not capture the complexity, incompleteness, or heterogeneity of real-world systems. The industrial case study in Chapter 11 complements these results by identifying such challenges.

10.6. Threats to Validity

Following the guidelines for empirical research in software engineering by Wohlin et al. [WRH+12], the following categories of validity threats are considered in this evaluation.

Construct Validity

Construct validity concerns whether the evaluation accurately measures what it intends to measure. In this case, the seeded faults are selected to represent common categories of integration faults, such as interface fault, miscoded call faults, and wrong function faults. While these fault types are relevant and grounded in prior classification schemes, the evaluation does not fully cover the diversity and complexity of faults in real-world CBSS, such as those involving concurrency, timing, or

system resource contention, which are also prevalent in CBSS but out of scope for this evaluation.

Internal Validity

Internal validity addresses whether the observed effects can be attributed to the manipulated variables, i.e., the seeded faults. Since only one fault is introduced per branch and the system configuration is otherwise kept constant, the observed results can be reliably traced to the injected fault. However, the same person implemented the system, defined the test cases, and seeded the faults, which may introduce a risk of confirmation bias.

External Validity

External validity concerns the generalizability of the results. The Piggy Bank system follows a realistic microservice architecture and uses common communication protocols (REST and AMQP), yet it remains a simplified system. The findings may not generalize to larger, more heterogeneous systems with more complex interaction topologies and external dependencies.

Reliability

Reliability refers to the repeatability of the evaluation. All branches, test data, and configurations are version-controlled and publicly available[1]. The test cases are automated and repeatable. This ensures that the demonstration case study can be replicated and extended by other researchers, enhancing the reliability of the results.

1. All artifacts of the Piggy Bank system are publicly available at `https://github.com/NilsWild/Piggy-Bank`

Chapter 11.

Exploratory Industrial Case Study

Table of Contents

11.1. Objectives 121
11.2. Case Study Design 122
11.3. Identified Limitations 125
11.4. Discussion 128
11.5. Threats to Validity 129

While the general feasibility of the IBI Testing Approach has been demonstrated in a demostration case study using the Piggy Bank system, its applicability in complex industrial settings remains an open question. Industrial software systems often exhibit interaction characteristics that go beyond those found in academic case studies, including legacy constraints, heterogeneous communication protocols, and evolving architectural patterns.

To explore the limitations of the IBI Testing Approach, an exploratory case study is conducted in collaboration with IVU Traffic Technologies. Rather than verifying the IBI Testing Approach through idealized examples, this study focuses on understanding whether the IBI Testing Approach can be applied to existing systems and which challenges and limitations emerge in doing so.

11.1. Objectives

This study focuses on identifying interaction patterns that illustrate conceptual and technical limitations of the IBI Testing Approach. These patterns are derived from real-world observations and serve as the basis for analyzing gaps in the IBI Testing Process, IBI Metamodel and InterACt.

The evaluation is guided by the following questions:

Q 1: Can existing unit tests be converted into InterACt unit tests without modifying the test scenario or assertions?

- What modifications are needed?
- What restrictions apply when using InterACt unit tests instead of standard JUnit 5 tests?

Q 2: Are there component behaviors or interaction patterns that the IBI testing process does not currently support?

- Do specific interaction patterns introduce challenges that require additional extensions to the IBI metamodel, process, or tooling?
- What extensions or modifications are required to support these scenarios?

Q 3: What additional requirements must be met for adoption in an industrial context?

11.2. Case Study Design

To answer the research questions, an exploratory industrial case study is conducted in collaboration with IVU Traffic Technologies that develop a large-scale public transport software suite. The suite consists of more than 30 products composed of numerous independently deployable components that communicate using a variety of protocols and integration patterns. Many of these components have evolved over several decades, resulting in a heterogeneous technology landscape.

Due to the implementation decisions made for InterACt and the defined scope of this thesis, the IBI Testing Approach is not intended to be applied to the entire industrial system. Instead, the evaluation focuses on a selected subset of components that meet the current technical requirements of InterACt. These constraints include specific technologies (Spring), protocols (REST, AMQP), and test framework compatibility (JUnit 5). Within this subset, the evaluation analyzes recurring interaction patterns. These patterns serve to represent conceptual and technical limitations in the IBI Metamodel, IBI Testing Process and their implementation in InterACt and assess the feasibility of applying the IBI Testing Approach in real-world industrial contexts.

11.2.1. Component Selection

The product portfolio of the industry partner has evolved over more than 40 years, resulting in a heterogeneous set of technologies, development practices, and integration patterns. Many of the older components rely heavily on low-level unit testing and manual or system-level integration testing, and thus do not fit the assumptions of the IBI Testing Approach. Furthermore, the components need to be compatible with the technical ecosystem of InterACt to ensure that the instrumentation and observation mechanisms can be applied:

- **Programming Language:** The component must be implemented in a JVM-based language. This ensures compatibility with InterACt, which is implemented in Kotlin.

- **Test Framework:** The component must use JUnit 5 for testing. The current implementation of the InterACt Test Framework Adapter is tailored to JUnit 5 and leverages its extension mechanism to control test execution and extract test information.
- **Application Framework:** The component must be based on the Spring Framework. This ensures compatibility with the existing InterACt instrumentation and observation mechanisms.
- **Communication Protocols:** The component must use at least one of the communication protocols currently supported by InterACt. These include REST and AMQP. Components using unsupported protocols, were excluded to avoid capturing incomplete interaction data.
- **Interface-Level Testing:** Existing unit tests must exercise the component via its defined interfaces. This is essential for extracting meaningful interaction expectations from observable behavior.

Applying these selection criteria significantly narrowed the set of eligible components. Out of the more than 30 products in the industry partner's suite and an even larger number of components, twelve fulfilled the technical and testing requirements. This number was further reduced to seven, as the remaining five components utilized additional protocols in their test scenarios that are not yet supported by InterACt. Including these components and scenarios would have resulted in incomplete observations and impaired the extraction of reliable interaction expectations.

A consequence of this restriction is that it was not possible to construct a complete potential behavior graph for any scenario within the selected subset of components. As a result, the evaluation focuses exclusively on capturing component behavior and extracting interaction expectations. Validation of these expectations had to be omitted in the industrial setting due to missing links in the interaction flow.

11.2.2. Preparation and Instrumentation

Before applying InterACt to the industry partner's components, several preparatory steps are necessary to enable the selected components to work with InterACt. These steps include adapting the existing unit test suites, integrating the required instrumentation to capture component behavior, and executing the test cases to record observable interactions.

Test Suite Adaptation The selected components already include unit test suites to verify their behavior. However, these test cases are not originally developed with InterACt in mind. For this evaluation, the test cases are adjusted to comply with the requirements of the InterACt JUnit 5 Test Framework Adapter. This includes replacing standard JUnit 5 annotations with the `@InterACtTest` annotation, ensuring

tests are parameterized to allow for message injection, and isolating assertions that rely on specific expected values rather than property-based verification. For example-based assertions InterACt's `forExample` lambda is used to skip these assertions during interaction test execution, whereas property-based assertions are wrapped in InterACt's `inherently` lambda.

Instrumentation of Observers To capture interaction data during test execution, the selected components are instrumented with the appropriate *Interface Observers*. These observers are implemented as Spring Boot autoconfiguration modules. Each observer records messages sent and received through the CUT's interfaces during test execution and forwards them to the Test Execution Manager for correlation with the active test case.

Test Execution Workflow Once the test suites are adapted and observers registered, the instrumented test cases are executed. After a test suite completes, the resulting IBI component model is sent to the Integration Controller for expectation extraction. The integration controller processes the IBI models, extracts interaction expectations and generates interaction test suites. The generated interaction test suites are then executed by re-running the test methods annotated with `@InterACtTests`.

Analysis Focus The evaluation focuses on analyzing integration-related behavior observed during test execution and identifying recurring patterns that reveal limitations in the current IBI Testing Approach. These patterns emerged naturally from the tested components and serve as the basis for the subsequent analysis. Each identified pattern highlights a specific challenge for the IBI testing process and informs the discussion of required conceptual extensions or implementation improvements.

11.3. Identified Limitations

The analysis is based on recurring interaction patterns observed across multiple components in the evaluated set of components. Each pattern highlights a distinct type of issue—conceptual or technical—that affects the ability to extract, model, or verify interaction expectations and therefore system behaviors.

The identified patterns were not preselected but derived from real-world observations across multiple components and unit test cases. They serve to illustrate key limitations in the current state of the IBI Testing Approach with respect to stimulus detection, message handling, interface modeling, and third-party integrations.

For each identified pattern, the following structure is used: the context in which the pattern occurred is described, followed by an explanation of the resulting limitations, and a discussion of how it could be addressed.

11.3.1. Conceptual Limitations

Conceptual Limitation 1: Internal Stimuli
In many cases a component behavior is triggered by internal application lifecycle events rather than an external incoming message. One example observed in the case study involved a component that initializes an internal cache during application startup. As part of this process, the component sends a REST request to retrieve configuration data and subsequently performs an AMQP call to request additional information.

From the perspective of the IBI testing process, this type of behavior presents a significant challenge: the component's actions are not preceded by an observable stimulus that could serve as the starting point of the interaction. Since the current IBI Metamodel assumes that each system behavior begins with a stimulus message received through one of the component's interfaces, no valid interaction expectation can be constructed for such cases—even though the rest of the behavior is observable and well-defined.

This pattern reveals a conceptual limitation. Conceptually, the IBI testing process lacks a mechanism to represent internal stimuli, such as lifecycle events or scheduled jobs. Those stimuli do not result from external components but are internal. Even though time based events could be modeled by a clock component it would complicate the integration process without additional benefits, since the integration does not need to be tested. Addressing this limitation would require extending the IBI Metamodel to explicitly allow for internal triggers and adapting the tooling accordingly.

Conceptual Limitation 2: Missing IBI Component Models
Many system behaviors involve components for which no unit test suites exist. These components may be external services or components that are not tested via their interfaces. Although they actively participate in interactions, they cannot be

directly instrumented with InterACt, and no IBI component models are generated for them.

This leads to an incomplete IBI system model, resulting in gaps in the potential behavior graph, which in turn prevents the verification of interaction expectations. Consequently, it becomes impossible to verify behaviors that involve such components.

This limitation is conceptual in nature. The IBI Testing Approach currently assumes that all relevant components are tested in isolation and instrumented for behavior capture. In real-world systems, however, it is common for some parts of the system to remain opaque—either due to organizational boundaries or because they are not under active development.

A potential solution would involve introducing integration mocks to represent the expected behavior of such components. These mocks could be derived from the observed interactions and used to complete the system model by modeling expected behavior explicitly. The data captured by InterACt in conjunction with production traces could support this process by identifying which components are missing and what minimal behavior needs to be emulated to complete interaction flows. However, support for such mocking is currently not part of the implementation or concept.

Conceptual Limitation 3: Message Order Guarantees

When a component behavior depends on the order of messages, the IBI Testing Process may verify the interaction expectations even though the message order is not guaranteed by the system. For example, a component may send a message to its environment and then receive two response messages in return. However, the IBI Testing Process only guarantees that both responses are sent by some component in the environment, but it does not check if their order is guaranteed, since it just verifies that there is one verifying system behavior and the order of messages is controlled by the abstract test case. In tradition integration testing, message order guarantees that are not kept, would lead to flaky tests, since the order of messages is not guaranteed and would thus cause the test to fail occasionally. The presented approach leads to false negatives for such integration faults.

A potential solution would need to be implemented in the IBI Metamodel and the IBI Testing Process. The IBI Metamodel would need to be extended to include message order guarantees, and the IBI Testing Process would need to be adapted to verify these guarantees during interaction expectation verification. This would require additional information about the system's communication protocols and their guarantees.

11.3.2. Technical Limitations

Technical Limitation 1: Exchange Formats

InterACt's implementation assumed JSON as the standard exchange format. However, in the industry partner's system, components communicate via AMQP using Protocol Buffers to serialize and deserialize structured domain messages. These binary-encoded messages could be successfully transmitted and received, but the message content could not be deserialized and therefore used as parameters of interaction test cases.

InterACt relied on the assumption that message payloads would be encoded in JSON and directly deserializable into data structures suitable for inclusion in the IBI component model and parameter injection in interaction test cases. Consequently, while the observers captured the message envelope and metadata, they were unable to interpret the serialized content or represent it meaningfully in the system model.

To address this issue, an abstraction layer for message (de-)serialization was introduced. This extension allows protocol-specific serializers and deserializers to be registered for supported formats, enabling InterACt to correctly interpret and store the actual message content. Once this mechanism was in place and a Protocol Buffers deserializer was added, the system was able to extract interaction expectations and reuse the deserialized message content as parameters in interaction tests.

This pattern highlights a technical limitation in the initial implementation of InterACt, rather than a conceptual flaw in the IBI Testing Approach itself. The evaluation shows that support for custom serialization formats is essential in real-world systems, where protocols such as AMQP or even REST may carry data in arbitrary formats. The added abstraction layer now provides a flexible extension point for future support of additional formats.

Pattern 3: Dynamically Created Interfaces

Systems that use messaging protocols often support dynamic endpoint creation for RPC calls. In the observed case, a component performs RPC calls using AMQP. For each request, the component dynamically generates a temporary reply queue and set its name in the `reply-to` header of the outgoing message. The corresponding response is expected on this temporary queue.

From a runtime perspective, this pattern is well-established and supported by the AMQP 0.9.1 specification. However, it poses significant challenges for the IBI component model. Since each test execution creates a new temporary queue with a unique name, the Interface Observers interpret each instance as a separate interface. As a result, the component model grows with each test run, accumulating structurally identical and semantically equivalent interfaces that differ only in their dynamically generated identifiers.

This leads to two main issues. First, the system model becomes polluted with redundant interface definitions, making it harder to analyze and reason about

the component's actual behavior. Second, interaction test cases generated from this data refer to different interface instances on each execution, which hinders expectation reuse and consistency across test runs.

Currently, no mechanism exists in InterACt to normalize or abstract over dynamically created interfaces. As a result, interaction expectations cannot be reliably extracted for such patterns, despite the underlying behavior being both stable and observable.

This pattern reflects a technical limitation in the implementation of the Interface Observers and the system model's reliance on concrete interface definitions. Conceptually, dynamic endpoints could be abstracted as logical interfaces rather than queue names. Addressing this limitation would require protocol-specific strategies for grouping dynamic interfaces or defining generalized interaction contracts for ephemeral endpoints.

11.4. Discussion

The results reveal several limitations that affect its practical adoption and effectiveness in such environments. These limitations span both conceptual assumptions in the underlying IBI Metamodel and IBI Testing Process as well as technical constraints in the current implementation of InterACt. A key factor influencing these results is the limited technical ecosystem currently supported by InterACt. As with many PoC implementations, the tool focuses on a specific combination of test frameworks and communication protocols. This restriction naturally limited the set of components that could be considered in the evaluation. Nonetheless, the selection process already provides valuable insights: in heterogeneous environments with a long evolution history, only a small subset of components may currently benefit from the IBI Testing Approach. This limitation directly affects the return on investment when applying the approach in such contexts. Broader adoption would require extending support to additional test frameworks and communication protocols, as well as introducing mechanisms for handling partial potential behavior graphs or mocking missing components.

The following analysis discusses how the evaluation results relate to the research questions. Each question is revisited in light of the observed limitations.

Q 1 asked whether existing unit tests can be converted into InterACt tests without modifying the test scenario or assertions. The evaluation showed that this was possible for all test cases but might require significant migration effort. While some test cases were easy to adapt others required a lot of manual effort. This includes identifying and isolating stimuli and responses, restructuring test data and expected values to support parameterized execution, and distinguishing between property-based and example-based assertions. These steps introduce non-negligible developer effort and highlight the need for guidance or tooling support

to ease the migration process.

Q 2 focused on unsupported behavior patterns. Several recurring patterns were identified where either the model or the implementation failed to support realistic scenarios: behaviors triggered by internal stimuli, dynamically generated interfaces, and the use of custom serialization formats. While some of these challenges (e.g., format handling) can be addressed technically, others (e.g., modeling internal stimuli or uncovered components) require conceptual extensions to the IBI Metamodel itself.

Q 3 concerned the additional requirements for industrial adoption. The findings show that the current implementation is not yet ready for use in heterogeneous systems without further extension. Only a subset of components could be included in the evaluation, and even within those, substantial manual adaptation was required. Broader adoption would depend on support for additional protocols, serialization formats, and integration strategies, as well as mechanisms to handle systems that are partially uncovered by appropriate test cases. Furthermore, the restrictions regarding the state setup of components limits the usefulness of the approach in systems with complex interaction patterns.

In summary, the evaluation identified multiple conceptual and technical limitations of the IBI Testing Approach that must be addressed before the approach can deliver meaningful value in industrial settings.

11.5. Threats to Validity

This section reflects on the limitations of the case study design and the validity of the conclusions drawn from the evaluation.

Internal Validity. The evaluation relies on the correct functioning of the instrumentation and observer components during test execution. While care was taken to verify the captured data, it is possible that certain interactions were not observed due to framework limitations or configuration errors. Additionally, the filtering of components based on technical compatibility may have introduced selection bias, focusing only on newer or more modular components, that use certain interaction patterns.

External Validity. The evaluation is conducted within a single industrial context, involving a specific domain (public transport), a limited set of components, and a JVM-based technology stack. As such, the findings may not generalize directly to other industries, technology ecosystems, or testing cultures. However, the identified

interaction patterns (e.g., internal stimuli, dynamic interfaces, message serialization formats) are common in many distributed systems and likely to appear in other domains as well.

Construct Validity. The evaluation focuses on interaction pattern identification and does not attempt to measure quantitative outcomes such as test coverage, defect detection rate, or performance. The absence of expectation verification in the industrial case study also limits the assessment of the IBI testing process. The abstraction of real-world issues into general patterns introduces a degree of interpretation, though these abstractions are grounded in recurring technical observations.

Conclusion Validity. The conclusions are based on a small number of components and test cases. While the findings are supported by repeated observations, some relevant patterns may not have been encountered due to the limited scope of the evaluation.

Chapter 12.

Conclusion

Table of Contents

12.1. Discussion of the Research Questions 132
12.2. Contributions 133
12.3. Future Work 135
12.4. Closing Remarks 137

Following the DSR methodology introduced in Chapter 1, this thesis pursued the development and evaluation of the IBI Testing Approach to address multiple challenges in the domain of integration testing of CBSSs. The DSR methodology guided the formulation of objectives (**O1–O3**) and the derivation of research questions (**RQ1-RQ3**), which shaped the design of the IBI Testing Approach, its implementation, and evaluation.

This thesis demonstrated that the integration of CBSSs can be effectively tested (**O1**) by systematically reusing and analyzing existing unit test cases (**O2**). To achieve this, the IBI Metamodel, the IBI Testing Process, and their implementation in InterACt were developed. Together, they constitute the IBI Testing Approach. A novel approach to integration testing of CBSSs.

Traditional integration testing approaches typically involve considerable manual effort to define test cases, prepare test data, and maintain dedicated integration environments. These challenges are particularly pronounced in evolving systems where frequent changes to components and system architecture require constant test adaptation. The IBI Testing Approach presented in this thesis addresses these limitations by shifting from manually constructed integration test cases to interaction-based verification of individual component interactions in context. This enables isolated and reusable testing of component interactions and supports early and continuous validation throughout development (**O3**).

The remainder of this chapter discusses how the research questions outlined in Section 1.3 have been addressed and identifies potential directions for future research.

12.1. Discussion of the Research Questions

RQ 1: How can interaction expectations be systematically extracted from unit test cases?

This question was addressed through the analysis of unit test cases, which commonly simulate interactions between a component and its environment by using mocks. The thesis demonstrated that these mocked interactions encode implicit expectations about how a component assumes other components to behave. By observing the messages exchanged with mocks during unit test execution, interaction expectations can be systematically extracted and made explicit.

RQ 1.1 explored which types of expectations can be extracted from unit tests. Unit test-based interaction expectations capture the assumptions that are implicitly expressed through the use of mocks. However, they do not cover interactions or expectations that are unrelated to mocked responses.

RQ 1.2 addressed the fact that not all expectations are implicitly defined by unit test cases. The IBI Testing Approach allows test developers to define abstract interaction expectations, which are then resolved using existing unit test data and verified in the same manner as those derived from the unit test cases.

RQ 2: How can interaction expectations be formalized to support automated integration verification?

Two types of interaction expectations are represented within the IBI Metamodel. Unit test-based interaction expectations are derived from unit test cases, while abstract interaction expectations are defined by the test developers. Both share a common structure. An interaction expectation consists of a stimulus interface, a reaction interface, and an assertion. The stimulus interface defines the input messages, the reaction interface defines the expected output messages, and the assertion specifies the expected behavior of the SUT.

RQ 3: How can integration testing be conducted based on formalized interaction expectations?

The IBI Testing Process operationalizes the IBI Metamodel by generating interaction test cases from unit test cases and verifying interaction expectations by validating component behaviors in context.

RQ 3.1 focused on determining whether a given set of components can be successfully integrated. The thesis addressed this by analyzing whether the behaviors required to fulfill interaction expectations can be derived from the component behaviors that are unit tested. If no such component behavior sequence can be found, this indicates a missing or incompatible integration path. However, this re-

lies on the completeness of unit test coverage, which may not always hold in practice.

RQ 3.2 examined whether expectations can be verified without dedicated integration tests. The IBI Testing Process uses the unit test cases to derive interaction test cases that are executed in the context of the SUT. The results of these executions are then correlated with the interaction expectations. This allows for verifying whether the system behavior fulfills the expectations without requiring to exercise the system in a dedicated integration test environment.

Although the research questions could be addressed using the proposed IBI Testing Approach, the evaluation highlighted several limitations that require further investigation.

12.2. Contributions

This thesis makes several contributions to the field of software testing, particularly in the context of CBSSs. These contributions span conceptual foundations, methodological advancements, and practical tool support. Together, they address the objectives defined in Section 1.2 and provide answers to the research questions discussed above.

1. A Metamodel for Integration Testing Based on Unit Test Cases

The thesis introduces the IBI Metamodel that captures the structural and behavioral aspects of CBSSs, unit test cases, interaction expectations, and their verification. The IBI Metamodel consists of four sub-models:

- The *Component-based System Metamodel*, which represents components, interfaces, messages, and their relationships.
- The *Test Metamodel*, which records unit test behavior as observed sequences of stimuli and responses.
- The *Interaction Expectation Metamodel*, which encodes the interaction expectations.
- The *Verification Metamodel*, which links expectations to observable behavior and captures whether they were fulfilled.

This IBI Metamodel supports fine-grained reasoning about integration behavior and enables automated integration test derivation and interaction expectation verification.

2. An Iterative Integration Testing Process

Building on the IBI Metamodel, the thesis defines the IBI Testing Process: a testing process that leverages unit test cases to derive, simulate, and iteratively verify system integration. The IBI Testing Process eliminates the need for manually constructed integration tests for requirements that can be encoded in interaction expectations and includes the following coordinated activities:

- Extraction of IBI component models and interaction expectations from unit test cases.
- Resolution of abstract interaction expectations based on existing test data.
- Derivation of potential system behaviors through derivation of component behavior sequences.
- Iterative generation and execution of interaction test cases using existing unit test infrastructure.
- Verification of interaction expectations and identification of coverage gaps.

The IBI Testing Process is formally defined using property graphs and regular path queries, providing a precise and extensible operational foundation.

3. A PoC Implementation: The InterACt Tool

The InterACt tool implements the IBI Metamodel and IBI Testing Process using a modular architecture that supports multiple communication protocols. It provides:

- Interface observers for protocol-specific message capturing (e.g., REST, AMQP) and interface binders for IBI system model construction.
- A test execution manager for aggregating and correlating interactions with the CUT.
- An integration controller that performs expectation resolution and interaction test generation.
- A user interface for visualizing IBI system models, interaction test cases, and interaction expectation verification results.

InterACt demonstrates the feasibility of the proposed IBI Testing Approach and supports its evaluation in real-world scenarios.

4. Evaluation in a Demonstration Case Study and Exploratory Case Study

The approach was evaluated in two complementary evaluations:

- A demonstration case study, using a microservice system (the Piggy Bank system), which served to validate the IBI Testing Approach in a controlled environment.
- An exploratory case study in collaboration with IVU Traffic Technologies, which evaluated the applicability of the IBI Testing Approach in an industrial setting.

These evaluations confirmed the conceptual viability of the IBI Testing Approach, identified practical limitations, and provided insights for future enhancements.

12.3. Future Work

The proposed IBI Testing Approach provides a strong foundation for interaction-based integration testing. Nevertheless, several directions remain for future research and development. These include extensions to protocol support, improvements in scalability and state management, enhanced integration into modern development workflows, and broader empirical validation.

1. Support for Additional Protocols

The current implementation supports communication protocols such as REST and AMQP. To increase applicability in diverse software systems, the framework could be extended to support additional protocols, including gRPC, Kafka, or GraphQL.

2. Improved Interaction Expectation Verification Strategies

Interaction expectation verification currently relies on unit test data and interface bindings. More sophisticated strategies could incorporate historical data from previous component versions or employ heuristics and machine learning to predict likely integration flows. These enhancements could reduce verification effort and also guide developers in identifying the root cause for failed interaction expectations.

3. Enhanced State Awareness and Stateful Verification

The current IBI Testing Approach does not explicitly model or track component or system state in context of interaction expectations. As a result, interaction expectations that span multiple steps or rely on prior state can not be verified. Two main limitations were identified:

- *Lack of expectation chaining:* Expectations are evaluated in isolation, preventing the expression of end-to-end flows that depend on prior interactions.
- *Insufficient control over state setup:* Interaction test cases rely solely on message stimuli, preventing explicit state setup.

Extending the IBI Testing Approach to model dependencies between interaction expectations, and enabling state transfer or snapshotting between interaction test cases, would improve support for complex and stateful systems.

4. Integration with Continuous Integration Pipelines

To increase practical utility, the IBI Testing Approach should be integrated into continuous integration and delivery (CI/CD) workflows. Automatically generating and executing interaction test suites in response to component behavior or interface changes would support continuous integration testing.

5. Scalability and Performance Optimization

As system size and the number of test cases grow, the performance of the IBI Testing Approach becomes increasingly important. The amount of generated interaction test cases grows exponentially with the number of components and potential component behavior sequences, that are directly correlated to the number of input and output equivalence classes handled by each interface. Thus changes to the component architecture could be explored as well. E.g. if a component replies to erroneous requests via a different interface the amount of potential component behavior sequences would be reduced. Scalability is critical for industrial adoption in large-scale software systems.

6. Precise Behavior Correlation via Distributed Tracing

Currently, interaction tracing assumes sequential test execution and relies on implicit correlation of messages and test cases. Integration with distributed tracing tools such as OpenTelemetry could enable precise correlation. This would improve traceability, support parallel test execution, and increase observability.

7. Broader Industrial Evaluation

While an initial exploratory case study was conducted in collaboration with IVU Traffic Technologies, broader empirical validation is needed. Future studies should investigate:

- Generalizability across different domains and architectures,
- Developer acceptance and learning effort,

- Long-term maintainability and cost-benefit trade-offs.

Such evaluations would provide deeper insights into the practical impact and guide further refinement of the IBI Testing Approach.

12.4. Closing Remarks

This thesis presented a novel approach to automating integration testing for CBSSs. Grounded in the DSR methodology, the work contributed both practically and scientifically by addressing a challenge in modern software development: verifying the integration of independently developed components in CBSSs.

Through the IBI Metamodel and the IBI Testing Process, the thesis introduced a conceptual foundation for leveraging unit test artifacts to verify system-level behavior. The IBI Testing Approach enables early, incremental verification of CBSSs without the need for additional test environments.

InterACt demonstrates the feasibility of the method. The evaluations confirmed the conceptual applicability of the solution while also revealing technical and practical limitations that inform future work.

By enabling systematic reuse of test knowledge and bridging the gap between unit testing and system verification, this research contributes to the advancement integration testing strategies.

Ultimately, this thesis provides both a foundation and a roadmap for future research into test reuse and automated integration testing in CBSS.

References

[ABM00] Colin Atkinson, Joachim Bayer, and Dirk Muthig. Component-based product line development: the kobra approach. *Software Product Lines: Experience and Research Directions*:289–309, 2000.

[AG02] Colin Atkinson and Hans-Gerhard Groß. Built-in contract testing in model-driven , component-based development. In 2002.

[ALR+04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[Ang18] Renzo Angles. The property graph database model. In Dan Olteanu and Barbara Poblete, editors, *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018*, volume 2100 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[BB05] F. Belli and C.J. Budnik. Towards self-testing of component-based software. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, volume 2, 205–210 Vol. 1, 2005.

[Bei95] Boris Beizer. *Vlack-Box Testing*. 1995.

[BG03] S. Beydeda and V. Gruhn. State of the art in testing components. In *Third International Conference on Quality Software, 2003. Proceedings.* Pages 146–153, 2003.

[BMK20] Mark Buenen, Anand Moorthy, and Sushil Kumar. Continous Testing Report 2020, Capgemini, 2020.

[BNdS22] Mark Buenen, Sathish Natarajan, and Rohit de Souza. World Quality Report 14th Edition, Capgemini, 2022.

[CK93] Kwang Ting Cheng and Avinash S Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th International Design Automation Conference*, pages 86–91, 1993.

[CMW87] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. *SIGMOD Rec.*, 16(3):323–330, 1987.

[Coh] Mike Cohn. The forgotten layer of the test automation pyramid. `https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid`. Accessed 2024-11-18.

[Cro93] Nigel Cross. Science and design methodology: a review. *Research in Engineering Design*, 5(2):63–69, June 1993.

[DMM01] Marcio Eduardo Delamaro, JC Maidonado, and Aditya P. Mathur. Interface mutation: an approach for integration testing. *IEEE transactions on software engineering*, 27(3):228–247, 2001.

[Dyn] Dynatrace. Technology support. `https://docs.dynatrace.com/docs/ingest-from/technology-support`. Accessed 2025-04-28.

[ECD+09] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Matthew Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Transactions on Software Engineering*, 35(1):29–45, 2009.

[Edw01] Stephen H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, 11(2):97–111, 2001.

[ER11] Emelie Engström and Per Runeson. Software product line testing–a systematic mapping study. *Information and Software Technology*, 53(1):2–13, 2011.

[FL00] Peter Fröhlich and Johannes Link. Automated test case generation from dynamic models. In *European Conference on Object-Oriented Programming*, pages 472–491. Springer, 2000.

[Fow] Martin Fowler. Microservices. `https://martinfowler.com/articles/microservices.html`. Accessed 2024-11-18.

[FRL+19] Stefan Fischer, Rudolf Ramler, Lukas Linsbauer, and Alexander Egyed. Automating test reuse for highly configurable software. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, SPLC '19, pages 1–11, Paris, France. Association for Computing Machinery, 2019.

[GLP+25] Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. Database Theory in Action: Cypher, GQL, and Regular Path Queries. In Sudeepa Roy and Ahmet Kara, editors, *28th International Conference on Database Theory (ICDT 2025)*, volume 328 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 36:1–36:5, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025.

[Goe14] Paulo Goes. Design science research in top information systems journals. *MIS Quarterly*, 38:iii–viii, March 2014.

[Gor11] Ian Gorton. *Essential Software Architecture*. Springer, 2nd edition, 2011.

[Gor22] Ian Gorton. *Foundations of Scalable Systems*. O'Reilly Media, 2022.

[GV10] Vahid Garousi and Tan 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*, 83(11):2251–2262, 2010.

[GWN07] Markus Gälli, Rafael Wampfler, and Oscar Nierstrasz. Composing tests from examples. *Journal of Object Technology*, 6:71–86, 2007.

[Hem15] Hadi Hemmati. How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156, 2015.

[hGmb] hello2morrow GmbH. Sonargraph – static analysis and architecture validation. `https://www.hello2morrow.com/products/sonargraph`. Accessed 2025-04-28.

[HMP+04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004. ISSN: 02767783.

[HSW19] Dominik Hellhake, Tobias Schmid, and Stefan Wagner. Using data flow-based coverage criteria for black-box integration testing of distributed software systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 420–429, 2019.

[HvH15] Wilhelm Hasselbring and André van Hoorn. Open-source software as catalyzer for technology transfer: kieker's development and lessons learned, September 2015.

[IEE17] IEEE Standards Association. ISO/IEC/IEEE 24765:2017(E) - Systems and Software Engineering – Vocabulary. International Standard ISO/IEC/IEEE 24765:2017(E), IEEE Standards Association, 2017, pages 1–541.

[IPJ+19] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 955–963, Tallinn, Estonia. Association for Computing Machinery, 2019.

[ISTQB] International Standard Testing Qaulifications Board. User acceptance test. `https://glossary.istqb.org/en˙US`. Accessed 2025-01-15.

[JJB+07] Muhammad Jaffar-ur Rehman, Fakhra Jabeen, Antonia Bertolino, and Andrea Polini. Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability*, 17(2):95–133, 2007.

[JM97] J.-M. Jazequel and B. Meyer. Design by contract: the lessons of ariane. *Computer*, 30(1):129–130, 1997.

[KG11] Usman Ali Khan and Kunal Gupta. Challenges in component based software engineering as the technology of the modern era. In 2011.

[LF17] Bruno Lima and João Pascoal Faria. A survey on testing distributed and heterogeneous systems: the state of the practice. In Enrique Cabello, Jorge Cardoso, André Ludwig, Leszek A. Maciaszek, and Marten van Sinderen, editors, *Software Technologies*, pages 88–107, Cham. Springer International Publishing, 2017.

[LGM+24] Alexander Lercher, Johann Glock, Christian Macho, and Martin Pinzger. Microservice api evolution in practice: a study on strategies and challenges. *Journal of Systems and Software*, 215:112110, 2024.

[LLS+12] Sascha Lity, Malte Lochau, Ina Schaefer, and Ursula Goltz. Delta-oriented model-based spl regression testing. In *2012 Third International Workshop on Product LinE Approaches in Software Engineering (PLEASE)*, pages 53–56. IEEE, 2012.

[LMM19] Jyri Lehvä, Niko Mäkitalo, and Tommi Mikkonen. Consumer-driven contract tests for microservices: a case study. In Xavier Franch, Tomi Männistö, and Silverio Martínez-Fernández, editors, *Product-Focused Software Process Improvement*, pages 497–512, Cham. Springer International Publishing, 2019.

[LW90] Hareton K. N. Leung and Lee J. White. A study of integration testing and software regression at the integration level. *Proceedings. Conference on Software Maintenance 1990*:290–301, 1990.

[MBS+19] Andi Mann, Alanna Brown, Michael Stahnke, and Nigel Kersten. State of DevOps Report. Technical report, Puppet, Circle CI, Splunk, 2019. Accessed 2024-10-14.

[McI68] M.D. McIlroy. Mass produced soptware components, by m.d. mcilroy. In *Software Engineering, Report on a conference sponsored by the NATO Science Committee*, Gamisch, Germany, October 1968.

[MDB+07] R. Moraes, J. Duraes, R. Barbosa, E. Martins, and H. Madeira. Experimental risk assessment and comparison using software fault injection. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 512–521, 2007.

[MFD06] 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)*, pages 34–34, 2006.

[mInc22] mabl Inc. Testing in DevOps 2022, mabl Inc., 2022.

[MK11] Sajjad Mahmood and Azhar Khan. An industrial study on the importance of software component documentation: a system integrators perspective. *Information Processing Letters*, 111(12):583–590, 2011.

[MMP+21] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. Semantic matching of gui events for test reuse: are we there yet? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, pages 177–190, Virtual, Denmark. Association for Computing Machinery, 2021.

[MTY01] E. Martins, C.M. Toyota, and R.L. Yanagawa. Constructing self-testable software components. In *2001 International Conference on Dependable Systems and Networks*, pages 151–160, 2001.

[MZ04] M Momotko and L Zalewska. Component+ built-in testing a technology for testing software components. *Foundations of Computing and Decision Sciences*, 29(1-2):133–148, 2004.

[NdCM+11] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D McGregor, Eduardo Santana De Almeida, and Silvio Romero de Lemos Meira. A systematic mapping study of software product lines testing. *Information and Software Technology*, 53(5):407–423, 2011.

[Neo24] Neo4j. Cypher and gql conformance. `https://neo4j.com/docs/cypher-manual/current/appendix/gql-conformance/`, 2024. Accessed 2025-04-28.

[Nor06] Dan North. Introducing BDD — dannorth.net. `https://dannorth.net/introducing-bdd/`, 2006. Accessed 2025-01-06.

[NuRH10] Furqan Naseer, Shafiq ur Rehman, and Khalid Hussain. Using metadata technique for component based black box testing. In *2010 6th International Conference on Emerging Technologies (ICET)*, pages 276–281, 2010.

[NW09] M.F.F. Nasution and H.R. Weistroffer. Documentation in systems development: a significant criterion for project success. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–9, 2009.

[ODR+07] Alessandro Orso, Hyunsook Do, Gregg Rothermel, Mary Jean Harrold, and David S. Rosenblum. Using component metadata to regression test component-based software. *Software Testing, Verification and Reliability*, 17(2):61–94, 2007.

[OG09] Erika Mir Olimpiew and Hassan Gomaa. Reusable model-based testing. In *Formal Foundations of Reuse and Domain Engineering: 11th International Conference on Software Reuse, ICSR 2009, Falls Church, VA, USA, September 27-30, 2009. Proceedings 11*, pages 76–85. Springer, 2009.

[OZL+11] Sebastian Oster, Marius Zink, Malte Lochau, and Mark Grechanik. Pairwise feature-interaction testing for spls: potentials and limitations. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, pages 1–8, 2011.

[Pac] Pact.io. Pact - microservices testing made easy. `https://pact.io/`. Accessed 2024-11-18.

[Pet68] Brian Randell Peter Naur, editor. *Software Engineering, Report on a conference sponsored by the NATO Science Committee*, number 1 in NATO Software Engineering Conference. Gamisch, Germany, October 1968.

[PHZ24] Luan Pham, Huong Ha, and Hongyu Zhang. Root cause analysis for microservices based on causal inference: how far are we? In *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 706–718, 2024.

[PSL14] David Parsons, Teo Susnjak, and Manfred Lange. Influences on regression testing strategies in agile software development environments. *Software Quality Journal*, 22(4):717–739, December 2014.

[PTR+07] Ken Peffers, Tuure Tuunanen, Marcus Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24:45–77, January 2007.

[RML+16] Leah Riungu-Kalliosaari, Simo Mäkinen, Lucy Ellen Lwakatare, Juha Tiihonen, and Tomi Männistö. Devops adoption benefits and challenges in practice: a case study. In Pekka Abrahamsson, Andreas Jedlitschka, Anh Nguyen Duc, Michael Felderer, Sousuke Amasaki, and Tommi Mikkonen, editors, *Product-Focused Software Process Improvement*, pages 590–597, Cham. Springer International Publishing, 2016.

[Rob06] Ian Robinson. Consumer-driven contracts: a service evolution pattern — martinfowler.com. `https://martinfowler.com/articles/consumerDrivenContracts.html`, December 2006. Accessed 2024-11-18.

[SAM09] Fernando R. C. Silva, Eduardo S. Almeida, and Silvio R. L. Meira. An approach for component testing and its empirical validation. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 574–581, Honolulu, Hawaii. Association for Computing Machinery, 2009.

[SBO18] Mali Senapathi, Jim Buchan, and Hady 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*, EASE'18, pages 57–67, Christchurch, New Zealand. Association for Computing Machinery, 2018.

[SCK04] D.P. Siewiorek, R. Chillarege, and Z.T. Kalbarczyk. Reflections on industry trends and experimental research in dependability. *IEEE Transactions on Dependable and Secure Computing*, 1(2):109–127, 2004.

[SCM22] André Santos, Alcino Cunha, and Nuno Macedo. Schema-guided testing of message-oriented systems. In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*, pages 26–37. INSTICC, SciTePress, 2022.

[Sil16] Alan Sill. The design and architecture of microservices. *IEEE Cloud Computing*, 3(5):76–80, 2016.

[SM07] Monalisa Sarma and Rajib Mall. Automatic test case generation from uml models. In *10th International Conference on Information Technology (ICIT 2007)*, pages 196–201. IEEE, 2007.

[SMB08] Philip Samuel, Rajib Mall, and Ajay Kumar Bothra. Automatic test case generation using unified modeling language (uml) state diagrams. *IET software*, 2(2):79–93, 2008.

[SMM10] Santosh Kumar Swain, Durga Prasad Mohapatra, and Rajib Mall. Test case generation based on use case and sequence diagram. *International Journal of Software Engineering*, 3(2):21–52, 2010.

[SOP07] F. Saglietti, N. Oster, and F. Pinte. Interface coverage criteria supporting modelübased integration testing. In *20th International Conference on Architecture of Computing Systems 2007*, pages 1–9, 2007.

[SP10] Bernhard Schätz and Christian Pfaller. Integrating component tests to system tests. *Electronic Notes in Theoretical Computer Science*, 260:225–241, 2010. Proceedings of the 5th International Workshop on Formal Aspects of Component Software (FACS 2008).

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., USA, 2nd edition, 2002.

[TS05] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 253–262, Lisbon, Portugal. Association for Computing Machinery, 2005.

[TSL04] 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, 2004. IRI 2004.* Pages 493–498, 2004.

[Var16] Moshe Y. Vardi. A theory of regular queries. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, pages 1–9, San Francisco, California, USA. Association for Computing Machinery, 2016.

[Vit03] Padmal Vitharana. Risks and challenges of component-based software development. *Commun. ACM*, 46(8):67–72, August 2003.

[WCO03] Ye Wu, Mei-Hwa Chen, and Jeff Offutt. Uml-based integration testing for component-based software. In *Proceedings of the Second International Conference on COTS-Based Software Systems*, ICCBSS ’03, pages 251–260, Berlin, Heidelberg. Springer-Verlag, 2003.

[Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *SIGPLAN OOPS Mess.*, 1(1):7–87, August 1990.

[Wey98] E.J. Weyuker. Testing component-based software: a cautionary tale. *IEEE Software*, 15(5):54–59, 1998.

[WKB+21] Robert White, Jens Krinke, Earl T. Barr, Federica Sarro, and Chaiyong Ragkhitwetsagul. Artefact relation graphs for unit test reuse recommendation. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 137–147, 2021.

[WKW99] Yingxu Wang, G. King, and H. Wickburg. A method for built-in tests in component-based software maintenance. In *Proceedings of the Third European Conference on Software Maintenance and Reengineering (Cat. No. PR00090)*, pages 186–189, 1999.

[WL23a] Nils Wild and Horst Lichter. InterACt: a tool for unit test based integration of component-based software systems. In *18th International Conference on Software Engineering Advances (ICSEA 2023)*, pages 58–63, Valencia, Spain. IARIA, November 2023.

[WL23b] Nils Wild and Horst Lichter. Unit test based component integration testing. In *30th Asia-Pacific Software Engineering Conference (APSEC 2023)*, pages 1–10, Seoul, South Korea. IEEE Computer Society, December 2023.

[WLK20] Nils Wild, Horst Lichter, and Peter Kehren. Test automation challenges for application landscape frameworks. In *International Conference on Software Testing, Verification and Validation, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020.

[WLM+20] Muhammad Waseem, Peng Liang, Gastón Márquez, and Amleto Di Salle. Testing microservices architecture-based applications: a systematic mapping study. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pages 119–128, 2020.

[WLM24] Nils Wild, Horst Lichter, and Constantin Mensendiek. Expectation-based integration testing of unidirectional interactions in component-based software systems. In *19th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2024)*, pages 202–213, Angers, France. SCITEPRESS – Science and Technology Publications, Lda, April 2024.

[WPC01] Ye Wu, Dai Pan, and Mei-Hwa Chen. Techniques for testing component-based software. In *Proceedings Seventh IEEE International Conference on Engineering of Complex Computer Systems*, pages 222–232, 2001.

[WRH+06] Michael W Whalen, Ajitha Rajan, Mats PE Heimdahl, and Steven P Miller. Coverage metrics for requirements-based testing. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 25–36, 2006.

[WRH+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer, Berlin, Heidelberg, 2012.

[WVS+21] Mario Winter, Karin Vosseberg, Frank Simon, Andreas Spillner, Annette Simon, Helmut Pichler, and Misperi Sakarya. Softwaretest in praxis und forschung, 2021.

[Xu11] Dianxiang Xu. A tool for automated test code generation from high-level petri nets. In Lars M. Kristensen and Laure Petrucci, editors, *Applications and Theory of Petri Nets*, pages 308–317, Berlin, Heidelberg. Springer Berlin Heidelberg, 2011.

[ZHM97] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.

List of Figures

1.1. Design Science Research framework 9

4.1. Architecture of the Piggy Bank system. 30

5.1. Structure of the IBI Metamodel 37
5.2. The Component-based System Metamodel. 38
5.3. The Test Metamodel. 42
5.4. The Interaction Expectation Metamodel. 44
5.5. The Interaction Expectation Verification Metamodel. 47

6.1. The simplified IBI Testing Process. 50
6.2. The $provides$ association in the IBI Metamodel (left) and its representation in the property graph (right). 55
6.3. The *Component Interaction Roles* in the IBI Metamodel (left) and their representation in the property graph (right). 56
6.4. The $boundTo$ association in the IBI Metamodel and its representation in the property graph. 58
6.5. IBI Testing Process 59
6.6. Capture IBI component models 60
6.7. Create IBI system model 61
6.8. Extract Unit Test-based Interaction Expectations 63
6.9. Derivation of Interaction Expectations. 64
6.10. Collect Interaction Expectations. 66
6.11. Derive Potential Component Behavior Sequences 67
6.12. Iterative Interaction Expectation verification 70
6.13. Verify Interaction Expectations 73
6.14. Collect Test Gaps 74
6.15. Create Report 75

7.1. Component diagram of the *PBS_1*. 78
7.3. Sequence diagram of the Transfer Gateway's unit test case *utc1_TG*. 82
7.4. IBI component model for the Transfer Gateway. 83
7.5. Sequence diagram of the Account Twin Service's unit test case *utc1_ATS* (credit transaction). 84
7.6. Sequence diagram of the Account Twin Service's unit test case *utc2_ATS* (debit transaction). 85
7.7. Sequence diagram of the Notification Service's unit test *utc1_NS*. . 85
7.8. Interface bindings for *PBS_1*. 86

7.9. First PBG for *die1* in *PBS_1*. . . . 88
7.10. Second PBG for *die1* in *PBS_1*. . . . 88
7.11. First PBG for *utie1_TG*. . . . 88
7.12. Second PBG for *utie1_TG*. . . . 88
7.14. Interaction test based on *cb2_ATS*. . . . 90
7.15. Interaction test based on *cb1_ATS*. . . . 90
7.16. Interaction test based on *cb1_NS*. . . . 91
7.18. The integration test case simulated by the generated interaction test cases. . . . 92

8.1. Overview of InterACt's architecture. . . . 95
8.2. A section of the IBI system model of the Piggy Bank system, as stored in Neo4j. . . . 97
8.3. System overview displaying detected components, test results, and expectation statuses. . . . 105
8.4. Visualization of integration tests for an interaction expectation. Two interaction test cases have been passed already, and the third one is still pending verification. . . . 106

Acronyms

ATS	Account Twin Service 84–86, 88
BIT	Built-in Testing 21, 22, 26
CBSS	Component-based Software System 3, 5–7, 11, 13, 15, 16, 18, 21, 24, 26, 31, 35, 36, 39, 44, 48, 49, 52, 118, 119, 131, 133, 137
CUT	Component Under Test 20, 42, 63, 93, 94, 98–101, 124, 134
DSR	Design Science Research 8, 9, 12, 13, 21, 93, 111, 131, 137
IBI	Interaction-based Integration 5–8, 10–13, 21–26, 35–37, 39, 41, 44–46, 48–52, 54–68, 72, 74, 75, 77–80, 82–87, 89, 92–94, 96, 97, 100–104, 107, 111, 113–118, 121, 122, 124–137, 149, 150
ITS	Interaction Test Suite 51, 71, 72
NS	Notification Service 85, 86
PBS	Piggy Bank System 86, 88, 89, 159
PoC	Proof of Concept 93, 101, 107, 111, 128, 134
RPQ	Regular Path Query 49, 52–54, 56, 63, 103
SUT	System Under Test 19, 36, 49, 50, 57, 58, 61, 62, 64, 66, 67, 71, 74, 79, 105, 132, 133, 153–156
TG	Transfer Gateway 82, 83, 86, 87, 89
UTS	Unit Test Suite 50, 51, 60, 71, 72, 79

Symbols

$?c$	A yet to be determined component. 63
$?q_{c_1}$	A yet to be determined state of a component. 63, 66
$?q_{c_i}$	A yet to be determined state of a component. 63, 71
$?q_{c_n}$	A yet to be determined state of a component. 63, 66
$Boolean$	A Boolean value (true or false). 54, 61, 64, 67, 155
c	A component. 17, 56–58, 60, 61, 63, 64, 69, 71, 72, 103, 153–156
c_1	A component. 58
CB	A set of component behaviors. 68, 69
cb	A component behavior. 69, 72, 154–156
cb_1	The first component behavior in a sequence. 60, 69, 71, 73
CB_c	The finite set of component behaviors of component c. 56, 58, 69, 71, 72, 153
cb_c	A component behavior in the set of component behaviors CB_c of a component c. 56, 57
CB_{ctc}	The set of component behaviors of a concrete test case ctc. 61, 69
CB_{exp}	The set of expected component behaviors. 69
cb_i	A component behavior. 60, 61, 69, 71, 72, 153, 156
cb_{i+1}	The next component behavior in a sequence. 60, 63, 156
cb_j	A component behavior. 69, 71
cb_n	The last component behavior in a sequence. 60, 69, 71, 73
cbr	A component behavior result. 56, 57, 69, 72
cbr_i	A component behavior result. 60, 63, 156
$CBSeq$	Set of component behavior sequences. 67, 71, 73
CB_{source}	The set of component behaviors that have no previous component behaviors in a potential behavior graph. 69
CB_{SUT}	The finite set of component behaviors of the SUT. 58, 68
cbt	A component behavior trigger. 56, 69
cbt_{adapt}	The adapted component behavior trigger. 71
cbt_i	A component behavior trigger. 60, 156
cbt_{i+1}	The component behavior trigger of the component behavior following cb_i. 60, 63, 156
cbt_{orig}	The original component behavior trigger. 71, 72
c_i	A component. 58, 63, 74, 154

c_j A component. 58, 74

c_n A component. 58

C_{SUT} The set of all components in the SUT. 58, 60–64, 69, 71, 74

ctc A concrete test case. 60, 61, 69, 153

CTC_c The set of concrete test cases for a component c. 60, 69

$depth(cb)$ The depth of a component behavior cb in a potential behavior graph. 69

ie_d A derived interaction expectation. 65

E A finite set of edges in a property graph PG. 52, 53, 68, 69, 154, 155

e_1 The first edge of a path p. 52, 53, 156

$edge(e)$ A total function that associates a pair of nodes to each edge in PG, formally defined as $edge : E \to N \times N$. 52, 53, 154

$edges(p)$ A function returning the sequence of edge labels for a given path p, formally defined as $edges : p \to L_{E^{\pm}}{}^{*}$. 53, 54, 154

e_i An edge of a path p. 53, 156

ei_1 The first environment interaction in a sequence. 60

ei_i An environment interaction in a sequence. 60, 63

ei_{n-1} The last environment interaction in a sequence. 60

e_{len} The last edge of a path p. 52, 53, 156

Env_c The set of all components interacting with component c in a SUT. 58, 63, 74

Env_{c_i} The set of all components interacting with component c_i in a SUT. 58

IE Set of interaction expectations. 66, 67, 73

ie An interaction expectation. 62, 68

IE_a Set of abstract interaction expectations. 64

iea An abstract interaction expectation. 64–66

IE_d Set of derived interaction expectations. 65, 67

IE_{utb} Set of unit test-based interaction expectations. 63, 67

$bound$ Function that determines if an outgoing interface is aligned with an incoming interface. 61, 62, 64, 66, 67

IF_c The set of interfaces of component c. 56, 103

IF_{in} The domain of incoming interfaces. 61, 64, 67, 154

if_{in} An incoming interface in the set of all incoming interfaces IF_{in} of the SUT. 57, 62, 63, 66, 68, 69, 72

$IF_{\text{in},c}$ The set of incoming interfaces of component c 56, 58, 103

$IF_{\text{in,exp}}$ The set of incoming interfaces by which stimuli are expected. 62–65, 67–69, 87

$IF_{\text{in,SUT}}$ The set of incoming interfaces contained in the SUT 58, 62, 66

IF_{out} The domain of outgoing interfaces. 61, 64, 67, 155

if_{out}	An outgoing interface in the set of all ougoing interfaces IF_{out} of the SUT. 57, 62, 63, 66, 68, 69, 72
$IF_{\text{out},c}$	The set of outgoing interfaces of component c 56, 58
$IF_{\text{out,exp}}$	The set of outgoing interfaces by which reactions are expected. 64–66
$IF_{\text{out,SUT}}$	The set of outgoing interfaces contained in the SUT 58, 62
K	A finite set of property keys in a property graph PG. 52, 155
l	A label from L_E. 53
$\lambda_E(e)$	A total function that associates each edge with exactly one label from L_E, formally defined as $\lambda_E : E \to L_E$. 52, 53, 155
$\lambda_N(n)$	A total function that associates each node with exactly one label from L_N, formally defined as $\lambda_N : N \to L_N$. 52, 56–58, 155
L_E	A finite set of edge labels in a property graph PG. 52, 53, 155
len	The length of a path p in PG, where $len \geq 1$. 53, 155, 157
$length(p)$	The length of a path p in a graph. 69
$L_{E^\pm}$	The set of original edge labels and their inverse labels, defined as $L_{E^\pm} = L_E \cup \{l^- \mid l \in L_E\}$. 53, 54, 154–157
L_N	A finite set of node labels in a property graph PG. 52, 155
m	A message. 57, 63, 66, 69, 72
$match(r, w)$	A total function determining whether a sequence of edge labels matches a regular expression, formally defined as $match : \mathcal{R}(L_{E^\pm}) \times {L_{E^\pm}}^* \to Boolean$. 54, 155
MSG	The set of all messages. 58, 66
N	A finite set of nodes in a property graph PG. 52, 53, 58, 154, 155
n	An index variable symbolizing the total number of elements. 62, 156
p	A finite sequence of nodes and directed edges forming a path in PG. 52–54, 69, 154–157
$Paths(PG)$	The set of all paths p in a property graph PG. 53, 54
PBG	A potential behavior graph. 68, 69, 104
PG	A property graph, representing a finite directed labeled graph with property-value pairs. 52–54, 154–157
$pos(cb)$	Fucntion returning the position of a component behavior cb in a concrete test case. 61, 69
$prop(n, k)$	A partial function that associates nodes with selected property values based on a key, formally defined as $prop : N \times K \to V$. 52, 54, 155
q_c	A state of component c. 156

q_{c_1}	The state of a component. 62
q_{c_n}	The state of a component. 62
q_{post}	The state after a component behavior cb. 57, 63, 69, 72
$q_{\text{post},i}$	The state after a component behavior cb_i. 60, 63
q_{pre}	The state before a component behavior cb. 56, 57, 69, 71
$q_{\text{pre,adapt}}$	The state before the adapted component behavior trigger. 71, 72
$q_{\text{pre},i}$	The state before a component behavior cb_i. 60
$q_{\text{pre},i+1}$	The state before a component behavior cb_{i+1}. 60, 63
q_{SUT}	A state of the SUT defined by a set of component states $\{q_{c_1}, \ldots, q_{c_n}\}$. 62, 71, 87
r	A reaction. 57, 68
$R_{selected}$	The set of sets of reactions to stimulate the SUT based on an abstract interaction expectation. 66
R_{cb}	The set of reactions resulting from a component behavior cb. 69
R	The set of reactions resulting from a component behavior cb. 57, 62, 66, 68, 69, 72, 87
$regex$	A regular expression over $L_{E^{\pm}}$. 53, 54
$regex_i$	A regular expression over $L_{E^{\pm}}$. 53, 54
$regex_j$	A regular expression over $L_{E^{\pm}}$. 53, 54
R_i	The set of reactions of a component behavior result cbr_i. 60, 63
$\mathcal{R}(L_{E^{\pm}})$	The set of valid regular expressions over $L_{E^{\pm}}$. 53, 54, 155
R_{pre,cb_i}	The set of reactions before a component behavior cb_i. 71, 72
RS_{cb}	The set of reactions to replace the stimuli of a component behavior cb. 69
S	The set of stimuli triggering a component behavior cb. 56, 57, 69, 71, 72
s	A stimulus. 56, 57, 63, 68
S_{adapt}	The adapted stimuli of a component behavior trigger. 72
sbt	A system behavior trigger. 62, 63, 65–69, 87
S_{cb}	The set of stimuli triggering a component behavior cb. 69
S_i	The set of stimuli of a component behavior trigger cbt_i. 60, 63
S_{i+1}	The set of stimuli of a component behavior trigger cbt_{i+1}. 63
σ_1	The traversal direction of the first edge e_1 of a path p 52, 53
σ_i	The traversal direction of the edge e_i of a path p 53
σ_{len}	The traversal direction of the last edge e_{len} of a path p 52, 53
ie_{utb}	A unit test-based interaction expectation. 63
V	An infinite set of property values in a property graph PG. 52, 155
ν	A node. 58
ν_0	The first node of a path p. 52

ν_1	The second node of a path p. 52
ν_i	A node. 53, 54, 157
ν_{i-1}	The node before node ν_i in a path p. 53
ν_j	A node. 53, 54
ν_{len}	The last node of a path p with length len. 52
w	A word over $L_{E^{\pm}}$, representing a sequence of edge labels along a path p in PG. 54

Identifiers

aie1	The abstract interaction exepectation representing the use case that both PBS *PBS_1* and *PBS_2* should implement. 81, 87, 92, 159, 160
assert1_ATS	The assertion of the unit test case *utc1_ATS*. 84
assert1_NS	The assertion of the unit test case *utc1_NS*. 86
assert1_TG	The assertion of the unit test case *utc1_TG*. 83, 87
assert2_ATS	The assertion of the unit test case *utc2_ATS*. 85
assert_aie1	The assertion of the abstract interaction expectation *aie1*. 81, 87
cb1'_ATS	The Account Twin Service's behavior when it has the state *q1'_ATS* and receives the stimulus message *m_r1_TG*. 90, 160
cb1'_NS	The Notification Service's behavior when it has the state *q1'_NS* and receives the stimulus message *m_r1'_ATS*. 160
cb1_ATS	The Account Twin Service's behavior when it has the state *q1_ATS* and receives the stimulus message *m_s1_ATS*. 84, 88–90, 150, 160, 161
cb1_NS	The Notification Service's behavior when it has the state *q1_NS* and receives the stimulus message *m_s1_NS*. 86, 89, 91, 150, 160, 161
cb1_TG	The Transfer Gateway's behavior when it has the state *q1_TG* and receives the stimulus message *m_s1_TG*. 82, 89, 160, 162
cb2_ATS	The Account Twin Service's behavior when it has the state *q3_ATS* and receives the stimulus message *m_s2_ATS*. 85, 88–90, 150, 161, 162
cb2_TG	The Transfer Gateway's behavior when it has the state *q2_TG* and receives the stimulus message *m_s2_TG*. 83, 162
die1	The derived interaction expectation that resulted from the abstract interaction expectation *aie1*. 87–91, 150
envi1_TG	The environment interaction started by the Transfer Gateway in the unit test case *utc1_TG*. 83, 87, 161, 162
if_in1_ATS	The Account Twin Service's incoming interface to receive transactions. 80, 81, 84, 88, 160, 161
if_in1_NS	The Notification Service's incoming interface to receive information about handled transactions. 81, 86, 88, 91, 161

if_in1_TG — The Transfer Gateway's incoming interface to receive transfers. 80, 82, 87, 160, 161

if_in2_TG — The Transfer Gateway's incoming interface to receive responses to the transactions it sends by the outgoing interface *if_out1_TG*. 80, 81, 83, 87, 88, 161

if_in_aie1 — The incoming interface definition that specifies the reaction interface expectation of the abstract interaction expectation *aie1*. 81, 87

if_out1_ATS — The Account Twin Service's outgoing interface to inform the environment about handled transactions. 81, 84–86, 88, 160, 161

if_out1_NS — The Notification Service's outgoing interface to notify about handled transactions. 81, 86, 160

if_out1_TG — The Transfer Gateway's outgoing interface to pass transactions to the environment. 80–82, 87, 88, 160

if_out2_ATS — The Account Twin Service's outgoing interface to respond to the messages it receives by its invoming interface *if_in1_ATS*. 80, 81, 84, 85, 88, 160, 161

if_out2_TG — The Transfer Gateway's outgoing interface to respond to the messages it receives by invoming interface *if_in1_TG*. 80, 83

if_out_aie1 — The outgoing interface definition that specifies the stimulus interface exepectation of the abstract interaction expectation *aie1*. 81, 87

m_r1'_ATS — The reaction message the Account Twin Service sends by its outgoing interface *if_out1_ATS* as a result of the component behavior *cb1'_ATS*. 90, 91, 159

m_r1'_NS — The reaction message the Notification Service sends by its outgoing interface *if_out1_NS* as a result of the component behavior *cb1'_NS*. 90, 91

m_r1_ATS — The two reaction messages the Account Twin Service sends by its outgoing interface *if_out1_ATS* as a result of the component behavior *cb1_ATS*. 84

m_r1_NS — The reaction message the Notification Service sends by its outgoing interface *if_out1_NS* as a result of the component behavior *cb1_NS*. 86

m_r1_TG — The reaction message the Transfer Gateway sends by the outgoing interface *if_out1_TG* as a result of the component behavior *cb1_TG*. 82, 83, 87, 89, 90, 159

m_r2'_ATS — The reaction message the Account Twin Service sends by its outgoing interface *if_out2_ATS* as a result of the component behavior *cb1'_ATS*. 90, 91

m_r2_ATS — The two reaction messages the Account Twin Service sends by its outgoing interface *if_out2_ATS* as a result of the component behavior *cb1_ATS*. 84

m_r2_TG — The stimulus message the Transfer Gateway receives by its incoming interface *if_in1_TG* as a result of the environment interaction *envi1_TG* in the unit test case *utc1_TG*. 83

m_r3_ATS — The reaction message the Account Twin Service sends by its outgoing interface *if_out1_ATS* as a result of the component behavior *cb2_ATS*. 85

m_r4_ATS — The reaction message the Account Twin Service sends by its outgoing interface *if_out2_ATS* as a result of the component behavior *cb2_ATS*. 85

m_s1_ATS — The stimulus message the Account Twin Service receives by its incoming interface *if_in1_ATS* in the unit test case *utc1_ATS*. 84, 159

m_s1_NS — The stimulus message the Notification Service receives by its incoming interface *if_in1_NS* in the unit test case *utc1_NS*. 86, 90, 159

m_s1_TG — The stimulus message the Transfer Gateway receives by its incoming interface *if_in1_TG* in the unit test case *utc1_TG*. 82, 87, 159

m_s2_ATS — The stimulus message the Account Twin Service receives by its incoming interface *if_in1_ATS* in the unit test case *utc2_ATS*. 84, 85, 159

m_s2_TG — The stimulus message the Transfer Gateway receives by its incoming interface *if_in2_TG* as a result of the environment interaction *envi1_TG* in the unit test case *utc1_TG*. 83, 87, 91, 159

PBS_1 — The Piggy Bank system composed of the Trasfer Gateway, Account Twin Service, and Notification Service. 78, 79, 81, 86, 88, 89, 91, 92, 149, 150, 159

PBS_2 — The Piggy Bank system composed of the Trasfer Gateway, and Account Twin Service. 78, 79, 81, 86, 88, 89, 91, 92, 159

q1'_ATS — The Account Twin Service's initial state in the interaction test case based on *utc1_ATS*. *Acc_2* with a balance of 100€ is in the set of Accounts maintained by the Account Twin Service. 89, 159

q1'_NS — The Notification Service's initial state in the interaction test case based on *utc1_NS*. Notifications for transactions involving *Acc_2* are enabled. 159

q1_ATS — The Account Twin Service's initial state in the unit test case *utc1_ATS*. *Acc_3* with a balance of 100€ is in the set of Accounts maintained by the Account Twin Service. 84, 89, 159

q1_NS — The Notification Service's initial state in the unit test case *utc1_NS*. Notifications for transactions involving *Acc_5* are enabled. 86, 159

q1_TG — The Transfer Gateway's initial state in the unit test case *utc1_TG*. *Acc_2* is in maintained by the Transfer Gateway. 82, 159

q2_ATS — The Acount Twin Service's state resulting from *cb1_ATS*. 84

q2_NSS — The Notification Service's state resulting from *cb1_NS*. 86

q2_TG	The Transfer Gateway's state resulting from *cb1_TG*. 82, 87, 159
q3'_ATS	The Account Twin Service's initial state in the interaction test case based on *utc2_ATS*. *Acc_2* with a balance of 600€ is in the set of Accounts maintained by the Account Twin Service. 89
q3_ATS	The Account Twin Service's initial state in the unit test case *utc2_ATS*. *Acc_4* with a balance of 600€ is in the set of Accounts maintained by the Account Twin Service. 84, 89, 159
q3_TG	The Transfer Gateway's state resulting from *cb2_TG*. 83
q4_ATS	The Acount Twin Service's state resulting from *cb2_ATS*. 85
utc1_ATS	The Account Twin Service's unit test case to test a credit transaction. 79, 84, 89, 149, 159, 161
utc1_NS	The Notification Service's unit test case to test notifications for transactions. 79, 85, 86, 149, 159, 161
utc1_TG	The Transfer Gateway's unit test case to test successful transfer handling. 79, 82, 83, 87, 89, 149, 159, 161
utc2_ATS	The Account Twin Service's unit test case to test a debit transaction. 79, 84, 85, 89, 149, 159, 161, 162
utie1_TG	The unit test-based interaction expectation extracted from the environment interaction *envi1_TG*. 87–89, 91, 150

Related Work from the SE Group, RWTH Aachen, June 25

The following section gives an overview of related work done at the SE Group, RWTH Aachen. More details can be found on the website `www.se-rwth.de/topics/` or in [HMR+19]. The work presented here mainly has been guided by our mission statement:

Our mission is to define, improve, and industrially apply *techniques*, *concepts*, and *methods* for *innovative and efficient development* of software and software-intensive systems, such that *high-quality* products can be developed in a *shorter period of time* and with *flexible integration* of *changing requirements*. Furthermore, we *demonstrate the applicability* of our results in various domains and potentially refine these results in a domain specific form.

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04c]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.", [JWCR18] addresses the question of how digital and organizational techniques help to cope with the physical distance of developers and [RRSW17] addresses how to teach agile modeling.

Modeling will increasingly be used in development projects if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKR+06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum11, Rum12] and [Rum16, Rum17], the UML/P, a variant of the UML especially designed for programming, refactoring, and evolution is defined.

The language workbench MontiCore [GKR+06, GKR+08, HKR21] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09], and refactoring in various modeling and programming languages [PR03]. To better understand the effect of an agile evolving design, we discuss the need for semantic differencing in [MRR10].

In [FHR08] we describe a set of general requirements for model quality. Finally, [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve the reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation. In [KMA+16] we have also introduced a classification of ways to reuse modeled software components.

Artifacts in Complex Development Projects

Developing modern software solutions has become an increasingly complex and time consuming process. Managing the complexity, the size, and the number of artifacts developed and used during a project together with their complex relationships is not trivial [BGRW17].

To keep track of relevant structures, artifacts, and their relations in order to be able, e.g., to evolve or adapt models and their implementing code, the *artifact model* [GHR17, Gre19] was introduced. [BGRW18] and [HJK+21] explain its applicability in systems engineering based on MDSE projects and [BHR+18] applies a variant of the artifact model to evolutionary development, especially for CPS.

An artifact model is a meta-data structure that explains which kinds of artifacts, namely code files, models, requirements files, etc. exist and how these artifacts are related to each other. The artifact model, therefore, covers the wide range of human activities during the development down to fully automated, repeatable build scripts. The artifact model can be used to optimize parallelization during the development

and building, but also to identify deviations of the real architecture and dependencies from the desired, idealistic architecture, for cost estimations, for requirements and bug tracing, etc. Results can be measured using metrics or visualized as graphs.

Artificial Intelligence in Software Engineering

MontiAnna is a family of explicit domain specific languages for the concise description of the architecture of (1) a neural network, (2) its training, and (3) the training data [KNP+19]. We have developed a compositional technique to integrate neural networks into larger software architectures [KRRW17] as standardized machine learning components [KPRS19]. This enables the compiler to support the systems engineer by automating the lifecycle of such components including multiple learning approaches such as supervised learning, reinforcement learning, or generative adversarial networks.

For analysis of MLOps in an agile development, a software 2.0 artifact model distinguishing different kinds of artifacts is given in [AKK+21].

According to [MRR11g] the semantic difference between two models are the elements contained in the semantics of the one model that are not elements in the semantics of the other model. A smart semantic differencing operator is an automatic procedure for computing diff witnesses for two given models. Such operators have been defined for Activity Diagrams [MRR11d], Class Diagrams [MRR11b], Feature Models [DKMR19], Statecharts [DEKR19], and Message-Driven Component and Connector Architectures [BKRW17, BKRW19]. We also developed a modeling language-independent method for determining syntactic changes that are responsible for the existence of semantic differences [KR18a].

We apply logic, knowledge representation, and intelligent reasoning to software engineering to perform correctness proofs, execute symbolic tests, or find counterexamples using a theorem prover. We have defined a core theory in [BKR+20], which is based on the core concepts of Broy's Focus theory [RR11, BR07], and applied it to challenges in intelligent flight control systems and assistance systems for air or road traffic management [KRRS19, KMP+21, HRR12].

Intelligent testing strategies have been applied to automotive software engineering [EJK+19, DGH+19, KMS+18], or more generally in systems engineering [DGH+18]. These methods are realized for a variant of SysML Activity Diagrams (ADs) and Statecharts.

Machine Learning has been applied to the massive amount of observable data in energy management for buildings [FLP+11, KLPR12] and city quarters [GLPR15] to optimize operational efficiency and prevent unneeded CO_2 emissions or reduce costs. This creates a structural and behavioral system theoretical view on cyber-physical systems understandable as essential parts of digital twins [RW18, BDH+20].

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P, [Hab16] for MontiArc is used in domains such as cars or robotics [HRR12], and [AMN+20a] for enterprise information systems based on the MontiCore language workbench [KRV10, GKR+06, GKR+08, HKR21].

In [KRV06], we discuss additional roles necessary in a model-based software development project. [GKR+06, GHK+15, GHK+15a] discuss mechanisms to keep generated and handwritten code separated. In [Wei12, HRW15, Hoe18], we demonstrate how to systematically derive a transformation language in concrete syntax and, e.g., in [HHR+15, AHRW17] we have applied this technique successfully for several UML sub-languages and DSLs.

[HNRW16] presents how to generate extensible and statically type-safe visitors. In [NRR16], we propose the use of symbols for ensuring the validity of generated source code. [GMR+16] discusses product lines of template-based code generators. We also developed an approach for engineering reusable language components [HLN+15, HLN+15a].

To understand the implications of executability for UML, we discuss the needs and the advantages of executable modeling with UML in agile projects in [Rum04c], how to apply UML for testing in [Rum03], and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML) & the UML-P Tool

Starting with the early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the books [Rum16, Rum17] and is implemented in [Sch12].

Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09], [BCGR09a], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied when checking variants of class diagrams [MRR11e] and object diagrams [MRR11c] or the consistency of both kinds of diagrams [MRR11f]. We also apply these concepts to activity diagrams [MRR11a] which allows us to check for semantic differences in activity diagrams [MRR11d]. The basic semantics for ADs and their semantic variation points are given in [GRR10].

We also discuss how to ensure and identify model quality [FHR08], how models, views, and the system under development correlate to each other [BGH+98b], and how to use modeling in agile development projects [Rum04c], [Rum03] and [Rum02].

The question of how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99a], [FEL+98] and [SRVK10].

The UML-P tool was conceptually defined in [Rum16, Rum17, Rum12, Rum11], got the first realization in [Sch12], and is extended in various ways, such as logically or physically distributed computation [BKRW17a]. Based on a detailed examination [JPR+22], insights are also transferred to the SysML 2.

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use than general-purpose programming languages but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08, HKR21] allows the specification of an integrated abstract and concrete syntax format [KRV07b, HKR21] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Voe11, HLN+15, HLN+15a, HRW18, BEK+18b, BEK+19, Sch12] and can, thus, easily be reused. We discuss the roles in software development using domain specific languages already in [KRV06] and elaborate on the engineering aspect of DSL development in [CFJ+16].

[Wei12, HRW15, Hoe18] present an approach that allows the creation of transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11, GMR+16]. [BDL+18] presents a method to derive internal DSLs from grammars. In [BJRW18], we discuss the translation from grammars to accurate metamodels. Successful applications have been carried out in the Air Traffic Management [ZPK+11] and television [DHH+20] domains. Based on the concepts described above, meta modeling, model analyses, and model evolution have been discussed in [LRSS10] and [SRVK10].

[BJRW18] describes a mapping bridge between both. DSL quality in [FHR08], instructions for defining views [GHK+07] and [PFR02], guidelines to define DSLs [KKP+09], and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

A broader discussion on the *global* integration of DSMLs is given in [CBCR15] as part of [CCF+15a], and [TAB+21] discusses the compositionality of analysis techniques for models.

The MontiCore language workbench has been successfully applied to a larger number of domains, resulting in a variety of languages documented, e.g., in [AHRW17, BEH+20, BHR+21, BPR+20, HHR+15, HJRW20, HMR+19, HRR12, PBI+16, RRW15] and Ph.D. theses like [Ber10, Gre19, Hab16, Her19, Kus21, Loo17, Pin14, Plo18, Rei16, Rot17, Sch12, Wor16].

Software Language Engineering

For a systematic definition of languages using a composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15a]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10, HR17, HKR21, HRW18, BPR+20, BEK+19].

In [SRVK10] we discuss the possibilities and the challenges of using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Voe11, Naz17, KRV08, HLN+15, HLN+15a, HNRW16, HKR21, BEK+18b, BEK+19] and the backend [RRRW15b, NRR16, GMR+16, HKR21, BEK+18b, BBC+18]. In [GHK+15, GHK+15a], we discuss the integration of handwritten and generated object-oriented code. [KRV10] describes the roles in software development using domain specific languages.

Language derivation is to our belief a promising technique to develop new languages for a specific purpose, e.g., model transformation, that relies on existing basic languages [HRW18].

How to automatically derive such a transformation language using a concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs.

We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15, HHK+13] that are derived from base languages to be able to constructively describe differences between model variants usable to build feature sets.

The derivation of internal DSLs from grammars is discussed in [BDL+18] and a translation of grammars to accurate metamodels in [BJRW18].

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services.

We use streams, statemachines [GKR+96], and components [BR07] as well as expressive forms of composition and refinement [PR99, PR97, RW18] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR10, HRR12] for architecture design and extensions for states [RRW13c, BKRW17a, RRW14a, Wor16]. In [RRW13], we introduce a code generation framework for MontiArc. [RRRW15b] describes how the language is composed of individual sublanguages.

MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. Other extensions are concerned with modeling cloud architectures [PR13], security in [HHR+15], and the robotics domain [AHRW17, AHRW17b]. Extension mechanisms for MontiArc are generally discussed in [BHH+17].

[GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants.

[MRR14b] provides a precise technique for verifying the consistency of architectural views [Rin14, MRR13] against a complete architecture to increase reusability. We discuss the synthesis problem for these views in [MRR14a]. An experience report [MRRW16] and a methodological embedding [DGH+19] complete the core approach.

Extensions for co-evolution of architecture are discussed in [MMR10], for powerful analyses of software architecture behavior evolution provided in [BKRW19], techniques for understanding semantic differences presented in [BKRW17], and modeling techniques to describe dynamic architectures shown in [HRR98, HKR+16, BHK+17, KKR19].

Compositionality & Modularity of Models

[HKR+09, TAB+21] motivate the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07, RW18] and algebraically grounded in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10, HKR21] that can even be used to develop modeling tools in a compositional form [HKR21, HLN+15, HLN+15a, HNRW16, NRR16, HRW18, BEK+18b, BEK+19, BPR+20, KRV07b]. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09].

[Voe11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15b] applies compositionality to robotics control.

[CBCR15] (published in [CCF+15a]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information, we have developed the concept of tagging languages in [GLRR15, MRRW16]. It allows the description of additional information for model elements in separated documents, facilitates reuse, and allows typing tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision, and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96, RK96]. An extended version especially suited for the UML is given in [GRR09], [BCGR09a] and in [BCGR09] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08] or sequence diagrams in [BGH+98a].

To better understand the effect of an evolved design, detection of semantic differencing, as opposed to pure syntactical differences, is needed [MRR10]. [MRR11d, MRR11a] encode a part of the semantics to handle semantic differences of activity diagrams. [MRR11f, MRR11f] compare class and object diagrams with regard to their semantics. Furthermore, [BKRW17] compares component and connector architectures similar to SysML' block definition diagrams and [RSR+99] discusses the combination of those architectures with the UML.

In [BR07, RR11], a precise mathematical model for distributed systems based on black-box behaviors of components is defined and accompanied by automata in [Rum96]. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of exemplary object interaction, today called sequence diagram. [BGH+98b] discusses the relationships between

a system, a view, and a complete model in the context of the UML.

[GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these to class and object diagrams in [MRR11f] as well as activity diagrams in [GRR10].

[Rum12] defines the semantics in a variety of code and test case generation, refactoring, and evolution techniques. [LRSS10] discusses the evolution and related issues in greater detail. [RW18] discusses an elaborated theory for the modeling of underspecification, hierarchical composition, and refinement that can be practically applied to the development of CPS.

A first encoding of these theories in the Isabelle verification tool is defined in [BKR+20].

Evolution and Transformation of Models

Models are the central artifacts in model driven development, but as code, they are not initially correct and need to be changed, evolved, and maintained over time. Model transformation is therefore essential to effectively deal with models [CFJ+16].

Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04c, MRR10], refinement [PR99, PR97, KPR97, PR94], decomposition [PR99, KRW20], synthesis [MRR14a], refactoring [Rum12, PR03], translating models from one language into another [MRR11e, Rum12], systematic model transformation language development [Wei12, HRW15, Hoe18, HHR+15], repair of failed model evolution [KR18a].

[Rum04c] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97] and refining pipe-and-filter architectures is explained in [PR99, PR97]. This has e.g. been applied for robotics in [AHRW17, AHRW17b].

Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. [HRRS11, HRR+11, HRRS12] encode these in constructive Delta transformations, which are defined in derivable Delta languages [HHK+13].

Translation between languages, e.g., from class diagrams into Alloy [MRR11e] allows for comparing class diagrams on a semantic level. Similarly, semantic differences of evolved activity diagrams are identified via techniques from [MRR11d] and for Simulink models in [RSW+15].

Variability and Software Product Lines (SPL)

Products often exist in various variants, for example, cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08, GKPR08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12].

Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRRS11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe special variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13, HHK+15] and [HRW15] describe an approach to systematically derive delta languages.

We also apply variability modeling languages to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02] and generators [GMR+16]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09], leverage features for their compositional reuse [BEK+18b, BEK+19], and applied it as a semantic language refinement on Statecharts in [GR11].

Digital Twins and Digital Shadows in Engineering and Production

The digital transformation of production changes the life cycle of the design, the production, and the use of products [BDJ+22]. To support this transformation, we can use Digital Twins (DTs) and Digital Shadows (DSs). In [DMR+20] we define: "A digital twin of a system consists of a set of models of the system, a set of digital shadows, and provides a set of services to use the data and models purposefully with respect to the original system."

We have investigated how to synthesize self-adaptive DT architectures with model-driven methods [BBD+21a] and have applied it e.g. on a digital twin for injection molding [BDH+20]. In [BDR+21] we investigate the economic implications of digital twin services.

Digital twins also need user interaction and visualization, why we have extended the infrastructure by generating DT cockpits [DMR+20]. To support the DevOps approach in DT engineering, we have created a generator for low-code development platforms for digital twins [DHM+22] and sophisticated tool chains to generate process-aware digital twin cockpits that also include condensed forms of event logs [BMR+22].

[BBD+21b] describes a conceptual model for digital shadows covering the purpose, relevant assets, data, and metadata as well as connections to engineering models. These can be used during the runtime of a DT, e.g. when using process prediction services within DTs [BHK+21].

Integration challenges for digital twin systems-of-systems [MPRW22] include, e.g., the horizontal integration of digital twin parts, the composition of DTs for different perspectives, or how to handle different lifecycle representations of the original system.

Modeling for Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12, BBR20] are complex, distributed systems that control physical entities. In [RW18], we discuss how an elaborated theory can be practically applied to the development of CPS. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12, KRRW17], autonomous driving [BR12b, KKR19], and digital twin development [BDH+20] to processes and tools to improve the development as well as the product itself [BBR07].

In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest to European avionics [ZPK+11]. Optimized [KRS+18a] and domain specific code generation [AHRW17b], and the extension to product lines of CPS [RSW+15, KRR+16, RRS+16] are key for CPS.

A component and connector architecture description language (ADL) suitable for the specific challenges in robotics is discussed in [RRW13c, RRW14a, Wor16, RRSW17, Wor21]. In [RRW12], we use this language for describing requirements and in [RRW13], we describe a code generation framework for this language. Monitoring for smart and energy efficient buildings is developed as an Energy Navigator toolset [KPR12, FPPR12, KLPR12].

Model-Driven Systems Engineering (MDSysE)

Applying models during Systems Engineering activities is based on the long tradition of contributing to systems engineering in automotive [FND+98] and [GHK+08a], which culminated in a new comprehensive model-driven development process for automotive software [KMS+18, DGH+19]. We leveraged SysML to enable the integrated flow from requirements to implementation to integration.

To facilitate the modeling of products, resources, and processes in the context of Industry 4.0, we also conceived a multi-level framework for production engineering based on these concepts [BKL+18] and addressed to bridge the gap between functions and the physical product architecture by modeling mechanical functional architectures in SysML [DRW+20]. For that purpose, we also did a detailed examination of the upcoming SysML 2.0 standard [JPR+22] and examined how to extend the SPES/CrEST methodology for a systems engineering approach [BBR20].

Research within the excellence cluster Internet of Production considers fast decision making at production time with low latencies using contextual data traces of production systems, also known as Digital Shadows (DS) [SHH+20]. We have investigated how to derive Digital Twins (DTs) for injection molding [BDH+20], how to generate interfaces between a cyber-physical system and its DT [KMR+20], and have proposed model-driven architectures for DT cockpit engineering [DMR+20].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, GKR+96, BCR07b, BCGR09a, BCGR09], (2) understanding the refinement [PR94, RK96, Rum96, RW18] and composition [GR95, GKR96, RW18] of statemachines, and (3) applying statemachines for modeling systems.

In [Rum96, RW18] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [GKR96, BR07].

We apply these techniques, e.g., in MontiArcAutomaton [RRW13, RRW14a, RRW13, RW18], in a robot task modeling language [THR+13], and in building management systems [FLP+11b].

Model-Based Assistance and Information Services (MBAIS)

Assistive systems are a special type of information system: they (1) provide situational support for human behavior (2) based on information from previously stored and real-time monitored structural context and behavior data (3) at the time the person needs or asks for it [HMR+19]. To create them, we follow a model centered architecture approach [MMR+17] which defines systems as a compound of various connected models. Used languages for their definition include DSLs for behavior and structure such as the human cognitive modeling language [MM13], goal modeling languages [MRV20, MRZ21] or UML/P based languages [MNRV19]. [MM15] describes a process of how languages for assistive systems can be created. MontiGem [AMN+20a] is used as the underlying generator technology.

We have designed a system included in a sensor floor able to monitor elderlies and analyze impact patterns for emergency events [LMK+11]. We have investigated the modeling of human contexts for the active assisted living and smart home domain [MS17] and user-centered privacy-driven systems in

the IoT domain in combination with process mining systems [MKM+19], differential privacy on event logs of handling and treatment of patients at a hospital [MKB+19], the mark-up of online manuals for devices [SM18a] and websites [SM20], and solutions for privacy-aware environments for cloud services [ELR+17] and in IoT manufacturing [MNRV19]. The user-centered view of the system design allows to track who does what, when, why, where, and how with personal data, makes information about it available via information services and provides support using assistive services.

Modeling Robotics Architectures and Tasks

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires the composition and the interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers the broad propagation of robotics applications.

The MontiArcAutomaton language [RRW12, RRW14a] extends the ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13c, RRRW15b, HKR21] that perfectly fit robotic architectural modeling.

The iserveU modeling framework describes domains, actors, goals, and tasks of service robotics applications [ABH+16, ABH+17] with declarative models. Goals and tasks are translated into models of the planning domain definition language (PDDL) and then solved [ABK+17]. Thus, domain experts focus on describing the domain and its properties only.

The LightRocks [THR+13, BRS+15] framework allows robotics experts and laymen to model robotic assembly tasks. In [AHRW17, AHRW17b], we define a modular architecture modeling method for translating architecture models into modules compatible with different robotics middleware platforms.

Many of the concepts in robotics were derived from automotive software [BBR07, BR12b].

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment, and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed, and tested. A consistent requirement management connecting requirements with features in all development phases for the automotive domain is described in [GRJA12].

The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. A methodical embedding of the resulting function nets and their quality assurance using automated testing is given in the SMaRDT method [DGH+19, KMS+18].

[HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses the means to extract a well-defined Software Product Line from a set of copy and paste variants.

Potential variants of components in product lines can be identified using similarity analysis of interfaces [KRR+16], or execute tests to identify similar behavior [RRS+16]. [RSW+15] describes an approach to using model checking techniques to identify behavioral differences of Simulink models. In [KKR19], we model dynamic reconfiguration of architectures applied to cooperating vehicles.

Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12b, BR12], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in the

development and the evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12b].

[MMR10] gives an overview of the state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions.

MontiSim simulates autonomous and cooperative driving behavior [GKR+17] for testing various forms of errors as well as spatial distance [FIK+18, KKRZ19]. As tooling infrastructure, the SSELab storage, versioning, and management services [HKR12] are essential for many projects.

Internet of Things, Industry 4.0 & the MontiThings Tool

The Internet of Things (IoT) requires the development of increasingly complex distributed systems. The MontiThings ecosystem [KRS+22] provides an end-to-end solution to modeling, deploying [KKR+22], and analyzing [KMR21] failure-tolerant [KRS+22] IoT systems and connecting them to synthesized digital twins [KMR+20]. We have investigated how model-driven methods can support the development of privacy-aware [ELR+17, HHK+14] cloud systems [PR13], distributed systems security [HHR+15], privacy-aware process mining [MKM+19], and distributed robotics applications [RRRW15b].

In the course of Industry 4.0, we have also turned our attention to mechanical or electrical applications [DRW+20]. We identified the digital representation, integration, and (re-)configuration of automation systems as primary Industry 4.0 concerns [WCB17]. Using a multi-level modeling framework, we support machine as a service approaches [BKL+18].

Smart Energy Management

In the past years, it became more and more evident that saving energy and reducing CO_2 emissions are important challenges. Thus, energy management in buildings as well as in neighborhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales.

During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for the technical specification of building services already.

We adapted the well-known concept of statemachines to be able to describe different states of a facility and validate it against the monitored values [FLP+11b]. We show how our data model, the constraint rules, and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing and Services

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality, and new application domains. It promises to enable new business models, facilitate web-based innovations, and increase the efficiency and cost-effectiveness of web development [KRR14].

Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15a], Big Data, Apps, and Service Ecosystems bring attention to aspects like responsiveness, privacy, and open platforms. Regardless of the application domain, developers of such systems need robust methods and efficient, easy-to-use languages and tools [KRS12].

We tackle these challenges by perusing a model-based, generative approach [PR13]. At the core of this approach are different modeling languages that describe different aspects of a cloud-based system

in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale.

We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our development platforms. New services, e.g., for collecting data from temperature sensors, cars, etc. are now easily developed and deployed, e.g., in production or Internet-of-Things environments.

Security aspects and architectures of cloud services for the digital me in a privacy-aware environment are addressed in [ELR+17].

Model-Driven Engineering of Information Systems & the MontiGem Tool

Information Systems provide information to different user groups as the main system goal. Using our experiences in the model-based generation of code with MontiCore [KRV10, HKR21], we developed several generators for such data-centric information systems.

MontiGem [AMN+20a] is a specific generator framework for data-centric business applications that uses standard models from UML/P optionally extended by GUI description models as sources [GMN+20]. While the standard semantics of these modeling languages remains untouched, the generator produces a lot of additional functionality around these models. The generator is designed flexible, modular, and incremental, handwritten and generated code pieces are well integrated [GHK+15a, NRR15a], tagging of existing models is possible [GLRR15], e.g., for the definition of roles and rights or for testing [DGH+18].

We are using MontiGem for financial management [GHK+20, ANV+18], for creating digital twin cockpits [DMR+20], and various industrial projects. MontiGem makes it easier to create low-code development platforms for digital twins [DHM+22]. When using additional DSLs, we can develop assistive systems providing user support based on goal models [MRV20], privacy-preserving information systems using privacy models and purpose trees [MNRV19], and process-aware digital twin cockpits using BPMN models [BMR+22].

We have also developed an architecture of cloud services for the digital me in a privacy-aware environment [ELR+17] and a method for retrofitting generative aspects into existing applications [DGM+21].

References

[ABH+16] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Model-Driven Separation of Concerns for Service Robotics. In *International Workshop on Domain-Specific Modeling (DSM'16)*, pages 22–27. ACM, October 2016.

[ABH+17] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. *Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse*. Aachener Informatik-Berichte, Software Engineering, Band 28. Shaker Verlag, December 2017.

[ABK+17] Kai Adam, Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Executing Robot Task Models in Dynamic Environments. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[AHRW17] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017.

[AHRW17b] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSER)*, 8(1):3–16, 2017.

[AKK+21] Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 55–68. ACM, October 2021.

[AMN+20a] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19)*, LNI P-304, pages 59–66. Gesellschaft für Informatik e.V., May 2020.

[ANV+18] Kai Adam, Lukas Netz, Simon Varga, Judith Michael, Bernhard Rumpe, Patricia Heuser, and Peter Letmathe. Model-Based Generation of Enterprise Information Systems. In Michael Fellmann and Kurt Sandkuhl, editors, *Enterprise Modeling and Information Systems Architectures (EMISA'18)*, CEUR Workshop Proceedings 2097, pages 75–79. CEUR-WS.org, May 2018.

[BBC+18] Inga Blundell, Romain Brette, Thomas A. Cleland, Thomas G. Close, Daniel Coca, Andrew P. Davison, Sandra Diaz-Pier, Carlos Fernandez Musoles, Padraig Gleeson, Dan F. M. Goodman, Michael Hines, Michael W. Hopkins, Pramod Kumbhar, David R. Lester, Bóris Marin, Abigail Morrison, Eric Müller, Thomas Nowotny, Alexander

Peyser, Dimitri Plotnikov, Paul Richmond, Andrew Rowley, Bernhard Rumpe, Marcel Stimberg, Alan B. Stokes, Adam Tomkins, Guido Trensch, Marmaduke Woodman, and Jochen Martin Eppler. Code Generation in Computational Neuroscience: A Review of Tools and Techniques. *Journal Frontiers in Neuroinformatics*, 12, 2018.

[BBD+21b] Fabian Becker, Pascal Bibow, Manuela Dalibor, Aymen Gannouni, Viviane Hahn, Christian Hopmann, Matthias Jarke, Istvan Koren, Moritz Kröger, Johannes Lipp, Judith Maibaum, Judith Michael, Bernhard Rumpe, Patrick Sapel, Niklas Schäfer, Georg J. Schmitz, Günther Schuh, and Andreas Wortmann. A Conceptual Model for Digital Shadows in Industry and its Application. In Aditya Ghose, Jennifer Horkoff, Vitor E. Silva Souza, Jeffrey Parsons, and Joerg Evermann, editors, *Conceptual Modeling, ER 2021*, pages 271–281. Springer, October 2021.

[BBD+21a] Tim Bolender, Gereon Bürvenich, Manuela Dalibor, Bernhard Rumpe, and Andreas Wortmann. Self-Adaptive Manufacturing with Digital Twins. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 156–166. IEEE Computer Society, May 2021.

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BBR20] Manfred Broy, Wolfgang Böhm, and Bernhard Rumpe. Advanced Systems Engineering - Die Systeme der Zukunft. White paper, fortiss. Forschungsinstitut für softwareintensive Systeme, Munich, July 2020.

[BCGR09] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BDH+20] Pascal Bibow, Manuela Dalibor, Christian Hopmann, Ben Mainz, Bernhard Rumpe, David Schmalzing, Mauritius Schmitz, and Andreas Wortmann. Model-Driven Development of a Digital Twin for Injection Molding. In Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'20)*, Lecture Notes in Computer Science 12127, pages 85–100. Springer International Publishing, June 2020.

[BDJ+22] Philipp Brauner, Manuela Dalibor, Matthias Jarke, Ike Kunze, István Koren, Gerhard Lakemeyer, Martin Liebenberg, Judith Michael, Jan Pennekamp, Christoph Quix, Bernhard Rumpe, Wil van der Aalst, Klaus Wehrle, Andreas Wortmann, and Martina Ziefle. A Computer Science Perspective on Digital Transformation in Production. *Journal ACM Transactions on Internet of Things*, 3:1–32, February 2022.

[BDL+18] Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. Deriving Fluent Internal Domain-specific Languages from Grammars. In *International Conference on Software Language Engineering (SLE'18)*, pages 187–199. ACM, 2018.

[BDR+21] Christian Brecher, Manuela Dalibor, Bernhard Rumpe, Katrin Schilling, and Andreas Wortmann. An Ecosystem for Digital Shadows in Manufacturing. In *54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0*. Elsevier, September 2021.

[BEH+20] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Journal of Object Technology (JOT)*, 19(3):3:1–16, October 2020.

[BEK+18b] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018.

[BEK+19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Composition of Independent Language Features. *Journal of Systems and Software (JSS)*, 152:50–69, June 2019.

[Ber10] Christian Berger. *Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles*. Aachener Informatik-Berichte, Software Engineering, Band 6. Shaker Verlag, 2010.

[BGH+97] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Towards a Precise Semantics for Object-Oriented Modeling Techniques. In Jan Bosch and Stuart Mitchell, editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader*, LNCS 1357. Springer Verlag, 1997.

[BGH+98a] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. *Journal Computer Standards & Interfaces*, 19(7):335–345, November 1998.

[BGH+98b] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. In *Part of the Grand Challenges in Modeling (GRAND'17) Workshop*, July 2017.

[BGRW18] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations*, LNCS 10748, pages 146–153. Springer, January 2018.

[BHH+17] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 53–70. Springer, July 2017.

[BHK+17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017.

[BHK+21] Tobias Brockhoff, Malte Heithoff, István Koren, Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, Merih Seran Uysal, Wil M. P. van der Aalst, and Andreas Wortmann. Process Prediction with Digital Twins. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 182–187. ACM/IEEE, October 2021.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BHR+18] Arvid Butting, Steffen Hillemacher, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. Shepherding Model Evolution in Model-Driven Development. In *Joint Proceedings of the Workshops at Modellierung 2018 (MOD-WS 2018)*, CEUR Workshop Proceedings 2060, pages 67–77. CEUR-WS.org, February 2018.

[BHR+21] Arvid Butting, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 217–234. Springer, July 2021.

[BJRW18] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)*, pages 174–186. ACM, 2018.

[BKL+18] Christian Brecher, Evgeny Kusmenko, Achim Lindt, Bernhard Rumpe, Simon Storms, Stephan Wein, Michael von Wenckstern, and Andreas Wortmann. Multi-Level Modeling Framework for Machine as a Service Applications Based on Product Process Resource Models. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control (ISCSIC'18)*. ACM, September 2018.

[BKR+20] Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla. *Towards an Isabelle Theory for distributed, interactive systems - the untimed case*. Aachener Informatik Berichte, Software Engineering, Band 45. Shaker Verlag, March 2020.

[BKRW17a] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Architectural Programming with MontiArcAutomaton. In *12th International Conference on Software Engineering Advances (ICSEA 2017)*, pages 213–218. IARIA XPS Press, May 2017.

[BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, April 2017.

[BKRW19] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. *Journal of Systems and Software (JSS)*, 149:437–461, March 2019.

[BMR+22] Dorina Bano, Judith Michael, Bernhard Rumpe, Simon Varga, and Matthias Weske. Process-Aware Digital Twin Cockpit Synthesis from Event Logs. *Journal of Computer Languages (COLA)*, 70, June 2022.

[BPR+20] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 35–46. ACM, October 2020.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12b] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[BRS+15] Arvid Butting, Bernhard Rumpe, Christoph Schulze, Ulrike Thomas, and Andreas Wortmann. Modeling Reusable, Platform-Independent Robot Assembly Processes. In *International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015)*, 2015.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15a] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14] Betty H.C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha M. Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In Nelly Bencomo, Robert France, Betty H.C. Cheng, and Uwe Aßmann, editors, *Models@run.time*, LNCS 8378, pages 101–136. Springer International Publishing, Switzerland, 2014.

[CFJ+16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, November 2016.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[DEKR19] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. Semantic Differencing of Statecharts for Object-oriented Systems. In Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19)*, pages 274–282. SciTePress, February 2019.

[DGH+18] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. Model-Based Testing of Software-Based System Functions. In *Conference on Software Engineering and Advanced Applications (SEAA'18)*, pages 146–153, August 2018.

[DGH+19] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. SMArDT modeling for automotive software testing. *Journal on Software: Practice and Experience*, 49(2):301–328, February 2019.

[DGM+21] Imke Drave, Akradii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. A Methodology for Retrofitting Generative Aspects in Existing Applications. *Journal of Object Technology (JOT)*, 20:1–24, November 2021.

[DHH+20] Imke Drave, Timo Henrich, Katrin Hölldobler, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 173–188. Gesellschaft für Informatik e.V., February 2020.

[DHM+22] Manuela Dalibor, Malte Heithoff, Judith Michael, Lukas Netz, Jérôme Pfeiffer, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Generating Customized Low-Code Development Platforms for Digital Twins. *Journal of Computer Languages (COLA)*, 70, June 2022.

[DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 245–255. ACM, September 2019.

[DMR+20] Manuela Dalibor, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr, editors, *Conceptual Modeling*, pages 377–387. Springer International Publishing, October 2020.

[DRW+20] Imke Drave, Bernhard Rumpe, Andreas Wortmann, Joerg Berroth, Gregor Hoepfner, Georg Jacobs, Kathrin Spuetz, Thilo Zerwas, Christian Guist, and Jens Kohl. Modeling Mechanical Functional Architectures in SysML. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 79–89. ACM, October 2020.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[EFLR99a] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language. «UML»'98: Beyond the Notation*, LNCS 1618, pages 336–348. Springer, Germany, 1999.

[EJK+19] Rolf Ebert, Jahir Jolianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rumpe, and Karin Samira Salman. Applying Product Line Testing for the Electric Drive System. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 14–24. ACM, September 2019.

[ELR+17] Robert Eikermann, Markus Look, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Architecting Cloud Services for the Digital me in a Privacy-Aware Environment. In Ivan Mistrik, Rami Bahsoon, Nour Ali, Maritta Heisel, and Bruce Maxim, editors, *Software Architecture for Big Data and the Cloud*, chapter 12, pages 207–226. Elsevier Science & Technology, June 2017.

[FEL+98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Journal Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FIK+18] Christian Frohn, Petyo Ilov, Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Alexander Ryndin. Distributed Simulation of Cooperatively Interacting Vehicles. In *International Conference on Intelligent Transportation Systems (ITSC'18)*, pages 596–601. IEEE, 2018.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. Der Energie-Navigator - Performance-Controlling für Gebäude und Anlagen. *Technik am Bau (TAB) - Fachzeitschrift für Technische Gebäudeausrüstung*, Seiten 36-41, März 2011.

[FLP+11b] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FND+98] Max Fuchs, Dieter Nazareth, Dirk Daniel, and Bernhard Rumpe. BMW-ROOM An Object-Oriented Method for ASCET. In *SAE'98, Cobo Center (Detroit, Michigan, USA)*, Society of Automotive Engineers, 1998.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference (IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Informatik Bericht 2008-02. TU Braunschweig, 2008.

[GHK+15] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015.

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 112–132. Springer, 2015.

[GHK+20] Arkadii Gerasimov, Patricia Heuser, Holger Ketteniß, Peter Letmathe, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In Judith Michael and Dominik Bork, editors, *Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers*, pages 22–30. CEUR Workshop Proceedings, February 2020.

[GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects*. Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0: Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKR+17] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[GKR+96] Radu Grosu, Cornel Klein, Bernhard Rumpe, and Manfred Broy. State Transition Diagrams. Technical report, TU Munich, 1996.

[GLPR15] Timo Greifenberg, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Energieeffiziente Städte - Herausforderungen und Lösungen aus Sicht des Software Engineerings. In Linnhoff-Popien, Claudia and Zaddach, Michael and Grahl, Andreas, Editor, *Marktplätze im Umbruch: Digitale Strategien für Services im Mobilen Internet*, Xpert.press, Kapitel 56, Seiten 511-520. Springer Berlin Heidelberg, April 2015.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GMN+20] Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In Bonnie Anderson, Jason Thatcher, and Rayman Meservy, editors, *25th Americas Conference on Information Systems (AMCIS 2020)*, AIS Electronic Library (AISeL), pages 1–10. Association for Information Systems (AIS), August 2020.

[GMR+16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016 Conference*, LNI 254, pages 141–156. Bonner Köllen Verlag, March 2016.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[Gre19] Timo Greifenberg. *Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte*. Aachener Informatik-Berichte, Software Engineering, Band 42. Shaker Verlag, August 2019.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR09] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System Model-based Definition of Modeling Language Semantics. In *Proc. of FMOODS/FORTE 2009, LNCS 5522*, Lisbon, Portugal, 2009.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[Her19] Lars Hermerschmidt. *Agile Modellgetriebene Entwicklung von Software Security & Privacy*. Aachener Informatik-Berichte, Software Engineering, Band 41. Shaker Verlag, June 2019.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15a] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Journal Future Generation Computer Systems*, 56:701–718, 2015.

[HHR+15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'15)*, CEUR Workshop Proceedings 1463, pages 18–23, 2015.

[HJK+21] Steffen Hillemacher, Nicolas Jäckel, Christopher Kugler, Philipp Orth, David Schmalzing, and Louis Wachtmeister. Artifact-Based Analysis for the Development of Collaborative Embedded Systems. In *Model-Based Engineering of Collaborative Embedded Systems*, pages 315–331. Springer, January 2021.

[HJRW20] Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Komposition Domänenspezifischer Sprachen unter Nutzung der MontiCore Language Workbench, am Beispiel SysML 2. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 189–190. Gesellschaft für Informatik e.V., February 2020.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HKR+16] Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In *Software Architecture - 10th European Conference (ECSA'16)*, LNCS 9839, pages 175–182. Springer, December 2016.

[HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021.* Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.

[HLN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous

Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 45–66. Springer, 2015.

[HLN+15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 19–31. SciTePress, 2015.

[HMR+19] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Innovations in Model-based Software and Systems Engineering. *Journal of Object Technology (JOT)*, 18(1):1–60, July 2019.

[HNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, pages 67–82. Springer, July 2016.

[Hoe18] Katrin Hölldobler. *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 36. Shaker Verlag, December 2018.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal*, 37(10):64–72, October 2004.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR10] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VI*, Informatik-Bericht 2010-01, pages 13 – 22. fortiss GmbH, Germany, 2010.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181-192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[HRW18] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Journal Computer Languages, Systems & Structures*, 54:386–405, 2018.

[JPR+22] Nico Jansen, Jerome Pfeiffer, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. The Language of SysML v2 under the Magnifying Glass. *Journal of Object Technology (JOT)*, 21:1–15, July 2022.

[JWCR18] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. Does Distance Still Matter? Revisiting Collaborative Distributed Software Design. *IEEE Software Journal*, 35(6):40–47, 2018.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KKR19] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *Journal of Object Technology (JOT)*, 18(2):1–20, July 2019.

[KKR+22] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. *Journal ACM Transactions on Internet of Things*, 3:1–30, November 2022.

[KKRZ19] Jörg Christian Kirchhof, Evgeny Kusmenko, Bernhard Rumpe, and Hengwen Zhang. Simulation as a Service for Cooperative Vehicles. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MASE*, pages 28–37. IEEE, September 2019.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KMA+16] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. VCU: The Three Dimensions of Reuse. In *Conference on Software Reuse (ICSR'16)*, LNCS 9679, pages 122–137. Springer, June 2016.

[KMP+21] Hendrik Kausch, Judith Michael, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications. In *Aerospace Europe Conference 2021 (AEC 2021)*. Council of European Aerospace Societies (CEAS), November 2021.

[KMR+20] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 90–101. ACM, October 2020.

[KMR21] Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 197–209. ACM, October 2021.

[KMS+18] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. Improving Model-based Testing in Automotive Software Engineering. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE'18)*, pages 172–180. ACM, June 2018.

[KNP+19] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*, pages 283–293. IEEE, September 2019.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[KPRS19] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. On the Engineering of AI-Powered Systems. In Lisa O'Conner, editor, *ASE19. Software Engineering Intelligence Workshop (SEI19)*, pages 126–133. IEEE, November 2019.

[KR18a] Oliver Kautz and Bernhard Rumpe. On Computing Instructions to Repair Failed Model Refinements. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 289–299. ACM, October 2018.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRR+16] Philipp Kehrbusch, Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, and Christoph Schulze. Interface-based Similarity Analysis of Software Components for the Automotive Industry. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 99–108. ACM, September 2016.

[KRRS19] Stefan Kriebel, Deni Raco, Bernhard Rumpe, and Sebastian Stüber. Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In Stephan Krusche, Kurt Schneider, Marco Kuhrmann, Robert Heinrich, Reiner Jung, Marco Konersmann, Eric Schmieders, Steffen Helke, Ina Schaefer, Andreas Vogelsang, Björn Annighöfer, Andreas Schweiger, Marina Reich, and André van Hoorn, editors, *Proceedings of the Workshops of the Software Engineering Conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19)*, CEUR Workshop Proceedings 2308, pages 87–94. CEUR Workshop Proceedings, February 2019.

[KRRW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113-116. VDI Verlag, 2012.

[KRS+18a] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447 – 457. ACM, October 2018.

[KRS+22] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software (JSS)*, 183:1–21, January 2022.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering Journal*, 27:119–151, April 2020.

[Kus21] Evgeny Kusmenko. *Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems*. Aachener Informatik-Berichte, Software Engineering, Band 49. Shaker Verlag, November 2021.

[LMK+11] Philipp Leusmann, Christian Möllering, Lars Klack, Kai Kasugai, Bernhard Rumpe, and Martina Ziefle. Your Floor Knows Where You Are: Sensing and Acquisition of Movement Data. In Arkady Zaslavsky, Panos K. Chrysanthis, Dik Lun Lee, Dipanjan Chakraborty, Vana Kalogeraki, Mohamed F. Mokbel, and Chi-Yin Chow, editors, *12th IEEE International Conference on Mobile Data Management (Volume 2)*, pages 61–66. IEEE, June 2011.

[Loo17] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE*. Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, March 2017.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MKB+19] Felix Mannhardt, Agnes Koschmider, Nathalie Baracaldo, Matthias Weidlich, and Judith Michael. Privacy-Preserving Process Mining: Differential Privacy for Event Logs. *Business & Information Systems Engineering*, 61(5):1–20, October 2019.

[MKM+19] Judith Michael, Agnes Koschmider, Felix Mannhardt, Nathalie Baracaldo, and Bernhard Rumpe. User-Centered and Privacy-Driven Process Mining System Design for IoT. In Cinzia Cappiello and Marcela Ruiz, editors, *Proceedings of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems*, pages 194–206. Springer, June 2019.

[MM13] Judith Michael and Heinrich C. Mayr. Conceptual modeling for ambient assistance. In *Conceptual Modeling - ER 2013*, LNCS 8217, pages 403–413. Springer, 2013.

[MM15] Judith Michael and Heinrich C. Mayr. Creating a domain specific modelling method for ambient assistance. In *International Conference on Advances in ICT for Emerging Regions (ICTer2015)*, pages 119–124. IEEE, 2015.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer Journal*, 43(5):42–48, May 2010.

[MMR+17] Heinrich C. Mayr, Judith Michael, Suneth Ranasinghe, Vladimir A. Shekhovtsov, and Claudia Steinberger. Model centered architecture. In *Conceptual Modeling Perspectives*, pages 85–104. Springer International Publishing, 2017.

[MNRV19] Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann, editors, *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 595–614. CEUR Workshop Proceedings, September 2019.

[MPRW22] Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. Integration Challenges for Digital Twin Systems-of-Systems. In *10th IEEE/ACM International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems*, pages 9–12. ACM, May 2022.

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, pages 230–254. Springer Berlin Heidelberg, 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11f] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR11g] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Summarizing Semantic Model Differences. In Bernhard Schätz, Dirk Deridder, Alfonso Pierantonio, Jonathan Sprinkle, and Dalila Tamzalit, editors, *ME 2011 - Models and Evolution*, October 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[MRR14a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views (extended abstract). In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Software Engineering 2014*, LNI 227, pages 63–64. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, 2014.

[MRR14b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *International Conference on Software Engineering (ICSE'14)*, pages 95–105. ACM, 2014.

[MRRW16] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Consistent Extra-Functional Properties Tagging for Component and Connector Models. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'16)*, CEUR Workshop Proceedings 1723, pages 19–24, October 2016.

[MRV20] Judith Michael, Bernhard Rumpe, and Simon Varga. Human behavior, goals and model-driven software engineering for assistive systems. In Agnes Koschmider, Judith Michael, and Bernhard Thalheim, editors, *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)*, pages 11–18. CEUR Workshop Proceedings, June 2020.

[MRZ21] Judith Michael, Bernhard Rumpe, and Lukas Tim Zimmermann. Goal Modeling and MDSE for Behavior Assistance. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 370–379. ACM/IEEE, October 2021.

[MS17] Judith Michael and Claudia Steinberger. Context modeling for active assistance. In Cristina Cabanillas, Sergio España, and Siamak Farshidi, editors, *Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017)*, pages 221–234, 2017.

[Naz17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.

[NRR15a] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. Mixed Generative and Handcoded Development of Adaptable Data-centric Business Applications. In *Domain-Specific Modeling Workshop (DSM'15)*, pages 43–44. ACM, 2015.

[NRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, LNI 254, pages 133–140. Bonner Köllen Verlag, March 2016.

[PR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PBI+16] Dimitri Plotnikov, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. NESTML: a modeling language for spiking neurons. In *Modellierung 2016 Conference*, LNI 254, pages 93–108. Bonner Köllen Verlag, March 2016.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[Pin14] Claas Pinkernell. *Energie Navigator: Software-gestützte Optimierung der Energieeffizienz von Gebäuden und technischen Anlagen*. Aachener Informatik-Berichte, Software Engineering, Band 17. Shaker Verlag, 2014.

[Plo18] Dimitri Plotnikov. *NESTML - die domänenspezifische Sprache für den NEST-Simulator neuronaler Netzwerke im Human Brain Project*. Aachener Informatik-Berichte, Software Engineering, Band 33. Shaker Verlag, February 2018.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR97] Jan Philipps and Bernhard Rumpe. Refinement of Information Flow Architectures. In M. Hinchey, editor, *ICFEM'97 Proceedings*, Hiroshima, Japan, 1997. IEEE CS Press.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen*. Aachener Informatik-Berichte, Software Engineering, Band 22. Shaker Verlag, May 2016.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, Aachen, Germany, December 2014.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[Rot17] Alexander Roth. *Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex*. Aachener Informatik-Berichte, Software Engineering, Band 31. Shaker Verlag, December 2017.

[RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.

[RRRW15b] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRS+16] Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, Christoph Schulze, Kevin Thissen, and Michael von Wenckstern. Test-driven Semantical Similarity Analysis for Software Product Line Extraction. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 174–183. ACM, September 2016.

[RRSW17] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17)*, pages 127–136. IEEE, May 2017.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In Seyff, N. and Koziolek, A., editor, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*, pages 133–146. Monsenstein und Vannerdat, Münster, 2012.

[RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13c] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RRW15] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development. In *MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 41–47. ACM, 2015.

[RSR+99] Bernhard Rumpe, M. Schoenmakers, Ansgar Radermacher, and Andy Schürr. UML + ROOM as a Standard ADL? In Frances M. Titsworth, editor, *Engineering of Complex Computer Systems, ICECCS'99 Proceedings*, IEEE Computer Society, 1999.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04c] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SHH+20] Günther Schuh, Constantin Häfner, Christian Hopmann, Bernhard Rumpe, Matthias Brockmann, Andreas Wortmann, Judith Maibaum, Manuela Dalibor, Pascal Bibow, Patrick Sapel, and Moritz Kröger. Effizientere Produktion mit Digitalen Schatten. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 115(special):105–107, April 2020.

[SM18a] Claudia Steinberger and Judith Michael. Towards Cognitive Assisted Living 3.0. In *International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops 2018)*, pages 687–692. IEEE, march 2018.

[SM20] Claudia Steinberger and Judith Michael. Using Semantic Markup to Boost Context Awareness for Assistive Systems. In *Smart Assisted Living: Toward An Open Smart-Home Infrastructure*, Computer Communications and Networks, pages 227–246. Springer International Publishing, 2020.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[TAB+21] Carolyn Talcott, Sofia Ananieva, Kyungmin Bae, Benoit Combemale, Robert Heinrich, Mark Hills, Narges Khakpour, Ralf Reussner, Bernhard Rumpe, Patrizia Scandurra, and Hans Vangheluwe. Composition of Languages, Models, and Analyses. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 45–70. Springer, July 2021.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Voe11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[WCB17] Andreas Wortmann, Benoit Combemale, and Olivier Barais. A Systematic Mapping Study on Modeling for Industry 4.0. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pages 281–291. IEEE, September 2017.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling*. Aachener Informatik-Berichte, Software Engineering, Band 25. Shaker Verlag, November 2016.

[Wor21] Andreas Wortmann. *Model-Driven Architecture and Behavior of Cyber-Physical Systems*. Aachener Informatik-Berichte, Software Engineering, Band 50. Shaker Verlag, October 2021.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.