

Knowledge-Driven Architecture Composition

Case-Based Formalization of Integration Knowledge to Enable Automated Component
Coupling

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim



vorgelegt von
Fabian Jonas Burzlaff
aus Lahr/Schwarzwald

Mannheim, 2021

Dekan: Dr. Bernd Lübcke, Universität Mannheim
Referent: Prof. Dr. Heiner Stuckenschmidt, Universität Mannheim
Korreferent: Prof. Dr. Colin Atkinson, Universität Mannheim

Tag der mündlichen Prüfung: 29.03.2021

Abstract

Service interoperability for embedded devices is a mandatory feature for dynamically changing Internet-of-Things and Industry 4.0 software platforms. Service interoperability is achieved on a technical, syntactic, and semantic level. If service interoperability is achieved on all layers, plug and play functionality known from USB storage sticks or printer drivers becomes feasible. As a result, micro batch size production, individualized automation solution, or job order production become affordable. However, interoperability at the semantic layer is still a problem for the maturing class of IoT systems.

Current solutions to achieve semantic integration of IoT devices' heterogeneous services include standards, machine-understandable service descriptions, and the implementation of software adapters. Standardization bodies such as the VDMA tackle the problem by providing a reference software architecture and an information meta model for building up domain standards. For instance, the universal machine technology interface (UMATI) facilitates the data exchange between machines, components, installations, and their integration into a customer- and user-specific IT ecosystem for mechanical engineering and plant construction worldwide. Automated component integration approaches fill the gap of software interfaces that are not relying on a global standard. These approaches translate required into provided software interfaces based on the needed architectural styles (e.g., client-server, layered, publish-subscribe, or cloud-based) using additional component descriptions. Interoperability at the semantic layer is achieved by relying on a shared domain vocabulary (e.g., an ontology) and service description (e.g., SAWSDL) used by all devices involved. If these service descriptions are available and machine-understandable knowledge of how to integrate software components on the functional and behavioral level is available, plug and play scenarios are feasible.

Both standards and formal service descriptions cannot be applied effectively to IoT systems as they rely on the assumption that the semantic domain is completely known when they are noted down. This assumption is hard to believe as an increasing number of decentralized developed and connected IoT devices will exist (i.e., 30.73 billion in 2020 and 75.44 billion in 2025). If standards are applied in IoT systems, they must be updated continuously, so they contain the most recent domain knowledge agreed upon centrally and ahead of application. Although formal descriptions of concrete integration contexts can happen in a decentralized manner, they still rely on the assumption that the knowledge once noted down is complete. Hence, if an interoperable service from a new device is available that has not been considered in the initial integration context, the formal descriptions must be updated continuously. Both the formalization effort and keeping standards up to date result in too much additional engineering effort. Consequently, practitioners rely on implementing software adapters manually. However, this dull solution hardly scales with the increasing number of IoT devices.

In this work, we introduce a novel engineering method that explicitly allows for an incomplete semantic domain description without losing the ability for automated IoT system integration. Dropping the completeness claim requires the management of incomplete integration knowledge. By sharing integration knowledge centrally, we assist the system integrator in automating software adapter generation. In addition to existing approaches, we enable semantic integration for services by making integration knowledge reusable. We empirically show with students that integration effort can be lowered in a home automation context.

Zusammenfassung

Service-Interoperabilität ist eine notwendige Eigenschaft für eingebettete Geräte in sich dynamisch verändernden Internet der Dinge sowie Industrie 4.0 Plattformen.

Service-Interoperabilität wird auf der technischen, syntaktischen und semantischen Ebene erreicht. Falls Service-Interoperabilität auf allen Ebenen erreicht ist, dann sind so genannte "Plug-and-Play"-Szenarien wie beispielsweise bekannt von USB-Sticks oder Drucker realisierbar. Dadurch werden beispielsweise Kleinserienproduktionen, individuelle Automatisierungslösungen oder Auftragsfertigungen finanzierbar. Für IoT Systeme ist Interoperabilität auf der semantischen Ebene aber immer noch ein Problem.

Aktuelle Lösungen zur Erreichung von semantischer Interoperabilität von IoT Geräten umfassen Standards, maschineninterpretierbare Servicebeschreibungen und die Implementierung von Software Adapter. Standardisierungsbestrebungen wie der VDMA bieten eine Referenzarchitektur sowie Informationsmodelle an, um Domänenstandards zu bauen. Beispielsweise wird mit der universellen Maschinenschnittstelle (UMATI) der Datenaustausch zwischen Maschinen, Komponenten, Installationen und deren Integration in kunden- und anwenderspezifische IT-Ökosysteme im Bereich des Maschinen- und Anlagenbaus weltweit ermöglicht. Es existieren auch automatisierte Integrationsansätze, welche nicht auf einem globalen Standard beruhen. Diese Ansätze übersetzen einen benötigte und eine angebotene Schnittstelle basierend auf dem zugrundeliegenden Architekturstil sowie zusätzlichen Beschreibungen. Dabei wird Interoperabilität auf der semantischen Ebene durch ein gemeinsames Vokabular (bspw. Ontologien) und Schnittstellenbeschreibungen (bspw. SAWSDL) welche von allen Geräten benutzt werden, erreicht. Falls die Schnittstellenbeschreibungen offen sind und maschineninterpretierbares Integrationswissen auf der funktionalen Ebene und bezüglich des Verhaltens des Services vorhanden ist, dann sind auch hier "Plug-and-Play"-Szenarien möglich.

Sowohl Standards als auch formale Servicebeschreibungen können jedoch nicht effektiv in IoT-Systemen angewendet werden, da sie eine vollständige Beschreibung der Anwendungsdomäne zum Zeitpunkt der Erstellung voraussetzen. Diese Annahme ist nicht glaubhaft, da die Anzahl der dezentral entwickelten und miteinander verbundenen IoT-Geräte immer stärker wächst (bspw. 30.73 Milliarden im Jahr 2020 und 75.55 Milliarden im Jahr 2025). Falls Standards in IoT-Systemen angewendet werden, dann müssen sie kontinuierlich aktualisiert werden um den aktuellen Zustand der Domäne wie von Standardisierungsorganisationen festgelegt zu beinhalten. Formale Schnittstellenbeschreibungen können auch basierend auf konkreten Integrationsfällen dezentral erstellt werden. Jedoch nehmen auch solche inkrementellen Ansätze an, dass sich die Domäne und damit das Integrationswissen nicht mehr verändert, sobald es gespeichert wurde. Falls ein neues Gerät mit einer interoperablen Schnittstelle verfügbar wird und es nicht während der initialen Beschreibung bedacht wurde, dann müssen auch hier Schnittstellenbeschreibungen und Integrationswissen erneut angepasst werden. Sowohl der Formalisierungsaufwand für Integrationswissen als auch die Pflege von Standards stellen einen hohen Implementierungsaufwand dar. Deshalb implementieren Praktiker manuelle Software Adapter. Dieser schwerfällige Lösungsansatz skaliert jedoch kaum mit der immer größer werdenden Anzahl an IoT-Geräten.

In dieser Arbeit stellen wir eine neue Entwicklungsmethode vor, welche unvollständige, semantische Domänenbeschreibungen explizit von Anfang an erlaubt. Dabei geht die Fähigkeit

zur automatisierten Komponentenintegration nicht verloren. Durch das Weglassen des Vollständigkeitsanspruchs muss unvollständiges Integrationswissen handhabbar gemacht werden. Durch das Teilen von Integrationswissen unterstützt die Methode den Systemintegrator in der automatisierten Erstellung des Softwareadapters. Im Gegensatz zu verwandten Forschungsansätzen ermöglichen wir die semantische Integration von Schnittstellen, weil wir Integrationswissen wiederverwendbar machen. Wir zeigen die Leistungsfähigkeit unserer Lösung im Kontext der Hausautomatisierung anhand von geringeren Integrationsaufwänden.

Danksagung

An dieser Stelle möchte ich allen beteiligten Personen meinen großen Dank aussprechen, die mich bei der Anfertigung und Bearbeitung dieser Dissertation unterstützt haben.

Zuerst möchte ich mich bei meinem Doktorvater Prof. Dr. Heiner Stuckenschmidt bedanken, der mich auf dem Weg zu dieser Dissertation stets mit viel Verständnis, Ehrlichkeit und Freiheit unterstützt hat. Für konstruktive Anregungen und hilfreichen Kommentare danke ich ebenso Prof. Dr. Colin Atkinson, der meine Dissertation als Zweitgutachter betreut hat.

Für die inhaltliche Ausrichtung und die thematische Eingrenzung sei Dr. Christian Bartelt besonders gedankt. Durch sein großes Engagement, fachliche Hinweise und ausführliche Forschungsdiskussionen hat er wesentlich zum erfolgreichen Abschluss der Arbeit beigetragen.

Für die vielfältige Unterstützung bin ich besonders mit der Forschungsgruppe "AI Systems Engineering" am Institut für Enterprise Systeme (InES) verbunden. In zahlreichen Vorträgen, Projekten und Konferenzen war es mir immer eine Freude mit allen Mitarbeitern in unterschiedlichen Konstellationen zusammenzuarbeiten. Im Speziellen möchte ich mich hier bei Christian Schreckenberger, Nils Wilken und Michael Pernpeintner bedanken.

Weiterhin danke ich allen Studierenden die mich im Rahmen von Abschlussarbeiten, Seminararbeiten und studentischen Teamprojekten unterstützt haben und ohne die diese Arbeit nicht möglich gewesen wäre. Insbesondere bedanke ich mich bei Steffen Jacobs, Lukas Adler und besonders bei Maurice Ackel für die Umsetzung diverser Prototypen.

Mein herzlicher Dank gilt schließlich meinen Eltern, Anette Zirlewagen-Burzlaff und Bernd Burzlaff, die meine Zeit als wissenschaftlicher Mitarbeiter mit großer Begeisterung verfolgt haben. Bei dieser Gelegenheit möchte ich mich auch bei meinen beiden Brüdern (Konstantin Burzlaff und Björn Burzlaff), meinen Freunden und meinen Kollegen bedanken. Ihre Motivation, ihr Lachen und ihre Ablenkung zum richtigen Zeitpunkt haben mich nachhaltig unterstützt.

Mein größter Dank gilt meiner Partnerin Claudine Schaz. Ohne ihr Einfühlungsvermögen, ihre positive Einstellung und ihre Geduld wäre vieles nicht möglich gewesen.

Contents

I. Introduction	1
1. Motivation	2
1.1. Context	3
1.2. Problem Statement	4
1.3. Research Question	5
1.4. Solution Overview	5
1.4.1. Running Example	7
1.5. Influences	9
1.6. Application Scenarios	10
1.6.1. Consumer IoT	10
1.6.2. Industrial IoT	10
1.6.3. Mobile Apps and Web Services	11
1.7. Structure of This Work	12
II. Background	13
2. Definitions and Context	14
2.1. Barriers	14
2.2. Terminology	15
2.2.1. Syntax and Semantics in Software Architectures	15
2.2.2. Engineering Approaches	17
2.2.3. Knowledge Management	18
2.3. Internet of Things Software Architectures	19
3. Methods to Achieve Semantic Interoperability in IoT	23
3.1. Software Adapter Implementation	23
3.2. Top-Down Engineering Methods	25
3.3. Bottom-Up Engineering Methods	28
4. Requirements for Semantic Service Interoperability in IoT	33
III. Knowledge-driven Architecture Composition	34
5. Formalization	35
5.1. Basics	35

Contents

5.2. Mapping Types	37
5.3. Reasoning Principles for Mappings	42
6. Integration Knowledge Management	45
6.1. Integration Knowledge Management Process	46
6.2. Algorithms	49
6.2.1. Composition of Operation Mappings	49
6.2.2. Composition of Identifier Mappings	51
6.2.3. Inverse of Mappings	52
7. Application	53
7.1. From Abstract to Concrete Integration Knowledge Management	55
7.2. Towards Software Adapter Generation	57
7.2.1. Client-Server	57
7.2.2. Publish-Subscribe	59
IV. Reference Implementation	62
8. Deployment and Technologies	63
8.1. Logical Architecture	63
8.2. Deployment Architecture	64
8.2.1. Interface Description Languages	65
8.2.2. Mapping Language L^*	66
8.2.3. Software Adapter Generation and API Endpoints	66
8.3. Application Examples	67
8.3.1. Client-Server Mapping Function	67
8.3.2. Client-Server One-to-One	67
8.3.3. Subscribe One-To-One	69
8.3.4. Publish One-To-Many	69
9. Safeguarding Expected Method Benefits	71
9.1. Speed	72
9.2. Reliability	73
V. Evaluation	76
10. Preliminaries	77
11. Empirical Evaluation 1: Mapping Generation for Sensor Values	78
11.1. Evaluation Setup	78
11.2. Evaluation Execution Process	79
11.2.1. Evaluation Steps	80
11.3. Implementation	81

Contents

11.4. Results	82
11.4.1. Break-Even Analysis	84
11.5. Threats to Validity	86
12. Empirical Evaluation 2: Mapping Generation for Services	87
12.1. Evaluation Setup	87
12.2. Evaluation Execution Process	88
12.2.1. Evaluation Steps	88
12.3. Implementation	91
12.4. Results	92
12.4.1. Break-Even Analysis	93
12.5. Threats to Validity	94
13. Empirical Evaluation 3: Adapter Generation for Services	95
13.1. Evaluation Setup	95
13.2. Evaluation Execution Process	96
13.2.1. Evaluation Steps	98
13.3. Implementation	98
13.4. Results	101
13.4.1. Break-Even Analysis	103
13.5. Threats to Validity	105
14. Performance Evaluation: Reasoning Algorithms and Architectures	106
14.1. Evaluation Setup	106
14.2. Fat Client Results	107
14.2.1. Discussion	109
14.3. Thin Client Results	109
14.3.1. Discussion	110
14.4. Threats to Validity	110
VI. Related Work, Limitations and Conclusion	112
15. Related Work	113
16. Discussion	117
16.1. Limitations	118
16.2. Future Work	119
17. Conclusion	122
A. List of Own Publications	133
B. Usage Examples	135
B.1. Link to Prototype	136

List of Algorithms

1.	Create Mapping Suggestions	50
2.	Find Transitive Mapping Chain	51
3.	Create Identifier Mapping Suggestions	52
4.	Create Mapping Suggestion with Ontology	91

List of Figures

1.1. Syntax and Semantics	3
1.2. Knowledge-Driven Architecture Composition Method	6
1.3. Running Example – Integration Knowledge Reuse and Reasoning	7
1.4. Perspectives on Semantic Service Interoperability	9
1.5. Structure of This Work	12
2.1. Barriers for Semantic Service Interoperability in IoT Systems	14
2.2. Top-Down vs. Bottom-Up System Design	17
2.3. Six Layer IoT Architecture	20
2.4. Service-oriented Architecture for Web Services	21
3.1. Software Adapter Implementation Method	23
3.2. Example – Software Adapter	24
3.3. Top-Down Engineering Method	26
3.4. Example – Top-Down Engineering Method	27
3.5. Bottom-Up Engineering Method	28
3.6. Example – Bottom-Up Engineering Method	31
5.1. Integration Cases	40
6.1. Knowledge-Driven Architecture Composition Method	47
6.2. Adapted Bottom-Up Engineering Method	48
7.2. Client-Server Split	57
7.3. Client-Server Aggregate	58
7.4. Client-Server Extend	58
7.5. Publish-Subscribe Split	59
7.6. Publish-Subscribe Aggregate	60
7.7. Publish-Subscribe Extend	61
8.1. Logical System Overview	63
8.2. Deployment Diagram	64
8.3. Example – JSONata	66
8.4. Example – Simple JSONata Mapping Function	67
8.5. Example – Complex JSONata Mapping Function	67
8.6. One-to-One Mapping using OpenAPI	68
8.7. Mapping Test & Validator using OpenAPI	68
8.8. One-to-One Subscribe using AsyncAPI	69

List of Figures

8.9. Example – One-to-Many Publish using AsyncAPI	70
9.1. Incomplete but Reliable Mappings	71
9.2. Incomplete Mappings	72
9.3. Ensuring Mapping Reliability	75
11.1. Semantic Interoperability Example for a Home Automation Platform	79
11.2. Eval 1 – Evaluation Steps	81
11.3. Eval 1 – High-Level System Architecture	82
11.4. Eval 1 – Reuse Task Time Comparison for Both Groups	83
11.5. Eval 1 – Integration Times Per Automation Rule	84
11.6. Eval 1 – Average Participant Performance	84
11.7. Eval 1 – First Integration Knowledge Reuse with Variable Channel Replacements	85
11.8. Eval 1 – Two Replacements with Variable Integration Knowledge Reuse	85
12.1. Eval 2 – Evaluation Steps	89
12.2. Eval 2 – High-Level System Architecture	90
12.3. Eval 2 – Integration Time	92
12.4. Eval 2 – Component Interaction Correctness	93
13.1. Eval 3 – Evaluation Steps	97
13.2. Eval 3 – High-Level System Architecture	99
13.3. Eval 3 – Average and Standard Deviation for All Integration Tasks	101
13.4. Eval 3 – Integration Time	102
13.5. Eval 3 – Errors	103
13.6. Eval 3 – First Integration Knowledge Reuse with Variable Mapping Reuse	104
13.7. Eval 3 – 50% Mapping Reuse with Variable Integration Knowledge Reuse	104
14.1. Eval 4 – Example for D=1	107
14.2. Eval 4 – Example for D=3	107
14.3. Eval 4 – Calculation Times for Fat Client	108
14.4. Eval 4 – Quadratic and Exponential Approximation Functions for Fat Client	108
14.5. Eval 4 – Calculation Times for Thin Client	109
14.6. Eval 4 – Quadratic and Exponential Approximation Functions for Thin Client	110
B.1. Eval 2 – Describing Context Based on An Ontology and JSON-LD	135
B.2. Eval 3 – Generated Software Adapter Project without Mappings	135
B.3. Eval 3 – Generated Software Adapter Project with Mappings	136

List of Tables

2.1. Comparisons of Code Repositories for Selected Home Automation Platforms	14
2.2. One Month Growth of Code Repositories for Selected Home Automation Platforms	15
2.3. Knowledge Management Activities for Software Architecture	18
4.1. Comparison of Methods to Achieve Semantic Interoperability in IoT	33
5.1. Complete Integration Knowledge for a Domain	42
14.1. Eval 4 – Parameters for Created Data Series	107
14.2. Eval 4 – Quadratic and Exponential Growth Rate Approximation	108
14.3. Eval 4 – Quadratic and Exponential Growth Rate Approximation	109
15.1. Related Solution Approaches	114
16.1. Related Bottom-Up Solution Approaches	117

Part I.
Introduction

1. Motivation

Service Interoperability for embedded devices is a mandatory feature for dynamically changing Internet-of-Things and Industry 4.0 software platforms. Interoperability can be achieved on the technical, syntactic, and semantic level. For instance, an HTTP endpoint running on a device could return a JSON payload when called by a client with a GET request.

```
{
  "volume": 20,
  "sourceName": 10
}
```

Listing (1.1) Client payload

```
{
  "volume": 20,
  "sourceName": 10
}
```

Listing (1.2) Server 1 payload

```
{
  "volume": 20,
  "input": 10
}
```

Listing (1.3) Server 2 payload

Assume the required payload of the client is defined as shown in listing 1.1 and the provided server payload is defined as shown in listing 1.2. On the technical layer, the client and the server must support the same network protocol. This enables the client and the server to address each other before talking. On the syntactic layer, the usage of the JSON standard regarding punctuation (e.g., keys are in quotation marks), keys are in quotation marks, parenthesis is determined. This enables the client and server to process the payload in a machine-readable form. On the syntactic layer, the client and the server know that "volume" means audio loudness and is measured in decibel and "sourceName" means HDMI port. Hence, words are linked to well-defined concepts based on a common domain model. This enables the client and the server to understand each other meaningfully.

Assume, we replace server 1 payload (see listing 1.2) with server 2 payload (see 1.3). Now, integration questions arise (i.e., service operation syntax and semantics do not change; no behavioral aspects are now important). These integration questions could be the following:

- Do sourceName and input refer to the same concept?
- Are volume and volume measured in the same unit?
- Can sourceName values be replaced with input values?
- How is such a transformation implemented so that we can execute it?
- Where are needed transformations deployed?
- ...

For humans, such questions can be answered by looking at PDF documents or machine-readable service descriptions. Based on their cognitive world model, their domain experience, and other information sources (e.g., an informal domain standard), they can efficiently reason about possible answers. For devices (i.e., automated component coupling approaches), the presented payload identifiers only consist of composing symbols with no links to concepts or things in reality.

1. Motivation

If a required and a provided interface should be coupled automatically, we must write these links and references in a formal (i.e., machine-understandable) way.

1.1. Context

In 2012, Barghani et al. [1] published their often cited survey about semantics in the Internet of Things. In this survey, the authors posed the following challenges related to applying semantic technologies into the IoT domain: dynamicity and complexity; scalability; semantic service computing for IoT; distributed data storage query; quality, trust and reliability of data; security; privacy, interpretation & perception of data. Achieving interoperability at the semantic level (we refer to this as semantic interoperability from now on) is well known since the 90s [2, 3]. Semantic interoperability ensures that services and data exchanges between a provided and a required interface make sense – that the requester and provider have a common understanding of the meaning of services and data [2]. Semantic interoperability in distributed systems is mainly achieved by establishing semantic correspondences (i.e., mappings) between vocabularies of different (data) sources [1, 3].

From a compiler viewpoint, component interface descriptions such as presented in Figure 1.1 are tokenized and parsed to build up an abstract syntax tree. This abstract syntax tree is then subject to semantic analysis. Here a compiler may check if all variables have been declared before their usage or if there exist type violations. However, the semantic analysis phase cannot identify the domain-specific meaning of a variable. For instance, the identifiers "volume" and "brightness" are both of type Integer but relate to different things in the real world.

For human-to-machine communication scenarios, this can be achieved by naming software identifiers according to a language that the human can understand. Hence, the software engineer builds the software, and the end user must share the same concept definitions that describe the associated real world entity. If not, the sender and receiver do not communicate meaningfully. For machine-to-machine communication, words and their associated meaning must be defined differently as machines operate on words without understanding their meaning. That means that devices need additional information to reason about the semantics of a word.

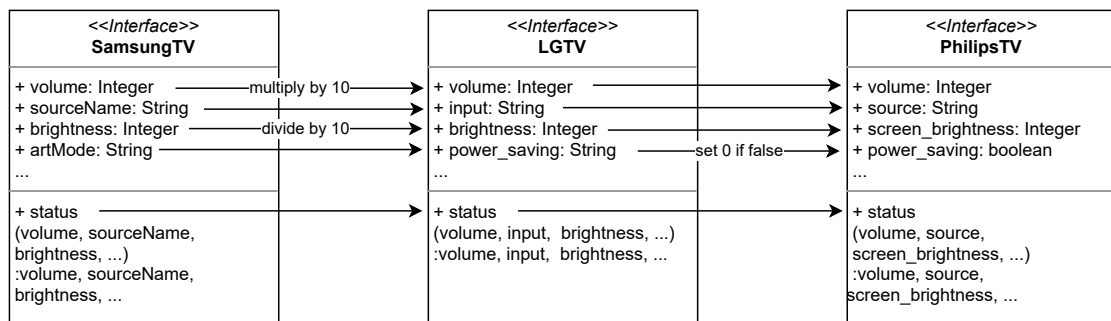


Figure 1.1.: Syntax and Semantics

Currently, most industrial or commercial software systems deal with semantic interoperability by using a domain-specific de-facto standard. For instance, a home infrastructure's cleaning robot

1. Motivation

is compatible with Google Home or Amazon Alexa. Consequently, the cleaning robot producer must implement the respective software adapters. Another example from the industrial automation domain is the ecl@ss standard [4] where each product must conform to a predefined product description. If the underlying software platform should support semantic interoperability in a more changeable way, we must engineer it somehow. More precisely, software adaptability is achieved by engineering principles (e.g., explicitly planned component configurations), emergent properties (e.g., implicitly derived from cooperation patterns of the participants), or evolutionary mechanisms (e.g., replacing components) [5].

Current solutions from the software architecture community already tackle semantic interface mismatch by automating the software adapter generation process. For example, Autili et al. [6] provide an automated synthesis method for mediators to achieve interoperability among heterogeneous systems that favours correctness.

It would be beneficial from an end user perspective if all devices (e.g., televisions) are controlled with a mobile remote control. If no domain standard exists, the application developer would be required to implement and manage many software adapters. In a worst case scenario, this would mean one adapter for each device in each distinct programming language.

1.2. Problem Statement

In the universe of IoT, there will not exist one distinct standard for each use case [1]. Agreeing on and keeping standards up to date is not feasible for dynamically changing IoT systems. Hence, system integrators are currently forced to implement software adapters. What is bad about this is not the manual implementation effort but the circumstance that the same integration knowledge is repeatedly implemented in these software adapters.

Bottom-up approaches that do not rely on a predefined standard try to automate service integration by describing each integration context based on service descriptions and interface mappings. These integration contexts are defined with a closed world assumption in mind. However, if an unforeseen integration case comes up, no automated service integration takes place. In dynamically changing IoT environments, this results in a manual service specification effort that does not yield the desired benefits as structural and behavioral interface mappings are assumed to be steady once they are defined. Formalizing possible integration contexts ahead to put them into inventory increases the specification effort even more as they may not be used [7].

Within this gap, we introduce a novel integration method [8]. This method does not aim at a fixed model of the desired domain. In contrast to existing bottom-up driven solution proposals, it refrains from formalizing concrete integration contexts in a big bang manner at a specific point in time. Instead, the proposed approach explicitly allows for interface mappings that are formalized incrementally. Therefore, interface mappings are only formalized in a machine-understandable way if a concrete integration case is present. The scientific foundations and principles are currently not available.

1.3. Research Question

The proposed method is an answer to our leading research question:

What does an integration method look like that can semantically integrate software services in an automated way based on incomplete integration knowledge?

Therefore, the following sub questions will be answered in this thesis:

Research Question 1. How can we make integration knowledge that is captured in imperative software adapter reusable?

Solution Approach: Describe interface mappings in a declarative way and store it centrally.

We provide a framework that enables a declarative description of integration knowledge. This declarative integration knowledge is then stored centrally so that it can be reused. Furthermore, we outline how we can integrate the proposed method into existing engineering processes. Last, we characterize existing interoperability methods on a conceptual level and provide multiple examples from existing solution proposals in contrast to our proposal.

Research Question 2. How well can we manage incomplete integration knowledge?

Solution Approach: Design, implement and evaluate an integration knowledge management process.

Writing integration knowledge using a formal language is a complicated and time consuming task. Hence, we outline algorithms that can reason about integration knowledge and introduce a process for capturing and reusing integration knowledge. The proposed algorithms work on graph structures that contain persisted integration knowledge. These algorithms can generate machine-readable interface mappings for unseen integration cases in a dependable way. The overall knowledge management process and its application to selected architectural styles are illustrated with realistic examples.

1.4. Solution Overview

The goal of the proposed method is to assist the system integrator in generating software adapters automatically. In addition to existing approaches, we enable semantic integration for services and make integration knowledge reusable [8, 9]. The technological leverage is to formalize and reason about declarative integration knowledge that evolves over time. The process leverage is to support the system integrator by generating a software adapter based on the computed interface mappings. To reduce the implementation effort, the method relies on the following principles:

- We do not require all component interfaces and mappings to be present at system design time but formalize them incrementally when a concrete use case is available

1. Motivation

- If all mappings for one integration case are present, then a working software adapter can be generated in a reliable way
- Evolution of the interface mapping model is allowed by construction

The set of all principles for semantic system integration can be subordinated under the term knowledge-driven architecture composition (KDAC).

From an engineering perspective, software adaptability is achieved by engineering principles (i.e., explicitly planned component configurations), emergent properties (i.e., implicitly derived from cooperation patterns of the participants), or evolutionary mechanisms (i.e., replacing components) [10]. KDAC tackles engineering principles, emergent properties and evolutionary mechanisms in the following way:

Engineering: At the core, KDAC is a software engineering method that tries to minimize the mapping formalization effort by relying on concrete integration cases instead of predefined composition models. We can integrate KDAC into current software engineering methods such as agile development or other incremental development modes. In addition to implementing an imperative software adapter, mappings are only formalized if a concrete integration case occurs at system design or run time (i.e., bottom-up). These mappings are stored incrementally using a declarative language. A declarative language allows for applying reasoning principles. In contrast to top-down methods (e.g., integration based on standardized and ontologies that are not expected to change anymore), incomplete integration knowledge is explicitly allowed.

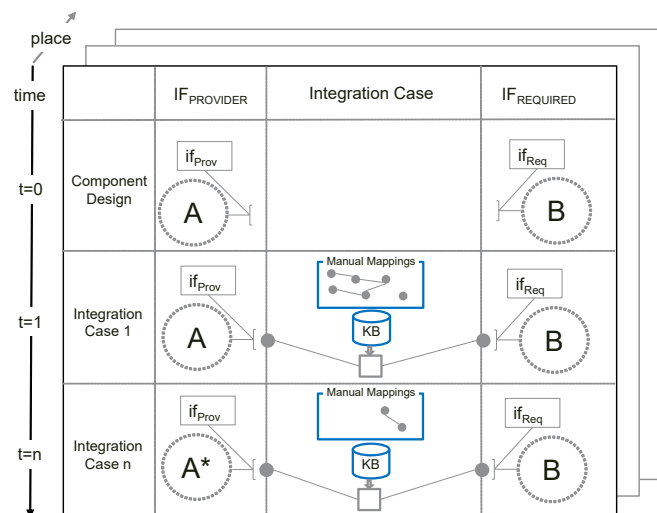


Figure 1.2.: Knowledge-Driven Architecture Composition Method

Evolution: In the beginning, the human-in-the-loop principle applies as the underlying knowledge base (see KB in Figure 1.2) is empty. Over time, integration knowledge is added to the knowledge base when new devices are integrated (see dots and lines at $t=1$ in Fig. 1.2). Hence, in the beginning, more formalization effort takes place. The declarative formalization allows for knowledge reuse from previous integration cases independent of the service model and service description syntax. Finally, the formalization effort is reduced by reuse and reasoning principles

1. Motivation

(see fewer dots and lines at $t=n$ in Fig. 1.2).

Emergent: Although integration knowledge is incomplete, automation is possible over time so that the system integrator fades out of the loop. Instead of integrating each device with one central domain model in a star like manner (i.e., the domain model acts similar to a "translator in the middle"), we can build up complex mapping chains. This structure allows for applying reasoning principles such as transitive relationships and inverse mappings across integration cases. Moreover, we can integrate unforeseen component replacements without human anticipation and intervention.

1.4.1. Running Example

This work's running example is a remote control application running on a mobile device that can manage televisions (e.g., set volume, change input, change the channel, etc.). Now, the remote control application should be adapted to support another television providing a semantically identical service interface. No existing application code should be adapted. Hence, the client's requests (i.e., mobile application) must be adapted from the currently supported television (e.g., POST Samsung) to another television (e.g., POST LG). Here service is defined as a networked interface that comprises a URL and data being sent as input and output (i.e., REST over HTTP/JSON).

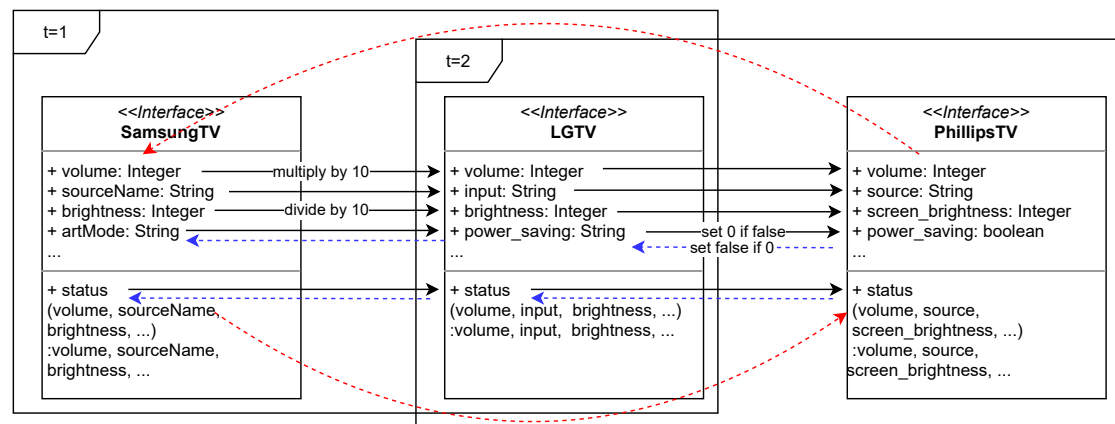


Figure 1.3.: Running Example – Integration Knowledge Reuse and Reasoning

Assume for component A (see Fig. 1.2) the interface of a Samsung TV and for component B the interface of an LG TV (see Fig. 1.3). At $t=1$, the action "status" and its input and output parameters are mapped. A formalized mapping can include an identifier replacement (i.e., black lines with no text) or an operation (i.e., black lines with text). As we can retrieve no mappings from the knowledge base (see Fig. 1.2) for the Samsung and LG interface, all mappings have to be created manually by the system integrator. At $t=n$, these mappings can be reused for the same integration case or for the inverse integration case (the Samsung TV substitutes the LG TV). Furthermore, we can also reuse formalized mappings for extensions of already seen interfaces (i.e., indicated by component A* in Fig. 1.2).

For a "transitive" mapping chain, assume another integration from LG TV to a Philips TV at

1. Motivation

t=2. Now, we can deduce the integration case from Samsung TV to Philips TV. Furthermore, the inverse integration case from Philips to Samsung may also be covered if there exists an inverse function for each formalized mapping within the chain Philips TV \leftrightarrow LG TV \leftrightarrow Samsung TV. Hence, as soon as the system integrator identifies the required and provided interfaces based on his available components, a software adapter can be (partially) generated automatically.

We evaluate the proposed method by conducting empirical experiments with students. Furthermore, our reference implementation is measured regarding its performance in computing new interface mappings.

In the experiments, we justify that incrementally specifying interface mappings and implementing a working software adapter require less engineering effort over time. We measure the effort based on the time needed for creating a software adapter and the number of errors made during classical software adapter implementation, and for using the proposed method.

In the performance evaluation, we provide insights into the developed reasoning algorithms and their applicability. Here we deploy our tooling as a serverless web application and as a containerized cloud instance. We measure the reasoning algorithms' time to compute complex mappings when the integration knowledge base contains many nodes and edges.

We have published parts of the approach at various maturity stages. The relevant core publications are:

- In "Knowledge-driven architecture composition: Case-based formalization of integration knowledge to enable automated component coupling" [8] we outlined the problem statement from a software-architecture viewpoint. A toy example, including a manufacturing execution system and an industrial oven, is discussed for illustration.
- In "Towards automating service matching for manufacturing systems: Exemplifying knowledge-driven architecture composition" [11] we illustrated the application of the proposed method based on SAWSDL service descriptions. We outlined how integration knowledge is formalized and reused based on our own ontology to improve matching results produced by the SAWSDL-MX matcher. We learned that ontologies are hard to use for our needs.
- In "Semantic Interoperability Methods for Smart Service Systems: A Survey" [12] we surveyed related approaches that also assist system integrator in creating software adapters.
- In "Automated configuration in adaptive IoT software ecosystems to reduce manual device integration effort: Application and evaluation of a novel engineering method" [9] we designed our first empirical experiment. Informatics students applied the proposed method within a home automation platform. They formalized and reused integration knowledge for devices using a standalone integration editor.
- A second empirical experiment based on OpenAPI specifications and JSONata is currently in the state of a working paper. Here students are equipped with a web based integration

1. Motivation

front end that exploits reference implementations of the reasoning algorithms. These algorithms can chain JSONata expressions on the payload and interface level by dependably preserving their meaning.

- The third empirical evaluation will be published at the International Conference on Web Engineering (accepted on 17.02.2021). In the paper "Knowledge-Driven Architecture Composition: Assisting the System Integrator to reuse Integration Knowledge" [13], we evaluated the end-to-end applicability of KDAC, including software adapter generation.

1.5. Influences

The proposed solution is influenced by a variety of topics (see Fig. 1.4). Its roots are to be found within the Software Architecture community. Here the underlying communication architecture such as Client/Server or Publish/Subscribe influence the service descriptions as well as the software adapter generation process for IoT systems [14]. We assume the availability of a common communication protocol and a structured data representation format. Within the adaptive system community and the web service community, multiple interoperability mechanisms have been suggested for various systems (e.g., web services, embedded systems, service-oriented architectures). The range of solution proposals include formal interface mapping approaches [7], domains-specific standards [15], service matching [16], service composition [17] and supervised learning for learning abstract syntax trees for distributed code [18]. Our proposed solution is influenced by building up a central knowledge base and the algorithms to calculate interface mappings.

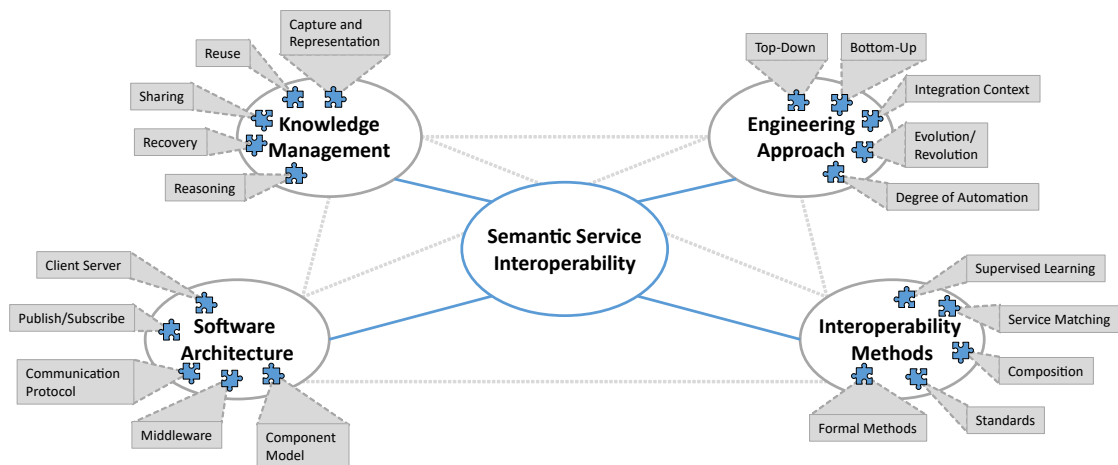


Figure 1.4.: Perspectives on Semantic Service Interoperability

The Knowledge Management community deals with knowledge representation, reuse, sharing, recovery, and reasoning. These concepts also have been recently transferred to the software architecture community [19]. Especially the topic of architectural mismatch is a well known problem for system reuse [20]. The proposed solutions contain reasoning algorithms that support the

1. Motivation

reuse characteristic for integration knowledge between a provided and a required interface. Apparently, a representation of the integration knowledge must be present. As a last point, the way distinct IoT systems are built and maintained determines which interoperability mechanisms can be applied during system design and run time. IoT systems can be engineered top-down using a waterfall-like engineering process or bottom-up in an agile development setting. Engineering tasks from both design dimensions can be automated. Here the proposed solution is influenced by a central knowledge base that is managed by system integrators in a decentralized way.

1.6. Application Scenarios

1.6.1. Consumer IoT

Currently, device manufacturers (e.g., smart lights or TVs) push their smart home devices into the market. Such commodity devices should be easily replaceable for homeowners. Most of these devices work together on a centrally managed standard (e.g., the Smart Applications REFERENCE ontology supervised by ETSI¹) or using a proprietary platform model (e.g., openHAB²). In the case of a standard, the corresponding interface elements and their input and output parameters are named according to the standard. Thereby, the semantics is checked by the software engineer so that the end user can work with the television in a plug and play manner.

In the case of a proprietary platform model, plug and play functionality in such software platforms is mainly achieved by creating a de-facto standard where the distinct platform manages all communication between devices. Devices do not communicate directly but can speak to each other by using the platform model as an intermediate language. Hence, integration happens manually between device and platform [21].

However, changing the centrally managed standard or the proprietary platform model requires significant effort if a smart home device that has not been captured so far should be supported. This is mainly due to the circumstance that the (proprietary) standard is considered to be complete. What is worse, proprietary domain models often contain platform dependant specialities. Here the proposed solution can help the system integrator to implement a software adapter by calculating interface mappings given required and provided interfaces. This is done by reusing integration knowledge from integration cases formalized by another building manufacturer in another home. Generated interface mappings may be incomplete such that the system integrator must formalize the missing mappings on his own. Nevertheless, this is more efficient than always having to formalize all mappings.

1.6.2. Industrial IoT

In industrial domains, plug and play is called plug and produce. Although the underlying concept for automated interoperability is similar, industrial interoperability solutions require a higher degree of safety and dependability.

Enabling production scenarios for micro batch sizes or commissioned production requires a digital production model that can be changed quickly. For achieving the required interoperability

¹<https://www.etsi.org/technologies/internet-of-things>

²<https://www.openhab.org/docs/concepts/items.html>

1. Motivation

at the semantic level, industry leaders currently cut the overall manufacturing domain into manageable pieces and create companion specifications (e.g., monitoring activities or automating processes). Companion specifications are part of the OPC UA framework, which includes a communication module and exposes a service model³. A technical companion specification can be enriched by domain-specific models that are published as a standard. Thereby, a variable's meaning can be determined by using a domain-specific standard such as ecl@ass⁴. Each embedded device manufacturer links internal variables used within programmable logical controllers to one item from the ecl@ass library. Hence, if every device manufacturer (e.g., for a commercial refrigerator to store vaccines) conforms to this standard, the software engineer ensures that each variable exposes the same meaning. Although there are fewer standards available compared to the consumer IoT domain, the goal of having only one standard (e.g., AUTOSAR [22] standard for the automotive domain) is hard to achieve as the manufacturing domain is more diverse than the automotive industry.

The proposed solution satisfies the required dependability properties as the system integrator is still in charge. Especially special purpose machinery manufacturers are a target group for storing and reusing integration knowledge as the available standards may not cover their devices. In contrast to smart home devices, integration knowledge is also needed to connect production devices to higher order information systems (e.g., manufacturing execution systems) and to connect a proprietary device with its standardized interface as offered by the digital twin. A digital twin is the virtual representation of a manufacturing device. Here system integrators can reuse centrally stored integration knowledge when they upgrade a production line at multiple clients on site. This allows for a swift change of production lines when a micro batch size of goods should be produced. The generated adapters can then be deployed as a component in the manufacturing service bus (e.g., BaSys 4.0)⁵.

1.6.3. Mobile Apps and Web Services

Flow-based programming allows end users to connect multiple web services in order to solve re-occurring tasks automatically. For example, if-this-then-that rules like "If a new file is uploaded, then notify me on my smartphone" reduces the time to check for new files manually. Therefore, most automation environments implement their software adapter to include new services (e.g., a new file upload service such as OwnCloud⁶).

Here semantic web services description techniques such as SAWSDL [23] can be applied. By doing so, a required and a provided service can be matched. For example, a file upload service is described as a required interface within the automation application, and each file upload service instance provides a service description that fulfills the required functionality. To reason about whether required and provided services match, a semantic service description contains links to an ontology. A reasoner then checks whether two concepts match exactly (i.e., the input of required and provided interface refer to the same model element) or can be computed (e.g., by class subsumption). However, most web services do not offer such a semantic service descrip-

³<https://opcua.vdma.org>

⁴<https://www.eclasscontent.com/index.php>

⁵<https://wiki.eclipse.org/BaSysx>

⁶<https://owncloud.com/>

1. Motivation

tion as the specification effort is perceived to be too high, and no de facto ontology exists for each use case (e.g., upload a file).

Nevertheless, many web services offer a syntactic service description, such as OpenAPI ⁷. This specification makes it easy for the human to understand what a web service does. Depending on whether the description is used as a template (i.e., API-driven development) or is generated from an existing interface (i.e., annotation-based development), the service description acts as a provided or a required interface. In the case of a template, the description acts as a required interface, and in the case of a generator, the description acts as a provided interface.

As OpenAPI descriptions are frequently used compared to SAWSDL, they can serve as a component model between which mappings can be formalized. Furthermore, the effort to create such a description is compared to a SAWSDL description lower. Using a declarative language for defining the mappings between OpenAPI description allows to store them in a knowledge base so that they become reusable. This allows to generate software adapters for mobile apps based on available services and to support the system integrator when new services become available.

1.7. Structure of This Work

Chapter 2 provides background context and relevant definitions. Furthermore, current method frames to achieve semantic interoperability are introduced (see Fig. 1.5). Chapter 3 formally describes our integration knowledge management process in detail and applies it to different architectural styles. We focus on Integration Knowledge Representation, Reuse, and Application. In chapter 4, we illustrate the theoretical concepts based on a reference implementation and selected examples. Chapter 5 describes four evaluations that focus on integration time savings, interface mapping correctness, and the system's performance. Last, we conclude our work in chapter 6 by outlining related work, limitations of our approach and future work.

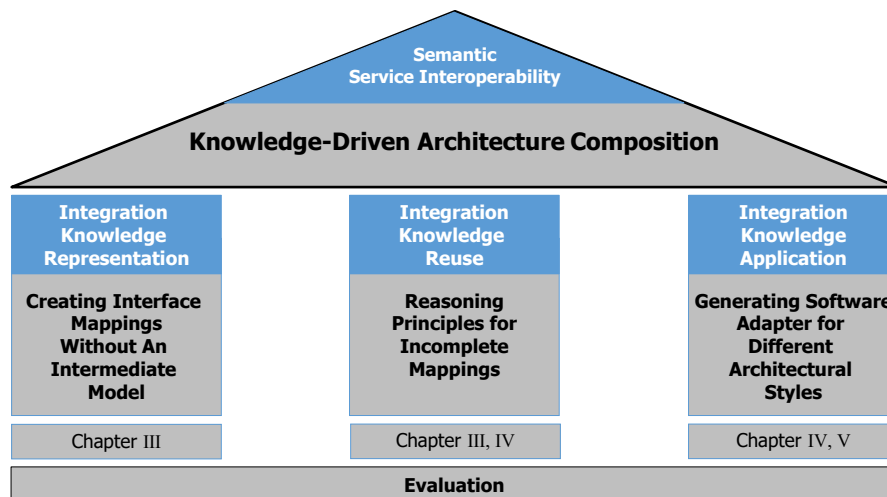


Figure 1.5.: Structure of This Work

⁷<https://www.openapis.org/>

Part II.

Background

2. Definitions and Context

Currently, there exists a dilemma for IoT systems between the considerable effort for creating service specifications with a formal semantic grounding and implementing many point-to-point adapters. Although dynamicity in those systems is a mandatory feature and different research communities have produced a significant set of solutions, the dilemma persists. This chapter will outline existing solution frames and argue why the interoperability problem for the still relatively new IoT system class cannot be solved by applying them.

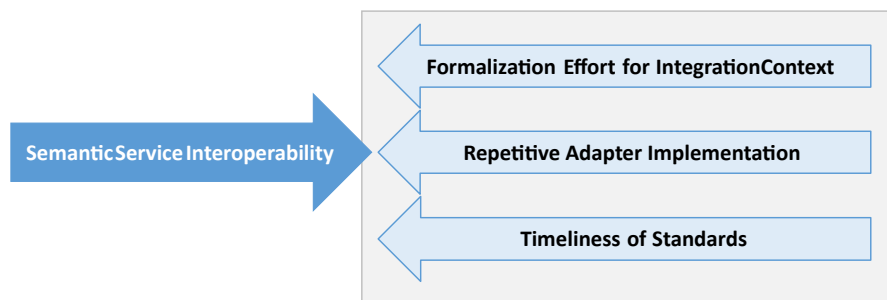


Figure 2.1.: Barriers for Semantic Service Interoperability in IoT Systems

2.1. Barriers

Most open-source IoT platforms provide many software adapters for IoT devices that should be supported. For instance, the home automation platforms iobroker, openHab, and homeassistant provide 341, 281, and 969 adapter projects in their github repositories (data was queried on 15.01.2021). An interesting statistic unfolds if we compare all adapter projects to their core platform logic and the user interface. For all home automation platforms, the code repositories for adapters are the largest in lines of code and also grow the fastest in absolute numbers (see Table 2.1, data was queried on the 10.01.2021 and Table 2.2, data was queried on the 10.02.2021).

Table 2.1.: Comparisons of Code Repositories for Selected Home Automation Platforms

Platform	Core			UI			Adapter		
	Files	Lines of Code	Comments	Files	Lines of Code	Comments	Files	Lines of Code	Comments
iobroker	166	188 574	6 858	140	35 749	31 33	17 231	3 642 123	271 114
openHab	2 269	151 678	64 494	2 121	196 561	7 130	10 661	914 498	240 498
homeassistant	2 619	369 861	30 442	1 332	298 785	3 946	1 2668	545 308	50 193

Although these statistics should be interpreted with a grain of salt, it provides a feeling of the dimension of how many lines of code are written within such open source projects. Most adapters

2. Definitions and Context

Table 2.2.: One Month Growth of Code Repositories for Selected Home Automation Platforms

Platform	Core			UI			Adapter		
	Files	Lines of Code	Comments	Files	Lines of Code	Comments	Files	Lines of Code	Comments
iobroker	+3 (1,81%)	+859 (0,46%)	+84 (1,22%)	0 (0%)	+39 (0,11%)	+1 (0,03%)	+344 (2,00%)	+119 176 (3,27%)	+3 710 (1,37%)
openHab	0 (0%)	0 (0%)	0 (0%)	+268 (12,64%)	+31 469 (16,01%)	0 (0%)	+354 (3,32%)	+34 888 (3,81%)	+5 714 (2,38%)
homeassistant	+91 (3,47%)	+25 101 (6,79%)	+1 099 (3,61%)	+33 (2,48%)	+10 528 (3,52%)	+9 406 (-9,28%)	+684 (5,40%)	+30 774 (5,64%)	+1 226 (2,44%)

must be written for each device and platform manually.

Hence, the first barrier to achieve semantic service interoperability is repetitive adapter implementation (see Fig. 2.1). The other two barriers are the timeliness of standards and the formalization effort for integration contexts. Timeliness of standards deals with the issue of keeping formal or informal domain standards up to date. Formalization effort for integration contexts deals with formal descriptions of how two interfaces should be coupled (e.g., platform and device or device and device).

All three barriers are in the scope of this work and will be discussed in this background chapter.

2.2. Terminology

In general, a **software architecture** can be defined as "the set of principal design decisions made about the system" [24]. These design decisions typically involve system functionality, data about the system state, and interaction of the composed software components. A **software component** is an architectural entity that encapsulates a subset of the system's functionality and/or data and restricts access to that subset via an explicitly defined Application Programming Interface (API) [24]. Software connectors handle communication between software components. A **software connector** is an architectural element tasked with effecting and regulating interactions among components [24]. For instance, a software connector can be operationalized by the software adapter pattern [25]. Such software connectors are needed if APIs do not match. Hence, reusing existing software components is hard if their architectures mismatch [20, 26]. One problem dimension responsible for the architectural mismatch is the semantics of API elements.

2.2.1. Syntax and Semantics in Software Architectures

Abstracting away from network protocol mismatches [27], component interface mismatches can occur on the syntactic and semantic level [28]. Among others, the component interface may also show mismatches regarding the quality of services, preconditions and postconditions, service granularity, and order of service invocation [28]. Component interfaces that expose the same functionality with regards to their semantics may come in different syntactical flavors. More formally, a model for an interface consists of a syntax, a semantic domain and semantic mappings [29]:

Syntax: We define syntax as the set of basic expressions constrained by a grammar that may be used in a language.

2. Definitions and Context

Semantic domain: Syntactic expressions must correspond to at least one element in a semantic domain (i.e. universe of discourse) to reveal their meaning. We can think of the elements from one semantic domain as a conceptual model. A conceptual model often serves as a source of knowledge about a problem area [30]. It represents the concepts and the associations among them and also attempts to clarify the meaning of various terms.

Semantic mapping: A semantic mapping is a function which maps every syntactic language expression to its semantic domain element.

One step towards a better human oriented understanding of APIs are interface descriptions. An **interface description** can either be used to express actions a component **requires** from its environment or to express actions that are **provided** to the environment. Then, software components are compatible if a contract between their interfaces can be defined that maps all necessary interface description elements (i.e., they can be integrated). In most software applications, this results in implementing a software adapter. From a component-based software development perspective, the concept of functional compatibility is used. Functional compatibility means that the required and provided high level functionalities are semantically equivalent [31]. In distributed systems (e.g., based on web services), this concept is similar to the concept of interoperability. In order to operationalize the semantic Mapping function for the HTTP/JSON service component model, a machine-understandable **interface description language** and a domain model are needed. This language L [32] is defined as

$$L = (C, A, S, M_S, M_C) \quad (2.1)$$

where C is the concrete interface syntax, A is the abstract syntax, and S defines the semantic domain elements. $M_C: C \rightarrow A$ is a function that assigns graphical or textual syntax elements C to one abstract syntax element A and $M_S: A \rightarrow S$ relates abstract syntax elements A to semantic domain elements S (see aforementioned definition of semantic mapping). If the semantic domain S is explicitly described in a machine-understandable way and all abstract interface syntax elements are mapped to S (e.g. using an OWL ontology), then a software adapter for two interfaces with different concrete syntax elements C can be generated in an automated way [7, 27]. The semantic domain S can be modeled as a closed world or open world model. A closed world model of a system directly represents the desired domain (e.g., SAREF ontology [33]). This means that there is a functional relation between language expressions and the modeled world. The represented knowledge is implicitly viewed as being complete [5]. Software (eco-) systems such as smart home systems can be designed with an open world (adaptable domain model) or closed world (static domain model) assumption in mind.

In an IoT platform, multiple roles are present. These are platform providers, application providers, system integrators, device providers, and operators. Multiple roles can be assigned to one person. Semantic interoperability is usually achieved by application providers, system integrators, and device providers. For these three roles, open source code, standards, and APIs are the most relevant aspects [34]. A **system integrator** benefits especially from standards concerning interoperability and open source when implementing software adapter for end-to-end IoT solutions. Because of the difficulties in creating standards with machine-understandable semantics mentioned earlier, most IoT platforms require software adapter implementation to support a new device. This manual process involves substantial development effort with no automation.

2. Definitions and Context

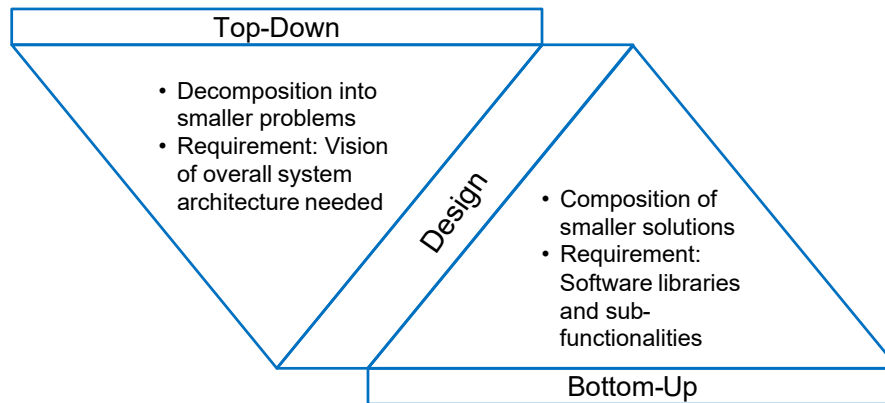


Figure 2.2.: Top-Down vs. Bottom-Up System Design

2.2.2. Engineering Approaches

Service compositions can be performed **bottom-up** or **top-down** [35]. Bottom-up refers to the composition of pre-existing service interfaces. Adapters may be required due to mismatches (i.e., from available software components [2]). Top-down refers to the service composition given a pre-existing composition model. Adaptations may be required to fit services into the composition model (i.e., from requirements [2]). Hence, the fundamental difference between top-down and bottom-up is whether the available integration case in an IoT system (e.g., home automation rule) is predefined or can dynamically emerge. This distinction is visualized in Fig. 2.2. Here any software artefact design can be driven by an overall vision of the IoT platform or can be composed of smaller software solutions or both.

Semantic integration is a well known topic regarding heterogeneous data sets. For instance, to analyze resulting data in a business intelligence system or applying data science algorithms. Here typical integration actions are extract, transform, and load. In this vein, semantic data integration based on machine-understandable ontologies has emerged [36]. **Semantic interoperability** ensures that services and data exchanges between a provided and a required interface make sense – that the requester and the provider have a common understanding of the meaning of services and data [2]. Semantic interoperability in distributed systems is mainly achieved by establishing semantic correspondences (i.e., **mappings**) between vocabularies of different (data) sources [1, 3]. From a software engineering viewpoint, domain standards, mapping instructions (e.g., companion standards), or individual coordination achieve semantic integration. Depending on the degree of formalization, domain knowledge can be queried and executed such that semantic integration scenarios (c.f., Figure 1.3) are automatable. Similar to semantic data integration, formal ontologies (e.g., using the RDF language) are also widely applied for software interfaces to relate a shared conceptualization of use cases to interface elements. Hence, semantic integration is similar to semantic interoperability by creating a mapping from one to another vocabulary but differs between research communities. Semantic integration has its roots in the artificial intelligence community [3], whereas semantic interoperability originated from software engineering processes for distributed systems. Semantic integration techniques are applied to data structures (e.g., databases), and semantic interoperability techniques deal with deployed

2. Definitions and Context

units of software so they can talk to each other using their interfaces in a meaningful way. Integration can happen at the **data** and at the **service** level. In this work, services are defined as software interfaces that are accessed over some application layer protocol. Services may require data integration regarding their payload (e.g., JSON document of an HTTP Method). Therefore, services are a candidate for data integration. Besides data integration, the software interface may also expose preconditions and postconditions, states, or generic method verbs (e.g., create-read-update-delete). For closed information systems, various languages such as WSDL [37], or SAWSDL (i.e., WSDL with links to a domain ontology) can be used at system design time to describe all relevant service characteristics. These interfaces and the underlying domain model is not expected to change. Consequently, the required formalization effort pays off over system operation years. However, data and services within IoT software systems change often and unexpectedly. Software systems such as smart home platforms or Industry 4.0 platforms require dynamics as a mandatory feature. Hence, the rate of change directly influences the formalization effort for service characteristics.

2.2.3. Knowledge Management

Software architecture knowledge management is a rather new field of research [38, 19]. Capilla et al. [19] describe the need for capturing architectural knowledge (e.g., integration knowledge) within the dimensions of sharing, compliance, discovery, and traceability. In contrast to technical activities, Li et al. rather look at knowledge-based activities based on concepts from the knowledge management framework [38]. They derive their knowledge management activities from theoretical system science [39].

Table 2.3.: Knowledge Management Activities for Software Architecture (adapted from [38])

Knowledge-based Approach	Knowledge Activity in the KM Framework in [39]
Knowledge Capture and Representation	Knowledge storage
Knowledge Reuse	Knowledge application
Knowledge Sharing	Knowledge transfer
Knowledge Recovery	Knowledge creation
Knowledge Reasoning	Knowledge creation

Li et al. [38] provide the following definitions for each knowledge-based approach in software architecture:

- Knowledge Capture and Representation (KCR) extracts knowledge from diverse sources as well as its acquisition directly from the stakeholders, and expresses knowledge in certain forms so that the knowledge can be used for automatic or human reasoning. On the one hand, knowledge representation accompanies knowledge capture since the knowledge should be represented in a particular form when captured; on the other hand, knowledge capture may happen during knowledge representation. For instance, when one uses a conceptual model to transform (e.g., annotate) implicit knowledge to explicit knowledge, knowledge capture takes place simultaneously. We thus consider knowledge capture and knowledge representation as a combined knowledge approach.

2. Definitions and Context

- Knowledge Reuse (KR) applies existing knowledge (e.g., architectural patterns or integration knowledge) in a particular context for various purposes.
- Knowledge Sharing (KS) exchanges knowledge (e.g., skills or expertise) among individuals in a community or an organization.
- Knowledge Recovery (KRv) recovers explicit knowledge from tacit knowledge, e.g., decision rationale that is not documented. In KM theory, knowledge is classified into tacit knowledge, which resides in people's heads, and explicit knowledge, which is codified in a certain form.
- Knowledge Reasoning (KRs) concludes and derives new knowledge from existing knowledge through inference. An example is reasoning based on the rationale knowledge of the existing architectural design decisions to make new decisions addressing new or modified design issues.

In general, **knowledge-driven** architecture composition relates to human knowledge about composing services in a meaningful way. In our understanding, an approach can be knowledge-driven as soon as at least one knowledge management activity is carried out. However, we define knowledge-driven more precisely. Knowledge-driven relates to knowledge about semantic connections between software interfaces (i.e., mappings). These semantic connections can be explicitly captured, shared, and reasoned about so that they become reusable in an automated way. In general, achieving reuse at the architectural level is considered a complex problem as the involved systems are specialized to different degrees [20]. For instance, generic styles such as call-return require other integration activities compared to software product lines. Garlan et al. [20] note that dynamically changing systems, trustworthiness (e.g., reliability of mappings), architecture evolution, and architecture lock-in are increasing problems to reusing software assets.

2.3. Internet of Things Software Architectures

The Internet of Things (IoT) can be defined from various perspectives. In a colloquial way, the IoT is a network of devices that can communicate with each other. This network of devices provides connectivity for everyone, and everything [40]. Within this work, we define **IoT** from an architectural viewpoint as "the internal/external communication of intelligent components via internet in order to improve the environment through providing smarter services" (as defined by Muccini et al. [14]).

In Fig. 2.3, a corresponding illustration for a six layer IoT architecture is given. We shortly describe the presented layers based on the explanations provided by Muccini et al. [14]. The perception layer consists of the physical objects and transfers them to the virtual space. The adaption layer facilitates the interoperability of heterogeneous IoT devices and may be omitted if no adapters are needed. The network layer adds networking protocols to the devices (e.g., Bluetooth or WiFi). The processing & storage layer analyzes and stores data provided by the connected devices via their services. On this layer, cloud computing, ubiquitous computing, databases and others can be applied. The application layer provides the service as requested by

2. Definitions and Context

the end user in the corresponding application domain. As a last point, the business layer contains everything to run the business model and generate revenue.

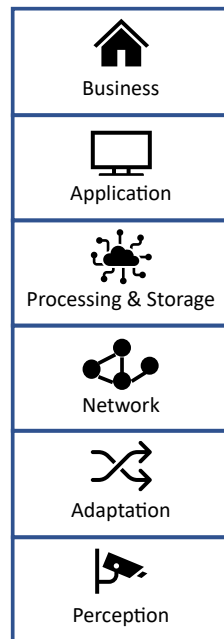


Figure 2.3.: Six Layer IoT Architecture (adapted from [14])

Depending on the underlying **distribution pattern**, the various layers can be deployed to different devices [14]. For example, a centralized distribution pattern assumes that there exists a central processing & storage layer that each device is connected to. In contrast, a decentralized distribution pattern assumes that all components (e.g., IoT devices) are fully connected and may expose their own resources to process, combine and provide services to other components in order to achieve a user goal. Regarding quality attributes, Muccini et al. [14] state that IoT systems should be scalable, secure, interoperable and should expose a good performance. According to their mapping study, privacy, availability, mobility, reliability, resiliency, and evolvability are also essential but of less concern according to their mapping study.

In this work, we focus on the **adaptation layer**. Regarding the mentioned quality attributes, interoperability and reliability are of importance. Especially interoperability at the semantic level is still a challenge for IoT systems [1, 41]. Therefore, Barnaghi et al. [1] identified and analyzed the following related challenges to achieve semantic interoperability such IoT systems: dynamicity and complexity; scalability; semantic service computing for IoT; distributed data storage query; quality, trust, and reliability of data; security and privacy and interpretation & perception of data. Semantic service computing, dynamicity and complexity are in the scope of this work.

Semantic service computing describes the application of semantic technologies within the concept of service-oriented architectures. Initially, service-oriented architectures (SOA) supported business enterprises using internet protocols [24]. The general idea behind service-orientation is choreography or orchestration of distinct services to achieve a user goal (e.g., book an airline

2. Definitions and Context

ticket). A user goal is usually associated with a software program that knows the services to be invoked to achieve the desired goal. For example, web services are one implementation of a service-oriented architecture (see Fig. 2.4). The core architectural activities to build such systems are describing the components and their services, determining the types of connectors and describing the application as a whole [24]. Core problems of SOA systems involve interoperability for open, distributed systems as heterogeneous services may come and go. The concept of SOA can be conceptually transferred to IoT systems. Hence, IoT devices offer their functionality by making their own services reusable from third parties. In order to solve interoperability at the semantic level for such IoT systems, semantic technologies are widely applied. Hence, the term of semantic service computing.

Applying semantic technologies such as SAWSDL [23] revealed new problems for IoT systems. Semantic technologies are suitable for interoperability, given that common ontology models are shared, and widely reused [1]. However, IoT services often operate in dynamic environments, constantly evolve and are subject to change frequently. Hence, they are different from most legacy services offered by information systems or the web. This also means that service composition is more challenging in the IoT domain, where reliable services are abundant.

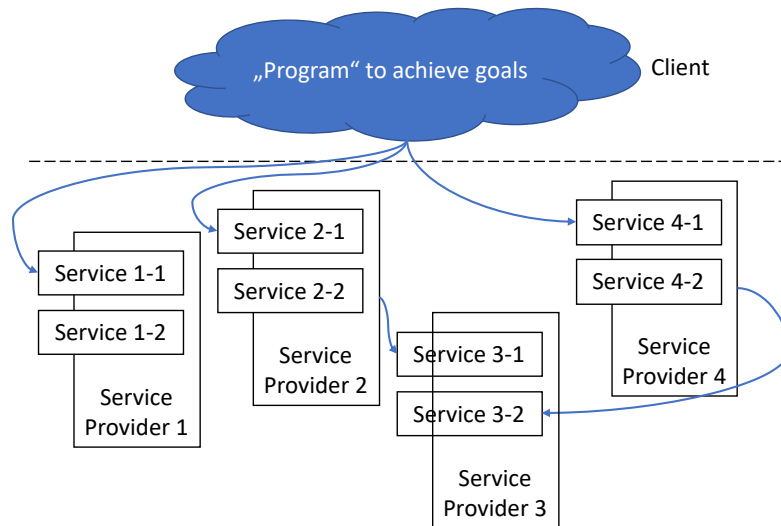


Figure 2.4.: Service-oriented Architecture for Web Services (adapted from [24])

Dynamicity relates to the pervasiveness and volatility of underlying devices and their environment that require continuous updates and monitoring [1]. Imagine that there exist not only one but multiple clients that access service 1-1 (see Fig. 2.4). If the interface of service 1-1 should ever change, then all users of that service must become aware of the change and modify their requests accordingly. For this and other reasons, many decentralized enterprise application developers have chosen to base their designs on REST. Thereby, they focus on the exchange of data rather than focusing on the invocation of functions with arguments. Although there are also service descriptions (e.g., JSON-LD [42]) for services following the REST paradigm, these descriptions still involve complex ontology engineering (e.g., adapting an ontology to the current integration case) and usage (e.g., reuse of share ontologies). The **complexity** involved in

2. Definitions and Context

describing services using common semantic web frameworks has hindered wide adaptation [1]. Dynamicity of the underlying devices also directly influences the adaptation layer (see Fig. 2.3) and the way it is engineered. Change in IoT software systems happens frequently. For example, Forbes predicts that until the year 2025, 75 billion devices will be connected to the internet¹. This translates to 20 million devices per day on average. However, innovative use cases such as drone delivery are not standardized yet. Hence the question arises when interface descriptions and their mappings are maintained. As most IoT systems require an always on operating mentality, it is debatable if the traditional software engineering life cycle consisting of a requirements phase, implementation phase, testing phase, and operating phase is still valid. Hence we only differentiated between **run time** and **design time** for software systems. From the viewpoint of semantic interoperability, the point of time for creating and/or updating mappings between interfaces is important. For example, systems that rely on a centralized distribution pattern usually rely on formal standards. Here all current and future use cases, as well as their associated domain model, are defined in a revolutionary way (i.e., in a top-down manner). Once fixed, they stay rather static and must be respected during system design time by the application and device manufacturer to ensure semantic interoperability at system run time. If no suitable domain model is available, an own domain model is built up based on the use case at hand. In contrast to top-down defined standards, this domain model is expected to change. Depending on the degree of formality, adapting the domain model can become complex. For example, adapting a domain model stored in a non relational database may be easier in contrast to adapting an OWL ontology. Hence bottom-up integration methods do have a concrete integration case at hand (i.e., required and provided services) but may suffer from (too) much formalization effort if integration knowledge is missing (e.g., interface mappings).

In the next subsection, the presented concepts are discussed in the context of frameworks for software engineering methods to achieve semantic interoperability. These methods are software adapter implementation, top-down engineering methods, and bottom-up engineering methods.

¹<https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>

3. Methods to Achieve Semantic Interoperability in IoT

3.1. Software Adapter Implementation

In the case of ad hoc integration activities, the structural design pattern of a wrapper or software adapter can be used. Generally, a software adapter allows incompatible interfaces to work together by assuming that they are functionally compatible (i.e., we can transform them into each other).

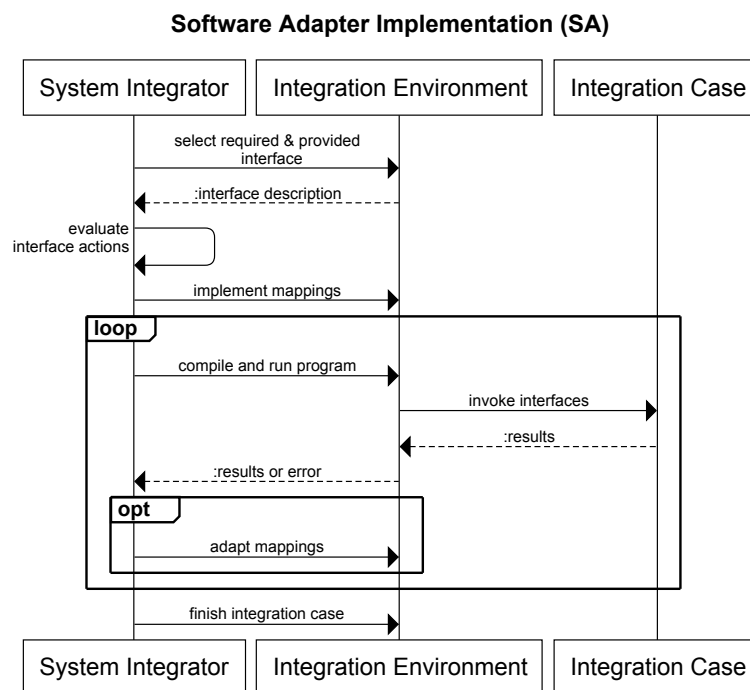


Figure 3.1.: Software Adapter Implementation Method

When a system integrator implements a software adapter, the mapping formalization style determines code reusability. For instance, a declarative programming style results in higher code reuse compared to an imperative, procedural programming style [43]. This is mainly because declarative programming styles focus on the intention of the mappings needed (i.e., describe the desired solution) rather than the programming constructs that control the state of execution (i.e., write instructions to reach the desired solution).

3. Methods to Achieve Semantic Interoperability in IoT

If an imperative, procedural programming style is used, then the software adapter is implemented manually every time the integration case occurs between two service endpoints. Therefore, Fig 3.1 illustrates the underlying method (SA) to integrate networked service components using a software adapter pattern (e.g., see Fig 3.2). Within the illustration, the integration environment relates to the tooling infrastructure. This is usually an Integrated Development Environment (IDE) such as Visual Studio Code. The Integration Case relates to the services that are called by a program. The main challenge here is the time and correctness of implementing a software adapter manually.

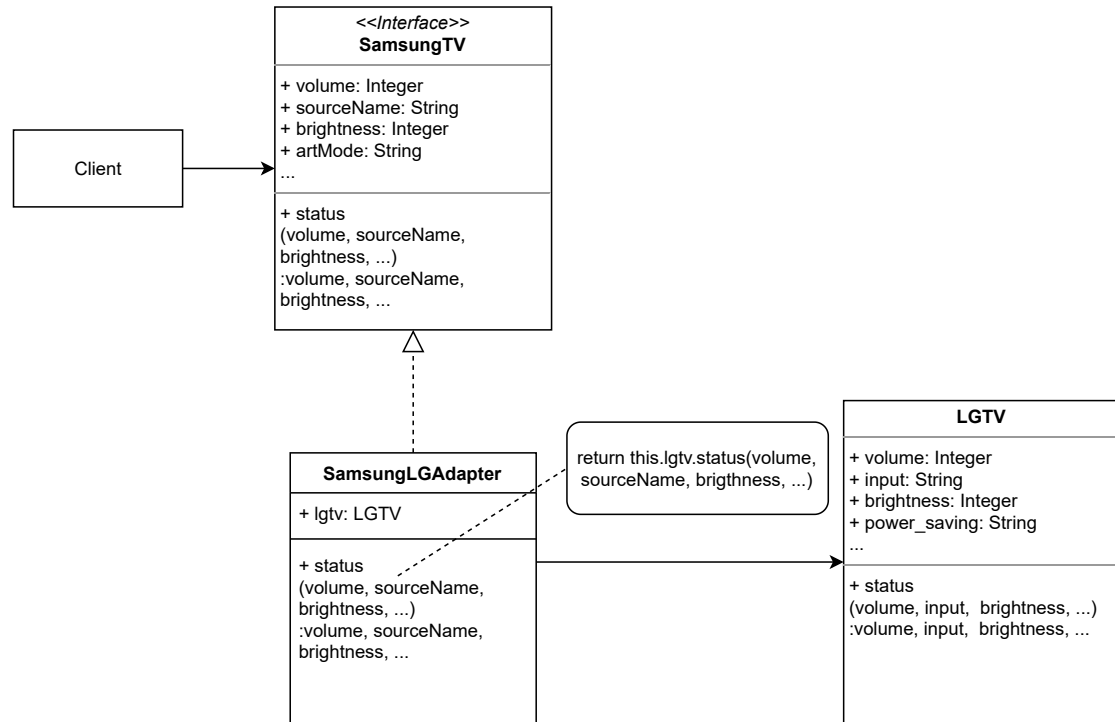


Figure 3.2.: Example – Software Adapter

Integration Example for Composition: For example, *artMode* is mapped to *power_saving* by the system integrator (see *implement/adapt mappings* in Fig. 3.1). Here the *semantic mapping function* M_S is implicitly codified in the imperative program code. For future integration cases, integration knowledge reuse is only possible for the system integrator (e.g., by reading code in adapter projects). This is because only a human can interpret and reason about the adapter code, but the platform itself cannot. Hence, agile development teams have to implement similar software adapters for the same integration context all over again on different platforms by interpreting someone else’s code.

Integration Knowledge Management: Integration knowledge is always incomplete as only a concrete integration case is known. Adding or updating an interface always requires code adaptations. The formalization effort in lines of code is low as only the software adapter must be implemented in the desired imperative programming language (e.g., JavaScript or Java). The

3. Methods to Achieve Semantic Interoperability in IoT

amount of time to write these lines of code is within minutes or hours as the system integrator is typically used to apply the chosen programming language. Integration knowledge is implicitly defined inside the software adapter. It cannot be queried and hence reused out of the box. The implicitly defined integration knowledge can be evolved by adapting the imperative code. The final software adapter is deployed in the adaptation layer (see Fig. 2.3).

Example Systems: Smart Home Systems such as openHAB offer a platform-specific domain model that must be used when implementing a software adapter for this system. Another example is IFTTT, where no domain model is predefined, but the end user must make sure that data is interoperable on the semantic level between interfaces.

Using the software adapter pattern, the system integrator can perform ad hoc integration activities. However, this approach is unstructured as integration problems were not considered at system design time. Hence, this approach relies mostly on manual coding and is thus error prone. For most dynamic IoT systems, a framework for integration activities should already be considered as a system requirement in the first place. Automated and tool supported adapter creation is a necessary design decision.

3.2. Top-Down Engineering Methods

Top-down engineering methods (TD) use interface mappings that are correct by construction. In contrast, web service matching approaches may produce only by probabilistic result values. Although matching solutions provide automation to software adapter implementation activities, we only care about methods that provide a perfect match. To achieve such results that are correct by construction, so called mapping approaches are used.

Top-down engineering methods rely on a complete, pre-defined domain model S (see Def. 1) that describes the application domain available at system design time. For instance, AUTOSAR provides a machine-readable domain standard for the automotive industry [44]. If a required and a provided service endpoint support one composition model, then top-down integration approaches can be applied (see Fig. 3.3). *Top-down* means that all independent parties involved in an application domain agree on standards regarding interface descriptions syntax A , a mapping function M_S , and a domain model S (c.f. AUTOSAR run time [44]).

If such a global agreement is not available, we can apply web service composition approaches [45]. Besides well-known approaches such as the Semantic Web Service Description Language (SAWSDL), more lightweight service description languages have recently been proposed to automate software component integration [45, 46]. For example, the JSON-LD language [42] is one promising interface description language that can automate service compositions based on machine-understandable domain models in data-driven environments [47, 48]. Therefore, JSON-LD combines the JSON syntax with Linked Data (LD in JSON-LD) principles to formalize the semantic mapping function $M_S: A \rightarrow S$ using an URL that points to a machine-understandable domain ontology S . However, this language does not support coordination or communication mismatches (i.e., service granularity and order of service invocation) as it relies on the HATEOAS principle for RESTful web services. Approaches that also respect coordination or communication mismatches during automated software adapter generations mainly rely

3. Methods to Achieve Semantic Interoperability in IoT

on Labeled Transition Systems (LTS) as an additional description document besides functionality [27, 7]. For a domain-specific model S these approaches rely on machine-understandable ontologies such as the Web of Things, Semantic Sensor Network, or other domain formalization (see [49] for an overview).

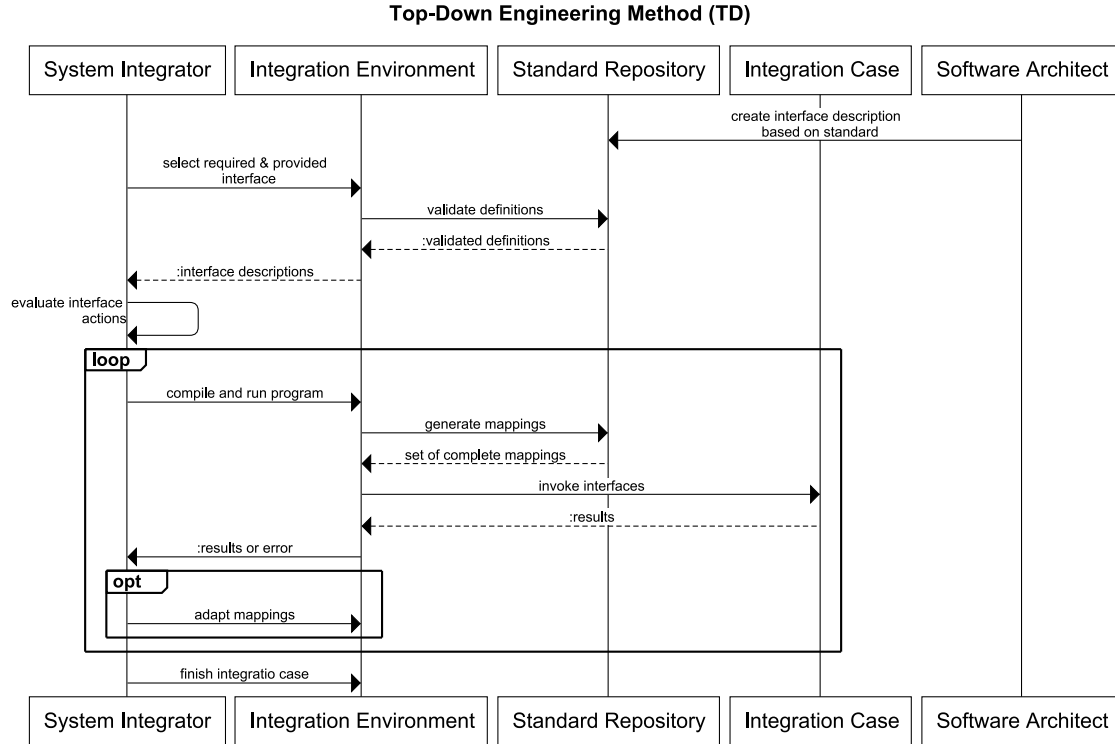


Figure 3.3.: Top-Down Engineering Method

The challenge for such standard-based approaches in software ecosystems is that the software component interface elements may exhibit another meaning in the describing context than in the integration context (see Fig. 1.3). In the describing context, the software architect links all interface syntax elements C to a standard S (see first line from Software Architect to Standard Repository in Fig. 3.3) using a description language L (see Equ. 2.1). Then, these interface descriptions are persisted in a standard repository. In service-oriented architectures, this is mainly done by using a service registry. In the integration context, the platform checks whether all keys from the provided and the required interface can be mapped (e.g., pointing to the same URL in one namespace) [45]. This means that top-down approaches rely on the assumption that a complete domain model can be derived in the requirements specification phase to ensure context awareness and reuse of service components at system run time [45]. If the standard repository should not contain all necessary mappings, the system integrator must adapt them manually. However, this case should not occur.

3. Methods to Achieve Semantic Interoperability in IoT

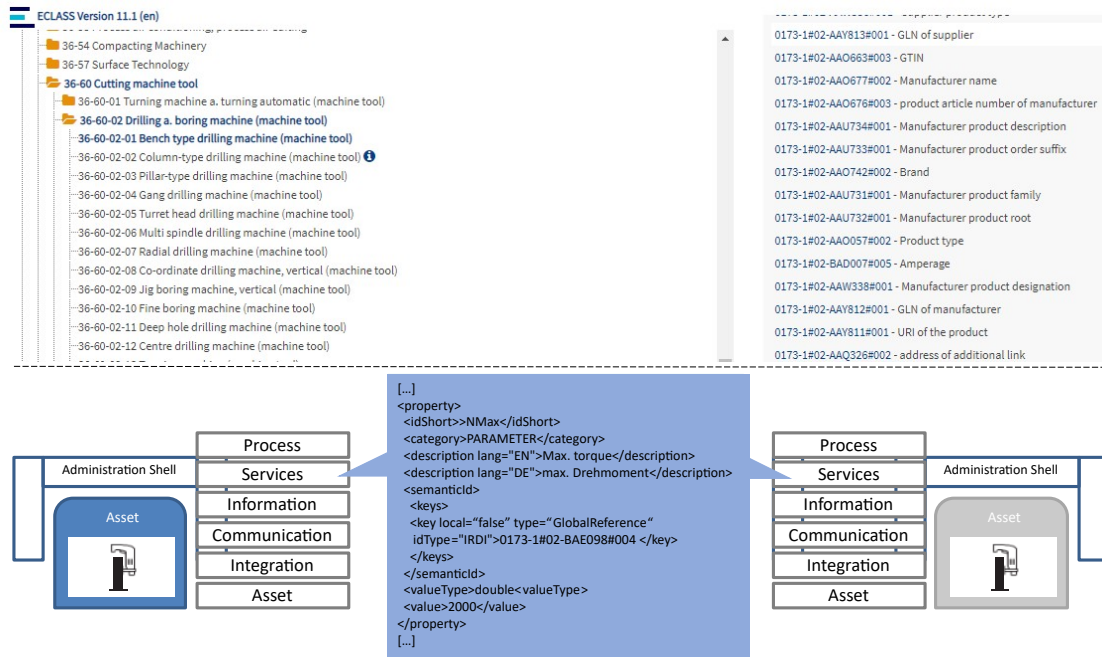


Figure 3.4.: Example – Top-Down Engineering Method

Integration Example for Standards: For example, *artMode* refers to <http://dbpedia.org/resource/Configuration> and *power_saving* refers to http://dbpedia.org/resource/Power_saving (see *generate mappings* in Fig. 3.3). If the standard in use does not contain a relationship from *.../Configuration* to *.../Power_saving*, then no automated interface mapping can occur although the identifiers are semantically identical. In agile development teams, integration knowledge reuse only happens if and only if the standard used is complete, unambiguous, and understood by all human developers involved. If not, the software architect's interface specification effort during *describing* context is not fully applicable. Hence, top-down approaches focus on reusing domain knowledge expertise as a result of an extensive agreement process.

Integration Knowledge Management: Integration knowledge for concrete integration cases is not needed. This is achieved by forcing all participants within a system to implement the same standard. The standard fixes syntax and semantics of all interface descriptions involved. Hence, there is no formalization effort in line of codes. The amount of time to create a complete domain standard is years, as it must be complete and used consistently among all participants. Interface descriptions can be validated against the definitions as specified in the standard. The standard repository can be queried and hence reused out of the box. However, standards are not expected to change and are introduced once in a revolutionary way. There is no need for an adaptation layer (see Fig. 2.3).

Example Systems: Industrial automation systems currently rely on OPC UA in combination with the domain standard ecl@ss [4]. For example, the ID 0173-1#02-AAO735#003 refers to the supplier name of a gauze dissecting swab or sponge (see Fig. 3.4). Scientifically driven solutions mostly rely on ontologies to model the domain (e.g., Linked Open Data Vocabularies

3. Methods to Achieve Semantic Interoperability in IoT

for the Internet of Things [15]).

Although top-down engineering methods to achieve semantic interoperability allow for automated plug and play scenarios for IoT applications, the downside of this approach is the formalization effort needed upfront. Each identifier has to be linked to a machine-understandable standard at design time. What is worse, an ontology is a formal (i.e., machine-readable), explicit specification of a shared conceptualization (i.e., domain knowledge). Shared is negative in this sense, as it refers to agreed upon by an exclusive group. The agreement must be reached before any application can be implemented if semantic interoperability is a requirement. However, this assumption contradicts the fact that millions of IoT devices are currently being connected.

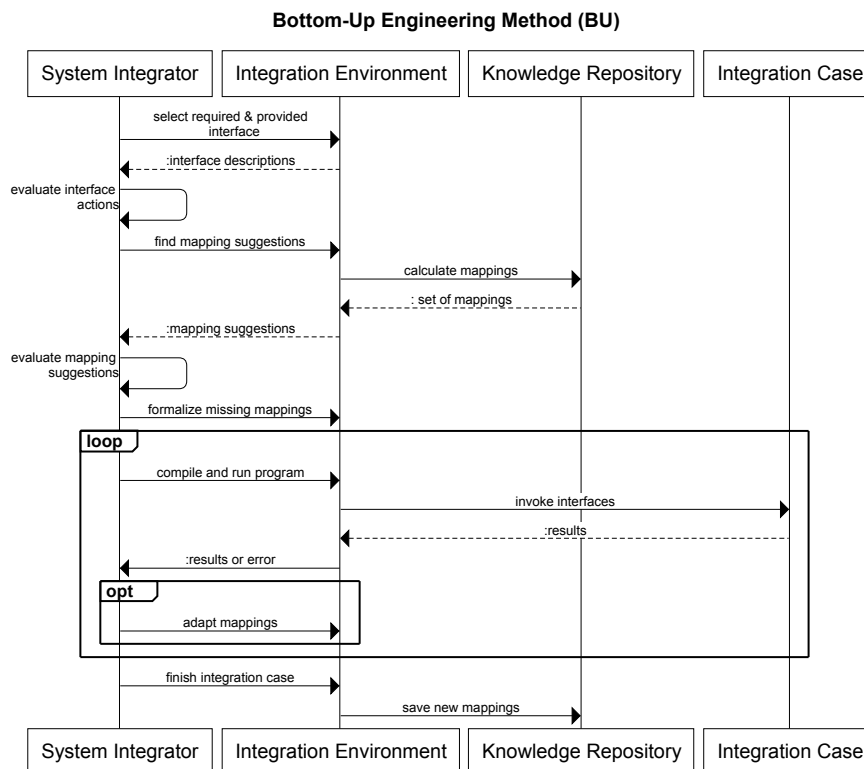


Figure 3.5.: Bottom-Up Engineering Method

3.3. Bottom-Up Engineering Methods

Innovative use cases emerge incrementally over time. This makes it hard for interested group members to come up with a shared conceptualization across company boundaries. However, device providers, platform owners, and application designers must cooperate in order to achieve interoperable solutions.

In contrast to top-down engineering methods, bottom-up engineering methods (BU) soften the claim for describing domain models completely as a requirement. This means that the domain

3. Methods to Achieve Semantic Interoperability in IoT

model is built up based on currently available software interfaces. A concrete set, if required, and provided interfaces is always given. Hence, integration shifts from a star-like integration graph (e.g., everyone references an ontology) to a chain-like integration graph (e.g., another interface that can be replaced by another interface that can again be replaced). Bottom-up favors an evolutionary creation of a domain model during system design time over a revolutionary domain model control.

Bottom-up engineering methods address the research challenge to automatically and dynamically compose IoT services by including an adaptation mechanism that can reconfigure delivery of services when the context changes [1]. This means that the software architect does not specify the meaning of software interface elements based on predefined requirements. Consequently, the *describing* context does not exist in these methods (see Fig. 3.3).

Community standards such as `schema.org` [50] or domain ontologies from the Semantic Web of Things [15] allow for a dynamic ecosystem evolution. However, such standards also suffer from a timeliness problem as more and more IoT devices emerge everywhere. Hence, the dynamically evolving domain model S may not be reusable by others, as it does not contain all required domain elements by C . Furthermore, there may exist no quality gate for new knowledge.

This has the following implications on the underlying engineering method (see Fig. 3.5). The integration environment typically involves a tool that can query the knowledge repository given the required and provided interfaces of the integration case. The resulting mapping suggestions can complete or incomplete. They are complete if the same integration case is implemented again. If an unknown or adapted interface is included in the integration case, changes are necessary within all used descriptions to reflect interface mappings correctly.

In this vein, Bennaceur et al. [51] proposes to use semantic web service descriptions that are linked to an ontology. This ontology is controlled by the system integrator and distinct from the current integration context. Furthermore, they provide a labeled transition system to describe the order of invocation of services. Their solution proposal can generate a software adapter at run time that is correct by construction based on these input parameters. Regarding the interface definition language L (see Def. 1), they formalize a mapping function M_S between the web service description and an intermediate ontology, which represents the semantic domain S .

To illustrate this setup, we reuse their own running example. In their running example, the operation *findTrip(...)* offered by the US Travel Agency system should be coupled to the interface EU Travel Agency (see Fig. 3.6). To fix each interface element's meaning, they link input and output data contained in both interface descriptions to their ontology (upper part of the figure). Thereby they integrate both interfaces on the semantic level. Furthermore, they specify that *findTrip(...)* as a required interface involves calling *selectCar(...)*, *selectFlight(...)*, *selectHotel(...)* and *makeReservation(...)* in that order as a labeled transition system (see Listing 3.1).

Listing 3.1: Labeled Transition System Example

```
SFI = ( findTripI -> selectFlightI -> END ).
SHI = ( findTripI -> selectHotelI -> END ).
SCI = ( findTripI -> selectCarI -> END ).
|| P1 = ( SFI || SHI || SCI ).
```

3. Methods to Achieve Semantic Interoperability in IoT

```
SFO = (selectFlightO -> findTripO ->END).
SHO = (selectHotelO -> findTripO ->END).
SCO = (selectCarO -> findTripO ->END).
|| P2 = (SFO || SHO || SCO).

P3 = (makeReservationI -> confirmTripI -> END).
P4 = (confirmTripO -> makeReservationO -> END).
P5 = (selectFlightO -> selectCarI -> END).
|| Onto = (P1 || P2 || P3 || P4 || P5).

US = (findTripI -> findTripO -> confirmTripI -> confirmTripO -> END).
EUF = (selectFlightI -> selectFlightO -> makeReservationI ->
      makeReservationO -> END).
EUH = (selectHotelI -> selectHotelO -> makeReservationI ->
      makeReservationO -> END).
EUC = (selectCarI -> selectCarO -> makeReservationI ->
      makeReservationO -> END).
|| EU = (EUF || EUH || EUC).
|| System = (EU || US || Onto).
```

Their ontology only captures the present integration case within a domain. Regarding our running example, this results in the following integration activities:

Integration Example for Formal Methods: Similar to the SA method, the System Integrator *artMode* maps to *power_saving*. Besides, the simple mapping function *replace* for required and provided keys are formalized in a declarative way and are stored in the Knowledge Repository for future reuse. In the future, the system integrator can query the Knowledge Repository based on required and provided interface descriptions, and an adapter is generated automatically. Hence, agile development teams continue their current state-of-practice by implementing a software adapter. Although they must invest additional mapping formalization effort in the short term, the reuse and reasoning capability speeds up their work in the long term.

Integration Knowledge Management: Integration knowledge is assumed to be complete once defined based on the set of concrete integration cases at hand. Adding or updating an interface typically requires updating the formal domain model as well as required transformation functions. Hence, the formalization effort in lines of code is moderate-high. It is moderate if the tooling infrastructure allows generating a software adapter automatically. It is high if mappings are only suggested but cannot be directly used within an executable software adapter. The amount of time to create these lines of codes is hours to days as it involves working with languages that are not applied by the system integrator on a regular basis. For instance, editing an OWL ontology or specifying labeled transitions systems for controlling the service invocation order is typically an expert skill. Integration knowledge can be queried and hence reused out of the box. Integration knowledge can evolve by updating the formal domain model and all accompanying integration specifications. In addition, the adapter code itself may be subject to manual adaptations based on the suggested mappings. The final software adapter is deployed in an adaptation layer (see Fig. 2.3).

3. Methods to Achieve Semantic Interoperability in IoT

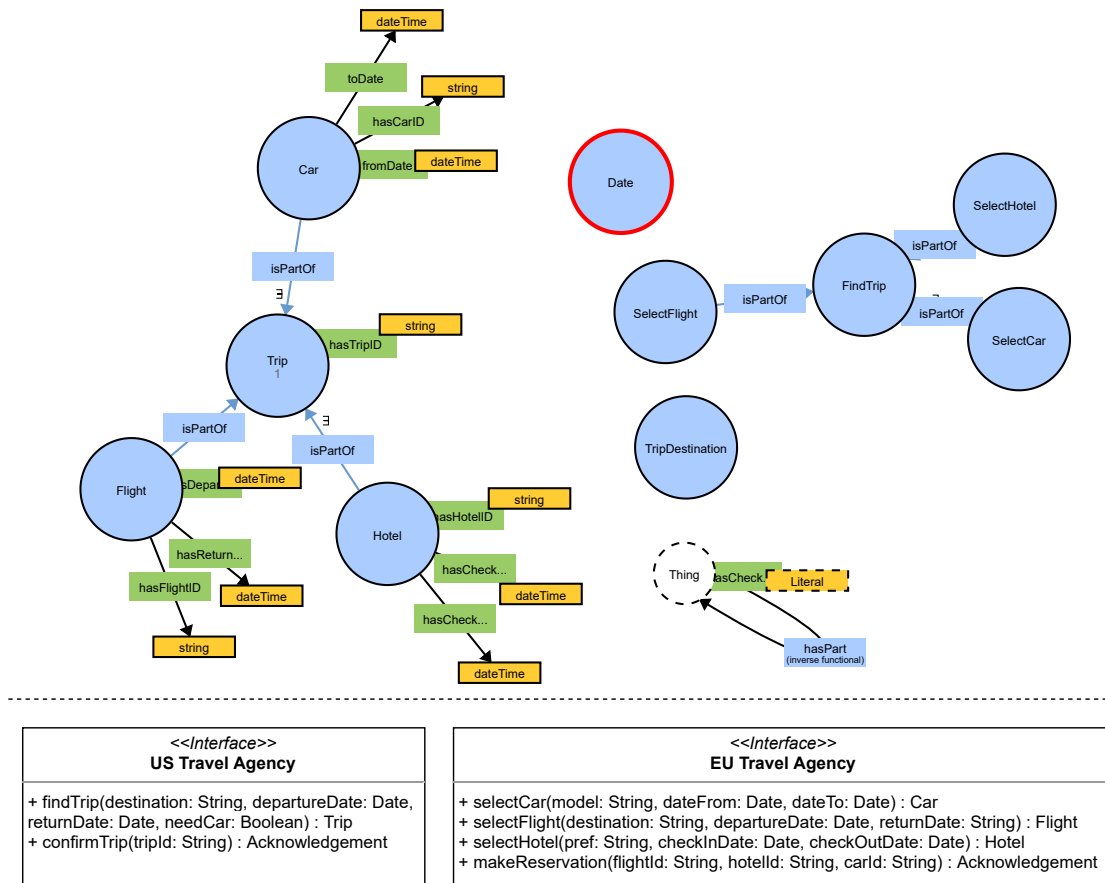


Figure 3.6.: Example - Semantic Integration based on Bennaceur et al. [7]

Example Systems: Defining interface mappings and transformation using a declarative language is currently widely done in practice. For example, flow-based programming environments such as NODE-RED¹ support the declarative mapping language JSONata [42]. Other examples are ioBroker² from the smart home domain and OpenIntegrationHub³ for data synchronization between business applications. However, these systems cannot share or reason in mappings. In contrast to scientific solutions such as Bennaceur et al. [51], these industrial systems suffer from the same problem as the software adapter implementation method. They are formalized in a repetitive way.

Like top-down engineering methods, current bottom-up engineering methods aim to achieve semantic interoperability in an automated way. However, bottom-up engineering methods suffer from the fact that the system integrator must perform more work compared to software adapter implementation.

¹<https://nodered.org/>

²<https://www.iobroker.net/>

³<https://www.openintegrationhub.org/>

3. Methods to Achieve Semantic Interoperability in IoT

On the one hand, we need a working software adapter. On the other hand, the integration knowledge must be efficiently formalized, so reuse is likely.

4. Requirements for Semantic Service Interoperability in IoT

We have seen that achieving semantic interoperability for IoT systems is still a challenge. Now, we relate the presented methods to our barriers. Therefore, we have summarized all the mentioned characteristics in Table 4.1. Although software adapter implementation does not expose the additional effort to control a shared domain model, software adapters are repetitively implemented, and the contained integration knowledge can not be reasoned about effectively. Top-down engineering methods rely on a predefined, complete domain model. However, such a domain model is static and cannot be changed easily as all participants rely on it. TD cannot cope with fast innovation cycles. Nevertheless, change is omnipresent in dynamic IoT scenarios. Bottom-up engineering methods, similar to software adapter implementation, do not rely on a common domain model but focus on concrete integration cases. Although these approaches do not attempt to formalize every possible detail of a domain, they also assume that their domain model is complete once defined. The main difference to standards is that they build up their domain model and the necessary transformation activities based on a set of concrete integration cases. Despite the fact that formalization effort is less, the formalization and time effort to adapt these formal models is not manageable over time and therefore not widely applied in practice.

Table 4.1.: Comparison of Methods to Achieve Semantic Interoperability in IoT

Method	Model		Formalization Effort	Time Effort	Evolvable	Adaptation Layer
	Incomplete	Complete				
SA	x		low	hours	yes	yes
TD		x	none	years	no	no
BU		x	moderate-high	hours-day	yes	yes

In the next chapter, we present our solution approach that is also based on the bottom-up principle. In contrast to existing bottom-up approaches, we explicitly allow for incomplete integration knowledge *at all time*. As a consequence, formalization effort for integration context becomes manageable as mappings are formalized incrementally. We still strive for automated software adapter generation for seen and unseen integration cases as we can query and reason about incomplete integration knowledge. Over time, this results in formalizing fewer mappings for similar integration cases. Still, the method and the tooling infrastructure should be executable without having expert knowledge in ontology engineering.

Part III.

Knowledge-driven Architecture Composition

5. Formalization

5.1. Basics

We begin this chapter by introducing the basic concepts using algebraic syntax. Therefore, we reuse the fundamentals of the theoretical system model from Bennaceur et al. [51]. Their component model is made up of capabilities, interfaces and actions. Later, we use this formal model to illustrate how our reasoning approach assists the system integrator. In the following, we refer to abstract concepts written as \mathcal{X} and concrete concepts in normal text style.

Let $C = \{c_1, \dots, c_n\}$ be a set of software components. The capability $Cap = \langle type, F \rangle$ of a component is defined as:

- $type \in \{Req, Prov, ReqProv\}$ where the $type$ specifies if a component requires or provides functionality to its environment. A component can also expose both types at the same time.
- F gives the semantics of the functionality by referring to concepts in a domain \mathcal{D} . For such a domain \mathcal{D} , an ontology \mathcal{O} explicitly models knowledge about the specific domain for all relevant provided and required functionality.

Capability can be interpreted as a service profile. A service profile has a name, a textual description, and it refers to concepts in the ontology \mathcal{O} . The service behaviour of a service describes how the capability can be achieved. For instance, a service process states the sequence of possible action invocations. An action involves an operation, input data, and output data. For example, a software component's actions are described both syntactically, using JSON schema, and semantically using ontological annotations. An action α is formally defined as $\alpha = \langle op, i, o \rangle (op, i, o \in \mathcal{O})$. Here an operation op is invoked by providing the appropriate input data i and consuming the produced output data o . The set of all actions that belong to a component is referred to as its interface \mathcal{I} . Each component only has one interface. The annotations of each component interface $\mathcal{I}(c)$ refer to a single ontology \mathcal{O} specific to the domain \mathcal{D} that the component belongs to.

In order to distinguish between the ontological concepts and the syntax of the input and output data (i, o) , we reuse the notion of a formal language L [32] from the previous chapter.

Definition 1 (Interface Description Language).

An interface description language L is defined as $L = (\mathcal{C}, \mathcal{A}, \mathcal{D}, M_{\mathcal{C}}, M_{\mathcal{D}})$ where \mathcal{C} is the concrete interface syntax, \mathcal{A} is the abstract syntax, and \mathcal{D} defines the semantic domain. $M_{\mathcal{C}}: \mathcal{C} \rightarrow \mathcal{A}$ is a function that assigns graphical or textual syntax elements $c \in \mathcal{C}$ to one abstract syntax element $a \in \mathcal{A}$ and $M_{\mathcal{D}}: \mathcal{A} \rightarrow \mathcal{D}$ relates abstract syntax elements $a \in \mathcal{A}$ to semantic domain elements $d \in \mathcal{D}$.

5. Formalization

In our running example (see Fig. 1.3), the concrete syntax refers to the JSON syntax, and the abstract syntax provides a higher level description from the concrete syntax in use. The relation of M_C realizes this. The semantic domain \mathcal{D} is associated with an ontology \mathcal{O} that models the knowledge about the specific domain. Consequently, the function $M_{\mathcal{D}}$ contains all annotations to refer to their ontology concept's syntactic elements. We denote the concrete syntax of input $\mathcal{C}(i)$ and output $\mathcal{C}(o)$ respectively.

Next we need the notion of a client. In our running example (see Fig. 1.3), the client is a remote application that runs on a mobile device. At design time of this application, the requirements stated that it should only support the Samsung TV. Hence, the client builds up the requests and handles responses as defined by the corresponding service description. In this case, the client requires the functionality defined by the Samsung TV.

Definition 2 (Required/Provided Interface).

When a client only functions by accessing actions $\{\alpha_1, \dots, \alpha_n\}$ from third party components, we describe such actions as *required* actions. In contrast, a *provided* action $\bar{\alpha}$ is an action that another client can access. Hence, the actions exposed by a third party component are provided to the environment.

Regarding the introduced notation, we state that provided actions or interfaces are equipped with an over line and required actions or interfaces are not.

Our next definition states how a capability Cap can be described within a semantic domain \mathcal{D} . Therefore, we specify the expressiveness of an ontology \mathcal{O} and introduce a labeled transition system. A labeled transition system is needed to capture the sequence of action invocations (i.e., service's behaviour).

Definition 3 (Semantic Domain).

We define the structural characteristics of the semantic domain \mathcal{D} using a description language that supports $SHOLN(\mathcal{D})$ expressiveness. We define the behavior of software components within a semantic domain using labeled transition systems. A labeled transition system LTS is defined by a quadruple $LTS = (Q, A, \rightarrow, Q_0)$ where Q is the nonempty, countable set of states, A is the set of labels (or actions), $\rightarrow \subseteq Q \times A \times Q$ the transition relation and $Q_0 \subseteq Q$ is the set of starting states. Q contains interface descriptions using L for required and provided interfaces. Q_0 contains interface descriptions using L for required interfaces. \rightarrow contains preconditions before an operation can be called.

The described concepts within the resulting ontology \mathcal{O} are used to relate input i and output data o using annotations. The order of invocations of operations (e.g., due to different granularity) is described using the labeled transition systems.

Given this formal model, functionally compatible components can be identified. Therefore, we assume that there exist an ontology \mathcal{O} that is used by the required as well as by the provided action (see Fig. 3.6). We say that a required capability $Cap_R = \langle Req, Op_R \rangle$ can be mapped to a provided capability $Cap_P = \langle Prov, Op_P \rangle$ iff $Op_P \sqsubseteq Op_R$ in the ontology of domain \mathcal{D} . This means that the concepts used by Op_P subsume the concepts used by Op_R . The idea behind that statement is that the required functionality is less demanding than the provided functionality.

The semantic domain \mathcal{D} can be described implicitly or explicitly. If it is described explicitly, it

5. Formalization

must be kept up to date and be integrated into the component development process. If it is not described explicitly, the actions described by the components interface must be interpreted by the person who maps required to provided actions.

Although we reuse the formal component model provided by Bennaceur et al. [7], we fundamentally differ in the way the formal domain model is built. Bennaceur et al. and all other approaches found assume that the semantic domain is given at a specific point in time. However, this is not the case for IoT systems. Hence, our approach provides a structured way to gain and formalize this domain knowledge incrementally. By doing so, the formalization effort (see Barriers) becomes manageable.

Now we return to the three general solution proposals software adapter implementation, top-down, and bottom-up interoperability approaches (see section Background II). All proposals try to establish semantic correspondences between provided and required interfaces [3, 1]. Furthermore, semantic interoperability ensures that services and data exchanges between a provided and a required interface make sense – that the requester and the provider have a common understanding of the "meaning" of services and data [2]. Hence, we can define the search for such a mapping between required and provided interfaces. Let \mathcal{I}_1 be the required interface by component c_1 and let \mathcal{I}_2 be the provided interfaces by component c_2 . Then, the search for mapping pairs (X_1, X_2) where $X_1 = \langle \alpha_1, \alpha_2, \dots, \alpha_n \rangle, \alpha_{i=1..n} \in \mathcal{I}_1$ and $X_2 = \langle \bar{\beta}_1, \bar{\beta}_2, \dots, \bar{\beta}_m \rangle, \bar{\beta}_{j=1..m} \in \bar{\mathcal{I}}$ such that X_1 can be mapped to X_2 is defined by the function $X_1 \mapsto X_2$. This means that the required actions of X_1 can be performed by calling the provided actions of X_2 . The relation $Map(\mathcal{I}_1, \mathcal{I}_2)$ can now be defined as follows:

Definition 4 (Mappings).

$$\begin{aligned}
 Map(\mathcal{I}_1, \mathcal{I}_2) = \\
 \{ (X_1, X_2) \mid X_1 = \langle \alpha_1, \alpha_2, \dots, \alpha_n \rangle, \alpha_{i=1..n} \in \mathcal{I}_1 \\
 \wedge X_2 = \langle \bar{\beta}_1, \bar{\beta}_2, \dots, \bar{\beta}_m \rangle, \bar{\beta}_{j=1..m} \in \mathcal{I}_2 \\
 \wedge X_1 \mapsto X_2 \}
 \end{aligned} \tag{5.1}$$

Next we turn towards formalizing different mapping types. Depending on the set of required actions that should be mapped to a set of provided actions, these cases are one-to-one, one-to-many, and many-to-many mappings.

5.2. Mapping Types

In Definition 4 for mappings, we have stated that we link a textual interface description element from the required interface and a textual interface description element from the provided interface with one concept $d \in \mathcal{D}$. In contrast to Bennaceur's et al. component model [51], we do not assume that integration cases are complete and do not change over time. Furthermore, our approach differs as mappings are defined between interfaces without a common, underlying ontology. Hence, an action $\alpha = \langle op, i, o \rangle (op, i, o \in \mathcal{O})$ is not related to \mathcal{O} but only to \mathcal{D} resulting in an action defined as $\alpha = \langle op, i, o \rangle (op, i, o \in \mathcal{D})$. This renders annotations as defined

5. Formalization

in the function $M_{\mathcal{D}}$ from Definition 1 impossible. However, when searching for mappings (see Def. 4), we can relate required actions to the ontological concepts as described by the provided actions.

Let an interface \mathcal{I} be described by the set of syntactic elements $c \in \mathcal{C}$ of the interface description language L . Now, the function $M_{\mathcal{D}}$ can be rewritten as $M_{\mathcal{D}}: M_{\mathcal{C}}(\mathcal{I}_1) \rightarrow M_{\mathcal{D}}(M_{\mathcal{C}}(\mathcal{I}_2))$. This means, that concrete ontology \mathcal{O} is replaced with the domain \mathcal{D} that is imagined by the system integrator by interpreting the provided interface description. Hence, the search for mappings is explicitly formalized by relating the concrete syntax $\mathcal{C}(\mathcal{I}_1)$ of a required interface to the concrete syntax of a provided interface \mathcal{I}_2 .

A required action $\alpha = \langle a, I_a, O_a \rangle \in \mathcal{I}_1$ can be mapped to a provided action $\bar{\beta} = \langle b, I_b, O_b \rangle \in \bar{\mathcal{I}}_2$ if the following requirements are met:

Definition 5 (One-to-One Mapping).

1. $b \sqsubseteq a$
2. $I_a \sqsubseteq I_b$
3. $I_a \sqcup O_b \sqsubseteq O_a$

This definition's general idea is that a provided operation can achieve a required operation if the required operation a is less demanding than the provided operation b . Furthermore, all required input data I_b for the provided action $\bar{\beta}$ can be deduced from the provided input data I_a for the required action α . Finally, the required output data O_a can be deduced by combining O_b and I_a . A one-to-many mapping extends this general idea. Again, let $\alpha = \langle a, I_a, O_a \rangle \in \mathcal{I}_1$ be the only required action. Now, a mapping to a sequence of provided actions $X_2 = \langle \bar{\beta}_1, \bar{\beta}_2, \dots, \bar{\beta}_m \rangle, \bar{\beta}_{j=1..m} \in \mathcal{I}_2$ must be achieved. Such a mappings exists if the following requirements are met:

Definition 6 (One-to-Many Mapping).

1. $\bigsqcup_{1 \leq j \leq m} b_j \sqsubseteq a$
2. $I_a \sqsubseteq I_{b_1}$
3. $I_a \sqcup \left(\bigsqcup_{1 \leq l \leq j-1} O_{b_l} \right) \sqsubseteq I_{b_j}$
4. $I_a \sqcup \left(\bigsqcup_{1 \leq l \leq m} O_{b_l} \right) \sqsubseteq O_a$

The first statement means that a subset of all provided operations b_j is subsumed by the required action α . The second statement means that the sequence of provided actions can be initiated since the input data of the first action I_{b_1} can be obtained from the input data of the required action I_a . The third statement means that the provided input data I_a subsequently merged by all outputs O_{b_l} finally allows calling all provided interfaces. This is necessary, as an action can

5. Formalization

only be executed if all required input data is present. The last statement ensures that the required output data O_a can be obtained from the set of generated output data by executing all provided actions.

As last type, the many-to-many mapping case is introduced. Let $X_1 = \{\alpha_1, \alpha_2, \dots, \alpha_n\}, \alpha_{i=1..n} \in \mathcal{I}_1$ be sequence of required actions and let $X_2 = \{\beta_1, \beta_2, \dots, \beta_m\}, \beta_{j=1..m} \in \mathcal{I}_2$ be the sequence of provided actions. Then a many-to-many mappings exists if:

Definition 7 (Many-to-Many Mapping).

1. $\bigsqcup_{1 \leq j \leq m} b_j \sqsubseteq \bigsqcup_{1 \leq i \leq n} a_i$
2. $\bigsqcup_{1 \leq i \leq l} I_{a_i} \sqsubseteq I_{b_1}$
3. $\left(\bigsqcup_{1 \leq j \leq l} I_{a_j} \right) \sqcup \left(\bigsqcup_{1 \leq h \leq j-1} O_{b_h} \right) \sqsubseteq I_{b_i}$
4. $\forall h \in [1, l[, O_{a_h} = \emptyset$
5. $\forall h \in [l, m], \left(\bigsqcup_{1 \leq i \leq h} I_{a_i} \right) \sqcup \left(\bigsqcup_{1 \leq k \leq n} O_{b_k} \right) \sqsubseteq O_{b_k}$

The first statement says that provided operations can perform all required operations. The second statement states that the execution of provided actions can be initiated if the necessary input data I_{b_j} can be computed based on the data previously received. The third statement states that all provided input parameters by the required operations merged with all output parameters produced by executing the provided operations sequentially allow to call all provided operations meaningfully.

Similar to Bennaceur et al., we assume synchronous invocation semantics. This means that a required action can only be called if its output is available, and analogously a provided action can be executed only if its input is available. However, we can accumulate the required actions' data and allow them to progress if they do not require any output. Hence, the fourth statement specifies that the first $l-1$ actions do not require any output and can be executed before the provided actions. The last statement specifies that all required outputs can be finally generated by merging all outputs generated by the provided operations.

This case would shift a lot of application logic into a software adapter. This is often not desired in reality, as it makes it hard to maintain such complex adapters.

We continue with our own model (i.e., own extensions to Bennaceur et al. [51] anymore) as we shift to the integration knowledge management process for knowledge-driven architecture composition. Here we especially focus on the knowledge reasoning aspect.

Throughout the definitions so far, we have applied the *Map* function that changes the syntactic representation of an identifier by preserving its meaning (e.g., "sourceName" is replaced

5. Formalization

Semantic Syntax	Equal	Not Equal
Equal	Case 1 (e.g. Formal Standard)	Case 2 (e.g. Homographs)
Not Equal	Case 3 (e.g. Synonyms)	Case 4 (i.e. incompatible)

Figure 5.1.: Integration Cases

by "input" in the running example 1.3). In this integration context, these two identifiers are homonyms. However, there are also integration cases on the identifier layer that require more complex transformations. Generally, we distinguish between four categories, as presented in Figure 5.1. Therefore, we can apply these categories not only on two identifiers but on all interface elements. With regard to the mapping definition 4, a set of required and a set of provided interfaces are functionally compatible if the function $M_{\mathcal{D}}$ relates operations to the same domain concepts $d \in \mathcal{D}$ and if the requirements from one mapping type can be applied. This scenario assumes that each identifier has a fixed meaning defined by its annotation. To enforce this distinct meaning, the underlying ontology \mathcal{O} must be available before the definition of a component interface (see Case 1 in Figure 5.1). Hence, no integration activities on the semantic layer are necessary for top-down approaches that rely on this assumption. Naturally, no integration activities occur if interfaces do not share common concepts as stated in Case 4 (e.g., two operations called "startEngine" and "openMailbox").

Cases 2 and 3 (see Fig. 5.1) can only occur if we use no common, centrally managed ontology. Here bottom-up integration approaches formalize interface mappings not based on a known standard but on what is available in a concrete integration context (i.e., driven by the interfaces that should be integrated). As seen in the *Background II* chapter, Bennaceur et al. [51] 1) build up an own ontology that suits their integration case, 2) link all actions to this ontology (i.e., $M_{\mathcal{D}}$ application to the newly created ontology) and 3) provide an LTS to describe the behavior of the components (see Definition 3). However, such bottom-up approaches allow for automated integration only if decentralized developers opt in for the same ontology and build up their operations so we can match them according to their behavioral specifications. Consequently, we need a modified definition for integration knowledge and incomplete mappings for our approach as our integration knowledge is always incomplete. Now we want to build up interface mappings incrementally without an intermediate \mathcal{O} .

Let $L = (\mathcal{C}, A, \mathcal{D}, M_{\mathcal{C}}, M_{\mathcal{D}})$ be the interface description language that is used by a required interface \mathcal{I}_1 and a provided interface \mathcal{I}_2 to describe their actions. Let $L^* = (\mathcal{C}^*, A^*, S^*, M_{\mathcal{C}}^*, M_{\mathcal{S}}^*)$ be a language that describes the relation between the required and provided interface. Hence, annotations from the syntax of actions to the ontology concepts are not part of the interface description anymore (i.e., $M_{\mathcal{D}}$). Therefore, we need an expressive language for annotations as the ontology concepts are not defined once they are needed. For instance, mathematical operations may be needed to transform input and output data. Based on L^* , the function $Map_{IntegrationCase}(\mathcal{I}_1, \mathcal{I}_2)$ can be defined as follows:

5. Formalization

Definition 8 (Case-based Mappings).

1. $\mathcal{D}_1 \sqcup \mathcal{D}_2 \neq \emptyset \wedge \mathcal{D}_1, \mathcal{D}_2 \in \mathcal{D}$
2. $M_{\mathcal{D}}(\mathcal{I}_1) : M_{\mathcal{C}}(\mathcal{I}_1) \rightarrow \mathcal{D}_1$
3. $M_{\mathcal{D}}(\overline{\mathcal{I}_2}) : M_{\mathcal{C}}(\overline{\mathcal{I}_2}) \rightarrow \mathcal{D}_2$
4. $Map_{IntegrationCase} : C^*(M_{\mathcal{C}}(\mathcal{I}_1) \rightarrow M_{\mathcal{C}}(\overline{\mathcal{I}_2})) \rightarrow M_{\mathcal{D}}(\overline{\mathcal{I}_2})$

In contrast to the definitions 4, the fourth statement gets rid of the intermediate ontology. Here C^* describes the concrete syntax of a mapping function that relates a set of required actions to the provided actions' domain concepts. Mappings (c.f., annotations) are not related from the required interface to the ontology and then to the provided interface but are related *directly* from the required interface to the provided interface. Of course, the intersection of the interpreted domain concepts from the required and provided actions must not be empty (see integration case 2 and 3 from Fig. 5.1). Otherwise, they would not be functionally compatible.

As a consequence of this case-based mapping formalization, the relation of domain concepts can be built up incrementally. Hence, domain concepts are never complete for all integration cases as a new interface can be added at any time. Here the key idea is to store integration knowledge between required and provided actions not in a star-like manner but as a chain of integration cases with *no fixed* intermediate ontology.

A requirement for applying reasoning principles on such chains is a graph structure where mappings are stored.

Let $Y = \{\mathcal{I}_1, \dots, \mathcal{I}_n\}$ be a set of interfaces. These interfaces can either provide actions or require actions based on an integration case. We can then define an incomplete, domain-specific mapping:

Definition 9 (Incomplete, Case-based Mappings).

Let $KB = (Y, Map_{IntegrationCase})$ be a knowledge base with a multi graph structure. A vertex v consists of an interface $y_i \in Y$ with actions $\{\alpha_1, \alpha_2, \dots, \alpha_n\}$. An edge $Map_{IntegrationCase}$ between required y_i and provided actions \bar{y}_j is inserted for each mapping type (e.g., one-to-one, one-to-many,..). Such mappings are incomplete, as only the needed mapping functions for the integration case at hand are formalized.

For example, assume that there are three interfaces $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3 \in Y$ where \mathcal{I}_1 has actions $\alpha_1, \dots, \alpha_3$, \mathcal{I}_2 has actions $\alpha_4, \dots, \alpha_6$ and \mathcal{I}_3 has actions $\alpha_7, \dots, \alpha_9$. Furthermore, assume the following functional compatibilities indicated by "x" between these actions exist (multiple "x" in a line indicate a one-to-many mapping).

Let us further assume an "x" includes all mappings for input and output between required and provided actions. Then we can state that the domain \mathcal{D} is completely described if all possible mappings are available in the knowledge base KB (see Table 5.1). Consequently, there is an ontology \mathcal{O} for the domain \mathcal{D} and a behavioral specification as an *LTS* for all possible invocation sequences (see definition 3). Let us assume the knowledge base is empty. An integration case then represents a subset of all "x"s. Further assume, that the integration case at hand (e.g.,

5. Formalization

	$\overline{\alpha_1}$	$\overline{\alpha_2}$	$\overline{\alpha_3}$	$\overline{\alpha_4}$	$\overline{\alpha_5}$	$\overline{\alpha_6}$	$\overline{\alpha_7}$	$\overline{\alpha_8}$	$\overline{\alpha_9}$
α_1				x					x
α_2					x	x			
α_3					x			x	x
α_4	x						x		
α_5		x					x	x	x
α_6			x				x		
α_7				x					
α_8		x		x		x			
α_9	x			x	x				

Table 5.1.: Complete Integration Knowledge for a Domain

replace Samsung with LG "/status" operation from our running example in Fig. 1.3), only needs mappings from α_1 to $\overline{\alpha_4}$. We store the corresponding mappings in the knowledge base using the language L^* . Hence, we can state that the mappings for the domain are incomplete and thus, the knowledge base is incomplete as well.

If the knowledge base is incomplete for the KDAC approach, then no automated integration is possible as integration knowledge is missing.

Up to now, mappings between actions have to be defined manually. To tackle this problem, we use reasoning principles to assist the system integrator in generating mappings automatically.

5.3. Reasoning Principles for Mappings

The mapping function $Map_{IntegrationCase} : C^*(M_C(\mathcal{I}_1) \rightarrow M_C(\overline{\mathcal{I}}_2)) \rightarrow M_{\mathcal{D}}(\overline{\mathcal{I}}_2)$ takes a required interface description \mathcal{I}_1 and a provided interface description $\overline{\mathcal{I}}_2$ as inputs. A mapping function's domain (see Def. 8) are all required actions $\alpha = \langle op, i, o \rangle (op, i, o \in \mathcal{D})$. Consequently, the image of a mapping function are all provided actions. The interfaces $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3$ define the functional interoperability between α_1 to α_4 and α_4 to α_7 with reasoning support. To reason about domains and images of mapping functions, we have to specify how a mapping function $Map_{IntegrationCase}$ is defined using the language L^* based on its concrete syntax C^* . Please note, that L^* refers to a purely technological language to define mappings with no relation to the application domain. L^* is applied to the input and output of an action. In fact, each input i and output o object can consist of multiple elements. Such elements are usually defined in a data model. This data model is unique for each interface description. Therefore, each interface description \mathcal{I} defines its distinct data model. This means that $Map_{IntegrationCase}$ actually defines a set of functions $\{map_{action_1}, map_{action_2}, \dots, map_{action_n}\}$ on the input and output data elements. Such a data element is uniquely recognized as an identifier id in the name space of the respective interface.

Let $data_1$ be the data model used by action α_1 , $data_4$ be the data model used by action α_4 and $data_7$ be the data model used by action α_7 . Now we can define the domain and the image of an operation.

5. Formalization

Definition 10 (Domain and Image of Case-based Mappings).

Each identifier $id \in data_1 \cup data_4 \cup data_7$ has a domain $DOM(id)$ and an image $IMG(id)$. The domain DOM and image IMG of a mapping function $map_{action} \in Map_{IntegrationCase}$ are restricted by the available input i and output o data of the respective operation op .

In the running example (see Fig. 1.3), we have stated that "sourceName" from the Samsung TV schema can be replaced with "input" from the LG TV. However, this can only be done when the identifier type can be transformed (e.g., from Integer to String) and the domain of definition (e.g., enumeration) of both identifiers is identical. If the domain of definition is not identical, the mapping function map_{action} must take care of the correct transformations. Whenever such a mapping function is defined, we can try to calculate its inverse function. This is our first reasoning principle.

Let $id_1 \in data_1, id_2 \in data_4$ be identifiers and $f : DOM(data_1) \rightarrow IMG(data_4)$ be the mapping function that relates data model $data_1$ to $data_4$. Let $r \subseteq id_1 \times id_2$ be the mapping function $r \subseteq f$ that relates identifier id_1 to id_2 .

Definition 11 (Inverse of Case-based Mappings).

An inverse mapping function r^{-1} for an identifier can be computed if r is a bijective function. Consequently, the inverse mapping function r^{-1} has the domain $DOM(data_4)$ and the image $IMG(data_1)$.

These mapping functions are inserted automatically into the knowledge base KB . Computed mappings based on the inverse property are complete for the inverse integration case if we can transform all identifiers from the provided action α_4 into the required action α_1 .

For the second reasoning principle, we have to return to our running example (see Fig. 1.3). Assume that there is another TV manufactured by Philips. Assume that the system integrator has examined two integration cases. The first one is integrating the Samsung TV with the LG TV, and the second is integrating the LG TV with the Philips TV. Then, the knowledge base KB contains the following integration knowledge:

- Vertexes: Interface descriptions $\mathcal{I}_{Samsung}, \mathcal{I}_{LG}$ and $\mathcal{I}_{Philips}$ where only the action "/status" and data models for input and output are defined.
- Edges: There is a one-to-one mapping from the required action $\alpha_{Samsung}$ to the provided action β_{LG} . Furthermore, there is a one-to-one mapping from the required action β_{LG} to the provided action $\gamma_{Philips}$.

Let $r \subseteq \alpha_{Samsung} \times \beta_{LG}$ and $s \subseteq \beta_{LG} \times \gamma_{Philips}$ be mappings on the respective actions provided by the interfaces.

Definition 12 (Composition of Case-based Operation Mappings). A composed mapping relation is defined as $S \circ R$ when $\{(\alpha_{\mathcal{I}_1}, \alpha_{\mathcal{I}_3}) \mid \exists \alpha_{\mathcal{I}_2} : (\alpha_{\mathcal{I}_1}, \alpha_{\mathcal{I}_2}) \in R \wedge (\alpha_{\mathcal{I}_2}, \alpha_{\mathcal{I}_3}) \in S\}$.

Derived mappings based on the composition property are complete for the integration case if we can transform all input and output data from the set of required actions from interface \mathcal{I}_1 into

5. Formalization

the set of provided actions from interface \mathcal{I}_3 . This definition can be applied to all mapping types.

The third reasoning principle is only applied to the mapping type one-to-many.

Assume the remote control application should control two other TVs (i.e., one-to-many mapping type). For instance, the request sent to the required interface $I_{Samsung}$ should be duplicated and sent to \mathcal{I}_{LG} and $\mathcal{I}_{Philips}$. Furthermore, the knowledge base KB contains the following integration knowledge:

- **Vertexes:** Interface descriptions $\mathcal{I}_{Samsung}, \mathcal{I}_{LG}$ and $\mathcal{I}_{Philips}$ where only the action "/status" and data models for input and output are defined.
- **Edges:** There is a one-to-one mapping from the action β_{LG} to the action $\gamma_{Philips}$

Now assume that a new one-to-many mapping from the required interface Samsung TV $\mathcal{I}_{Samsung}$ to the provided interfaces $\overline{\mathcal{I}}_{LG}$ and $\overline{\mathcal{I}}_{Philips}$ should be added.

Let $S \subseteq a_{\mathcal{I}_2} \times a_{\mathcal{I}_3}$ be the available mapping and let $data_1, data_2$ and $data_3$ be the respective data models for the interfaces $\mathcal{I}_1, \mathcal{I}_2$ and \mathcal{I}_3 . Then, after a new mapping function map_{action} for an identifier $id \in R \subseteq \alpha_{\mathcal{I}_1} \times \alpha_{\mathcal{I}_2}$ is added to the knowledge base KB , we can apply the following reasoning principle:

Definition 13 (Composition of Case-based Identifier Mappings).

Let $id_i \in data_1, id_j \in data_2$ and $id_k \in data_3$. Due to the new mapping $(id_i, id_j) \in R$, the mapping (id_i, id_k) can be calculated when a mapping $(id_j, id_k) \in R$ exists in the knowledge base KB .

The major difference between the second and third reasoning principle is the abstraction level. Def. 12 is applied on the action level and Def. 13 is applied on the data mode level. This can be done independently of the selected action as all actions of an interface relate to the same data model.

In the next section, we provide a sophisticated process description of how KDAC uses the reasoning principles and we lay out the details of the so far graphical representations.

6. Integration Knowledge Management

Similar to the presented integration methods in the Background section, knowledge-driven architecture composition (KDAC) aims to establish interoperability concerns as a *first-class* citizen in IoT development – just like test-driven software development revolutionized software engineering. Concerning the running example (see Fig. 1.3), it would be beneficial for the application developer to generate a software adapter automatically. Therefore, integration knowledge must be stored in a machine-understandable way and made publicly available. The additional formalization effort for interface mappings is motivated by a "give a little:receive a lot" mentality. For example, imagine that there already exists integration knowledge from the LG television to 5 other televisions. Now, the mapping from Samsung to LG allows not only the remote control application to speak to one but six other devices at once.

This may appear promising but must be put into context to the content presented so far. We do this by answering three leading questions that may have arisen until now.

1) *Why not just keep implementing software adapters manually when standards are not covering a use case and use machine-readable standards in all other cases?* There exists a dilemma between searching for a standard that supports a novel use case or building up an own domain ontology. Reusing existing solutions always inherits all possible domain elements because of the completeness claim. Using multiple standards for distinct use cases requires integration between standards. Then the amount of work spent on integrating standards increases, and the working hours spent only have a minimal influence on the use cases that should be supported. To avoid such a scenario, each IoT platform can define its individual domain model. Now the question arises whether the platform provider should provide means to implement software adapters in the platform environment or whether the device is implemented according to the individual platform model. In both cases, standards and software adapters, the human must reason about the mappings. However, this process is slow and error prone. Furthermore, the value added for customers to use their individual IoT application independent of the device manufacturer can only be achieved manually.

2) *So, how is KDAC different from just chaining software adapters stored in a code repository?* Search-based software engineering can be applied to software adapters [52, 53] or software adapters can be integrated into recommendation systems for software reuse [54, 55]. However, to identify chains between required and provided software interfaces, programming language independent information about its functionality must be created. Furthermore, the integration knowledge is trapped within the imperative code and cannot be reasoned about by other software components. Testing and debugging a chain of imperative software adapters would require stubs for each intermediate interface, and errors are hard to fix as wrong input data in another adapter may cause them. Hence, there is a need to store mappings in a declarative way so that reasoning algorithms can reuse existing mappings efficiently or create new mappings by means

6. Integration Knowledge Management

of rigorous reasoning rules.

3) Finally, how is KDAC different from just updating ontologies incrementally with new domain knowledge? Ontologies from symbolic artificial intelligence research have been applied widely by researchers from the software architecture or web service community, as ontologies are one way to solve semantic ambiguities. However, trained knowledge engineers primarily define ontologies for the purpose of semantic data integration in static information systems. Now, reusing these concepts in IoT systems is challenging, as constant innovation and system dynamism are omnipresent. Constantly updating ontologies would require a knowledge engineer or training the system integrator accordingly. Furthermore, ontologies were initially designed to express static domain-specific data relations but not to express preconditions and postconditions of services. Hence, KDAC relies on more radical thinking by eliminating ontologies from the infrastructure support completely. Consequently, there is a need for the use of a declarative language that can be applied by a system integrator quickly.

6.1. Integration Knowledge Management Process

As with most IoT-related communication, the messages being sent within an environment are mostly data-driven (i.e., following the HATEOS principle for REST services). Like most engineering approaches that aim to achieve semantic interoperability, KDAC also abstracts away from networking protocols (e.g., HTTP) and syntactic characteristics (e.g., JSON). Overall, the method aims at 1) to store mappings between two devices based on their interface definition using a declarative language and 2) to reason about these mappings. Finally, platform-specific software adapters are automatically generated based on derived mappings or by reusing stored mappings. The underlying tool support aims at finding all necessary mappings between a provided and required interface. If all mappings for one integration case are present, then a software adapter can be generated automatically.

During the component design phase at time $t=0$, the component provider and the requester can design their service interfaces without using an interface description language that contains semantic data annotations based on a machine-understandable domain model (see Fig. 6.1). Furthermore, an application is created which uses software component B.

Assume at time $t=1$, integration is necessary as a new device is present, and it should not change the IoT application logic. In addition to writing an individual software adapter to connect component model A to the component model of B, the system integrator adds declarative mappings between both actions and stores these mappings in a knowledge base KB (e.g., specifying $Map_{IntegrationCase}$) as defined in Def. 8). The following definitions and knowledge management activities are typically applied at this stage using the language L^* for mappings:

- Knowledge Capture and Representation: Mapping Types (Def. 5,6,7) and Case-based Mappings (Def. 8)
- Knowledge Sharing: Store all mappings in knowledge base

Over time, various other components are integrated in a decentralized setting and various physical places (i.e., shown by frames in Fig. 3), and new mappings are added to the central knowl-

6. Integration Knowledge Management

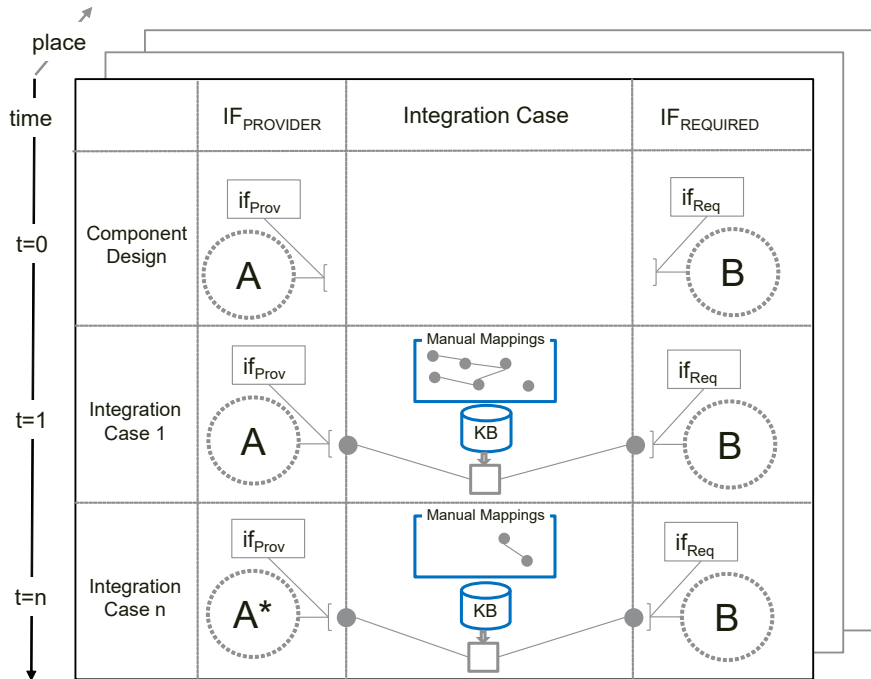


Figure 6.1.: Knowledge-Driven Architecture Composition Method

edge base. For example, software component A^* uses the same domain model as software component A . Hence, the stored mappings for component A can be reused. Now assume that there are no mappings between components A and A^* are available. Furthermore assume, that mappings between components A and C (component C are not shown in Fig. 6.1) and between component C to A^* exist. Because of the transitive relationship $A \leftrightarrow C \leftrightarrow A^*$, the mappings from component $A \leftrightarrow A^*$ can be calculated by applying the presented reasoning principles.

For example, a reasoning algorithm could infer new integration knowledge for the transitive characteristics for the identifier "volume" from the running example as a simple example. So, suppose the integration knowledge base contains the transformation functions $\{map_{action_1}, map_{action_2}, \dots, map_{action_n}\}$ from volume (SamsungTV) to volume (LGTV) and from volume (LGTV) to volume (PhillipsTV). In that case, the reasoning algorithm can directly infer the transformation function from volume (SamsungTV) to volume (PhillipsTV). The presented reasoning principles and the reuse of machine-understandable integration knowledge from previous integration cases can partially automate component composition (e.g., towards realizing a plug and play principle).

The following definitions are typically applied at this stage using the language L^* for mappings when integration knowledge is mostly incomplete:

- Knowledge Capture and Representation: Mapping Types (Def. 5,6,7) and Case-based Mappings (Def. 8)
- Knowledge Sharing: Store (missing) mappings in knowledge base

6. Integration Knowledge Management

- Knowledge Reuse: Query the knowledge base for mappings based on available interface descriptions
- Knowledge Reasoning: Inverse of Mappings (Def. 11) and Identifier Composition (Def. 13)

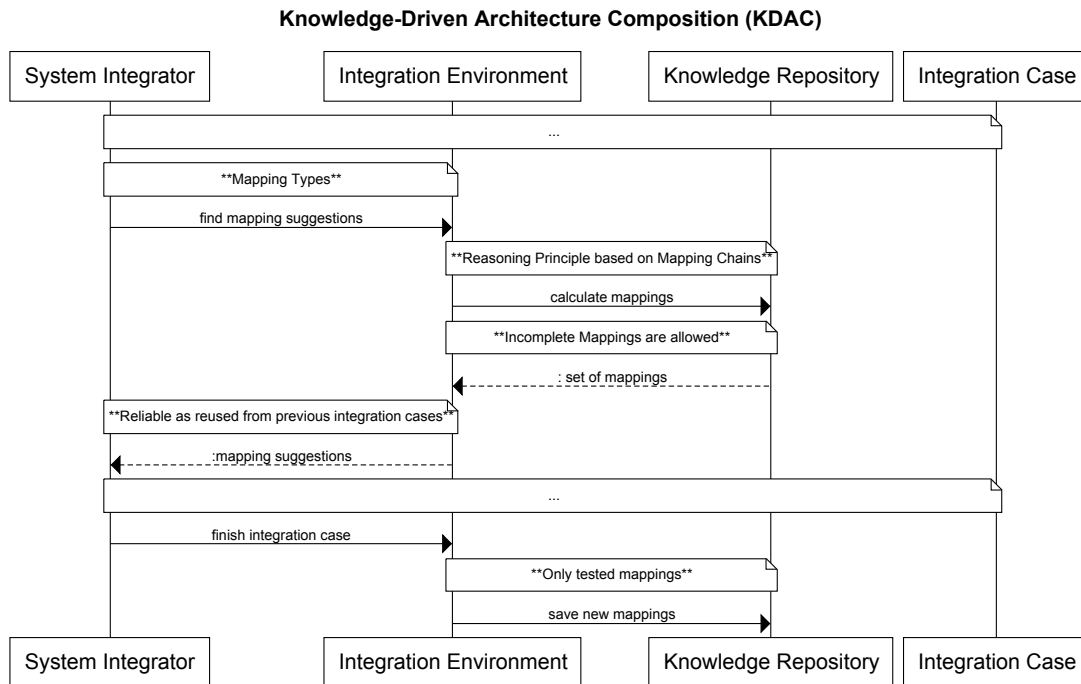


Figure 6.2.: Adapted Bottom-Up Engineering Method

At time $t=n$, only a few new mappings are required, which could ultimately result in fully automated component integration at run time by generating the required software adapter. Automated component coupling is achievable as soon as all functional action and data characteristics are present and/or can be deduced using the knowledge base. One can think of the reasoning process over integration knowledge as playing the game Sudoku: As soon as enough integration knowledge for an unseen integration case is present, we can calculate the missing information based on domain-specific rules (e.g., realizing a plug and play principle). The following definitions are typically applied at this stage using the language L^* for mappings when integration knowledge is mostly complete:

- Knowledge Reuse: Query the knowledge base for mappings based on available interface descriptions
- Knowledge Reasoning: Inverse of Mappings (Def. 11), Identifier Composition (Def. 13) and Operation Composition (Def. 12)

KDAC itself is a bottom-up engineering method. Therefore, the aforementioned formal description can be related to the bottom-up engineering method illustration from section 3.3. At this point of writing, we adapt the illustration for BU approaches in the following way (see Fig. 6.2). When searching for mapping suggestions, the mapping types must be specified. When calculating mappings, the presented reasoning principles based on available mapping chains are exploited. The resulting mapping chains from previous integration cases are reliable. Reliability is enforced by design as only tested mappings are allowed to be stored in the knowledge repository.

In the next section, we outline how the formal definitions that describe the conceptual knowledge activities are designed as algorithms. Before that, we will shortly outline how KDAC relates to the presented engineering methods presented in chapter II.

6.2. Algorithms

6.2.1. Composition of Operation Mappings

The first two algorithms perform mapping suggestions based on the operation mapping chain between the selected source and target interface(s) (see "calculate mappings" in Fig. 6.2). Within the algorithms, a source relates to a required interface, and a target relates to a provided interface. Within a mapping chain, intermediate targets can act as a provided or required interface.

To do so, it first extracts all mappings from the knowledge base. Afterwards, the algorithm tries to find a transitive mapping tree for each of the selected target interfaces. We build up a tree here because we allow for one-to-many mappings, which inherently leads to a tree structure when processed. After a transitive mapping tree is built for each target interface, we execute all mappings (i.e., integration cases). In this step, the request and response mappings of each mapping *map_{action}* in the tree get recursively applied to each other. Finally, this leads to a mapping from the source to the target interface(s). The algorithm sanitizes the final request mapping, so it does not contain any references to API's other than the source or target. A similar sanitization step is performed for the response mapping.

Algorithm 1 performs the search for a transitive mapping chain from the source interface to one of the target interfaces. It relies on recursion and hence performs a depth-first search, with the search space being all mappings of the knowledge base. In the first step, algorithm 1 retrieves all mappings from the knowledge base that map the current source interface. While doing so, the algorithm ensures that no interface is visited twice during the search process. This means that each walk from the root to a leaf in the final tree will be a path from the source to one target interface. This restriction reduces the maximum depth of the tree. In the next step, algorithm 1 iterates over all identified mappings and all of those mapping's target interfaces. This means that one-to-many mappings are treated as n one-to-one mappings. In each iteration step, we check if the current mapping's target is equal to our final target, and if so, we add a node with the current mapping and no children to the tree. If not, we start the recursion step to find transitive mapping chains from the current target to the final target. Finally, this leads to a tree structure in which each node contains mappings and potentially some children.

6. Integration Knowledge Management

Algorithm 1 Create Mapping Suggestions

```
0: procedure BUILDMAPPINGSUGGESTIONS(sourceInterface, targetInterfaces)
0:   mappings  $\leftarrow$  getAllMappingsFromKB()
0:   mappingTrees  $\leftarrow$  []
0:   for each targetInterface  $\in$  targetInterfaces do
0:     // Find mapping trees for each of the target interfaces
0:     mappingTrees.addAll(
0:       treeSearch(sourceInterface, targetInterface, mappings)
0:     )
0:   end for
0:   requestMapping  $\leftarrow$  {}, responseMapping  $\leftarrow$  {}
0:   for each mappingTree  $\in$  mappingTrees do
0:     // Execute all mapping trees and combine results
0:     result  $\leftarrow$  executeMappingTree(mappingTree)
0:     requestMapping  $\leftarrow$  requestMapping.merge(result.reqMapping)
0:     responseMapping  $\leftarrow$  responseMapping.merge(result.resMapping)
0:     // Early return
0:     if result.complete then
0:       break
0:     end if
0:   end for
0:   // Remove any references to APIs other than the targets from mapping
0:   requestMapping  $\leftarrow$  removeNonTargetReferences(requestMapping)
0:   return requestMapping, responseMapping
0: end procedure=0
```

Algorithm 2 works on the results of algorithm 1 and aims to build up a mapping from a source to a target interface by repetitively combining intermediary mappings so that the final mapping only includes the desired source and target. Overall, two mappings $Map_{IntegrationCase}$ need to be created for the request and response mapping. However, each of the mappings needs to be built up in different "directions". While the request mapping is created by repetitively performing the request mappings from the root to the leaves on each other, the response mapping is created by executing the response mappings from the leaves to the tree's root (see definitions for mapping types).

For the request mapping, the algorithm passes on the current intermediary result to the next recursion step. In the first step, there is no input yet. Hence, the request mapping of the root node's mapping is passed directly. In each subsequent step, the current node's request mapping is performed on the previous step's input. This means that each step combines two mappings, deleting the intermediary one. The newly generated mapping always maps from the source interface to some target interface along the path from the root to the leaf. Once the recursion meets a leaf, the generated mapping maps from the source to the target interface. However, several mappings from source to target will be created if there is more than one leaf. All those mappings get merged during the recursion step.

Response mapping works differently. At each node in the tree, there is a mapping with a response mapping that needs the input of several target interfaces. However, to get this input, we first need to perform a recursion step to get the previous input. This continues until the recursion

6. Integration Knowledge Management

reaches a leaf node. We can simply return the response mapping from the target interface to the intermediate source interface in those nodes. On the level above, we combine all those intermediary inputs. Once all those inputs are computed, the algorithm executes the current node's response mapping on the combined input. This leads to a new mapping that always maps from the target interface to some source interface.

Algorithm 2 Find Transitive Mapping Chain

```

0: procedure TREESEARCH(sourceInterface, finalTragetInterface, mappings,
0: visitedInterfaces = [])
0:   sourceMappings ← getUnvisitedMappingsWithSource(
0:     sourceInterface, visitedInterfaces, mappings)
0:   tree ← new Tree()
0:   for each srcMapping ∈ sourceMappings do
0:     for each trgInterface ∈ srcMapping.targetInterfaces do
0:       // Iterate over each mapping and each target of the mapping
0:       if trgInterface == finalTragetInterface then
0:         tree.addNodeWithoutChildren(srcMapping)
0:       else
0:         childTree ← treeSearch(
0:           trgInterface, finalTragetInterface
0:           mappings, visitedInterfaces.concat(sourceInterface)
0:         )
0:         if childTree.hasNodes then
0:           tree.addNodeWithChildren(srcMapping, childTree)
0:         end if
0:       end if
0:     end for
0:   end for
0:   return tree
0: end procedure=0

```

During the entire process, the algorithm checks in each step, whether all required properties of the source and the target interfaces are already mapped. Once this is the case, the algorithm returns early from the recursion as it needs no more computations. Once the recursion is back at the required interface, the final mapping will map the target's response to the desired source interface. We can feed the final mapping $Map_{IntegrationCase}$ and all included mapping functions $\{map_{action_1}, map_{action_2}, \dots, map_{action_n}\}$ into an adapter generation framework.

In the running example (see Fig. 1.3), we would integrate the generated adapter within the mobile application source code. However, other environments that allow for implementing software adapters are also possible (e.g., IoT platforms such as openHAB).

6.2.2. Composition of Identifier Mappings

Algorithm 3 is also based on the knowledge graph. Given this multigraph, finding semantically equal identifiers boils down to finding the graph component that contains the identifier of interest. After this graph component is identified, the resulting identifiers are filtered based on

6. Integration Knowledge Management

whether they belong to any selected target interface. In contrast to the algorithm for operation mappings, this algorithm is always invoked when a new identifier mapping is added during mapping creation.

Algorithm 3 Create Identifier Mapping Suggestions

```
0: procedure BUILDIDENTIFIERMAPPINGSUGGESTIONS(sourceIdentifier, targetInterfaces)
0:   // Get graph where previously mapped identifiers are connected
0:   knowledgeGraph ← getIdentifierEqualityGraphFromKB()
0:   component ← getGraphComponent(sourceIdentifier)
0:   for each identifier ∈ component do
0:     // Only return relevant identifiers
0:     if identifier in any targetInterfaces then
0:       yield identifier
0:     end if
0:   end for
0: end procedure=0
```

6.2.3. Inverse of Mappings

This reasoning principle does not need any specific logic as the inverse of a function can be computed based on existing libraries. Such a mapping is inserted into the knowledge base as soon as the original mapping is stored. From the viewpoint of the presented algorithms, such an inverse mapping is just another edge that may lead to the desired target interfaces in the multigraph.

In the next chapter, we illustrate the different integration cases that can be solved by the KDAC method and the presented algorithms. Therefore, we rely on client-server and the publish-subscribe communication style.

7. Application

So far, we have argued that there must exist an interface description L as well as a declarative language L^* to formalize the mappings for an integration case. Next, we introduce a more realistic interface description. In contrast to other bottom-up integration approaches such as Bennaceur et al. [51], we do not need to manage any concepts in an intermediate ontology. Hence, we can rely on syntactic interface descriptions that do not contain language features to express semantic annotations (e.g., JSON-LD). An example of such semantic annotations according to the JSON-LD standard is shown in Fig. 7.1.

```
{
  "@context": {
    "volume": "http://dbpedia.org/resource/Volume",
    "sourceName": "http://dbpedia.org/resource/Name",
    "brightness": "http://dbpedia.org/resource/Brightness",
    "artMode": "http://dbpedia.org/resource/Power_saving"
  },
  "volume": "20",
  "sourceName": "input_1",
  "brightness": "100",
  "artMode": "deactivated"
}
```

Listing 7.1: JSON-LD Example for Samsung Television based on DBpedia Knowledge Base

For illustrating the syntactic interface descriptions, listing 7.2 represents the interface description for Samsung TV, listing 7.3 represents the interface description for the LG TV, and listing 7.4 represents the interface description for the Philips TV. All interface descriptions represent a REST service. Hence, they typically rely on the HTTP/JSON communication protocol. The keys, as shown in the listing, have the following meaning:

- "url": The URL where the instance implementing the interface (i.e., service) is running
- "paths": The actions that are offered by this service
- "parameters": Input data that is required by the service
- "responses": Output data that is provided by the service after the action is invoked
- "schemas": The data model used by this interface for input and output data

These interface descriptions act as a provided or required interface. For example, the mobile remote control application from our running example implements the provided interface Samsung TV and connects it to its application logic. When the LG TV should be integrated into the application, the Samsung TV interface description becomes the required interface description, and the LG TV acts as the provided interface description.

7. Application

```
{
  "servers": [
    {
      "url": "https://iot.informatik.uni-
        mannheim.de:8088"
    }
  ],
  "paths": {
    "/status": {
      "post": {
        "parameters": [
          {
            "in": "body",
            "name": "body",
            "description": "Values to be
              updated",
            "required": true,
            "schema": {
              "$ref": "#/components/
                schemas/SamsungTvInfo"
            }
          }
        ]
      },
      "responses": {
        "200": {
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/
                  schemas/SamsungTvInfo"
              }
            }
          }
        }
      }
    }
  },
  "components": {
    "schemas": {
      "SamsungTvInfo": {
        "type": "object",
        "properties": {
          "volume": {
            "type": "integer",
            "example": 1
          },
          "sourceName": {
            "type": "string",
            "example": "source"
          },
          "brightness": {
            "type": "integer",
            "example": 1
          },
          "artMode": {
            "type": "string",
            "example": "channel"
          }
        }
      }
    }
  }
}
```

Listing (7.2) Service Description for Samsung TV

```
{
  "servers": [
    {
      "url": "https://iot.informatik.uni-
        mannheim.de:8089"
    }
  ],
  "paths": {
    "/status": {
      "post": {
        "parameters": [
          {
            "in": "body",
            "name": "body",
            "description": "Values to be
              updated",
            "required": true,
            "schema": {
              "$ref": "#/components/schemas/
                LgTvInfo"
            }
          }
        ]
      },
      "responses": {
        "200": {
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas
                  /LgTvInfo"
              }
            }
          }
        }
      }
    }
  },
  "components": {
    "schemas": {
      "LgTvInfo": {
        "type": "object",
        "properties": {
          "volume": {
            "type": "integer",
            "example": 1
          },
          "input": {
            "type": "string",
            "example": "input 1"
          },
          "brightness": {
            "type": "integer",
            "example": 1
          },
          "power_saving": {
            "type": "string",
            "example": "true"
          }
        }
      }
    }
  }
}
```

Listing (7.3) Service Description for LG TV

7.1. From Abstract to Concrete Integration Knowledge Management

This section provides examples of the different mapping types and reasoning principles (see formalization section). We assume that required interface descriptions should be replaced with provided interface descriptions (e.g., moving to a different home). However, this replacement action is only for illustrative purposes. The name of each example acts as a reference for each corresponding definition. We do not provide an example for a many-to-many mapping as such a complex mapping can also be constructed by multiple, simpler one-to-many mappings. Furthermore, such a mapping may result in too much application logic in the generated software adapter, which decreases code maintainability.

Example 1 (One-to-One Mapping).

In our running example, this means that the action `"/status"` offered by the Samsung TV (see listing 7.2) can be mapped to the operation `"/status"` offered by the LG TV (see listing 7.3). Therefore, all input data must be mapped to each other (e.g. `volume` \rightarrow `volume`, `sourceName` \rightarrow `input`, `brightness` \rightarrow `brightness` and `artMode` \rightarrow `power_saving`). Here `a` \rightarrow defines a mapping function in one direction (e.g., from Samsung TV to LG TV). We must formalize the same mappings for the output data in the opposite direction (i.e., from provided to required output).

Example 2 (One-to-Many Mapping).

In our running example (see Fig. 1.3), this situation is not directly depicted in the presented service descriptions as it symbolizes two one-to-one mappings for the action `"/status"`. However, we can construct a one-to-many mapping example by assuming there are two instead of one TV in a room. Hence, the Samsung TV is replaced by the LG TV

```
{
  "servers": [
    ...
  ],
  "paths": {
    "/status": {
      "post": {
        "parameters": [
          {
            ...
            "schema": {
              "$ref": "#/components/schemas/PhillipsTvInfo"
            }
          }
          ...
        ],
        "responses": {
          "200": {
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/PhillipsTvInfo"
                }
              }
            }
          }
          ...
        }
      },
      ...
    },
    "components": {
      "schemas": {
        "PhillipsTvInfo": {
          "type": "object",
          "properties": {
            "volume": {
              "type": "integer",
              "example": 1
            },
            "source": {
              "type": "string",
              "example": "source"
            },
            "screen_brightness": {
              "type": "integer",
              "example": 1
            },
            "power_saving": {
              "type": "boolean",
              "example": true
            }
          }
        }
        ...
      }
    }
  }
}
```

Listing 7.4: Service Description for Philips TV

7. Application

and the Philips TV. Now, the required action "/status" from the Samsung TV results in two action invocations. One "/status" invocation for the LG TV and one "/status" invocation for the Philips TV. Another example would be that the "/status" operation requires two action invocations offered by one other TV.

Example 3 (Domain and Image of Mappings).

In our running example, the input data identifier "volume" of the Samsung TV $volume_{Samsung}$ and the input data identifier "volume" $volume_{LG}$ of the LG TV can be mapped. Both parameters have the data type integer. Assume that the Samsung TV measures volume (the sound level of speakers) from 0 to 10 and the LG TV measures volume from 0 to 100. Then, the domain $DOM(volume_{Samsung})$ of a mapping function $Map_{action}(volume)$ is the value range 0 to 10 and its image $IMG(volume_{LG})$ is the value range from 0 to 100.

Example 4 (Inverse of Mappings).

Let the mapping function $volume$ has the domain $DOM(volume_{Samsung})$ and the image $IMG(volume_{LG})$. Then, the mapping function $volume : volume_{Samsung} \mapsto volume_{Samsung} * 10$ translates the identifier values $volume_{Samsung}$ to identifier values of $volume_{LG}$. We can now define the inverse of a mapping function.

Assume the mapping function $volume : volume_{Samsung} \mapsto volume_{Samsung} * 10$ is bijective. Then, the inverse mapping function $volume^{-1}$ is $volume^{-1} : volume_{LG} \mapsto volume_{LG}/10$ with domain $DOM(volume_{LG})$ and image $IMG(volume_{Samsung})$.

Example 5 (Composition of Operation Mappings). In our running example, the composition of operation mappings is directly visualized. Assume that the knowledge base contains a mapping from action "/status" (Samsung TV) to action "/status" (LG TV) and from action "/status" (LG TV) to action "/status" (Philips TV). Hence there exists a path within the graph from action $status_{Samsung} \rightarrow status_{LG} \rightarrow status_{Philips}$. If the integration case $status_{Samsung} \rightarrow status_{Philips}$ occurs, mappings can be computed.

In addition, if all mapping functions also have the inverse property (see Def. 11), then the mappings from $status_{Philips}$ to $status_{Samsung}$ can be computed. This is because of the path within the graph from the action $status_{Samsung} \leftrightarrow status_{LG} \leftrightarrow status_{Philips}$.

Example 6 (Composition of Identifier Mappings).

Assume there is a mapping function $brightness : brightness_{LG} \mapsto brightness_{Philips}$ with $DOM(brightness_{LG})$ and $IMG(screen_brightness_{Philips})$ for the action "/status" available from another integration case. Now, a one-to-many mapping from the action $status_{Samsung}$ to $status_{LG}$ and $status_{Philips}$ should be created. Consequently, the mapping function $brightness : screen_{Samsung} \mapsto brightness_{Samsung}/10$ with $DOM(brightness_{Samsung})$ and $IMG(brightness_{LG})$ is inserted into the knowledge base. After this mapping function is added, the mapping function $brightness : brightness_{Samsung} \mapsto brightness_{Philips}/10$ with $DOM(brightness_{Samsung})$ and $IMG(brightness_{Philips})$ can be calculated as there exists a path in the graph from $brightness_{Samsung} \rightarrow brightness_{LG} \rightarrow brightness_{Philips}$.

To recap, the main difference between the composition of identifier and operation mappings is the moment of application. For operation mappings, the reasoning principle is applied after the selection of action. The reasoning principle is applied for the identifier mappings after a mapped mapping function based on identifiers is created.

In the next sections, we will connect mapping functions to the client-server and the publish-subscribe communication style.

7.2. Towards Software Adapter Generation

Up to now, we have seen how integration knowledge for different mapping types can be managed. Furthermore, we introduced reasoning algorithms that assist the system integrator in reusing integration knowledge. As the last part of this chapter, we outline how available integration knowledge (especially mapping type one-to-many) is related to two exemplary architectural styles. Consequently, we can generate the corresponding software adapter and deploy it on systems that follow these architectural styles.

7.2.1. Client-Server

A client-server model [24] is an architectural style where clients send requests to a server, which performs the required functions and replies with the requested information. For example, clients initiated the communications by remote procedure calls.

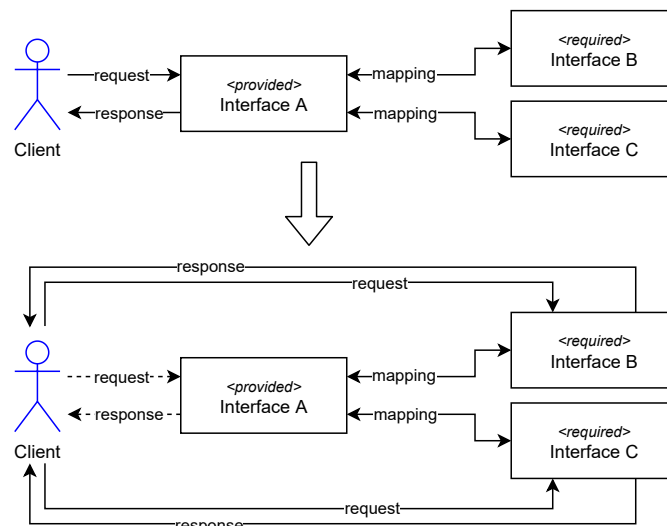


Figure 7.2.: Client-Server Split

Our first example deals with splitting an action invocation (see Fig. 7.2). The upper part of the figure illustrates the context before, and the lower part of the figure illustrates the context after the integration has been finished. Dotted lines symbolize that these remote procedure calls are no longer used.

Before: A client sends a request to a provided action from interface A. The instance of interface A should be replaced by actions offered by interface B and interface C. Now, the request is split up into two requests.

After: After the mapping is finished, the client application logic still assumes to call actions as

7. Application

defined by interface description A. After deployment into the client application code, the generated software adapter transforms the mapped action requests to the available interface instances B and C.

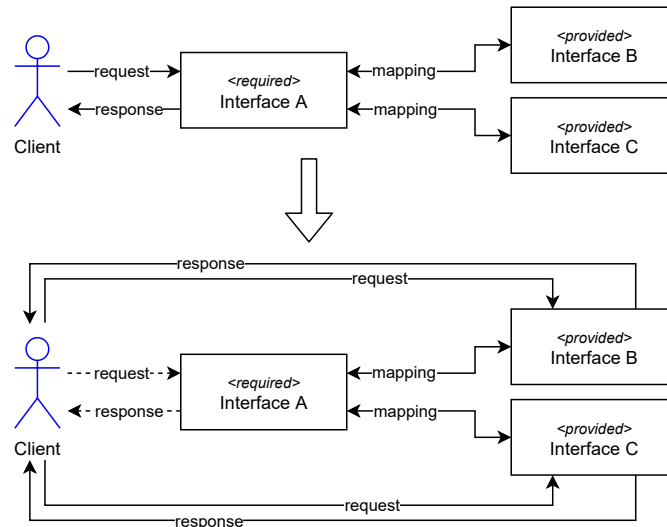


Figure 7.3.: Client-Server Aggregate

The second example deals with aggregating data produced by action invocations (see Fig. 7.3). In contrast to the split example, the response data is to be aggregated.

Before: A client expects response data from an action as defined in interface description A. The required output data is mapped to the provided output data from actions as offered by interface B and C.

After: After the mapping is finished, the client application logic still assumes to call actions as defined by interface description A. After deployment into the client application code, the generated software adapter transforms the mapped actions responses as produced by the interface instances B and C.

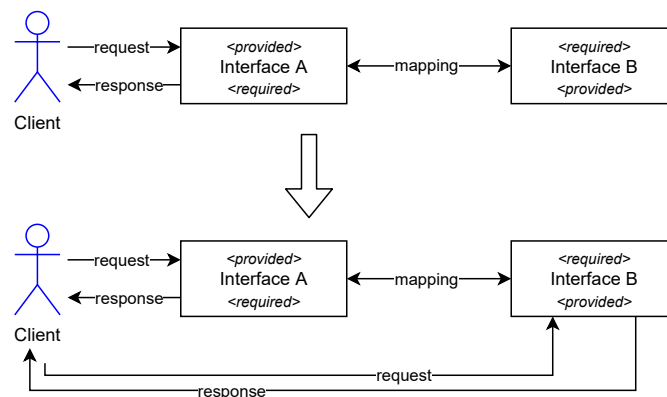


Figure 7.4.: Client-Server Extend

7. Application

The last example deals with extending an action invocation (see Fig. 7.4). In contrast to the split example, the required actions are not replaced. Hence, they act as both required and provided interfaces.

Before: A client expects input and output data as defined by an action from interface description A. Now, the client is expected to invoke actions from interface A and actions from interface B. Hence, the mappings from the required actions offered by interface A to provided actions offered by interface B are created.

After: After the mapping is finished, the client application logic still calls actions as offered by interface instance A. After deployment into the client application code, the generated software adapter extends the mapped action requests to the initial interface A and the additional interface B.

It is up to the client how both responses affect the application logic. This is not part of the software adapter anymore.

7.2.2. Publish-Subscribe

In contrast to the client-server model, the publish-subscribe model decouples clients from the server. A client must not know where the requested functionality is executed. A publish-subscribe model [24] is an architectural style where subscriber register/deregister to receive specific messages or content. The publisher maintains a subscription list and broadcasts messages to subscribers either synchronously or asynchronously.

In contrast to the client-server style, requests are not directly sent to an action but to channels that contain the payload for a specific topic. This channel topic is comparable to the action invocation. Arrows from the client to other interfaces define data published from the client (i.e., a provided channel). Arrows to the client define data that the client subscribes to (i.e., a required channel). Again, dotted lines symbolize that these channels are no longer used.

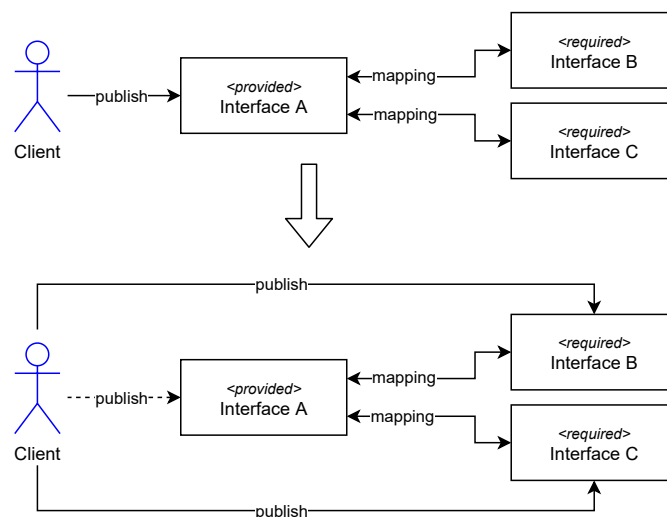


Figure 7.5.: Publish-Subscribe Split

7. Application

The first example deals with splitting published data (see Fig. 7.5).

Before: The client publishes data to a channel offered by interface A. The corresponding interface instance is replaced by channels offered by interfaces B and C. Hence, the mappings based on topic and payload are created.

After: After the mapping is finished, the client application logic still assumes to publish payload as defined by interface description A. After deployment into the client application code, the generated software adapter transforms the mapped payload and channel topics to the interface instances B and C.

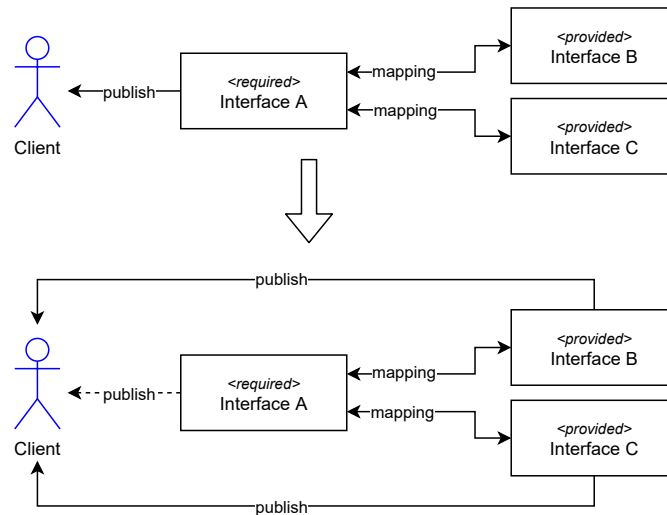


Figure 7.6.: Publish-Subscribe Aggregate

The second example deals with aggregating payload that the client subscribes to (see Fig. 7.6).

Before: A client subscribes to payload from a channel as defined in interface description A. The required payload is mapped to the provided payloads from channels where B and C's interfaces publish their data.

After: After the mapping is finished, the client application logic still assumes to subscribe to channel and payload as defined by interface description A. After deployment into the client application code, the generated software adapter transforms the mapped payload and responses as published by the interface instances B and C.

The last example deals with extending a subscription (see Fig. 7.7). Again, the required channel is not replaced, but more payload is consumed by the client as another interface also publishes its payload to the existing channel.

Before: A client subscribes to the payload as produced by a channel from interface A. Now, the client should deal with payload from interface A and payload as published by interface B. Hence, the mappings from the required channel offered by interface A to provided channels offered by interface B are created.

After: After the mapping is finished, the client application logic still receives payload as published by channels from interface instance A. After deployment into the client application code,

7. Application

the generated software adapter extends the mapped channel topics as published by the provided channels by the additional interface B. Again, it is up to the client how more payload affects the application logic.

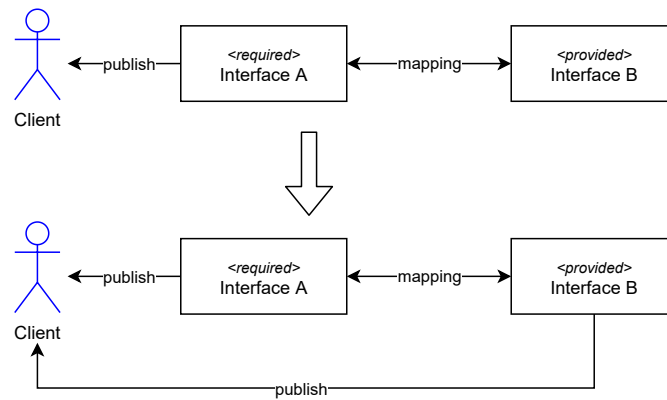


Figure 7.7.: Publish-Subscribe Extend

The interested reader may have noticed that we did not give much attention to the behavioral aspects of services (e.g., using a labeled transition system). In the scope of this work, we will only apply KDAC to stateless services. Therefore, we will not consider behavioral service aspects as we move to our approach's technological part. However, we will discuss this topic within the future work section.

In the next part, we will take the final steps towards operationalizing integration knowledge formalization by introducing the missing mapping language L^* .

Part IV.

Reference Implementation

8. Deployment and Technologies

In this chapter, we first provide a short overview of the technology frameworks we reuse for implementing the presented algorithms. However, the central part will be answering our first research question "How can we make integration knowledge that is captured in imperative software adapter reusable?". Therefore, we will give concrete examples of how interface mappings are described in a declarative way using the language L^* .

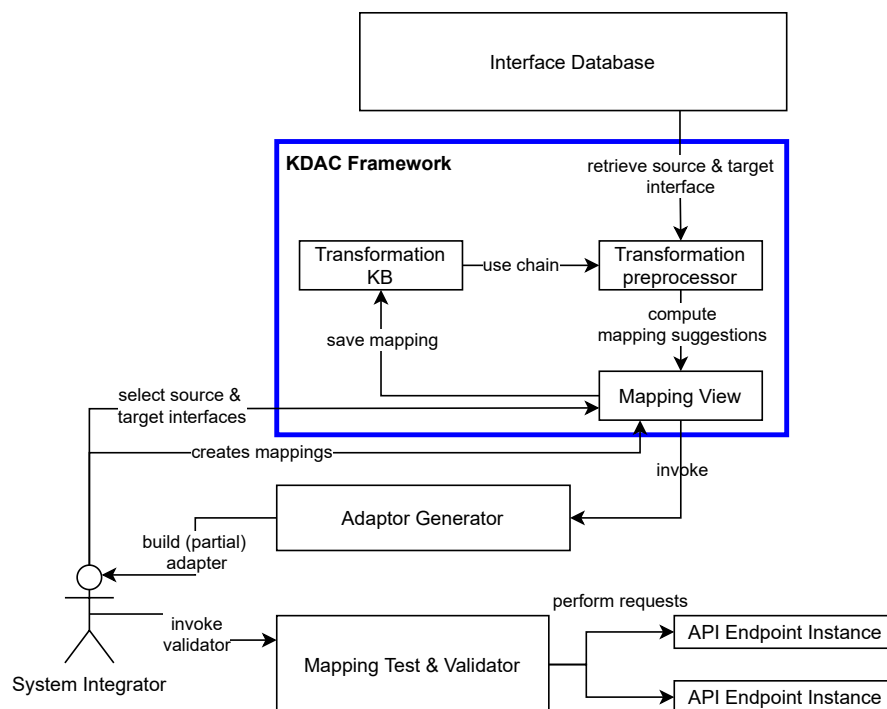


Figure 8.1.: Logical System Overview

8.1. Logical Architecture

The logical core elements necessary to store and reason about integration knowledge are displayed in the KDAC Frame in Fig. 8.1. This frame contains the Transformation KB, the Transformation preprocessor, and the Mapping View. The Transformation KB is responsible for storing mappings. The Transformation Preprocessor is accountable for executing the presented algorithms. Therefore, it retrieves interface descriptions from the Interface Database and applies the mappings retrieved from the Transformation KB. The Mapping View displays the

8. Deployment and Technologies

algorithms' results based on the selected required and provided operations and can be used to save modified mappings.

Besides, the logical components Mapping Test & Validator, API Endpoint Instances, and Adapter Generator are necessary to accompany the integration knowledge management process. The Mapping Test & Validator can be used to make a validation query based on the defined mappings within the Mapping View. By doing so, the system integrator can verify if requests succeed and responses are transformed in the desired way. In practice, this is necessary as interface descriptions may contain other identifier definitions compared to the actual service instance. If the validation succeeds, the Adapter Generator can be invoked such that a software adapter project is generated in the desired programming language. Depending on the execution environment, this adapter project is then processed further. For instance, it is imported into the code repository of the mobile application client (see running example 1.3).

8.2. Deployment Architecture

The presented logical components are deployed using a fat client model. This means that the client executes all reasoning logic. In total, we have three deployment targets and one user interface. The Presentation Server contains the Web Application, User Management, Authentication, and a Graph Database.

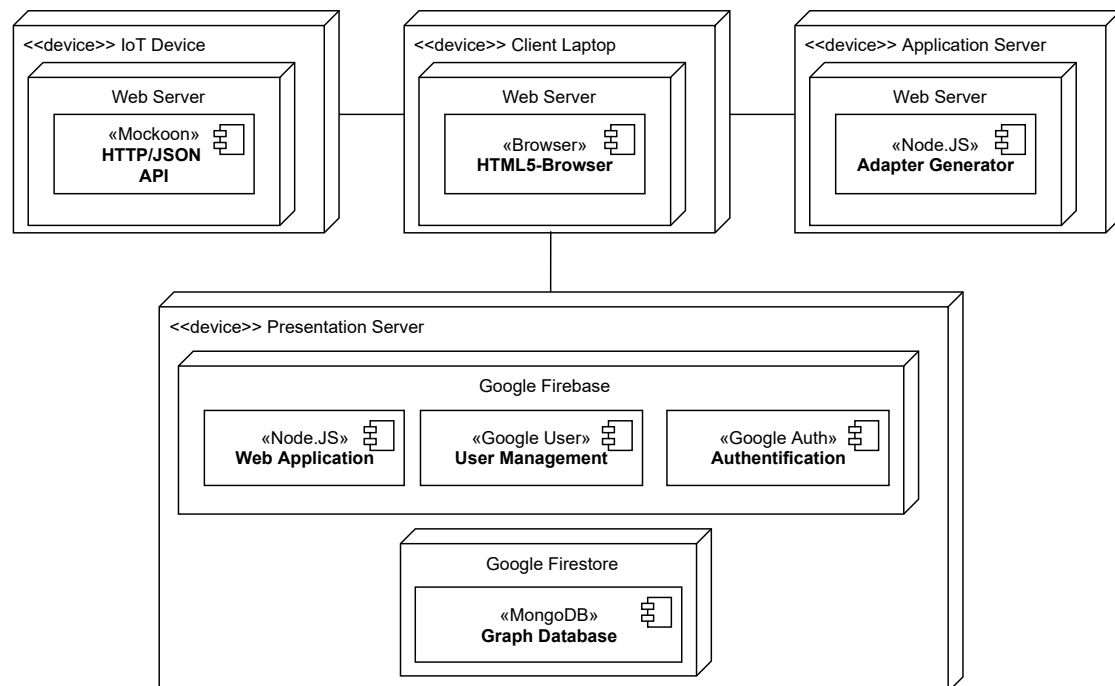


Figure 8.2.: Deployment Diagram

The Web Application is served to the client laptop and contains the logical elements of Transformation preprocessor and Mapping View. User Management and Authentication are infras-

structure services. The Graph Database stores interfaces and mappings and hence contains an execution environment for the Interface Database and the Transformation KB. The accompanying logical components API Endpoint Instance (simulating all needed IoT devices) and Adapter Generator are both realized as a component running in a web server environment.

8.2.1. Interface Description Languages

As an interface description language we rely on OpenAPI [56] and AsyncAPI [57].

An example for OpenAPI descriptions has already been discussed in listings 7.2, 7.3 and 7.4. According to the creators, the goal of an OpenAPI specification is to "...define a standard, programming language agnostic interface description for REST APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic. When properly defined via OpenAPI, a consumer can understand and interact with the remote service with minimal implementation logic. Similar to what interface descriptions have done for lower-level programming, the OpenAPI Specification removes guesswork in calling a service" [56].

An example of an AsyncAPI description can be seen in the listings 8.1. According to the creators, the goal of an AsyncAPI specification is to "...make working with EDAs as easy as it is to work with REST APIs. The AsyncAPI Specification is a project used to describe and document message-driven APIs in a machine-readable format. It is protocol agnostic, so you can use it for APIs that work over any protocol (e.g., AMQP, MQTT, WebSockets, ...)." [57].

Both descriptions are equipped with a specification parser, schema validators, code generators for many programming languages (e.g., JavaScript, Java, and many others), and annotation libraries for generating interface descriptions from code.

```

{
  "servers": {
    "production": {
      "url": "test.mosquitto.org:1883",
      "protocol": "mqtt"
    }
  },
  "channels": {
    "smartylighting/lighting/measured": {
      "publish": {
        "operationId": "receiveLightMeasurement",
        "message": {
          "$ref": "#/components/schemas/lightMeasuredPayload"
        }
      }
    },
    "smartylighting/streetlights/turn/on": {
      "subscribe": {
        "operationId": "turnOn",
        "message": {
          "$ref": "#/components/schemas/turnOnOffPayload"
        }
      }
    }
  },
  "components": {
    "schemas": {
      "lightMeasuredPayload": {
        "type": "object",
        "properties": {
          "lumens": {
            "type": "integer",
            "minimum": 0,
            "description": "Light intensity measured in lumens."
          }
        }
      }
    }
  }
}

```

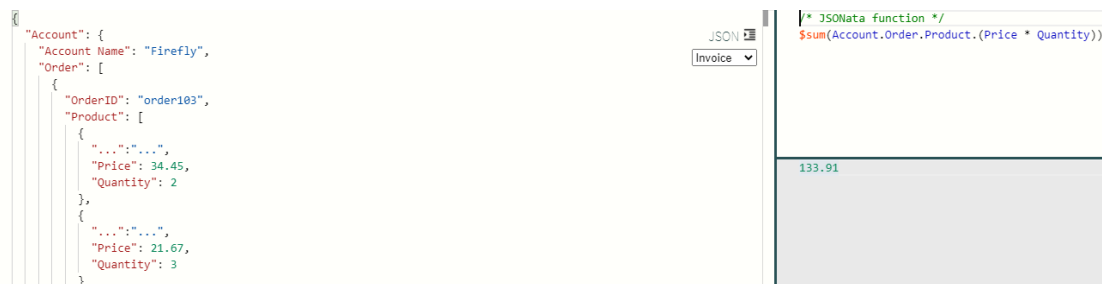
Listing 8.1: AsyncAPI Example

8. Deployment and Technologies

We parse these specifications in the implemented prototype to render the API endpoints' interfaces within the Mapping View. Furthermore, they are stored as nodes within the knowledge graph.

8.2.2. Mapping Language L^*

To describe mappings between required and provided interface descriptions, we use JSONata [58]. JSONata is a "...lightweight query and transformation language for JSON data. Inspired by the 'location path' semantics of XPath 3.1, it allows sophisticated queries to be expressed in a compact and intuitive notation. The JSONata path expression is a declarative, functional language. It is functional because it is based on the map/filter/reduce programming paradigm as supported by popular functional programming languages through higher-order functions. It is declarative because these higher-order functions are exposed through a lightweight syntax which lets the user focus on the intention of the query (declaration) rather than the programming constructs that control their evaluation" [58]. An example of such a query string can be seen in Fig. 8.3.



```

{
  "Account": {
    "Account Name": "Firefly",
    "Order": [
      {
        "OrderID": "order103",
        "Product": [
          {
            "Price": 34.45,
            "Quantity": 2
          },
          {
            "Price": 21.67,
            "Quantity": 3
          }
        ]
      }
    ]
  }
}

```

```

JSON
Invoice
/* JSONata function */
$sum(Account.Order.Product.(Price * Quantity))
133.91

```

Figure 8.3.: JSONata Example from <https://try.jsonata.org/>

8.2.3. Software Adapter Generation and API Endpoints

Both OpenAPI and AsyncAPI are equipped with code generators to generate client code to access a web service and generate server stubs. For code generation, they rely on the Moustache template engine¹. We inject our mappings into their generation process and provide our own adapter template. A web server serves the generated code. This allows the system integrator to download a software adapter project using the Mapping View.

To manage and make IoT devices available via their interface fast, we rely on Mockoon². Mockoon provides an easy way to mock APIs. Based on a configuration script, we deploy their command line interface on a web server. We started the interface instances based on the interface descriptions as provided by OpenAPI and AsyncAPI. This allows for executing the Mapping Test & Validator component on desired input and output data.

¹<https://mustache.github.io/>

²<https://mockoon.com/>

8.3. Application Examples

To use the prototype, the system integrator must be authenticated. Furthermore, it must be selected whether mappings for OpenAPI or AsyncAPI specification should be created or queried.

8.3.1. Client-Server Mapping Function

An example for a simple and a more complex mapping function is visualized in Fig. 8.4 and Fig. 8.5. This text editor opens when selecting a mapping from the Mapping Area. In a simple case, one required identifier can be replaced by a provided identifier. In a more complex case, all functions offered by JSONata can be used to specify an arbitrary complex mapping function. Therefore, all provided keys can be selected and are then available for usage within the text editor to compute the required key.

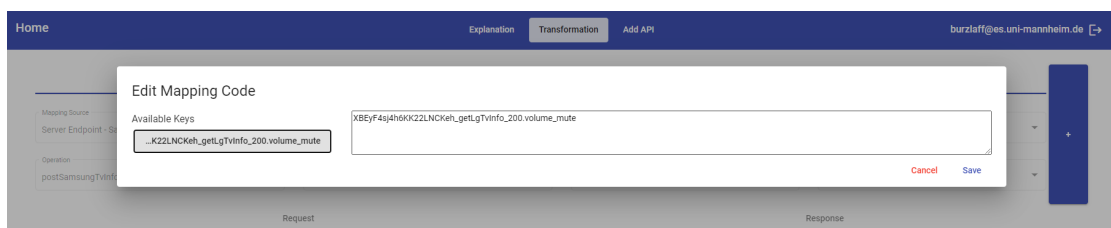


Figure 8.4.: Example – Simple JSONata Mapping Function

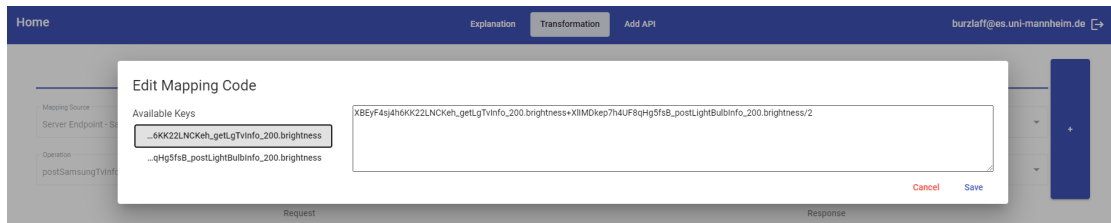


Figure 8.5.: Example – Complex JSONata Mapping Function

8.3.2. Client-Server One-to-One

In Fig. 8.6 a One-to-One mapping example for a OpenAPI specification is visualized. After the required source interface (left side) and provided target interface (right side) are selected, the operation to be integrated are chosen. Then, the mappings for the request as well as the response are created. Here the color code of each box within the Mapping Area indicates whether it is an invalid (red), a manual (blue), or an inferred mapping (green and brown). The algorithms are invoked as soon as a required operation is selected (c.f. definition 12) or a mapping has been modified (c.f. definition 13).

8. Deployment and Technologies

The screenshot shows the OpenAPI Mapping Tool interface. At the top, there are tabs for 'Home', 'Explanation', 'Transformation', and 'Add API'. The user's email 'burzloff@es.uni-mannheim.de' is visible in the top right. The main area is divided into 'Source' and 'Target 1' sections. The 'Source' section shows a 'Mapping Source' of 'Server Endpoint - Phillips' and an 'Operation' of 'postLightBulbInfo'. The 'Target 1' section shows a 'Mapping Target' of 'Server Endpoint - Yeelight' and an 'Operation' of 'postYeelightLightBulbInfo'. Below these, there are 'Request' and 'Response' fields. The central 'Mapping Area' is the focus, showing a 'Map same keys' button and a 'Map selected' button. It contains five mapping rules, each with a red minus sign in a circle. The rules map source keys to target keys: 'power' to 'power', 'color_yee' to 'color', 'brightness_yee' to 'brightness', 'time_Day' to 'time', and 'time_Minute' to 'time'. The 'Source Response Body' on the left lists keys like 'switch_philips', 'color', 'brightness', 'color_temperature', and 'time'. The 'Target Response Body' on the right lists keys like 'power', 'color_yee', 'brightness_yee', 'colorTemperature', and 'time'.

Figure 8.6.: One-to-One Mapping using OpenAPI

The screenshot shows the OpenAPI Mapping Test & Validator interface. At the top, there are two JSON snippets. The left one is a request body with keys like 'volume', 'mute', 'brightness', 'contrast', 'sharpness', 'colorTemperature', 'sourceName', and 'sourceId'. The right one is a response body with a single key '1'. Below the snippets is a 'Test Request' button. A red error message is displayed: 'ERROR: Missing request mappings color_temperature2,input2,power,sharpness_screen. Missing response mappings volume,mute,brightness,contrast,sharpness,colorTemperature,sourceName,channelName,power,keyCode,time,day,time,month,time,year,time,hour,time,minute,time,second'. Below the error message is a 'Finish Mapping' button. The bottom section is titled 'Errors in mapping' and contains two columns of missing keys: 'Missing in Request Mapping' (color_temperature2, input2, power, sharpness_screen) and 'Missing in Response Mapping' (volume, mute, brightness, contrast, sharpness, colorTemperature, sourceName, channelName, power, keyCode, time.day, time.month, time.year, time.hour, time.minute, time.second).

Figure 8.7.: Mapping Test & Validator using OpenAPI

In Fig. 8.7, the Mapping Test & Validator is displayed. The created JSONata mappings are applied by issuing a test request, and a request is made against all servers as specified in the interface specification. If a mapping function can fulfil not all required operation elements, an error listing all missing keys is displayed.

8.3.3. Subscribe One-To-One

The user interface for Publish-Subscribe integration cases is slightly different. In Fig. 8.8 a one-to-one subscription example is displayed. In contrast to the client-server style, the operation now represents the channel where the device should subscribe or publish to. Furthermore, the distinction between request keys and response keys is no longer needed as the client does not wait for an answer (i.e., asynchronous communication). In this figure, the target operation is marked with an asterisk. This means that the reasoning algorithms (i.e., Def. 12) found a matching provided operation. Hence, the system integrator can choose this operation directly.

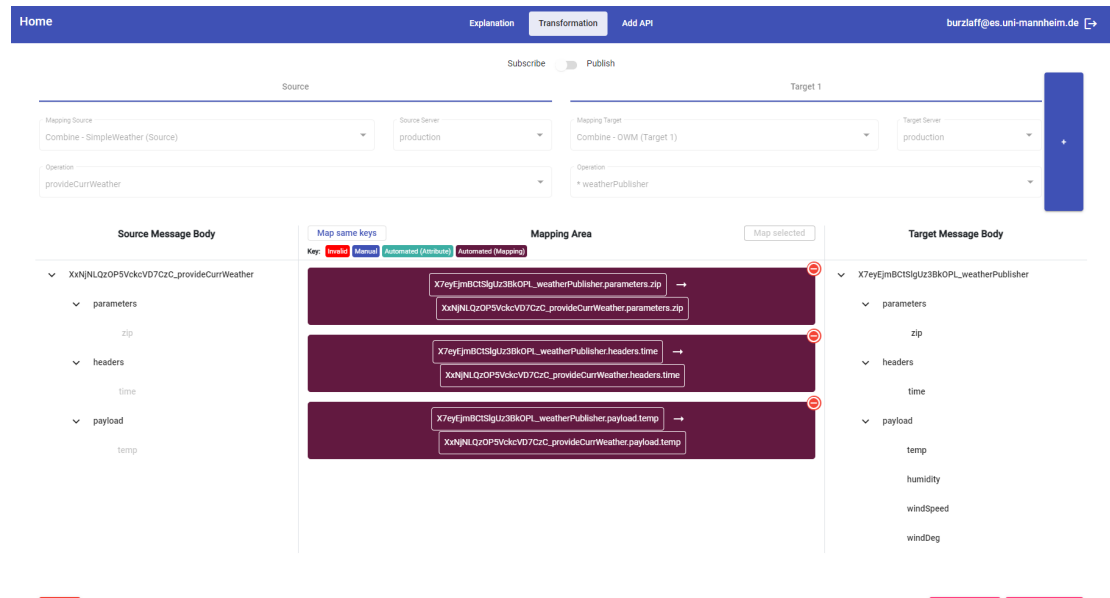


Figure 8.8.: One-to-One Subscribe using AsyncAPI

8.3.4. Publish One-To-Many

In Fig. 8.9, a one-to-many integration scenario for a publisher is visualized. This can be seen when having two (i.e., Target 1 and Target 2) instead of one target. Furthermore, the switch states that the source interface (required) is of type publish. On the target side, the new publishing setting is depicted. In this case, it is based on the extend example (see 7.7). For instance, the Mapping Source "Bosch Grid" acts as a required and as a provided interface. Hence, the interface list on the right side also contains the "Bosch Grid" as a provided interface.

Overall, all features (e.g., split or extend) discussed within the examples are available for both architectural styles. What is left is the software adapter generation part.

The software adapter generation mechanism is a supportive feature for the overall process. However, it does not directly influence the integration knowledge management process. At the time of writing, we support Node.js adapters. Other programming languages are also possible but must be implemented with another template.

8. Deployment and Technologies

The screenshot displays the MuleSoft Anypoint Studio interface for configuring a publish flow. At the top, the navigation bar includes 'Home', 'Explanation', 'Transformation', and 'Add API', along with the user email 'burztaff@es.uni-mannheim.de'. The main workspace is divided into three sections: 'Source', 'Target 1', and 'Target 2'. The 'Source' section is configured with 'Extend - Bosch Grid (Source & Target)' as the Mapping Source, 'production' as the Source Server, and 'handshaking' as the Operation. The 'Target 1' section is configured with 'Extend - Philips Hub (Target)' as the Mapping Target, 'production' as the Target Server, and 'receiveDiscovery' as the Operation. Below these sections is the 'Mapping Area', which is currently empty. On the left, the 'Source Message Body' is expanded to show a tree structure: 'XnMJ9KYRovWSFUjmMvUK_handshaking' (expanded), 'parameters' (expanded), 'deviceid', 'headers' (expanded), 'time', and 'payload' (expanded), 'category'. On the right, the 'Target Message Body' is also expanded to show a tree structure: 'XnMJ9KYRovWSFUjmMvUK_handshaking' (expanded), 'parameters' (expanded), 'deviceid', 'headers' (expanded), 'time', 'payload' (expanded), 'category', 'XnE62AEwmiHSLXv7JbPP1V_receiveDiscovery' (expanded), 'headers' (expanded), 'groupid', 'payload' (expanded), and 'deviceid'. The 'Mapping Area' has a 'Map selected' button and a 'Key:' dropdown with options: 'Invert', 'Manual', 'Automated (Attribute)', and 'Automated (Mapping)'.

Figure 8.9.: Example – One-to-Many Publish using AsyncAPI

9. Safeguarding Expected Method Benefits

Dealing with incomplete interface mappings is central to KDAC as integration knowledge is formalized incrementally. In addition, all mappings formalized over time should be defined in a reliable way (i.e., they should be trustworthy for other system integrators such that they are actually reused). Therefore, this chapter takes a closer look at the necessary engineering steps to identify missing mappings and mapping reliability.

Fig. 9.1 illustrates situations within the formalization phase based on the reference implementation. First, the system integrator selects the required and the available provided interface from the knowledge base (see number 1). Here we already check if a provided action has already been mapped to the selected required action based on the stored integration knowledge. We indicate available action mappings with an asterisk as a prefix before the action signature. Then the reasoning algorithms are invoked. After the knowledge base is queried and the mapping calculation is finished, not all keys may have a mapping. Naturally, this is the case if no mappings are found or if one of the interfaces has just been added to the knowledge base. If some but not all map-

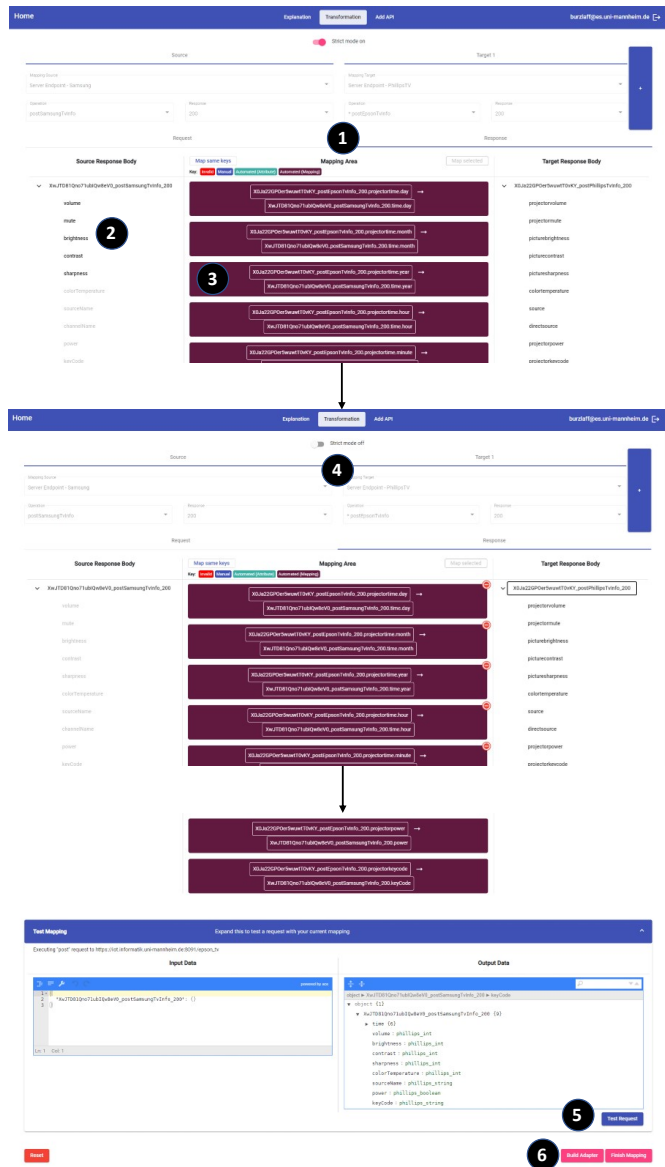


Figure 9.1.: Incomplete but Reliable Mappings

9. Safeguarding Expected Method Benefits

pings are retrieved, then all required keys without a mapping are highlighted with black color (see number 2). Next, the missing mappings are added such that all required keys are associated with a set of provided keys. As soon as all mappings are formalized, the following reliability checks are performed:

As a preset, mappings that are calculated or retrieved from the knowledge base can not be edited. This can only be done by deactivating the associated "strict" mode (see number 4). Assuming the provided mappings are correct, then the added mappings must be checked if they are reliable. This can be done by issuing a test request against the required interface where only the provided interface is actually available (see number 4). After the test request is executed, the system integrator inspects the result. Furthermore, the system integrator verifies if the device or mocked version of the device also exposes the desired behavior (e.g., the TV channel is switched correctly). Only then the system integrator is expected to store the defined mappings (see number 6 in Fig. 9.1).

Based on the presented engineering steps, we will look at the details in the next two sections.

9.1. Speed

Speed in formalizing interface mappings is mandatory as formalization effort adds to the integration time by just writing a software adapter. In order to detect missing interface mappings, all required keys must be mapped. The occurrence of incomplete mappings depends on the mapping type and the transformation function. Therefore, we look at the results that are produced by the algorithms as presented in section 6.2.

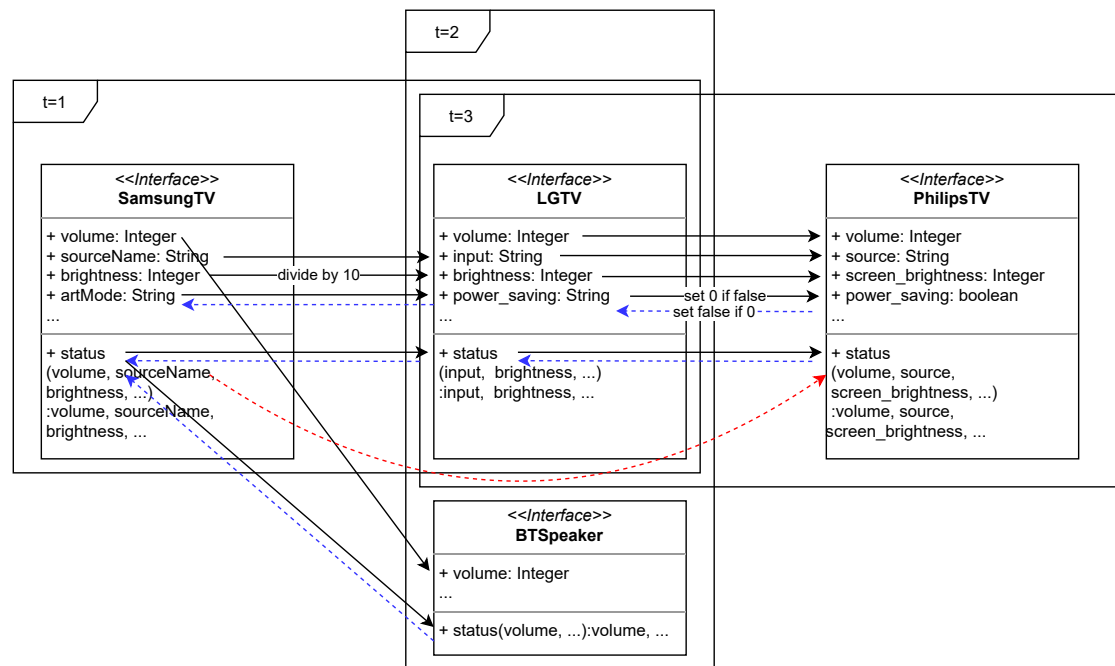


Figure 9.2.: Incomplete Mappings

9. Safeguarding Expected Method Benefits

The algorithms "Composition of Operation Mappings" (see Algo. 1) and "Composition of Identifier Mappings" (see Algo. 3) produce incomplete mappings if there exists at least one-to-many mapping within the chain of mappings and not all intermediate nodes have a distinct path to the target action.

Assume that the running example is extended by a third device, a speaker (see Fig. 9.2). At $t=2$, a one-to-many mapping from SamsungTV to LGTV and BTSpeaker is created. For instance, a movie's sound should not be issued by the TV but by a separate speaker. Therefore, the volume identifier of SamsungTV is mapped to the volume identifier of the BTSpeaker. At $t=3$, another system integrator maps the LGTV to the PhilipsTV. This means that there exists a path within the knowledge base from $SamsungTV \rightarrow LGTV \rightarrow PhilipsTV$. In an integration case between SamsungTV and PhilipsTV, not all mappings can be calculated by the algorithms. In fact, there will not exist a mapping for the volume identifier.

Inverse of Mappings (see Def. 11) can also cause incomplete mappings. If no inverse function exists or multiple provided identifiers are used to calculate a required identifier, then the calculated reverse mapping is incomplete. As a synthetic example, we can assume that the identifier brightness of the SamsungTV interface is calculated by multiplying the brightness_dimmer (not displayed) and the brightness identifier from the LG device. If such a mapping exists within a chain of mappings, then the inverse mapping is not calculated. The reason for this is that we cannot deal with systems of linear equations. For example, the linear equation

$$brightness_{Samsung} = brightness_dimmer_{LG} * brightness_{LG} \quad (9.1)$$

can be rewritten as

$$brightness_dimmer_{LG} = brightness_{Samsung} / brightness_{LG} \quad (9.2)$$

The algorithm can then not determine the next mapping as the LG identifier is present in both sides of the equation (i.e., there is a loop).

If the mentioned cases do not apply, then either no mapping or all mappings are calculated. Finally, the system integrator finishes the integration by generating the software adapter. This software adapter is then imported into the client's software project. In an ideal case, the application logic can then talk to the newly added device without changing any other piece of code. From a practical point of view, adjusting the mappings within the tooling infrastructure is generally faster than editing code within the generated software project.

9.2. Reliability

Mapping reliability over time is crucial. All mappings inserted into the knowledge base and calculated by the reasoning algorithms must be correct. Therefore, we implemented a correct by construction principle within the prototype. This principle is enforced within the prototype by the strict mode (see 4 in Fig. 9.1). This mode controls whether mappings can be saved or not. A mapping $Map_{IntegrationCase}$ can only be saved if the set of mappings is incomplete for the

9. Safeguarding Expected Method Benefits

selected required and provided action. Hence, it must be ensured that only the correct mappings are stored. If a manual mapping already exists, then only the initial creator of the mapping can change it. In order to enforce this correct by construction principle, we have implemented the following measures:

1. As soon as a mapping is formalized using the JSNOata syntax, the specified syntax is validated. If the syntax is not valid, the corresponding mapping is highlighted.
2. All necessary mappings between the required and the provided interface actions must be present. For request and publish mappings, all selected actions and their identifiers for the provided interfaces must be assigned (i.e., needed to call a provided action). For response and subscribe mappings, the selected action and its identifiers for the required interface must be assigned (i.e., needed to construct the required output ultimately). Only in this case can the system integrator issue a test request.
3. When all mappings are formalized, then value transformation is inspected by issuing a test request. Therefore, a mock instance or a physical device must be present. The system integrator decided if the mapping is correct based on the received values. If a transformation fails, the corresponding error is displayed in the application (e.g., multiplying two values of type string).
4. The system integrator is allowed to save the mapping only if the values are correct.

If there are mappings that cannot be reused for any other reason, the strict mode can be deactivated. Then, all mappings can be edited regardless of their ownership. However, these mappings cannot be saved. Only the software adapter can be generated. In this way, we ensure that all mappings are correct by constructions, and thus reliability is ensured.

However, there exist two cases where reliability is favoured over mapping reuse. The first case involves multiple paths from source to target actions, and the second case involves setting a static value using L^* . We will discuss both cases based on the situation as depicted in Fig. 9.3.

Assume four integration contexts have been formalized. At $t=1$, a mapping from $key1_A$ to $key3_B$ is inserted. At $t=2$, a mapping from $key1_A$ to $key5_C$ and at $t=3$ a mapping from $key5_C$ to $key7_D$ is inserted. Lastly, a mapping from $key3_B$ to $key7_D$ is inserted. Now, a conflict arises if interface A is selected as a required interface, and interface D is selected as a provided interface. The composition algorithm for operation mappings (see Algo. 1) will now find two paths for $key1_A$ where the first path is $key1_A \mapsto key5_C \mapsto key7_D$ and the second path is $key1_A \mapsto key3_B \mapsto key7_D$. This case can occur without violating the correct by construction measures as each manually inserted mapping is correct. Despite this circumstance, we can choose any of the mappings as there is no mapping that is "more" correct. This is mainly achieved by not allowing to store incomplete mappings for actions.

Nevertheless, the second case involving static values renders choosing between mappings impossible. Let us assume that the mapping $key1_A$ to $key3_B$ is altered such that $key1_A$ is set to $true$ at $t=1$. Then the mapping suggestion algorithm for operation mappings returns the paths $key1_A \mapsto true$ and $key1_A \mapsto key5_C \mapsto key7_D$. However, this static value, which can be set using a valid JSONata expression, is specific to the initial integration context (i.e., from interface A to interface B) and only set if provided the required interface exposes fewer identifiers

9. Safeguarding Expected Method Benefits

than the other. Therefore, mapping chains that include a static value mapping are dropped by the algorithm. In the example depicted in Fig. 9.3 this would mean that only the mapping $key1_A \mapsto key5_C \mapsto key7_D$ is returned as a result. If only static value mappings exist, no mapping would be returned, potentially resulting in an additional mapping to be defined.

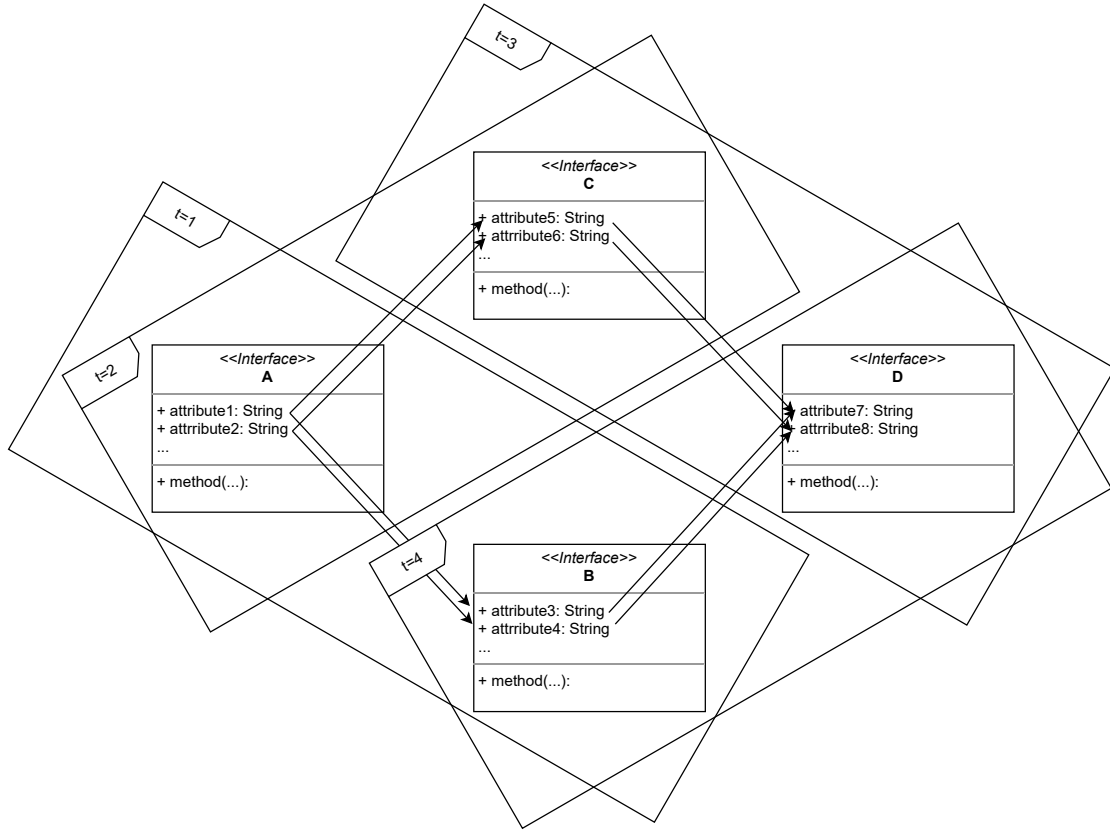


Figure 9.3.: Ensuring Mapping Reliability

The presented measures ensure high reliability regarding manual and calculated mappings. Therefore, we focus on performance metrics, such as integration time and algorithmic performance in the upcoming evaluation chapter.

However, we acknowledged that reliability must be ensured. In case of unexpected issues, we assume that these issues are solved out of band. We will further discuss this topic in our future work section.

Part V.
Evaluation

10. Preliminaries

This chapter provides three empirical experiments to evaluate the proposed method and one evaluation setup to illustrate the implemented algorithms' performance. Each evaluation has been carried out with different groups of students. Each evaluation was part of a university course that had to be completed by the students on their way to obtaining their degree. In the previous section we use examples to illustrate the different problems that can be solved by our reference implementation "How can we make integration knowledge that is captured in imperative software adapter reusable". The following experiments tackle the second research question, "How well can we manage incomplete integration knowledge?" and provide evidence regarding the proposed integration knowledge management process's efficiency.

- Evaluation 1 deals with the reconfiguration of home automation platforms. Based on a distinct infrastructure, we evaluate KDAC based on sensor values. Here semantics for data channels of IoT devices are changing.
- Evaluation 2 extends data channels to services as illustrated in listings 7.2, 7.3 and 7.4. The infrastructure, as presented in the previous chapter, is now applied. Within this evaluation, we focus on comparing KDAC to the integration methods software adapter implementation (see SW method 3.1) and bottom-up (see BU method 3.5) using an ontology. Thereby the application of reasoning principles is evaluated. We do not include TD methods in this comparison as extending a standard or selecting the right standard for an integration case is a challenge itself.
- Evaluation 3 extends the infrastructure used in evaluation 2 by adding adapter generation to it. We focus on comparing KDAC with software adapter implementation in Node.js.
- Evaluation 4 illustrates the implemented reasoning algorithm's performance for operation composition (see algorithm 1 using a fat client and thin client setup. Thereby, we provide evidence on our solution's usability when the knowledge base contains many nodes and edges.

11. Empirical Evaluation 1: Mapping Generation for Sensor Values

In this experiment, we provide our first evaluation of the KDAC method [9]. Therefore, we manipulate the data channels of a home automation platform at run time. Please note that in this first experiment, we only look at identifiers. As presented in the running example, services are not respected at this maturity stage (see Fig. 1.3). The central evaluation goal is to compare manually configuring a home automation platform to using the KDAC method. As an effort indicator, we measure the integration time to evaluate if specifying integration knowledge in addition to reconfiguration activities pays off over time.

11.1. Evaluation Setup

We designed a controlled experiment based on a within-subject evaluation design. Therefore, we used well-known design principles for empirical studies in software engineering [59, 60]. Two groups of students were formed, and these groups competed against each other [61]. By assigning students to one group, it was ensured that they were balanced in terms of experience and knowledge. The control group could not reuse interface mappings, but the experimental group could. This means that the control group had to reconfigure the underlying automation rule manually. However, if no mapping was found, they had to reconfigure the system (i.e., perform the control group's task) and specify a mapping using JOLT [62] as well. JOLT is a JSON to JSON transformation library where the specification for the transformation itself is a JSON document. JOLT acts as the mapping language L^* in this experiment.

Challenge: Our goal is to provide evidence that, over time, reusing interface mappings formalized based on concrete use cases speeds up integration tasks. Hence, additional specification effort should pay off regarding system reconfiguration time.

Participants: We conducted the experiment within a project cooperation between a German and a Romanian university. All students studied within an informatics related profession and can either speak English, German, or Romanian. Overall, 15 students participated in the evaluation.

- Seven students are currently pursuing their master studies, and eight students are pursuing their bachelor studies
- Four out of 15 students own IoT devices, and two out of 15 have already been involved in IoT software development projects
- Four out of 15 students have read about home automation platforms like openHAB [21], and three have already worked with If-This-Then-That (IFTTT) rules

11. Empirical Evaluation 1: Mapping Generation for Sensor Values

- Five out of 15 students knew the IoT-related protocol MQTT [63]

Experiment Scope: We only look at identifiers. Services, as presented in the running example, are not part of the component model. However, building up mapping chains for identifiers is already supported.

Metrics: We measure the time in seconds per home automation rule until the correct data channels are found. Furthermore, we measure the time for additional specification creation and the time for specification reuse.

Hypothesis: The independent variable is either determined by using the conventional approach (i.e., configuring the platform each time a new IoT device enters the environment) or by using the KDAC method. The dependent variable is the required time for solving integration tasks [60]. We suspect that component integration time is higher when reusing integration knowledge than manually configuring the home automation platform each time a component changes.

Technology Stack: As a home automation platform we chose openHAB [21]. OpenHAB exposes a management API that can be accessed at run time to manipulate the platform configuration using HTTP/JSON calls. All data channels have been designed based on publicly available adapter repositories (e.g., <https://www.openhab.org/addons/>). Devices and their values are simulated within the openHAB platform using the built in scripting engine.

For specifying interface mappings in a declarative style, we use JOLT [62]. Interface mappings are stored in a MongoDB database. The user interface is implemented using the Java Swing framework and deployed as a standalone JAR file.

11.2. Evaluation Execution Process

Assume that there exist multiple software component interfaces that were developed by independent software vendors.

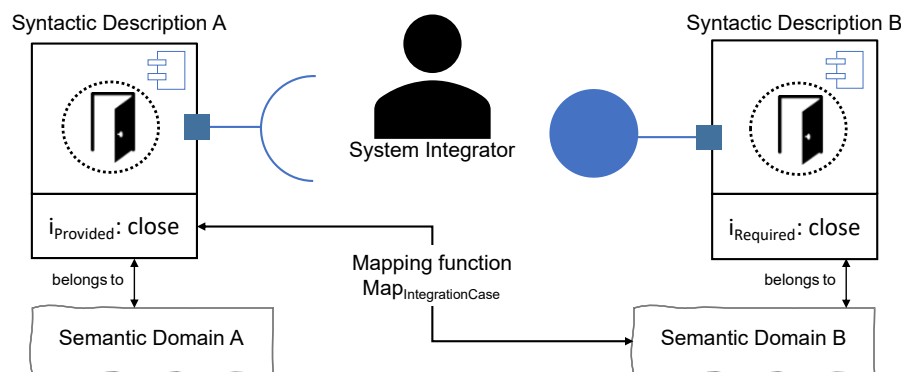


Figure 11.1.: Semantic Interoperability Example for a Home Automation Platform

An integration effort at the semantic level between provided and required software component interface exists when domain models are used by heterogeneous parties A and B (see Fig. 11.1). This is mainly because the device developer determines the concrete syntax based on a self-created semantic domain S for each interface element at component design time. The semantic

11. Empirical Evaluation 1: Mapping Generation for Sensor Values

domain is imagined (e.g., *Open door in the living room*), and the identifier is named accordingly (e.g., *close*). At component integration time, a system integrator can identify a mapping to map two syntaxes, each from one distinct vocabulary. This mapping does not only take place on the syntactic but also on the semantic interface level. Depending on the use case, *close* can mean to *undo open* or *close in proximity*. This semantic integration knowledge is codified into the respective software adapter by defining a mapping or by not defining one.

11.2.1. Evaluation Steps

The storyline presented to the students was the following:

A new automation rule has been downloaded to your home automation platform. However, the rule is not working as other devices have been initially used. Your task is to replace all data channels until the graphical state visualization provided by the home automation platform of each device is acting accordingly to the meaning conveyed by the displayed automation rule. Overall, all participants worked on six automation rules. As an example for the experimental group, one automation rule was:

Task 1: Find the correct item for the rule *Turn on Heating* based on the data channels

1. *Living_Heating*
2. *Heating_Living*
3. *Heating_GF_Living*

on the ground floor.

If the correct item is found, select (1) *Living_Heating* from the Remote Item Panel and create the mapping specification to the working item. Hence, the automation rule was initially configured with the data channel *Living_Heating*, but the device that provided this data channel was no longer available. The other automation rules exposed a similar structure.

The participants were instructed to follow the given order of data channels replacements and to then perform the following loop:

- *Configuration Time*: Configure the next data channel from the task (see 1 in Fig. 11.2) and export it to the connected home automation platform (see 2 in Fig. 11.2).
 - The experimental group could also directly select a correct data channel if a mapping was retrieved from the knowledge base (i.e., indicated by a green background). Hence, existing mappings were automatically evaluated, and the necessary reconfiguration calls to the management API of the platform were generated, but the students had to trigger their invocation manually. Otherwise, they had to stick to the data channel order from the task.
- *Testing Time*: Next, the participant switched to the home automation platform user interface and executed the adapted rule. The respective device state icon was then inspected if the desired action had been executed (e.g., the heating icon label switched its status from OFF to ON). If the Item changed its status according to the rule, then the task is solved. If not, the next Item had to be tested.

11. Empirical Evaluation 1: Mapping Generation for Sensor Values

- *Specification Time*: As soon as the correct item was found, then a mapping specification had to be created based on a template (experimental group only). Therefore, the students had to select the initial remote item (see 3 in Fig. 11.2) and had to formalize the mapping in the text editor (see 4 in Fig. 11.2)

Finally, all created mapping specifications were stored in the knowledge base (see 5 in Fig. 11.2) and were available to other students that were assigned to the experimental group.

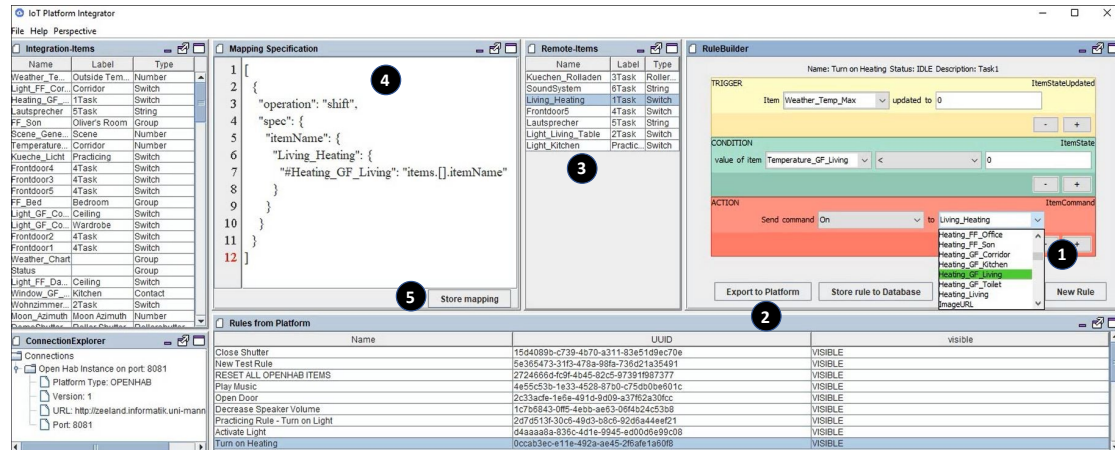


Figure 11.2.: Eval 1 – Evaluation Steps

11.3. Implementation

The main architectural components are depicted in Fig. 11.3. These components have the following functionality:

Formalization Editor: For specifying interface mappings in a declarative style, we use JOLT [62]. An example of a declarative JOLT specification can be seen at number four in Fig. 11.2.

Smart Home Platform: We used openHAB [21] as a home automation platform. OpenHAB can syntactically integrate various IoT components out of the box by providing over 200 adapters from heterogeneous device manufacturers. In addition, openHAB can be accessed using a REST-like interface to manipulate home automation rules, data channels offered by the IoT devices, a rule execution environment, and a user interface to monitor all devices' state.

Knowledge Base: The Knowledge Base stored formalized mappings in JOLT. Here, a graph-based structure was implemented where each node represents a data channel, and each edge represents a mapping specification. This allowed for calculating new mappings (e.g., traversing the graph from a required to a provided interface to identify transitive relationships).

Component: A component is an IoT device which was connected to the platform by using its interface. The platform provided the required software adapter that made all data channels syntactically available within the platform.

User Interface for Formalization Editor: Automation Rules within the IoT context often follow the IFTTT structure. This structure also holds for openHAB. For example, the automation rule

11. Empirical Evaluation 1: Mapping Generation for Sensor Values

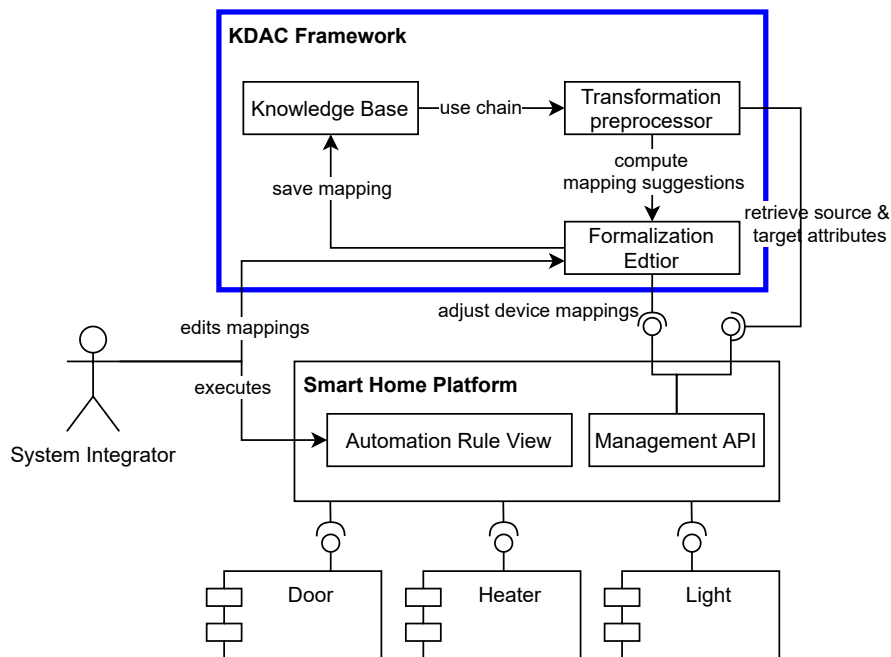


Figure 11.3.: Eval 1 – High-Level System Architecture

Turn on Heating contained a trigger, a condition, and an action (see *RuleBuilder* panel in Figure 11.2). Each rule part contained a drop-down menu where all available data channels provided by the connected devices were listed. All currently provided data channels were displayed in the *Integration Items* perspective, and all required items for one rule were displayed in the *Remote Items* panel.

If no mapping for a data channel existed, a mapping must be created by the system integrator in addition to reconfiguring the automation rule (i.e., adapting the software adapter over a user interface). The operation *shift* displayed in the *Mapping Specification* panel is part of the JOLT language.

The main effect of applying the proposed method was helping to achieve automated adaptability in software ecosystems. In particular, we focused on *engineered adaptability* and *evolutionary adaptability*. Engineered Adaptability as the platform was now able to reconfigure itself during run time and executed context-sensitive automation rules. Evolutionary adaptability as new components could be integrated manually by the system integrator.

11.4. Results

The main reason for carrying out an empirical evaluation was the trade-off between the cost of having additional formalization effort for automated data channel replacement and the benefits of reusing interface mappings for configuring the home automation platform (i.e., software adapter generation).

11. Empirical Evaluation 1: Mapping Generation for Sensor Values

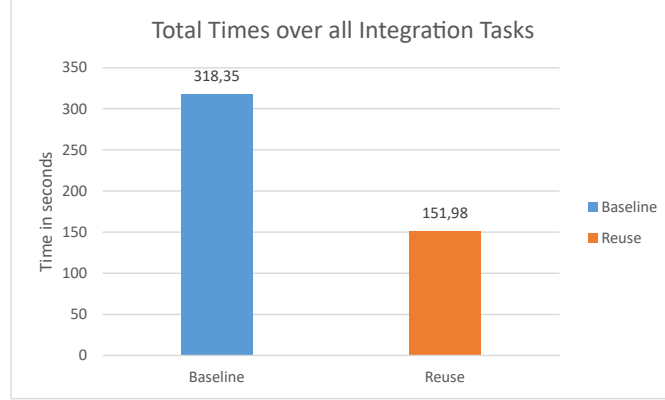


Figure 11.4.: Eval 1 – Reuse Task Time Comparison for Both Groups

Each task involved different amounts of item replacements. Only tasks 1, 3, and 5 were equipped with mapping specifications for the experimental group. Figure (see Figure 11.4) shows the total times for all reuse rules that were measured.

Therefore, we first calculated the total time in seconds for the rule using 11.3 for the control group and 11.4 for the experimental group. Here, Y returned the average time in seconds and X was the amount of replacement operations per rule as an integer. Then the sum of each rule result per group was calculated.

$$X = \text{AmountOfReplacementOperationsPerRule} \quad (11.1)$$

$$Y = \frac{\text{TotalAmountOfReplacementOperations}}{\text{TotalTimeForAllRules}} \quad (11.2)$$

$$\text{TimeControlGroup} = X * Y \quad (11.3)$$

$$\text{TimeReuseGroup} = \frac{(X - 1) * Y + \text{ReuseTime}}{X} \quad (11.4)$$

Figure 11.5 displays the averages and variances per rule in more details for both groups. Here, *Baseline* refers to the sum of *Configuration Time* and *Testing Time* for the control group. *Specification Time* means the time to create a mapping specification in JOLT (experimental group only). *Specification Time* only occurred once to one student of the experimental group. This specification then influenced the *Configuration Time* for the next student in the experimental group as it was automatically evaluated. *Testing Time* was almost identical for all students and groups.

If a green data channel (i.e., indicated by a green background as depicted in Figure 11.2) was present, measured *Reuse* times also referred to the sum of *Configuration Time* and *Testing Time* for the experimental group. However, the difference between the control group and the experimental group was the number of replacement operations. For instance, assume that 5 data channels must be tested in the given order. Furthermore, data channel 4 is the correct one, and there exists a mapping from data channel 1 to 4. The control group must then perform three

11. Empirical Evaluation 1: Mapping Generation for Sensor Values

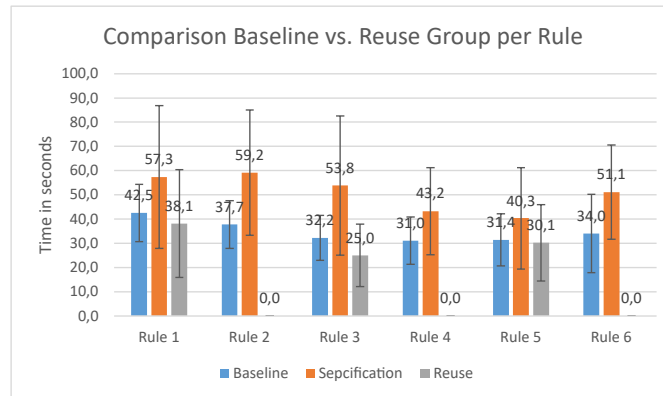


Figure 11.5.: Eval 1 – Integration Times Per Automation Rule

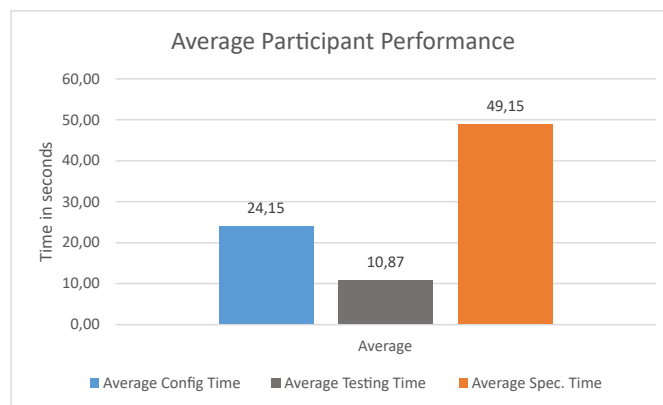


Figure 11.6.: Eval 1 – Average Participant Performance

replacement operations (i.e., 1-2, 2-3, 3-4), and the experimental group must perform one replacement operation (i.e., 1-4).

Overall, 211 item replacement operations were measured (i.e., Average Config Time) and tested within the home automation environment (i.e., Average Testing Time). For specification tasks, we measured 54 runs. On average, configuration time lasted 24 seconds, testing time lasted 11 seconds and specification time 49 seconds (see Figure 11.6).

Figure 11.4 suggests that the experimental group (i.e., reuse) was faster than the control group (i.e., baseline). Hence, our initial claim to outline the applicability of KDAC within an IoT software ecosystem was fulfilled. However, the point at which specification effort payed off over time can only be estimated (i.e., based on the metrics in Figure 11.6).

11.4.1. Break-Even Analysis

Time is a critical factor for KDAC as the additional specification overhead must pay off over time. The proposed method should be faster than implementing the same adapter all over again.

11. Empirical Evaluation 1: Mapping Generation for Sensor Values

Therefore, we provide a forecast based on the collected data to illustrate when the break-even point in contrast to software adapter implementation is reached in theory.

Within this first experiment, the extrapolation is influenced by how many replacement operations must be performed manually (i.e., the provided list of potential data channels from the task description) and the repetition of the same integration task. A mapping specification speeds up the selection process of the correct data channel. For example, if there exists a mapping from the first data channel *Living_Heating* to the correct data channel *Heating_GF_Living*, then two manual steps can be skipped. This also means that this integration context must occur at least twice. Otherwise, the manual adaptation process is always faster as it does not require any mapping specification.

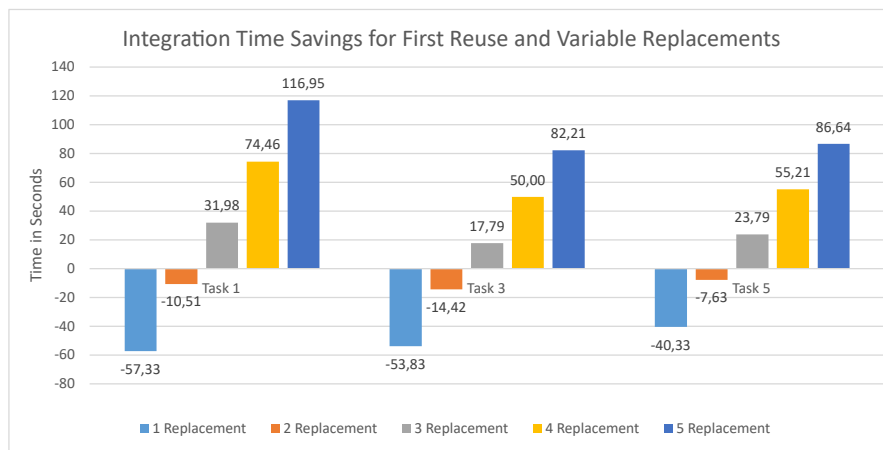


Figure 11.7.: Eval 1 – First Integration Knowledge Reuse with Variable Channel Replacements

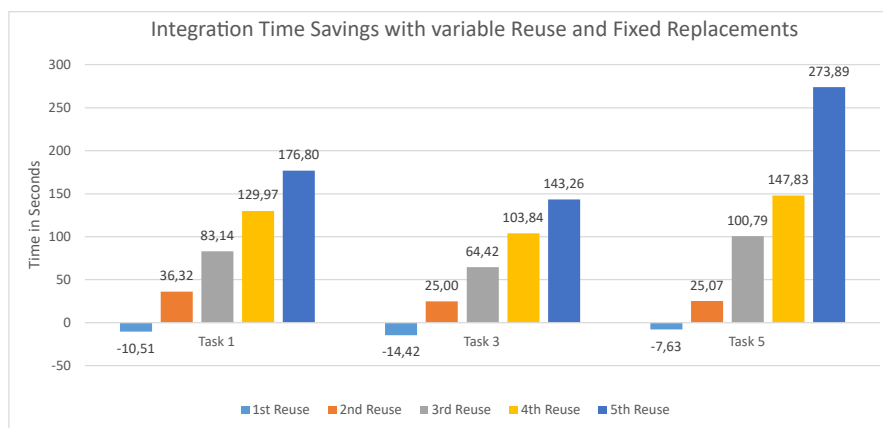


Figure 11.8.: Eval 1 – Two Replacements with Variable Integration Knowledge Reuse

For the extrapolation, we assume that either the data channel replacements (i.e., the number of relevant data channels) are variable (see Fig 11.7) or the amount of reuse scenario of the formalized mapping (see Fig. 11.8). Regarding the variable data channel replacements, we only

11. Empirical Evaluation 1: Mapping Generation for Sensor Values

considered the first reuse of the formalized mapping. Here, we can see that on first integration knowledge reuse, at least three replacements tasks must be made in order to compensate for the additional formalization time at first reuse of specified mappings.

Regarding the variable reuse scenarios, we only consider tasks with two data channel replacements. This number was chosen as it represents the average data channel replacements from the experiment's integration tasks. We can see that the additional formalization effort for creating a data channel mapping is caught up after the second time the integration knowledge is applied.

Please note that reasoning in this extrapolation was only applied to data elements. Furthermore, reverse mappings were not created and evaluated at all.

11.5. Threats to Validity

There are threats to internal validity and to external validity.

Internal Validity: Our evaluation targeted the causal relationship between either reconfiguring the home automation platform each time a new IoT device enters the system or using KDAC (independent variable) and reconfiguration time (dependent variable). However, the presented results provide one distinct result for one concrete implementation that may be subject to change in another run. This is mainly due to confounding factors (e.g., User Interface Design). Furthermore, the evaluation design ensured the early applicability of data channel mappings. In large scale engineering projects it is unclear when and how often such mappings can actually be reused.

External Validity: The population size is too small to be generalized from. Hence, we cannot say whether the presented results are statistically significant. The respective variances strengthen this circumstance. However, the empirical evaluation, the presented architecture, and the technical implementation illustrate how the novel engineering method KDAC can be applied in a home automation system.

Furthermore, we could not spot any link between students that had already worked within the field of IoT or and students that had worked on integration tasks and their performance. All students were able to finish all integration tasks. This may be interpreted as a hint that the approach and the tooling were applicable without further training.

12. Empirical Evaluation 2: Mapping Generation for Services

In this experiment, we shift away from data-driven sensor integration. Therefore, we use the architecture and reasoning principles presented in the previous chapter. The central evaluation goal in this evaluation was to compare the software adapter (SA) implementation method (see Fig. 3.1) with the KDAC method (see Fig. 6.2) and bottom-up methods using an ontology (BU) method (see Fig. 3.5). As an effort indicator, we measured the integration time and the component interaction correctness by counting needed manual adaptations.

12.1. Evaluation Setup

Challenge: It is unclear how the presented engineering methods to produce mappings perform in a simple IoT system. Therefore, we empirically compared the methods applying a within-subject design. SA did not store any mappings, BU evaluated mappings based on URLs to a common namespace, and KDAC transitively chained mappings from previous integration cases.

Participants: The participants studied Informatics at the Bachelor (4 students) or Master level (3 students). All students did not have working experience in using any of the applied technology frameworks or IoT projects in general.

Experiment Scope: As we are interested in the performance of the methods and not in the underlying technology, we chose a technology stack that all methods can utilize. We did not support adapter generation in this version but validated all mappings directly within the web based tool.

Metrics: Method performance is measured in integration time and component interaction correctness by letting seven students perform integration cases. Integration time is measured from *select required & provided interface to finish mapping* for all three methods in minutes (see Fig. 3.1, 6.2 and 3.5). The amount of required manual adaptations measured component interaction correctness after the respective pre-processors had computed all mapping suggestions for an integration case at hand.

Hypothesis: The independent variable was the engineering method. The dependent variables were integration time and component interaction correctness. We suspected that component interaction correctness and integration time was highest using the SA method.

Technology Stack: We relied on the HTTP/JSON component model using POST service calls. For specifying mapping functions $M_{IntegrationCase}$ in a declarative way, we used JSONata [58] (i.e., L^*) and for specifying domain mappings (i.e., $M_{\mathcal{D}}$) we used the JSON-LD syntax [48, 46]. The web-based evaluation prototype supported one-to-one interface mappings. All HTTP/JSON endpoints used had either been extracted from OpenAPI repositories (e.g. <https://rapidapi.com/>) or Smart Home Adapter repositories (e.g. <https://www.openhab.org/>).

org/addons/). OpenAPI does not support any relationships to a machine-readable or machine-understandable domain standard. Then, service instances of the extracted OpenAPI specifications have been created and deployed using MOCKOON and have been mocked with the dummy-json library.

12.2. Evaluation Execution Process

The leitmotif for the students was that a client requests a required server interface (e.g., POST Samsung), but only a semantically identical provided interface instance (e.g., POST LG) is available. Here, semantically identical means that the needed software adapter translates one interface to precisely one other interface. Thus, all request identifiers from the provided interface must be present in the required request, and all required response identifiers must be present in the response message of the provided interface.

A student either described interfaces using JSON-LD or used the available mappings directly. Eight measurement runs using three different integration cases were executed autonomously by the students. The integration contexts were based on the use case "As a mobile application user, I want to control all available devices in my current room by only using one application". The following scope restrictions apply:

For the BU method, the students were instructed to use `dbpedia.org` as a namespace. However, if DBpedia did not include a suitable entity, then `wikidata.org` was allowed to be used as well. This choice was mainly made as these open linked data vocabularies are frequently updated. During describing time, the students only used the service and the ontology. This is similar to describing an IoT component interface at design time using a machine-understandable domain standard (e.g., an ontology).

When selecting the OpenAPI descriptions, it was made sure that each integration context fulfilled the technical one-to-one interface mapping constraint. Furthermore, at least three similar interfaces had to be integrated so that the transitive mapping chain could be computed for the KDAC method (e.g., a music player from Bose, Sony and Sonos). The JSON-LD descriptions, abbreviated with BU-D (where *D* stands for *Description*), have been specified using a text editor. The resulting time to describe an interface is measured independently and is not included in the integration time measurement.

12.2.1. Evaluation Steps

As a first step, the students select the assigned tasks (see 1 in Fig. 12.1). The students mapped request and response by incrementally selecting one key on both sides and then mapping them (i.e., the concrete syntax *C* for the interface description language *L* in Def. 1 is a graphical one). Consequently, a mapping function *map_{action}* in JSONata also has a graphical syntax. For this evaluation, only simple mappings were permitted. A set of mapping function can be tested (i.e., perform a request against a provided service instance) at any time (see 4a, 4b and 4c in Fig. 12.1).

Additionally, the following process restrictions hold:

SA: When a mapping source and mapping target was selected from the Interface Database (see

12. Empirical Evaluation 2: Mapping Generation for Services

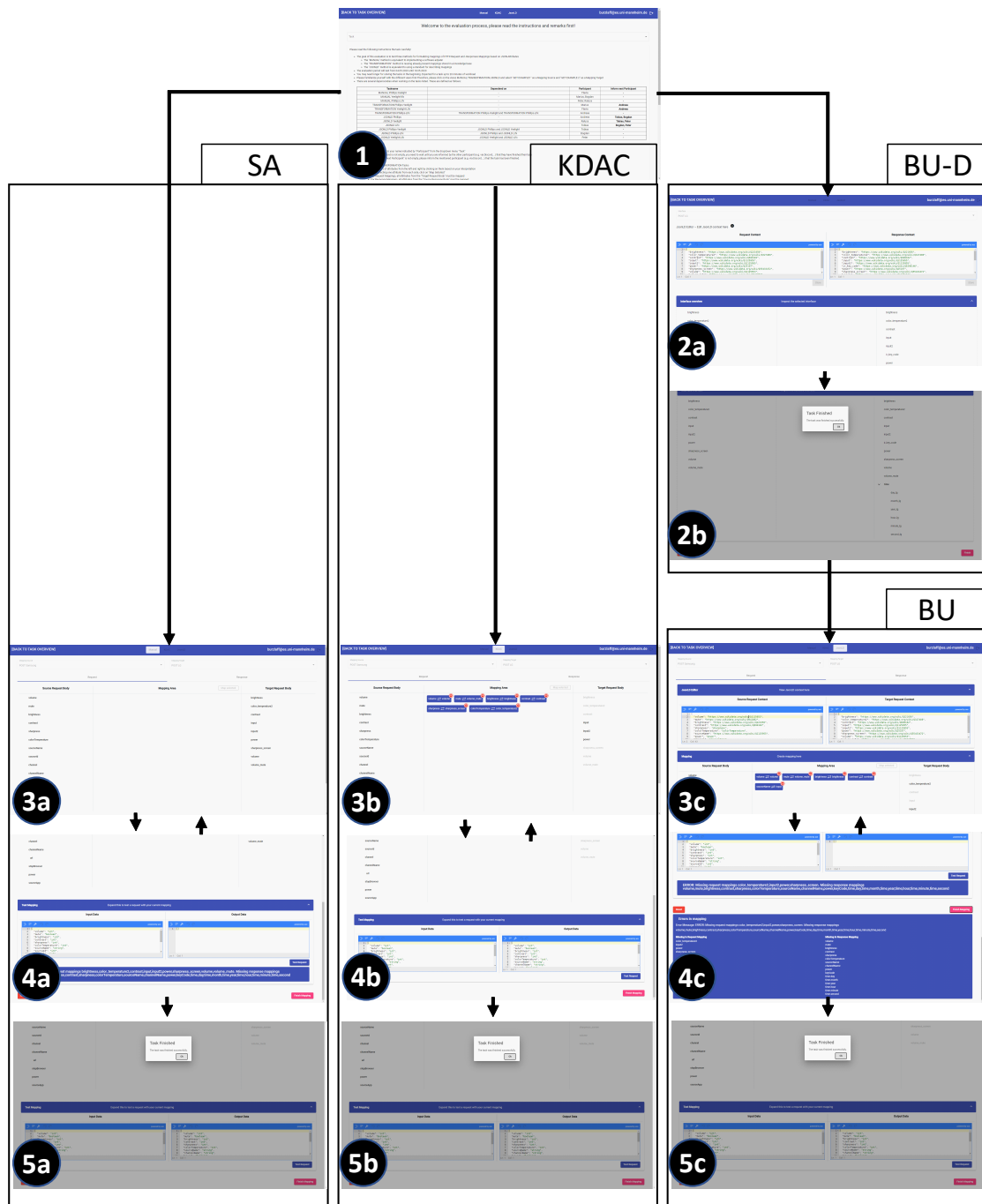


Figure 12.1.: Eval 2 – Evaluation Steps

12. Empirical Evaluation 2: Mapping Generation for Services

Fig. 12.2), then no mappings were shown at any time. This is a proxy to implementing a software adapter for a use case using a textual programming language. Hence, all mappings have to be inserted manually using the graphical syntax (see 3a in Fig. 12.1).

BU: When a mapping source and mapping target was selected from the Interface Database, then the associated JSON-LD descriptions (see Fig. 7.1 for an example) form the JSON-LD KB (see Fig. 12.2) were evaluated by the *JSON-LD preprocessor*. Suppose two keys were linked to the same entity in one namespace (i.e., identical URL). In that case, a mapping suggestion is automatically inserted in the *Mapping Area* (see 3c in Fig. 12.1). These JSON-LD descriptions have been created by the students beforehand (see BU-D lane in Fig. 12.1). Therefore only one interface at a time was presented to the students (see 2a in Fig. 12.1). Hence, the students did not know the integration context (i.e., the concrete required and provided interface to be mapped) but interpreted the interface based on a domain standard (i.e., DBpedia or wikidata). This was done to simulate a new integration case based on an ontology produced by a BU method. Only a syntactic validation was performed before the description is saved (see 2b in Fig. 12.1). Different students always performed the respective tasks BU-D and BU.

KDAC: When a mapping source and mapping target were selected from the Interface Database, then a breadth-first search on the *Transformation KB* by the *Transformation preprocessor* was performed (see Fig. 12.2). All computed mapping functions based on algorithm 1 for the source (i.e. required) and target (i.e. provided) interface were automatically inserted in the *Mapping Area* (see 3b in Fig. 12.1).

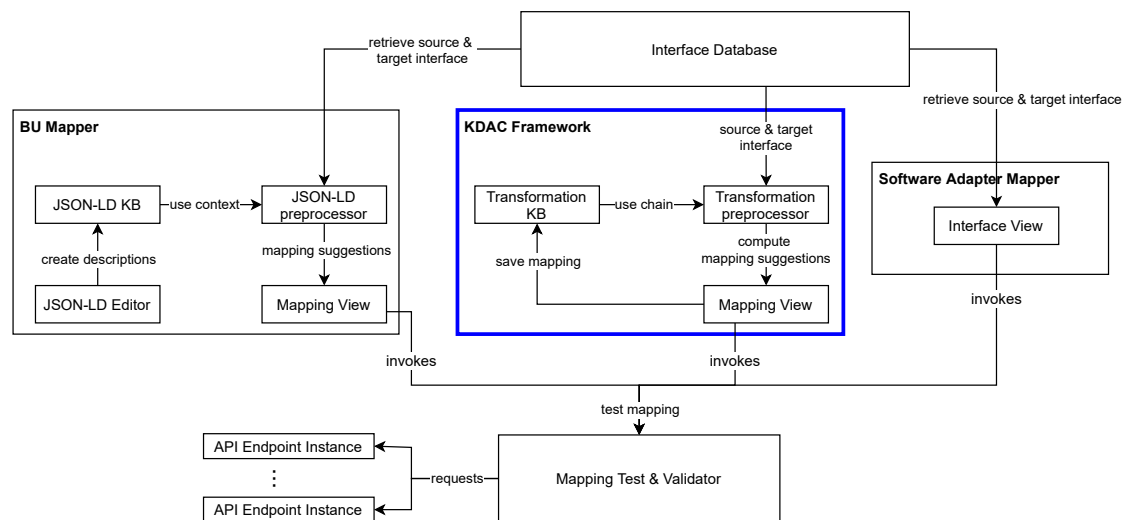


Figure 12.2.: Eval 2 – High-Level System Architecture

A is finished if the request to a required service was successfully transformed to the request of a provided service instance and vice versa for the respective response (see 5a, 5b, and 5c in Fig. 12.1). Generating a software adapter in an imperative language (e.g., Adapter Pattern implemented in Java) was not part of this architecture as we used an extension point for JSONata to integrate it into the web front end. Thus, simple transformations could be executed and tested.

12.3. Implementation

The overall system architecture was built up of three main parts, which are responsible for generating the interface mappings. In addition, a generic *Mapping Test & Validator* for testing the created mappings was implemented (see Fig. 12.2). Depending on which type of method was used, different preprocessors might be applied. Their task was to populate the *Mapping View* with automatically created suggestions of mapping key pairs.

In the case of the KDAC method, the *Transformation pre-processor* was invoked (see presented algorithms for details). To recap, it first tries to find a transitive mapping chain between the selected source and target interface using the breadth-first search on the *Transformation KB*. If the search was successful, a linked list of *Mappings* was returned. In this list, the first mapping source is equal to the selected source interface (i.e., POST Samsung), and the source of each subsequent mapping is equal to the target of the previous one. Finally, the last mapping target is equal to the selected target interface (i.e., POST LG). Once such a chain was identified, the preprocessor recursively applied the mappings stored in JSONata to each other, producing a final mapping from the source to the target interface (i.e., POST Samsung \rightarrow POST LG). This was done for both the request and the response data.

Algorithm 4 Create Mapping Suggestion with Ontology

```

0: procedure BUILDMAPPINGSUGGESTIONS(sourceInterface, targetInterface)
0:   expSourceRequest  $\leftarrow$  expand(sourceInterface.jsonLdRequestDefinition)
0:   expTargetRequest  $\leftarrow$  expand(targetInterface.jsonLdRequestDefinition)
0:   requestMapping  $\leftarrow$  new Mapping()
0:   for each Property requiredProp  $\in$  expTargetRequest do
0:     for each Property providedProp  $\in$  expSourceRequest do
0:       requestMapping.add(providedProp, requiredProp)
0:     end for
0:   end for
0:   expSourceResponse  $\leftarrow$  expand(sourceInterface.jsonLdResponseDefinition)
0:   expTargetResponse  $\leftarrow$  expand(targetInterface.jsonLdResponseDefinition)
0:   responseMapping  $\leftarrow$  new Mapping()
0:   for each Property requiredProp  $\in$  expSourceResponse do
0:     for each Property providedProp  $\in$  expTargetResponse do
0:       responseMapping.add(providedProp, requiredProp)
0:     end for
0:   end for
0:   return requestMapping, responseMapping
0: end procedure=0

```

The preprocessor for the BU method followed a different algorithm. It first used the *expand* functionality of the underlying JSON-LD interface description *C* which was stored in the *JSON-LD KB*. Afterwards, it traversed both expanded interface definitions and tried to find keys that were linked to the same entity in one namespace. Once a match was found, it created a mapping suggestion for the two keys. This simple algorithm is depicted in Fig. 4.

Once a mapping was completed, it can be tested and validated using the *Mapping Test & Validator*. This component tested statically whether all required source and target parameters were

12. Empirical Evaluation 2: Mapping Generation for Services

mapped and it dynamically executes the mapping against the provided API Endpoint instance. Hence, the system integrator could ensure that the resulting mapping performs the expected transformations.

12.4. Results

In order to validate our hypothesis, 54 one-to-one interface integration tasks were captured. There are 18 integration tasks for the Software Adapter Implementation method (see Fig. 3.1), 18 for the bottom-up engineering method (see Fig. 3.5), and 18 for the KDAC method (see Fig. 6.1).

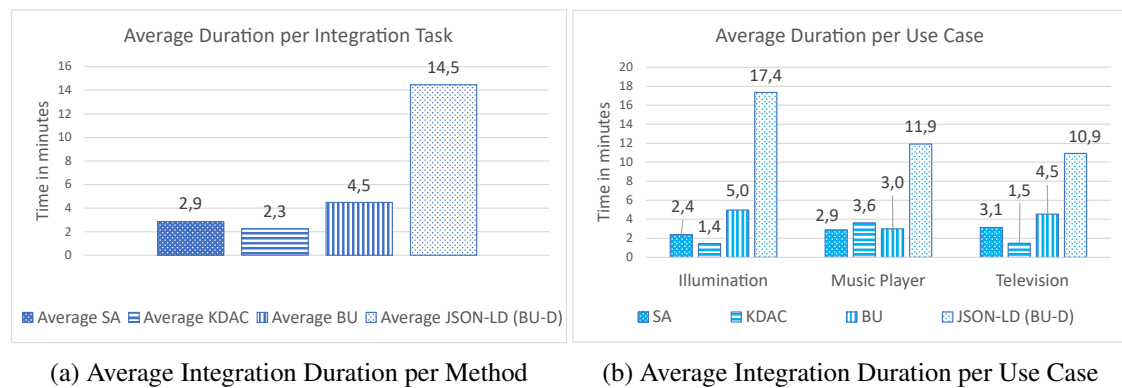


Figure 12.3.: Eval 2 – Integration Time

Fig. 12.3a illustrates the average integration time per engineering method and Fig. 12.3b the average integration time per use case. On average, the participants needed 14.5 minutes to create JSON-LD specifications (i.e., BU-D in all figures), which were required by the BU method using ontologies. Although the time for creating the interface descriptions was not included in the method result comparison, the BU method took, on average, two and a half minutes longer to complete than KDAC. KDAC was the fastest method except for the integration context *Music Player*. As the use cases exposed different amounts of JSON keys, the average//variance statistics for a normalized use case with 59 keys per task were 2.9//0.01 minutes for SA, 2.3//0.1 minutes for KDAC, 4.5//0.1 for minutes BU, and 14.5//2.2 minutes for BU-D.

For the BU method, the number of manual adaptations required after algorithms 1 produced the suggestions can be seen in Fig. 12.4a. Again, percentages are used as the interfaces contain different amounts of keys. A common theme for the BU method was using different namespaces during execution, as these were only partially restricted. This resulted in no automated mapping suggestions as the algorithm for BU cannot cope with URLs between different namespaces. However, when sticking to one namespace only, the percentage for manual adaptations only decreased about 21%.

Among others, the reasons for the manual adaptations for BU were the following: 1) Incomplete interface description stored by the student; 2) URLs pointing to type property, datatype or resource instead of entity; 3) Missing equivalence relationship in namespaces (e.g., Entities

12. Empirical Evaluation 2: Mapping Generation for Services

currentTitle and *Music Track*); 4) References to a complete ontology rather than an entity; 5) Combined key names such as *sourceID* are linked to only one entity; 6) Wrong URLs in interface descriptions (e.g., entity *Track* refers to railway and music).

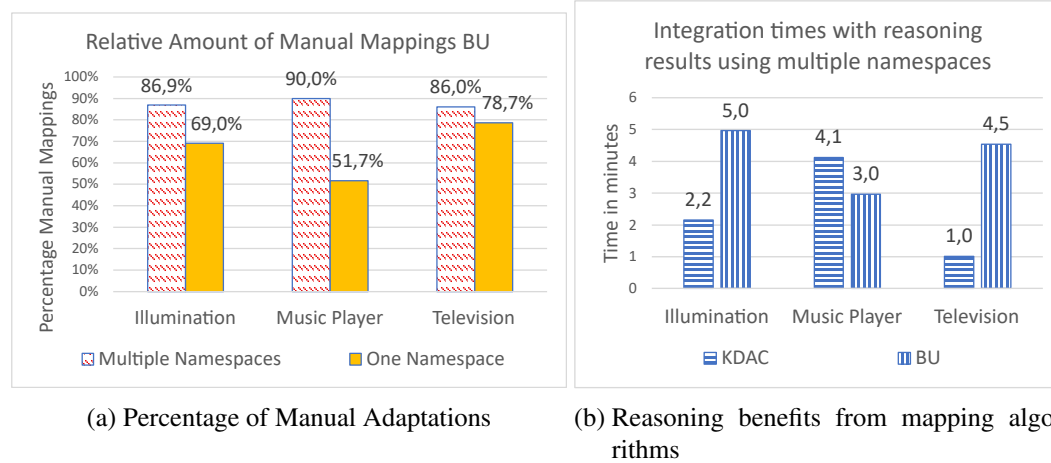


Figure 12.4.: Eval 2 – Component Interaction Correctness

For the KDAC method, no manual adaptations were needed whenever a transitive relationship was calculated. As the SA method deliberately lacked a suggestion algorithm, no mappings were automatically generated. However, time savings could be calculated whenever an algorithm produced a non empty result (i.e., JSON-LD descriptions and JSONata mappings for an unseen combination of provided and required interface). Fig. 12.4b outlines that the KDAC method was quicker (KDAC: 7.3 and BU: 12.5 min in total) and produced no manual mappings when compared to the BU method (see Fig. 12.4a). SA also did not make mapping errors but required more time compared to KDAC. However, no mapping errors for KDAC means that an adapter based on explicitly formalized mappings can be generated automatically, whereas the adapter generated by BU would contain errors.

We suspected that component interaction correctness and integration time was highest using the SA method. Based on the data collected, we can summarize that the SA method and the KDAC method had the highest component interaction correctness (i.e., no manual adaptations required), and integration time was lowest using the KDAC method. The observed errors for BU-D were mostly due to inconsistent vocabularies or outdated interface descriptions. This finding is consistent with the known problem of describing IoT interfaces from literature [1].

12.4.1. Break-Even Analysis

We do not provide a break-even analysis for this experiment. The main reason for this is the evaluation setting where we implemented three approaches (i.e., software adapter implementation, KDAC, and the bottom-up approach) on a common technical infrastructure.

In particular, software adapter implementation is abstracted by a graphical user interface to de-

12. Empirical Evaluation 2: Mapping Generation for Services

fine mappings. The same user interface is also applied within the KDAC approach (see Fig. 13.1). Hence, the additional formalization effort is the same. Consequently, we cannot add the additional formalization effort to the time for implementing the software adapter.

Regarding the BU approach using JSON-LD description, we decided to formulate the descriptions using a textual syntax. Comparing this formalization effort with the graphical syntax to formalize mappings in a concrete integration context is not possible. Due to the presence of errors in the JSON-LD descriptions during mapping generation by the corresponding algorithm (see Algo. 4), a comparison to the BU approach would be tampered.

12.5. Threats to Validity

Overall, the presented evaluation design favors internal over external validity. Hence, confounding factors for the independent variable *engineering method* (i.e. SA in Fig. 3.1, BU in Fig. 3.5 and KDAC in Fig. 6.2) were eliminated as much as possible. Tasks are randomly assigned to the students, but it is made sure that no student works on the same integration task in subsequent measurement runs.

Internal Validity: Describing interfaces with JSON-LD and linking interface elements to a domain standard is error-prone. These errors show that the integration based on ontologies to achieve semantic interoperability is not the fastest one. Naturally, if the JSON-LD descriptions did not contain any errors, all mapping functions would have been generated correctly, and the BU approach would produce no integration time. However, the presented results show that applying and reusing this method in a decentralized development setting shows that the used ontologies expose themselves heterogeneity as entities are defined at different granularity levels. Nevertheless, reusing an existing ontology is perceived easier compared to creating a new ontology for inexperienced students.

External Validity: All participants were able to solve all tasks using the web tool. Nevertheless, generalized statements based on this experiment must be discussed within the following frame: There may be a selection bias as the agile development team consisted of seven students. The representativeness of use cases is ensured by using OpenAPI specifications from external product vendors. However, during OpenAPI interface description selection it was made sure that a transitive mapping chain could be built early in the experiment. This may not hold in practice. Furthermore, it may not always be the case that there is a one-to-one mapping between a set of interfaces. Finally, the evaluation focuses on the engineering method. Therefore, a technology stack is chosen to be embedded into a single web-app to minimize confounding factors (e.g., a long learning curve for the students to use an interface description language). Nevertheless, different technologies and textual usage instead of graphical mapping languages L (e.g., SAWSDL) might have produced other results.

13. Empirical Evaluation 3: Adapter Generation for Services

In this experiment, we illustrate and test the end-to-end application of the proposed method for web services [13]. The participants had to work in two environments. This allowed for editing mappings within the mapping and coding environment. The central evaluation goal was to compare implementing software adapters, generating software adapters without reasoning principles, and generating software adapters with reasoning principles. As an effort indicator, we measured the integration time and the number of mapping errors and discuss problems during software adapter implementation.

13.1. Evaluation Setup

We empirically compared the software adapter implementation method against KDAC applying a within-subject design [64, 65]. SA represents implementing a software adapter. The KDAC method was available in two variants. The first variant involved generating software adapter without any mappings stored in the knowledge base (variant 1). The second variant used mappings stored in the knowledge (variant 2). Hence, we could compare the mapping time and errors made by the system integrator and, if any, made by the reasoning algorithms.

Challenge: It is unclear how well KDAC can assist the system integrator during software adapter (SA) implementation. In particular, the additional time to formalize mappings should result in achieving a working software adapter faster.

Participants: Each week, 3 to 5 integration cases have been assigned to four students. The students studied Informatics at the Bachelor (two students) or the Master (two students) level. All students did not have working experience in implementing software adapters in the given programming language or in IoT in general.

Experiment Scope: As we were interested in the performance of the method and not in the underlying technology, we chose a technology stack that can be utilized by both methods (i.e., SA and KDAC).

Metrics: The quantitative implementation effort was measured in integration time and component interaction correctness. Integration time was measured from starting the integration task until the students finished the mapping in the KDAC tool in minutes. Component interaction correctness was measured by the number of retries needed when the test criterion was not met.

Hypothesis: The independent variable was the engineering method. The dependent variables were integration time and component interaction correctness. We suspected that the component interaction correctness and integration time was highest using the KDAC method (see Fig. 6.2).

13. Empirical Evaluation 3: Adapter Generation for Services

Technology Stack: We relied on the HTTP/JSON component model using POST and GET service calls. For specifying mapping functions in a declarative way, we used JSONata [58], and for implementing the software adapter, we used the Visual Studio Code Web IDE. The web-based evaluation prototype supported one-to-one and one-to-many interface mappings. For implementing the software adapter, we chose NodeJS. A project setup script was provided so that the participants could resolve all necessary dependencies by issuing one command line statement within the Web IDE.

All HTTP/JSON endpoints have been designed based on publicly available endpoints from the OpenAPI repositories (e.g., <https://rapidapi.com/>) or Smart Home Adapter repositories (e.g., <https://www.openhab.org/addons/>).

An integration task was finished if the request to a required service instance was successfully transformed to the request of a provided service instance and vice versa for the respective response. There was a test criterion for each integration task that tells the students whether their mapping was correct or not. In essence, the test criterion contains all identifiers as defined for the required operation and the values as produced by the provided operations. This test criterion was checked every time the student runs the software adapter. If the test fails, a snapshot of the software adapter was stored. This allowed for a qualitative evaluation of the implementation.

13.2. Evaluation Execution Process

The leitmotif for the students was that a client requests a required server interface (e.g., POST Samsung), but only a semantically identical provided interface instance (e.g., POST LG) is available. This means that the needed software adapter translates one interface to one other interface. Thus, all request parameters from the provided interface must be present in the required request, and all required response parameters must be present in the response message of the provided interface.

Six measurement runs using three different use cases were executed autonomously by the students. Again, the integration contexts were illumination, music player, and television (see running example 1.3).

When selecting the OpenAPI descriptions, it was made sure by the experiment conductors that each integration context fulfilled the technical one-to-one interface mapping constraint. Furthermore, at least three similar interfaces had to be integrated so that the transitive mapping chain could be computed (e.g., a music player from Bose, Sony, and Sonos).

Software Adapter: When a mapping source and mapping target were selected, then no mappings were shown at any time. Hence, the students could only continue to generate the software adapter and start imperatively implementing the necessary transformation code.

KDAC: In both variants, a mapping could be tested before generating the software adapter within the KDAC tool (i.e., perform a request to a provided service instance) as soon as all request keys from the provided interface and all response keys from the required interface were mapped.

A search over all stored mappings was performed when a mapping source and mapping target is selected. All computed mapping functions for the source (i.e., required) and targets (i.e., provided) interfaces were automatically inserted and visualized in the KDAC tool. They could be

13. Empirical Evaluation 3: Adapter Generation for Services

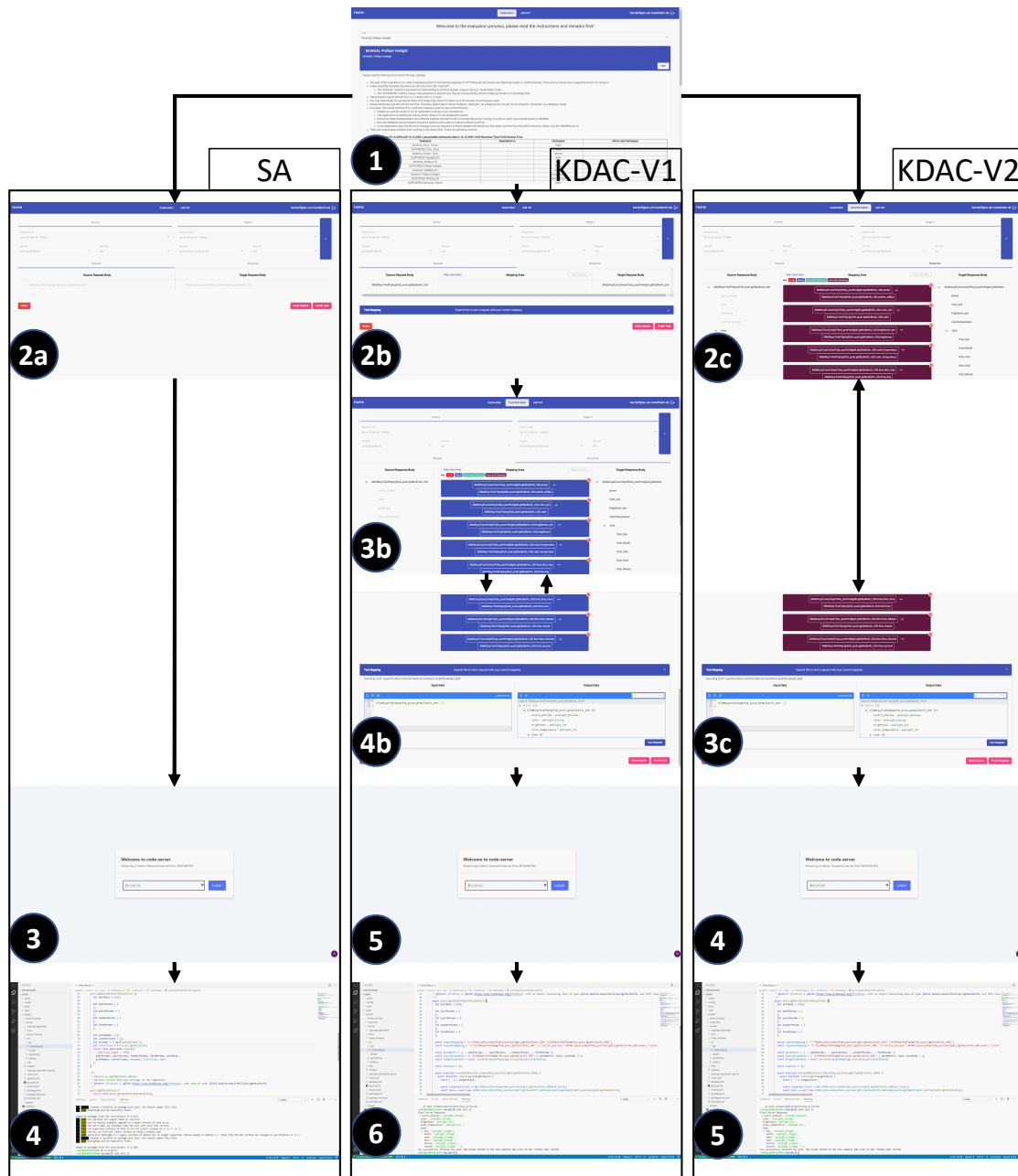


Figure 13.1.: Eval 3 – Evaluation Steps

edited at any time during the evaluation process by writing the correct JSONata syntax. All keys from the selected provided and the required interfaces could be used within one mapping operation.

13.2.1. Evaluation Steps

The students performed the following steps (see Fig. 13.1). First, they selected a task from the task overview (see 1 in Fig. 13.1).

Second, the type of tasks determines how the mappings were populated. If the tool was only used to generate the software adapter project, the students could only do so (see SA lane 2a). If the tool was used to create mappings between operations, the students could specify mappings (see KDAC-V1 lane 2b). This view also symbolizes the first variant of the KDAC method. Suppose there were already mappings within the knowledge base. In this case, the reasoning principles were applied, and the results were populated within the mapping view. They were annotated with a different color than manually specified mappings (i.e., blue) (see KDAC-V2 lane 2c). This view symbolizes the second variant of the KDAC method.

Next, the students could either directly login into the Web IDE (see SA lane 3, KDAC-V1 lane 5, and KDAC-V2 lane 4) or adjust the created mappings until all calls succeed (see KDAC-V1 lane 4b and KDAC-V2 lane 3c). When all calls succeed, then the mappings were stored within the knowledge base, and the students also proceed to login into the Web IDE (see SA lane 3, KDAC-V1 lane 5 and KDAC-V2 lane 4).

Last, the students must resolve all dependencies in the underlying Node.js environment by executing an install script and providing their username and password for authentication towards the knowledge base. If the tool was only used to generate the adapter skeleton, then the method that contained the actual transformations had to be implemented (see SA lane 4).

Suppose the tool was used to formalize mappings or mappings have been computed based on the reasoning principles. In that case, these mappings were inserted into the software adapter code that had to be implemented (see KDAC-V1 lane 6 and KDAC-V2 lane 5). At this stage, the students could not store mappings anywhere. Finally, the students could check anytime if their operationalized mappings were correct by executing a test script (see SA lane 4, KDAC-V1 lane 6, and KDAC-V2 lane 5). If this was the case, then a corresponding message is printed on the terminal, and the students ended the task by switching back to the KDAC tool and by clicking the finish task button.

13.3. Implementation

The overall system architecture was built up of three main parts responsible for generating the interface mappings. In addition, a generic *Mapping Test & Validator* for testing the created mappings was implemented (see Fig. 13.2). Depending on which type of method was used, a preprocessor might be applied. Their task was to populate the *Mapping View* with automatically created suggestions of mapping key pairs.

In the case of the first variant of the KDAC method (i.e., no reasoning principle application), the web tool supporting KDAC only provided a graphical user interface for specifying mappings

13. Empirical Evaluation 3: Adapter Generation for Services

with JSNOata. Here, the web tool was used to generate the software adapter project skeleton so that both approaches were as similar as possible. Hence, the first and second variant only differentiated in whether existing mappings were evaluated or not. In the case of the second variant of the KDAC method, the *Transformation preprocessor* was invoked.

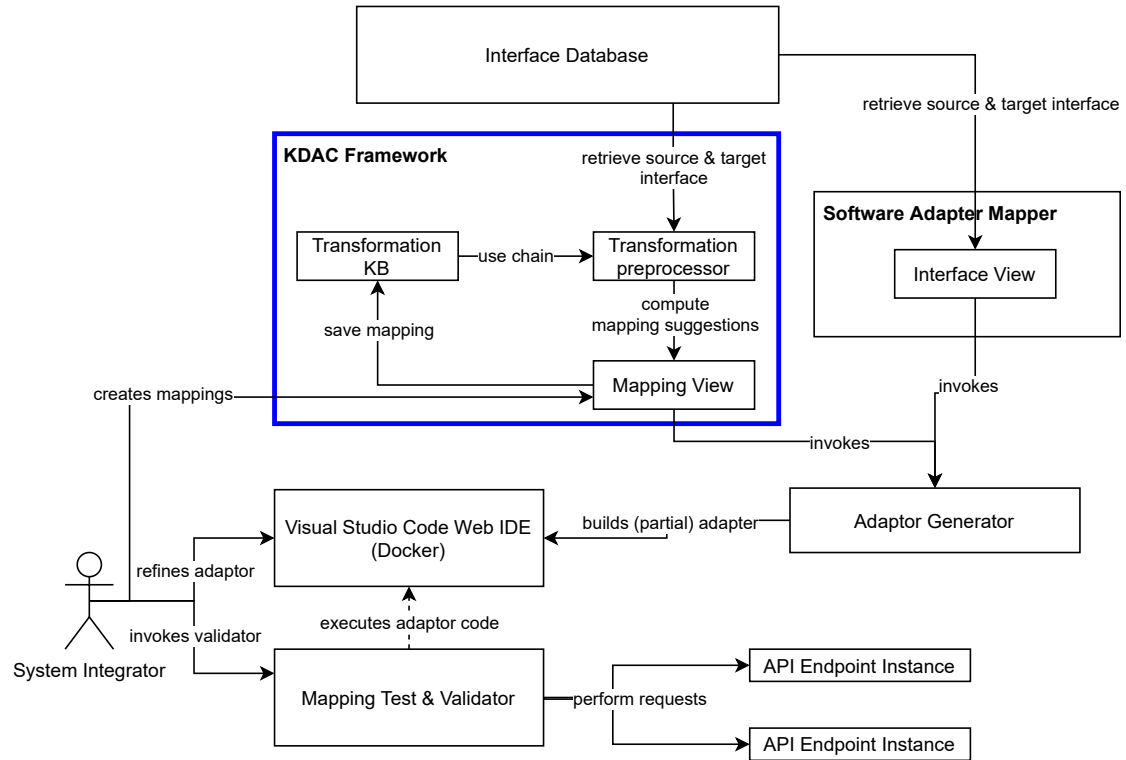


Figure 13.2.: Eval 3 – High-Level System Architecture

A finished example for the adapter to be implemented manually and the adapter generated based on the formalized mappings can be seen in listing 13.1 and listing 13.2.

```

constructor(apiClient) {
  this.apiClient = apiClient || ApiClient.instance;
  this.targetApi = new targetDefaultApi();
}
/**
 * Returns a TvInfo object.
 * Returns current date and settings of the tv.
 * @return {Promise} a {@link https://www.promisejs.org/|Promise}, with an
  object containing data of type {@link module:model/SamsungTvInfo} and
  HTTP response
 */
async postSamsungTvInfoWithHttpInfo() {
  const response = await this.targetApi.getLgTvInfo();
  return {
    data: {
      volume: response.volume,

```


13. Empirical Evaluation 3: Adapter Generation for Services

```
    sourceName: response.input,
    brightness: response.brightness,
    ....
  }
}
}
postSamsungTvInfo() {
  return this.postSamsungTvInfoWithHttpInfo()
    .then(function(response_and_data) {
      return response_and_data.data;
    });
}
```

Listing 13.1: Software Adapter JavaScript Example for Samsung and LG

```
/**
 * Returns a TvInfo object.
 * Returns current date and settings of the tv.
 * @return {Promise} a {@link https://www.promisejs.org/|Promise}, with an
 * object containing data of type {@link module:model/SamsungTvInfo} and
 * HTTP response
 */
async postSamsungTvInfoWithHttpInfo() {
  let postBody = null;
  const response = {};

  const requestMapping = "{\\"X0Ja22GPOer5wuwtT0vKY_postEpsonTvInfo_200\\":
XwJTD81Qno71ubIQw8eV0_postSamsungTvInfo_200}";
  const responseMapping = "{\\"XwJTD81Qno71ubIQw8eV0_postSamsungTvInfo_200
\\":{\\"volume\\":X0Ja22GPOer5wuwtT0vKY_postEpsonTvInfo_200.projectorvolume
,\\"brightness\\":X0Ja22GPOer5wuwtT0vKY_postEpsonTvInfo_200.
picturebrightness,\\"sourceName\\":
X0Ja22GPOer5wuwtT0vKY_postEpsonTvInfo_200.input,\\"...}}";

  const sourceInputData = { "XwJTD81Qno71ubIQw8eV0_postSamsungTvInfo_200":
{ body: postBody } };
  const targetInputData = await jsonata(requestMapping).evaluate(
sourceInputData);

  async function handleX0Ja22GPOer5wuwtT0vKY_postEpsonTvInfo_200() {
    async function callTarget(targetInput) {
      const { } = targetInput;
      const targetApiClient = new
X0Ja22GPOer5wuwtT0vKY_postEpsonTvInfo_200ApiClient();
      const data = await new
X0Ja22GPOer5wuwtT0vKY_postEpsonTvInfo_200TargetApi(targetApiClient).
postEpsonTvInfo();
      return data;
    }
    const {body: targetBody} = targetInputData["
X0Ja22GPOer5wuwtT0vKY_postEpsonTvInfo_200"];
    response["X0Ja22GPOer5wuwtT0vKY_postEpsonTvInfo_200"] = await
callTarget();
  }
}
```

13. Empirical Evaluation 3: Adapter Generation for Services

```

await Promise.all([handleX0Ja22GPOer5wuwtT0vKY_postEpsonTvInfo_200(), ]);
const { "XwJTD81Qno71ubIQw8eV0_postSamsungTvInfo_200": sourceOutputData }
= jsonata(responseMapping).evaluate(response);
return { data: sourceOutputData };
}
}
postSamsungTvInfo() {
return this.postSamsungTvInfoWithHttpInfo()
.then(function(response_and_data) {
return response_and_data.data;
});
}
}

```

Listing 13.2: Generated Software Adapter JavaScript Example for Samsung and LG

13.4. Results

We captured 108 one-to-one interface integration tasks to validate our hypothesis. There are nine integration tasks for the Software Adapter Implementation method, nine for the first KDAC variant, and nine for the second KDAC variant. Each integration task has been repeated four times during the evaluation period.

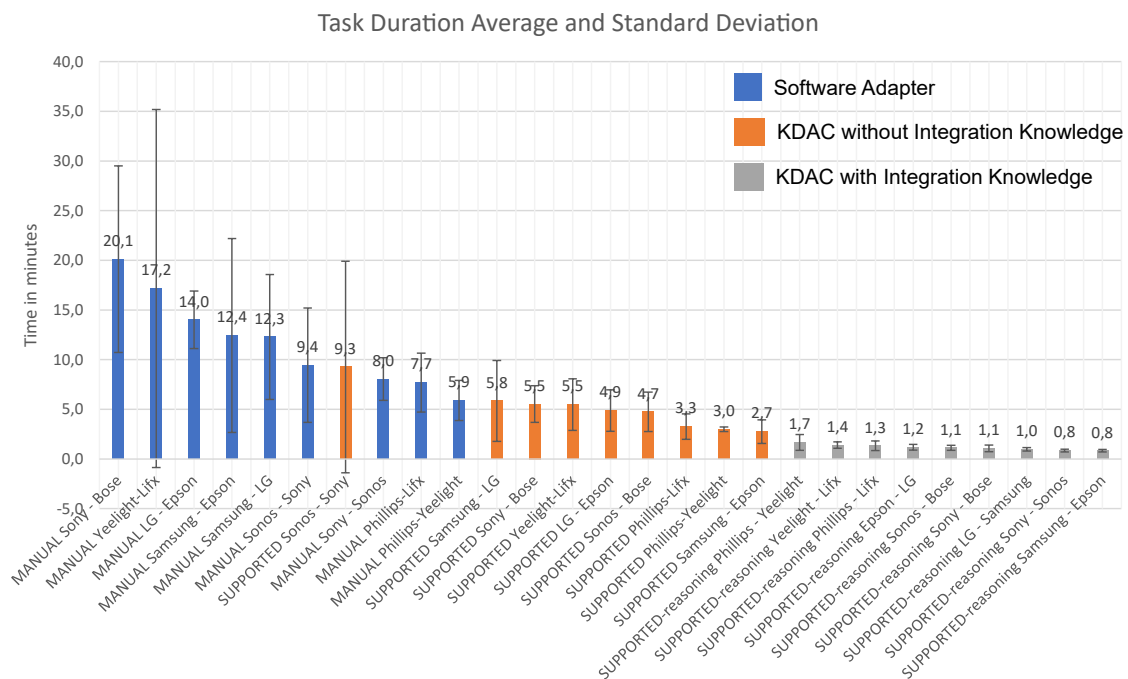


Figure 13.3.: Eval 3 – Average and Standard Deviation for All Integration Tasks

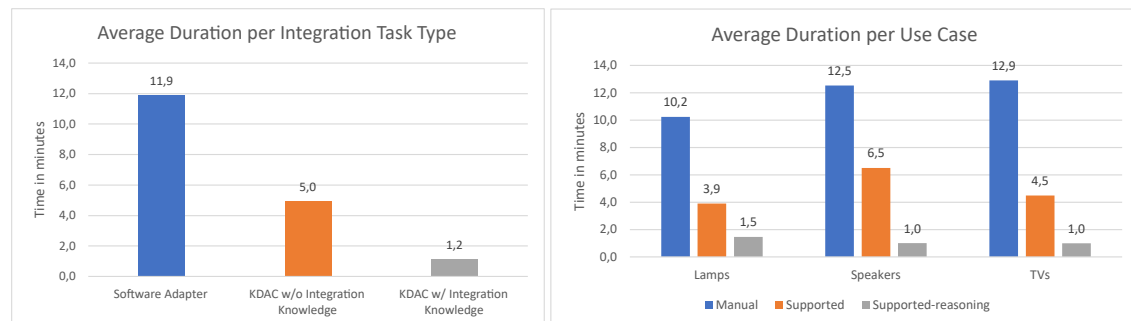
Fig. 13.3 outlines the duration average for all integration tasks. An integration task involved ten to 16 keys that had to be mapped. The integration time was measured in minutes, and the description of each task involved the integration task type. Here, "MANUAL" corresponds to

13. Empirical Evaluation 3: Adapter Generation for Services

only using the tool (see 13.1) as a software adapter generation environment where all mapping logic had to be implemented in the generated adapter project. "SUPPORTED" relates to the first variant of the KDAC method, where mappings between interfaces were defined using JSONata. Last, "SUPPORTED-reasoning" is the second variant of the KDAC method with reasoning and integration knowledge reuse. Mocked devices from Sony, Bose, and Sonos were speakers, Yee-light, Lifx, and Philips are lamps, and Epson, LG, and Samsung are TVs.

Overall, the average time needed for constructing a working software adapter was the highest for implementing software adapters and the lowest when mappings could be reused. Furthermore, the manual task's standard deviation is higher than the second variant of the KDAC method. This is mainly because of the presence or absence of errors during code writing. The number of keys did not directly affect the average integration time as the highest value of 20.1 minutes had 13 keys to be mapped. The first integration task with 16 keys scored an average duration of 14 minutes.

Fig. 13.4a illustrates the average integration time per engineering method and Fig. 13.4b illustrates the average integration time per use case. On average, the participants needed 11.9 minutes to implement a software adapter, 5 minutes to create mappings in the tool and then generate a software adapter, and 1.2 minutes when mappings could be reused. For all integration task types, the same number of keys had to be mapped (i.e., 117 keys). For the three use cases, this equality did not apply. However, this does not necessarily result in higher average integration times. Concerning the traditional software adapter implementation method, the use case with lamps (90 keys) lasted 10.2 minutes, the use case with the speakers (117 keys) lasted 12.5 minutes, and the use case with the TVs (141 keys) lasted 12.9 minutes. The average integration times was highest for the manual integration task types and lowest for the second variant of the KDAC method.



(a) Average Integration Duration per Method

(b) Average Integration Duration per Use Case

Figure 13.4.: Eval 3 – Integration Time

Fig. 13.5a and Fig. 13.5b illustrate the amount of retries from the viewpoints of integration task types and use cases. Naturally, the sum of retries per use case equals the number of retries per integration task type. It can be stated that the errors made were highest for the manual software adapter implementation method and lowest for the second variant of the KDAC method. This circumstance is straight forward as the number of errors possibly made by the students increases

13. Empirical Evaluation 3: Adapter Generation for Services

if no automation is involved (e.g., manually coding a software adapter). Hence, we list the most common errors for each method based on a manual inspection of code snapshots. For the manual method, the most common errors using Node.js were: 1) Missing or wrong keys in the result; 2) result object is undefined; 3) result object is empty; 4) key hierarchy was ignored; 5) key values not correctly assigned; 6) wrong encapsulation of result data; 7) import of the provided interface failed. For the first variant of the KDAC method, the most common error was a wrongly mapped key. No errors have been made for the second variant of the KDAC method.

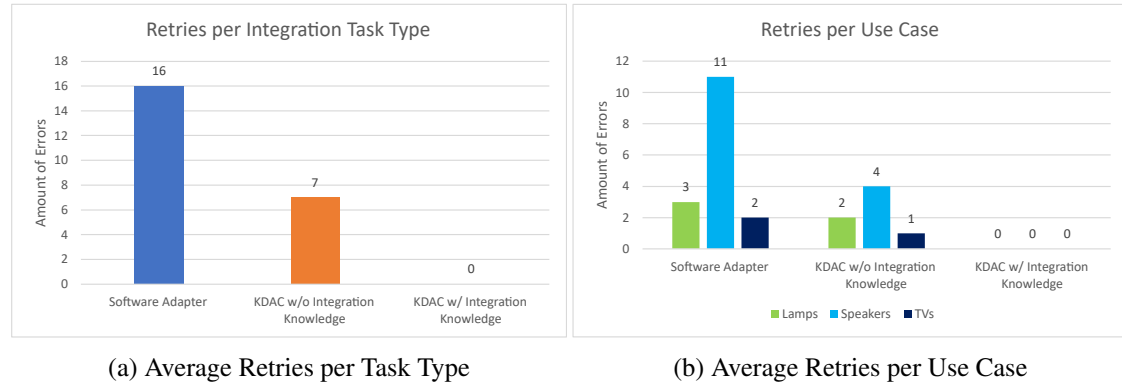


Figure 13.5.: Eval 3 – Errors

Error resolving strategies for all methods included the usage of logging functionality offered by the IDE. Regarding the manual method, this allowed for mostly identifying keys with different semantics as the retrieved values from the provided interfaces did not match the specified test criterion. Regarding the KDAC method's first variant, errors made in mapping from within the tool resulted in wrong JSONata transformations. These errors were mainly resolved by adjusting the inserted JSONata mapping strings directly in the software adapter. However, this error could be traced back to non-use of the Mapping Test & Validator (see Fig. 13.2) as no incorrect mappings should be stored in the knowledge base.

We suspected that the component interaction correctness and integration time was highest using the KDAC method. Based on the data collected, we can summarize that the second variant of the KDAC method had the highest component interaction correctness (i.e., no errors made), and the integration time was lowest using the second variant of the KDAC method as well. However, the first variant of the KDAC method involved some errors.

13.4.1. Break-Even Analysis

Similar to the break-even analysis from the first experiment, we extrapolate the needed integration time again. However, there are some differences. In contrast to the first extrapolation, we do not only look at one data channel but at services with keys (as specified in chapter 5). Hence, we look at the number of mappings that can be reused within the integration context after the first formalization. This means that we assume that a similar integration context occurs again after some time. The similarity is extrapolated by a percentage value of how many mappings

13. Empirical Evaluation 3: Adapter Generation for Services

mapaction can be reused.

We look at the first integration case after mapping formalization and provide insights into time savings based on variable mapping reuse (see Fig. 13.6). In this figure, we group each device category. We can see that the additional specification effort cannot be minimized if no mappings can be reused. With an increasing percentage of mapping reuse in this first integration case, the additional time for mapping formalization is surpassed by integration cases Lamps and TVs. We cannot equalise the formalization time for Speakers during the first reuse scenario.

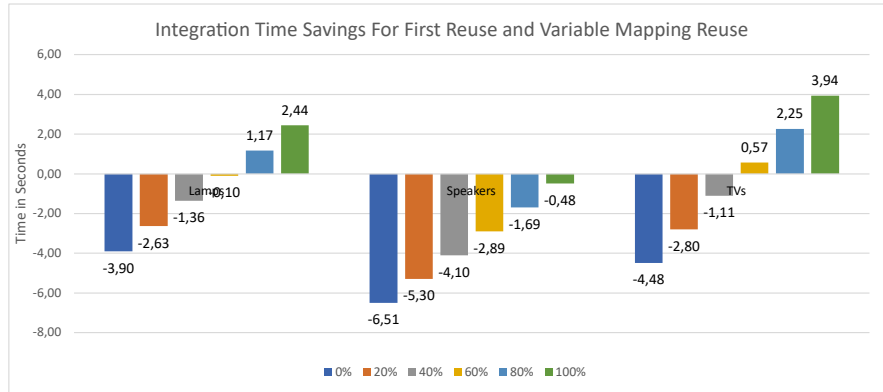


Figure 13.6.: Eval 3 – First Integration Knowledge Reuse with Variable Mapping Reuse

Now, we fix the percentage of mapping reuse at 50% and look at the number of reuse scenarios (see Fig. 13.7). We can see that after the second reuse scenario, the additional formalization time is surpassed in all use cases.

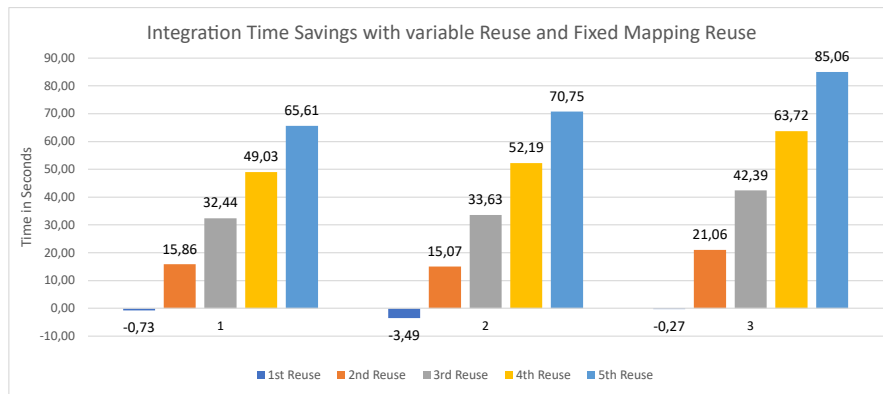


Figure 13.7.: Eval 3 – 50% Mapping Reuse with Variable Integration Knowledge Reuse

Overall, both extrapolations assume that integration knowledge can be reused in every subsequent integration context. However, this may not be the case if there are more use cases than the mentioned ones. Hence, the collected data only allows for an idealistic extrapolation as the experiment was designed in a way such that integration knowledge could be reused as early as possible.

13.5. Threats to Validity

Overall, the presented evaluation design favors internal over external validity. Hence, we eliminated the confounding factors for the independent variable *engineering method* as much as possible. Tasks were randomly assigned to the students, but it is made sure that no student works on the same integration task in subsequent measurement runs.

Internal Validity: Apparently, implementing interface mappings in a textual programming language and implementing interface mappings in a graphical web tool poses a different challenge for novices. Therefore, we ensured that the students working on software adapter implementation tasks also could rely on the NodeJS project skeleton generation service. Furthermore, we measured the results for using the graphical tool without reuse and reasoning functionality (i.e., KDAC variant 1). Consequently, we could identify the time saved by switching from the textual to the graphical syntax for mapping creation. Although we can see that the second variant of KDAC is the fastest, we can only approximate the point where using the tool in addition to implementing the software adapter pays off. This is mainly due to the challenge of collecting enough realistic engineering data.

External Validity: All participants were able to solve all tasks given the tooling infrastructure. Nevertheless, we can only discuss generalized statements based on this experiment within the following frame: There may be a selection bias as only four students were serving as study population members. The representativeness of use cases is ensured by using OpenAPI specifications from external product vendors. However, it was made sure during OpenAPI interface description selection that mappings could be chained early in the experiment. This may not hold in practice. Finally, the evaluation focuses on the engineering method. Hence, different technologies might have produced other results. Last, there existed a learning curve by the students for all use cases. The first integration contexts worked on (i.e., lamps) had a higher standard variation than the following use case (i.e., TVs). For instance, the highest (18 minutes) and second-highest standard deviation (10.7 minutes) have been measured for the tasks "MANUAL Yeelight-Lifx" (i.e., lamps) and "SUPPORTED Sonos-Sony" (i.e., speakers). The lowest (0.1) and second-lowest (0.1) standard deviations have been measured for the tasks "SUPPORTED-reasoning Samsung - Epson" and "SUPPORTED-reasoning Sony-Sonos". However, this learning curve applied to all students as they had no prior experience in implementing software adapters or using the KDAC tooling environment. In this evaluation context, no experience can be measured more precisely than some experience.

14. Performance Evaluation: Reasoning Algorithms and Architectures

In this experiment, we benchmark the implemented reasoning algorithms using data sets produced by a supporting tool. Therefore, we deployed the implemented reasoning component on two realistic architectures (i.e., fat client and thin client model). The central evaluation goal was to approximate if the tooling environment can also efficiently support the system integrator when the knowledge base becomes large.

14.1. Evaluation Setup

Challenge: It is unclear how long the implemented reasoning algorithms need to compute mappings based on action composition when the multigraph contains many nodes (i.e., interface descriptions) that are connected by many edges (i.e., mappings).

Experiment Scope: The web-based tool was requested to deduce a mapping from the requested to the provided action as defined by the interface description. The time needed to calculate the mapping was measured in milliseconds by the tool itself and printed onto the available console. The measurements were mostly conducted on different days.

Fat Client: The web application as used in evaluations 2 and 3 was directly connected to a cloud hosted NoSQL database. Therefore, all computations (e.g., mapping calculations) were done within the browser. A Google Cloud Firestore provides the data access layer within the Google Firebase environment. The data process layer was running on the device, using the application. The experiment was conducted on a Windows 10 laptop with an Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz (2 cores) and 16 GB of RAM.

Thin Client: The web application as used in evaluations 2 and 3 is packaged as a container. The underlying tool architecture utilizes a Kubernetes cluster. Therefore all necessary functionalities such as the transformation pre-processor, adapter generator, authentication, and user management were deployed as pods. Hence, the data processing layer was running on the back end (e.g., calculating mappings). The services are running on an Kubernetes cluster that has access to up to 8 CPUs with 2.6 Ghz and 16 GB of RAM.

In total, three data series were created (see Table 14.1). A data series consists of a sequence of measurement runs. In each measurement run, the respective algorithms calculated available mappings. The following parameters influenced the associated network of node and edges for each calculation:

- N is the number of nodes
- D is the maximum number of edges starting at each node in addition to the first edge

14. Performance Evaluation: Reasoning Algorithms and Architectures

- L is a binary variable that states whether loops are allowed or not

Table 14.1.: Eval 4 – Parameters for Created Data Series

Data series	N Values	D	L	Reference Color
1	10, 100, 500, 1000, 2000, ..., 6000	1	true	blue
2	10, 100, 500, 1000, 2000, ..., 6000	3	true	orange
3	10, 100, 500, 1000, 2000, ..., 6000	1	false	grey

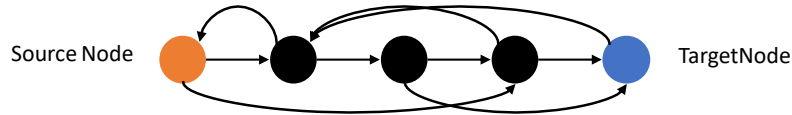


Figure 14.1.: Eval 4 – Example for $D=1$

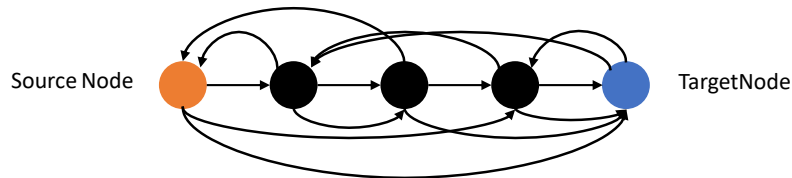


Figure 14.2.: Eval 4 – Example for $D=3$

An example for different values for parameter D is illustrated in Fig. 14.1 and Fig. 14.2. In addition to D , the parameter L has been sent to true as both graphs contain loops.

14.2. Fat Client Results

The calculation time results are shown in Fig. 14.3. The graph demonstrates the calculation time dependent on the N value of the test data set. For example, the calculation time to create an adapter from node 1 to node 500 was 7966 ms (8 seconds) for a network created during data series 1. It was 14871 ms (15 seconds) for a network created during data series 2 and 9071 ms (9 seconds) for a network created during series 3. The calculation time increased with an increasing N or increasing D . The highest values from node 1 to node 6000 were achieved with 139647 milliseconds (2.3 minutes) for data series 1, 300910 milliseconds (5 minutes) for data series 2 and 152521 milliseconds (2.5 minutes) for data series 3.

To approximate whether the collected data exposes quadratic or exponential growth, we will only look at data series 1 and 3 as their measurements are quite similar and hence can be combined. For the approximation, we allowed quadratic functions of type $ax^2 + bx$ and exponential functions of type $a \times e^{bx}$.

We can see that both generated approximation functions expose a high R^2 value (see Table 14.2).

14. Performance Evaluation: Reasoning Algorithms and Architectures

The coefficient of determination or R^2 value gives information of the goodness of fit of a model. R^2 is 1 at max, and a high R^2 value indicates a good fit. A graphical representation of both approximation functions can be seen in in Fig. 14.4.

Model	$f(x)$ Approximation	R^2 value
Quadratic	$0.00641 \times x^2 - 12.47791 \times x$	0.99
Exponential	$21670.70934 \times e^{0.00045 \times x}$	0.97

Table 14.2.: Eval 4 – Quadratic and Exponential Growth Rate Approximation

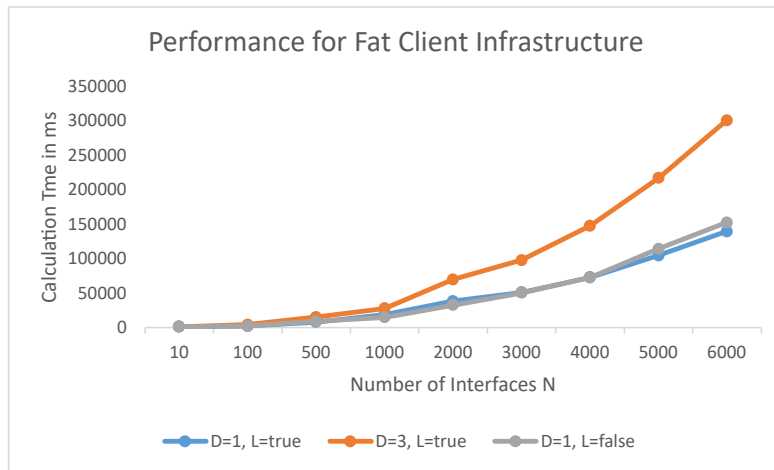


Figure 14.3.: Eval 4 – Calculation times in Milli Seconds (ms) for Fat Client

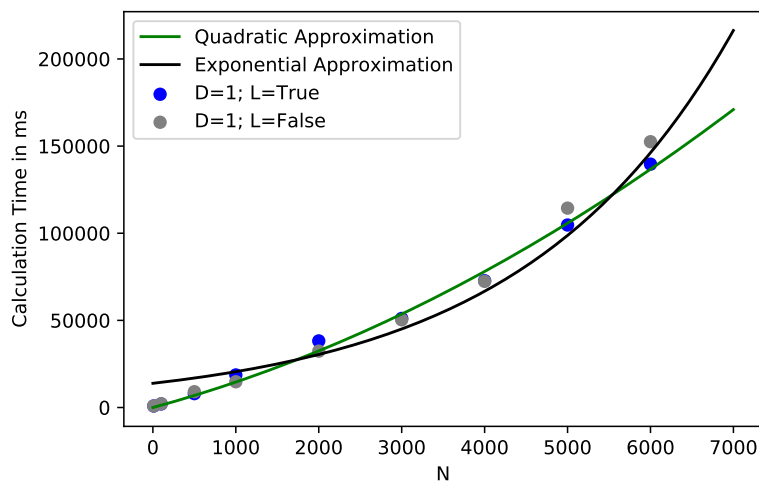


Figure 14.4.: Eval 4 – Quadratic and Exponential Approximation Functions for Fat Client

14.2.1. Discussion

The experiment indicates that the tool can effectively reason about mappings if the knowledge graph’s size is beneath a certain limit. The calculation times are growing quadratic to the complexity of the domain. A quadratic growth rate doubles with each addition to the input data set. This growth rate is reasonable, but not ideal [66]. Loops do not seem to slow the implemented algorithms down.

14.3. Thin Client Results

The calculation time results for the thin client infrastructure are shown in Fig. 14.5. Again, the graph demonstrates the calculation time dependent on the N value of the test data set. For example, the calculation time to create an adapter from node 1 to node 500 was 9287 milliseconds (9 seconds) for a network created during data series 1. It was 17904 milliseconds (18 seconds) for a network created during data series 2 and 9198 milliseconds (9 seconds) for a network created during sequence 3. The highest values from node 1 to node 6000 were achieved with 310299 milliseconds (5.2 minutes) for data series 1, 705128 milliseconds (11.8 minutes) for data series 2 and 295690 milliseconds (5 minutes) for data series 3.

Again, we approximate the functions as described in the fat client results section. We can see that both generated approximation functions expose a high R^2 value (see Table 14.3). A graphical representation of both approximation functions can be seen in Fig. 14.6.

Model	$f(x)$ Approximation	R^2 value
Quadratic	$0.00164 \times x^2 - 12.95678 \times x$	0,99
Exponential	$13883.74020 \times 2e^{0.00039 \times x}$	0,97

Table 14.3.: Eval 4 – Quadratic and Exponential Growth Rate Approximation

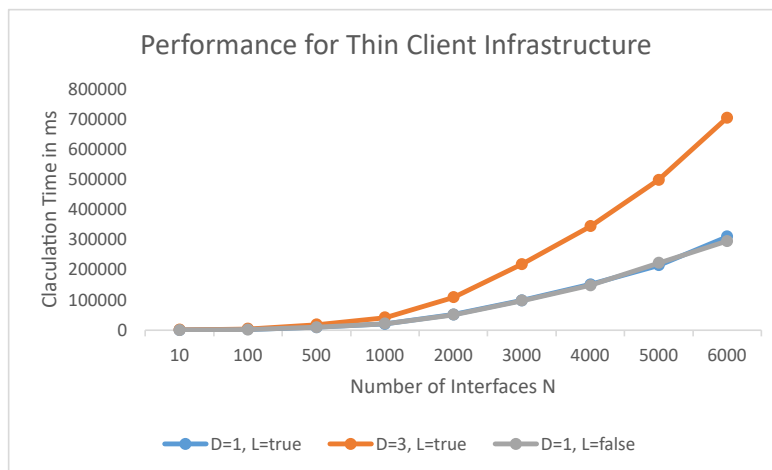


Figure 14.5.: Eval 4 – Calculation times in Milli Seconds (ms) for Thin Client

14. Performance Evaluation: Reasoning Algorithms and Architectures

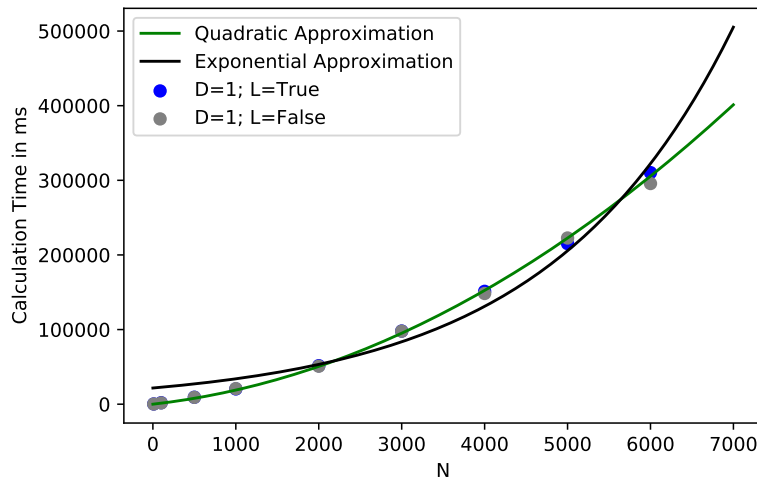


Figure 14.6.: Eval 4 – Quadratic and Exponential Approximation Functions for Thin Client

14.3.1. Discussion

The calculation times are growing quadratic to the complexity of the domain. In comparison to the fat client infrastructure, the thin client approach is visibly slower. For example, a mapping time calculation from node 1 to 6000 in data series 1 from the fat client infrastructure was 2.3 minutes, where the corresponding data series from the Thin Client was 5.2 minutes. This is remarkable in the sense that the algorithm returns a result earlier when running in the browser with fewer CPUs and similar main memory. We suspect two things. The Kubernetes cluster is using more calculation time for providing its service than expected. Furthermore, the algorithms in both approaches are implemented using Node.js. Node.js only runs in one thread and does not provide parallelization out of the box.

Independent of the differences in measurement, we cannot state how many nodes and edges a knowledge graph for integration knowledge may have in industry. Hence, we cannot state when mapping generation time is tolerable or not.

As a side note, we have to mention that HTTP is not the best choice for long running processes (i.e., several minutes). During some experimental runs with $N > 6000$, we quickly ran into multiple timeout errors.

14.4. Threats to Validity

Internal Validity: One limitation for both experiment's internal validity is measurement inaccuracies. The measurements were all conducted on different days resulting in slightly different circumstances for each measurement. Therefore a varying internet connection, different geographic locations, and varying available capacities of the laptop conducting the test may affected calculation times. Nevertheless, the experiment displays a multitude of data points, revealing a definite correlation between the calculation times and the complexity of the knowledge graph. However, the collected data set is small.

14. Performance Evaluation: Reasoning Algorithms and Architectures

External Validity: The external validity of this study is influenced by two factors. First, the test data consists of homogeneous interface descriptions. Every OpenAPI created during the test only differs from its title. Hence, only simple mappings were inserted as edges. Also, every interface description is displaying a basic API with only one resource. Therefore the test data does not properly represent the practical IoT domain with many profoundly different devices. Second, the test data is evenly connected. Every device was mapped to an exact number of other devices. Such an evenly distributed linking is unlikely to exist in a domain. Domain models of real world scenarios may be sparsely or irregularly connected and may have unreachable nodes, dead ends, or central nodes acting as a major intersection.

The named factors decrease the external validity of the experiment. The degree of this influence is undetermined. A heterogeneous set of interface descriptions in the domain model might have an increasing effect on calculation times.

Part VI.

Related Work, Limitations and Conclusion

15. Related Work

Four different research streams deal with semantic interoperability for various system classes. These are symbolic artificial intelligence, component-based software development, software architecture, and web services [12].

In symbolic artificial intelligence, semantic interoperability is discussed from the perspective of semantic service descriptions [67], ontologies [68, 69], ontologies for IoT [70], case-based reasoning [71, 72], empirical data integration [73], and knowledge base evolution [74]. A influential, theoretical approach for semantic interoperability was proposed by Euzenat [36].

In component-based software development, semantic interoperability is achieved by semantic component retrieval [52, 55], integration approaches for dynamic systems [75], reuse-oriented recommender systems [54], ontology-based software reuse [76], run time healing of integration problems [77], software interface adaptation [78], and semantic interoperability testing [79].

In software architectures, semantic interoperability is discussed regarding architectural styles [14], fuzzy service matching [16], software adapters [27, 7, 80], service-oriented architectures [81], architectural knowledge management [19, 38], architectural mismatch and reuse [20], interoperability for information systems [82], interface adaptations [83], and interoperability between software ecosystems [84].

In the web service community, semantic interoperability is examined regarding service description matching [85, 86, 87], semantic degrees [88], service composition [47, 46], description techniques [48], web of things [45], and matching of web service engineering data [89].

From a software engineering perspective, application examples of semantic technologies within IoT platforms include discovery service architecture patterns [90, 91], component-based architectures [92], ontology proposals [69], smart home applications [93], engineering frameworks [94], and pervasive computing architectures [95]. In industrial IoT systems (also called Industry 4.0), software engineers are supported to harness the benefits of semantic technologies by providing administrative shells [96], standard integration [97, 98], plug and play architectures for embedded systems [99, 100], description standards [101], and ontology proposals [102].

Although there are existing surveys about semantic interoperability for IoT [103] and industrial IoT systems [104], these surveys rather focus on domain-specific technological solutions or a system architecture and do not focus on the underlying engineering method to achieve interoperability. Hence, we conducted a fine grained search that mentioned knowledge management processes between stakeholders explicitly [12]. In table 15.1, we categorize formal approaches based on the applied integration methods (i.e., as presented in the background chapter), use of software knowledge and assets, mapping creation time, and viewpoint. All papers test their solution proposals based on reconfiguration time, degree of automation for software adapter generation, reasoning principles applied to reduce mapping specification time, and software adapter correctness. As a consequence, a distinct separation of method and technological solution or

15. Related Work

architecture proposal is not possible. Otherwise, an evaluation would not be feasible. However, all papers explicitly focus on humans-in-the-loop to achieve semantic interoperability:

Table 15.1.: Related Approaches based on Formal Mappings Tested in IoT and Industrial IoT scenarios

Study	Year	Research Focus	Method			Software and Assets	Knowledge	Mapping Creation Time	View-point
			Top Down	Bottom Up	Fully Automatable				
Ali et al. [105]	2015	Web Services	x		x	Domain Ontology, Semantic Sensor Descriptions		Requirements	Data
Khodadadi et al. [87]	2017	Web Service Discovery		x		Domain Ontology, JSON-LD, Thing Description		Desing Time	Service
Nostro et al. [80]	2016	Software Adapter	x		x	Domain Ontology, Labeled Transition Systems		Implementation	Service
Bennaceur et al. [7]	2015	Software Adapter		x	x	Domain Ontology, Finite State Machine, Model checker		Run Time	Service
Gyrard et al. [106]	2015	Logical AI		x		Domain Ontology, Dataset and Rules, SPARQL Queries		Implementation	Data
Patel et al. [107]	2015	Software Architecture		x		Conceptual Model		Implementation	Service
Grangel-González et al. [108]	2018	Logical AI		x		Device Models (AutomationML), Domain Model (Probabilistic Soft Logic)		Implementation	Data
Kovatsch et al. [109]	2015	Web Service		x	x	Semantic Service Description (RESTdesc), Resource Directory, Domain Ontology		Run time	Data
Koziolek et al. [99]	2018	Software Architecture	x		x	Device Models (PLCopen), OPC UA Service descriptions		Run time	Service
Nagib et al. [110]	2016	Software Architecture	x			Sensor Ontology, Domain Model (DBpedia)		Requirements	Data
Mooij [111]	2013	Software Adapter (Syntax only)		x		Software Adapters, Database Abstraction Models		Development	Service
Yang et al. [112]	2019	Software Architecture		x	x	Proprietary Semantic Documents (Table-based)		Development	Data
Prinz et al. [113]	2019	Software Architecture		x	x	oneM2M, BPMN Modeling Editor		Development	Service

Ali et al. [105] present a semantic processing framework for IoT-enabled communication systems. They base their method on queries for static and dynamic data, and make it easier and more cost-effective to add new external sources. They test their approach based on semantic data, which links to a personalized profile in a publish-subscribe architecture.

Khodadadi et al. [87] suggest a framework for service definition and discovery. This framework relies on ontologies paired with JSON-LD and is a prime example for bottom-up engineering, as services are annotated incrementally. Their evaluation shows their solution is also scalable.

Nostro et al. [80] introduce an approach to achieve functional and non-functional interoperability through synthesized connectors. All devices involved support a BPMN-like language. By using a stochastic model-based implementation and a dependability analysis, this approach allows software adapters to be generated in an automated way.

Bennaceur et al. [7] present another fully automatable approach that achieves interoperability through semantic technologies. Their approach uses a domain-specific ontology, already annotated services, and model checking techniques to generate correct by construction mediators

15. Related Work

automatically. In contrast to Nostro et al., they target the run time phase and minimize additional specification effort using reasoning principles.

Gyrard et al. [106] assist IoT developers in designing interoperable semantic web of things applications. To overcome manual mapping work, they design a generator that links all available data to a collection of ontologies. Therefore, they collected multiple ontologies from research projects and integrated them into the generator. Hence, the software engineer does only need to check if the generator suggests the correct mappings.

Patel et al. [107] enable high-level application development for the Internet of Things by using a common vocabulary defined by (non-technical) domain experts. Therefore, they separate IoT application development into different concerns and they provide a conceptual framework and an implementation framework that supports various stakeholders in their actions.

Grangel-González et al. [108] present knowledge graphs for semantically integrating cyber-physical Systems. They claim that integration knowledge is trapped within informal documents that describe cyber-physical systems. Hence, they propose a tool called SemCPS that combines Probabilistic Soft Logic and Knowledge Graphs to describe both a CPS and its components semantically. Their approach checks whether one component model conforms to another component model based on RDF principles.

Kovatsch et al. [109] introduce a practical approach to semantics for the IoT regarding physical states and device mashups. Their approach calculates an execution plan based on RESTdesc service descriptions to facilitate service composition. They note that calculating an execution plan took longer than expected and that this is a potential obstacle to applying their approach out of the box.

Koziolok et al. [99] show a self-commissioning industrial IoT Systems in Process Automation from an architectural point of view. This reference architecture is a prime example of top-down driven interoperability as they do not involve human interaction and they solely based their solution on standardized information models (i.e., PLCOpen and OPC UA). They could realize plug and play scenarios with decreased commission time to execute closed loop control programs.

Nagib et al. [110] present a framework for IoT devices. This framework facilitates on-the-fly dynamic integration, discovery, and access to heterogeneous sensor data. This is mainly achieved by reusing sensor data integrated from multiple heterogeneous sources, so that building innovative applications and services is accelerated.

Mooij [111] outlines an approach for system integration by developing adapters using a database abstraction. Based on a model-based specification, domain experts can reason about the specification. This specification is then used to generate software adapters. This is mainly achieved by modeling system interactions as declarative database operations instead of message communications. Hence, their approach only relies on a human to check semantic interoperability (i.e., domain experts interpret the database operation syntax).

Yang et al. [112] introduce an approach based on information fusion to implement semantic interoperability between IoT devices and end users. In contrast to the approaches presented so far, this approach focuses not on devices to talk to each other but on how the interaction of the user differs based on the same data in different contexts. Therefore, they attach a semantic document representation to the payloads being rendered by the IoT application. Context-aware system research also seems to influence this approach.

Finally, Prinz et al. [113] propose a novel I4.0-enabled engineering method. In their work, they

15. Related Work

require a human to model the desired production process with the Business Process Modelling Notation (BPMN). An orchestration engine executes this process at run time, where each process step is linked to available I4.0 components by relying on oneM2M service descriptions. Therefore, each BPMN process element contains metadata which identifies the required oneM2M service.

This overview of methods is based on all presented abstract methods. Now, we only look at BU methods as these methods are directly related solution proposals to our approach.

16. Discussion

KDAC classifies as a bottom-up approach. As seen throughout this work, the corresponding approaches aim at solving the semantic interoperability problem by composing autonomously developed required and provided interfaces. To outline the differences between KDAC and these existing approaches, we categorize found bottom-up approaches based on their goal. These goals are common domain model generation, software template generation, software adapter generation, and semantic service orchestration. Each subsequent goal conceptually requires all previous goals to be achieved.

Category	Author	Title
Common Domain Model	Yang et al. [112]	Tabdoc Approach: An Information Fusion Method to Implement Semantic Interoperability Between IoT Devices and Users
	Grangel-González et al. [108]	Knowledge Graphs for Semantically Integrating Cyber-Physical Systems
API Template Generation	Gyrard et al. [106]	Assisting IoT Projects and Developers in Designing Interoperable Semantic Web of Things Applications
	Patel et al. [107]	Enabling high-level application development for the Internet of Things
Software Adapter Generation	Mooij [111]	System integration by developing adapters using a database abstraction
	Bennaceur et al. [7]	Automated Synthesis of Mediators to Support Component Interoperability
Service Orchestration	Khodadadi et al. [87]	Simurgh: A framework for effective discovery, programming, and integration of services exposed in IoT
	Kovatsch et al. [109]	Practical semantics for the Internet of Things: Physical states, device mashups, and open questions
	Prinz et al. [113]	A novel I4.0-enabled engineering method and its evaluation

Table 16.1.: Related Bottom-Up Solution Approaches

Common domain model generation refers to unifying different service descriptions on the same domain or proposing a new domain model. API template generation builds upon such a domain model to assist the system integrator. These approaches generate code artefacts such that interfaces must not be built from scratch. Software adapter generation adds the ability to compose

a required and a provided interface. In contrast to template generation, the generated code is executable but must be manually imported into the software project during implementation. Semantic service orchestration generation shifts software adapter generation to system run time. Here, the software adapter includes all logic to call provided services in order to fulfil the required service.

16.1. Limitations

We will now look at the four categories and discuss limitations of KDAC in contrast to the identified existing approaches (see Table 16.1).

Regarding creating a common domain model, we must differentiate between proposing a new domain model and building up a common domain model based on integration cases. KDAC stores mappings and builds up a common domain model for integration cases based on available devices. KDAC cannot be used to define a domain model without these use cases as we require concrete interface descriptions. The resulting domain model will always be incomplete. Therefore, we conclude that KDAC cannot be used at the moment to propose a new domain model.

API Template Generation already takes a specific viewpoint on the application development process and does not only look at modelling the desired domain. For instance, the domain model (e.g., described in mappings or using an ontology) can be included in generating abstract software stubs that are further implemented by the system integrator. These stubs contain interfaces based on the available domain knowledge but must still be connected to the available concrete instances. KDAC assists the system integrator by proposing such mappings. Furthermore, KDAC can be used to generate an API template for further implementation only if the strict mode is deactivated.

Software Adapter Generation can be automated on the technical (i.e., networking protocol), syntactical (e.g., payload structured as JSON) and semantic (i.e., common domain vocabulary) level. KDAC provides a structured approach to build up the semantic level. Based on the distinction between required and provided interfaces, KDAC can generate a software adapter automatically. Depending on the incompleteness of mapping knowledge between required and provided interfaces, software adapter can also be incomplete. This means that for certain identifiers, no mapping may be defined. Although the missing mappings can also be inserted into the generated software code, these mappings are not synced to the knowledge base. Hence, software adapters should only be generated when all mappings for an integration context are specified and tested. Service Orchestration aims to discover relevant services with respect to the required user request and to create a form of computation (e.g., flow-based) at run time. Instead of using semantically annotated services based on a common standard (e.g., SAWSDL [23]), the knowledge base used by KDAC as well as the reasoning algorithms can be invoked with the required service as an input. As an output, the tool infrastructure then invokes all identified provided services, maps the values to the required service specifications and then returns the desired results. Here, the central characteristic of incomplete integration knowledge comes into play. Although KDAC is conceptually ready to be applied at run time, the downside of missing integration knowledge must be tackled. Especially when the knowledge base contains few mappings, a mechanism must be provided such that new mappings can be inserted at run time fast.

Most of the identified related bottom-up approaches are using distinct technologies for describing interfaces, creating a common domain model and providing some intelligence to assist the system integrator.

Obviously, the selected technologies for KDAC restrict the applicability of the method as well. As KDAC relies on JSONata to store mappings, service descriptions must be exportable as JSON. Hence, other service descriptions such as OPC UA [114], which is mainly used in manufacturing scenarios or Eclipse Vorto¹, which is used in home automation scenarios, require adaptations. If a proprietary description standard is used (e.g., a database abstractions as presented in [111]), major software refactoring is to be expected.

Within the world of web service, OpenAPI [56] and AsyncAPI [57] are relatively new, although they are already widely applied. More mature standards such as the web service description language (WSDL [37]) are also a candidate for describing web services that do not follow REST principles and that may require other networking protocols. Supporting such a standard would require adaptations within the tooling infrastructure.

On a more abstract layer, semantic web service descriptions (e.g., SAWSDL [23], JSON-LD [42] or RESTdesc²) are not compatible with the current tooling infrastructure. For describing the semantics of interface elements, most identified approaches rely on a formal ontology (e.g., supporting $\mathcal{SHOIN}(\mathcal{D})$). Furthermore, the reasoner can infer (new) knowledge based on such a formalization. Although different profiles restrict the expressiveness of formal ontologies (e.g., OWL-Full, OWL-DL or OWL-Lite), engineering knowledge within these technologies is complex and requires trained system integrators as compared to teaching them JSONata. Therefore, KDAC does not support ontologies. As a consequence, we do not inherit the same formalization and modelling efforts.

Furthermore, we learned early on that modeling the T-Box (i.e., concepts to represents interface mappings) such that reasoners can infer (new) mappings is hard and would result in poor applicability for students (see JSON-LD in empirical experiment 2 and [11]). Hence, we implemented our own knowledge base and reasoning algorithms. The downside is that we cannot benefit from the time tested and validated reasoning technologies from the artificial intelligence community. However, this does not invalidate our initial claim that having an ontology as a central knowledge base for semantic integration is not beneficial for IoT applications as they must be updated continuously.

16.2. Future Work

The presented approach speeds up software adapter implementation in a reliable way. The scope of KDAC includes web services based on the REST paradigm and event-driven architectures. The applicability is evaluated within home automation scenarios. In this section, we will highlight the most important next steps.

First, the tool currently assists the system integrator at design time. This means that code has to

¹<https://www.eclipse.org/vorto/>

²<https://restdesc.org/>

16. Discussion

be recompiled every time a new software adapter is produced and inserted into the corresponding software project. Although this process can be automated using continuous integration or delivery pipelines, the underlying system cannot change itself autonomously. In order to reuse integration knowledge in a self-adaptive way, integration knowledge must be evaluated at system run time. However, this would only be an intermediate step towards emergent systems. In an emergent system, device integration is not planned or reasoned based on human integration knowledge, but the system learns necessary communications based on observed device interactions. However, such plug and play scenarios can only work if integration reliability is given.

Second, reliability in KDAC is currently achieved by a testing process. This testing process involves human knowledge when a test is passed (i.e., when devices in the physical world show the expected behaviour). However, this mechanism is not suitable for device integration at run time. The underlying challenge is to deal with missing integration knowledge. At the moment, missing integration knowledge is handled by the system integrator. Although effective, this process step cannot be handled at run time efficiently. Hence, an automated way to resolve missing integration knowledge at run time to enable device integration is necessary.

Third, we are currently relying on syntactic interface specifications in the OpenAPI and AsyncAPI format. Although this type of interface specification is more widely used as semantic interface specifications (e.g., SAWLSDL [23]), they must be created in the first place. The process to achieve such descriptions still involves a high degree of manual work and technical knowledge. If no such descriptions exist, our approach (and most other relevant approaches) will not work. Here, more research on self-describing is necessary. Schwichtenberg et al. [115] take a first step by presenting an approach that can semi-automatically link OpenAPI specifications to a semantic grounding (e.g., a domain ontology) and then generate code adapters from it. Although this approach reduces human formalization effort, it still assumes that there exists one central domain ontology. Hence, more radical approaches towards self-describing systems are needed. For instance, Alon et al. [18] present an approach to learn distributed representations of code as vectors. However, such approaches are in an early research stage, and it may take years until they become applicable.

Fourth, other technical extensions to our work so far are A) labeled transitions systems, B) generating standard enhancements, and C) formal proofs A). At the moment, KDAC can only be applied to stateless services. This means that the sequence of service invocation does not matter. However, if service invocations do depend on each other (e.g., in one-to-many mapping), labelled transition systems may be integrated as another specification language. B) The integration knowledge produced by the reasoning algorithms may be used to generate standard enhancements. For instance, an ontology could be *just another* node within a mapping chain. Thereby, service instances for the A-BOX may be identifiable and transferred to an existing ontology. However, this ontology must be carefully selected, C) Our reasoning algorithms are well documented and extensively tested within our evaluations. However, we did not formally prove if they actually generate all possible mappings. Furthermore, their performance can be improved.

Last, a more business-related topic is named. Integration knowledge is a competitive advantage in the IoT market. Hence, not everybody may be eager to share this knowledge. Although the presented approach assumes that generating software adapters directly outweighs the additional formalization effort in future cases, sharing intellectual property (e.g., integration knowledge)

16. Discussion

may be subject to restrictions. Such restrictions (e.g., only selected system integrators can use individual bits of integration knowledge) are currently not supported by the tool.

17. Conclusion

In an ideal world, all software components would be self-descriptive such that they can unambiguously communicate in an automated way. However, service interoperability of embedded devices is still a problem for dynamically changing Internet of Things and Industry 4.0 software platforms. Similar to the Babylonian confusion, distributed software components currently suffer from communication problems. Although they may communicate with each other in the same language, they cannot understand each other. In this work, we introduced a novel engineering method that explicitly allows for an incomplete semantic domain description without losing the ability for automated IoT system integration. We utilize system integrators as a communication expert to translate between device descriptions in a meaningful way. By sharing integration knowledge centrally, we assist the system integrator in automating software adapter generation in the future. Our evaluation shows that students can efficiently apply the presented method and beat classical software adapter implementation. Hence, the presented approach only formalizes relevant integration knowledge that can evolve over time.

We hope that other researchers find our work inspiring in order to ultimately solve the interoperability challenge for distributed, self-adaptive, self-aware and emergent software systems.

Bibliography

- [1] P. Barnaghi, W. Wang, C. Henson, and K. Taylor, “Semantics for the internet of things: early progress and back to the future,” *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 8, no. 1, pp. 1–21, 2012.
- [2] S. Heiler, “Semantic interoperability,” *ACM Computing Surveys (CSUR)*, vol. 27, no. 2, pp. 271–273, 1995.
- [3] N. F. Noy, A. Doan, and A. Y. Halevy, “Semantic integration,” *AI magazine*, vol. 26, no. 1, pp. 7–7, 2005.
- [4] ecl@ass, “ecl@ass.” [Online]. Available: <https://www.eclasscontent.com/index.php?language=en>[retrieved:15.02.2021]
- [5] A. Rausch, C. Bartelt, S. Herold, H. Klus, and D. Niebuhr, “From software systems to complex software ecosystems: model-and constraint-based engineering of ecosystems,” in *Perspectives on the Future of Software Engineering*. Springer, 2013, pp. 61–80.
- [6] M. Autili, P. Inverardi, R. Spalazzese, M. Tivoli, and F. Mignosi, “Automated synthesis of application-layer connectors from automata-based specifications,” *Journal of Computer and System Sciences*, vol. 104, pp. 17–40, 2019.
- [7] A. Bennaceur and V. Issarny, “Automated Synthesis of Mediators to Support Component Interoperability,” *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 221–240, Mar. 2015.
- [8] F. Burzlaff and C. Bartelt, “Knowledge-driven architecture composition: Case-based formalization of integration knowledge to enable automated component coupling,” in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 108–111.
- [9] F. Burzlaff, S. Jacobs, and C. Bartelt, “Automated configuration in adaptive iot software ecosystems to reduce manual device integration effort: Application and evaluation of a novel engineering method,” *ADAPTIVE 2020*, 2020.
- [10] A. Rausch, C. Bartelt, S. Herold, H. Klus, and D. Niebuhr, “From Software Systems to Complex Software Ecosystems: Model- and Constraint-Based Engineering of Ecosystems,” in *Perspectives on the Future of Software Engineering: Essays in Honor of Dieter Rombach*, J. Münch and K. Schmid, Eds. Berlin, Heidelberg: Springer, 2013, pp. 61–80.

Bibliography

- [11] F. Burzlaff, C. Bartelt *et al.*, “Towards automating service matching for manufacturing systems: Exemplifying knowledge-driven architecture composition,” *Procedia CIRP*, vol. 72, pp. 707–713, 2018.
- [12] F. Burzlaff, N. Wilken, C. Bartelt, and H. Stuckenschmidt, “Semantic Interoperability Methods for Smart Service Systems: A Survey,” *IEEE Transactions on Engineering Management*, pp. 1–15, 2019.
- [13] F. Burzlaff and C. Bartelt, “Knowledge-Driven Architecture Composition: Assisting the System Integrator to reuse Integration Knowledge (to be published),” in *International Conference on Web Engineering*. Springer, 2021, p. tba.
- [14] H. Muccini and M. T. Moghaddam, “Iot architectural styles,” in *European Conference on Software Architecture*. Springer, 2018, pp. 68–85.
- [15] Lov4IoT, “Lov4iot.” [Online]. Available: <https://lov4iot.appspot.com/>[retrieved:15.02.2021]
- [16] M. C. Platenius, “Fuzzy matching of comprehensive service specifications,” PhD Thesis, Universitätsbibliothek, Paderborn, 2016.
- [17] M. Klusch and P. Kapahnke, “Semantic Web Service Selection with SAWSDL-MX,” in *Proceedings of the Second International Conference on Service Matchmaking and Resource Retrieval in the Semantic Web - Volume 416*, ser. SMRR’08. Aachen, Germany, Germany: CEUR-WS.org, 2008, pp. 2–16. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2889945.2889947>
- [18] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “code2vec: Learning distributed representations of code,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.
- [19] R. Capilla, A. Jansen, A. Tang, P. Avgeriou, and M. A. Babar, “10 years of software architecture knowledge management: Practice and future,” *Journal of Systems and Software*, vol. 116, pp. 191–205, 2016.
- [20] D. Garlan, R. Allen, and J. Ockerbloom, “Architectural mismatch: Why reuse is still so hard,” *IEEE software*, vol. 26, no. 4, pp. 66–69, 2009.
- [21] openHAB Foundation e.V., “openHAB,” 2019. [Online]. Available: <https://www.openhab.org/>[retrieved:15.02.2021]
- [22] S. Bunzel, “AUTOSAR – the Standardized Software Architecture,” *Informatik-Spektrum*, vol. 34, no. 1, pp. 79–83, Feb. 2011. [Online]. Available: <http://link.springer.com/10.1007/s00287-010-0506-7>
- [23] SAWSDL, “SAWSDL,” 2021. [Online]. Available: <https://www.w3.org/2002/ws/sawSDL/>[retrieved:15.02.2021]

Bibliography

- [24] N. Medvidovic and R. N. Taylor, "Software architecture: foundations, theory, and practice," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2. IEEE, 2010, pp. 471–472.
- [25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *European Conference on Object-Oriented Programming*. Springer, 1993, pp. 406–431.
- [26] M. Shaw, "Architectural issues in software reuse: It's not just the functionality, it's the packaging," *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. SI, pp. 3–6, 1995.
- [27] R. Spalazzese and P. Inverardi, "Mediating Connector Patterns for Components Interoperability," in *Software Architecture*, ser. Lecture Notes in Computer Science, M. A. Babar and I. Gorton, Eds. Springer Berlin Heidelberg, 2010, pp. 335–343.
- [28] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli, "Towards an Engineering Approach to Component Adaptation," in *Architecting Systems with Trustworthy Components*, ser. Lecture Notes in Computer Science, R. H. Reussner, J. A. Stafford, and C. A. Szyperski, Eds. Springer Berlin Heidelberg, 2006, pp. 193–215.
- [29] D. Harel and B. Rumpe, "Meaningful modeling: What's the semantics of "semantics"?" *Computer*, vol. 37, pp. 64–72, Nov. 2004.
- [30] M. Fowler, *Analysis patterns: reusable object models*. Addison-Wesley Professional, 1997.
- [31] A. Bennaceur, V. Issarny, R. Spalazzese, and S. Tyagi, "Achieving Interoperability through Semantics-Based Technologies: The Instant Messaging Case," in *The Semantic Web – ISWC 2012*, ser. Lecture Notes in Computer Science, P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. X. Parreira, J. Hendler, G. Schreiber, A. Bernstein, and E. Blomqvist, Eds. Springer Berlin Heidelberg, 2012, pp. 17–33.
- [32] G. Karsai, J. Sztipanovits, A. Ledeczki, and T. Bapty, "Model-integrated development of embedded software," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 145–164, Jan. 2003, conference Name: Proceedings of the IEEE.
- [33] "Smart appliances reference SAREF ontology," 2020. [Online]. Available: <https://sites.google.com/site/smartappliancesproject/home2/>[retrieved:15.02.2021]
- [34] J. Mineraud, O. Mazhelis, X. Su, and S. Tarkoma, "A gap analysis of internet-of-things platforms," *Computer Communications*, vol. 89, pp. 5–16, 2016.
- [35] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, "Web services," in *Web services*. Springer, 2004, pp. 123–149.
- [36] J. Euzenat, "Towards a principled approach to semantic interoperability," 2001.
- [37] WSDL, "WSDL," 2021. [Online]. Available: <https://www.w3.org/TR/2001/NOTE-wsdl-20010315>[retrieved:15.02.2021]

Bibliography

- [38] Z. Li, P. Liang, and P. Avgeriou, "Application of knowledge-based approaches in software architecture: A systematic mapping study," *Information and Software technology*, vol. 55, no. 5, pp. 777–794, 2013.
- [39] M. Alavi and D. E. Leidner, "Knowledge management and knowledge management systems: Conceptual foundations and research issues," *MIS quarterly*, pp. 107–136, 2001.
- [40] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future internet: the internet of things architecture, possible applications and key challenges," in *2012 10th international conference on frontiers of information technology*. IEEE, 2012, pp. 257–260.
- [41] M. Noura, M. Atiquzzaman, and M. Gaedke, "Interoperability in internet of things: Taxonomies and open challenges," *Mobile Networks and Applications*, vol. 24, no. 3, pp. 796–809, 2019.
- [42] JSON-LD, "JSON-LD," 2021. [Online]. Available: <https://www.w3.org/TR/json-ld11/> [retrieved:15.02.2021]
- [43] D. H. Lorenz and B. Rosenan, "Code reuse with language oriented programming," in *International Conference on Software Reuse*. Springer, 2011, pp. 167–182.
- [44] S. Bunzel, "Autosar—the standardized software architecture," *Informatik-Spektrum*, vol. 34, no. 1, pp. 79–83, 2011, publisher: Springer.
- [45] A. J. Jara, A. C. Olivieri, Y. Bocchi, M. Jung, W. Kastner, and A. F. Skarmeta, "Semantic web of things: an analysis of the application semantics for the iot moving towards the iot convergence," *International Journal of Web and Grid Services*, vol. 10, no. 2-3, pp. 244–272, 2014.
- [46] K. Kurniawan, F. J. Ekaputra, and P. R. Aryan, "Semantic Service Description and Compositions: A Systematic Literature Review," in *2018 2nd International Conference on Informatics and Computational Sciences (ICICoS)*. IEEE, 2018, pp. 1–6.
- [47] M. Cremaschi and F. De Paoli, "A practical approach to services composition through light semantic descriptions," in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2018, pp. 130–145.
- [48] M. Lanthaler and C. Gütl, "On using JSON-LD to create evolvable RESTful services," in *Proceedings of the Third International Workshop on RESTful Design*, 2012, pp. 25–32.
- [49] SWoT, "Semantic Web of Things." [Online]. Available: <http://sensormeasurement.appspot.com/> [retrieved:15.02.2021]
- [50] Schema, "Home - iotschema.org." [Online]. Available: <http://iotschema.org/> [retrieved: 15.02.2021]
- [51] A. Bennaceur, "Dynamic Synthesis of Mediators in Ubiquitous Environments," phdthesis, Université Pierre et Marie Curie - Paris VI, Jul. 2013. [Online]. Available: <https://tel.archives-ouvertes.fr/tel-00849402>

Bibliography

- [52] O. Hummel and C. Atkinson, “Automated Creation and Assessment of Component Adapters with Test Cases,” in *Component-Based Software Engineering*, ser. Lecture Notes in Computer Science, L. Grunske, R. Reussner, and F. Plasil, Eds. Springer Berlin Heidelberg, 2010, pp. 166–181.
- [53] F. Meyerer and O. Hummel, “Towards Plug-and-play for Component-based Software Systems,” in *Proceedings of the 19th International Doctoral Symposium on Components and Architecture*, ser. WCOP '14. New York, NY, USA: ACM, 2014, pp. 25–30. [Online]. Available: <http://doi.acm.org/10.1145/2601328.2601334>
- [54] W. Janjic, O. Hummel, and C. Atkinson, “Reuse-oriented code recommendation systems,” in *Recommendation Systems in Software Engineering*. Springer, 2014, pp. 359–386.
- [55] O. Hummel and C. Atkinson, “Extreme harvesting: Test driven discovery and reuse of software components,” in *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004*. IEEE, 2004, pp. 66–72.
- [56] “OpenAPI Specification,” 2020. [Online]. Available: <https://swagger.io/specification/v2/> [retrieved:15.02.2021]
- [57] “AsyncAPI Specification,” 2020. [Online]. Available: <https://www.asyncapi.com/> [retrieved:15.02.2021]
- [58] JSONata, “Json query and transformation language.” [Online]. Available: <https://jsonata.org/> [retrieved:15.02.2021]
- [59] A. J. Ko, T. D. LaToza, and M. M. Burnett, “A practical guide to controlled experiments of software engineering tools with human participants,” *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, Feb. 2015. [Online]. Available: <https://doi.org/10.1007/s10664-013-9279-3>
- [60] G. Leroy, *Designing User Studies in Informatics*. Springer Science & Business Media, Aug. 2011, google-Books-ID: IqR7M1h1yDQC.
- [61] D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, and M. Oivo, “Empirical software engineering experts on the use of students and professionals in experiments,” *Empirical Software Engineering*, vol. 23, no. 1, pp. 452–489, Feb. 2018. [Online]. Available: <https://doi.org/10.1007/s10664-017-9523-3>
- [62] “Jolt,” 2020. [Online]. Available: <https://github.com/bazaarvoice/jolt> [retrieved:15.02.2021]
- [63] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, “MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks,” in *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on*. IEEE, 2008, pp. 791–798. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4554519

Bibliography

- [64] A. J. Ko, T. D. Latoza, and M. M. Burnett, “A practical guide to controlled experiments of software engineering tools with human participants,” *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, 2015.
- [65] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.
- [66] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.
- [67] M. Klusch, P. Kapahnke, and I. Zinnikus, “SAWSDL-MX2: A Machine-Learning Approach for Integrating Semantic Web Service Matchmaking Variants,” in *2009 IEEE International Conference on Web Services*, Jul. 2009, pp. 335–342.
- [68] M. Uschold and M. Gruninger, “Ontologies and semantics for seamless connectivity,” *ACM SIGMod Record*, vol. 33, no. 4, pp. 58–64, 2004.
- [69] I. Szilagyi and P. Wira, “Ontologies and semantic web for the internet of things-a survey,” in *IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2016, pp. 6949–6954.
- [70] N. Seydoux, K. Drira, N. Hernandez, and T. Monteil, “Iot-o, a core-domain iot ontology to represent connected devices networks,” in *European Knowledge Acquisition Workshop*. Springer, 2016, pp. 561–576.
- [71] B. Limthanmaphon and Y. Zhang, “Web service composition with case-based reasoning,” in *Proceedings of the 14th Australasian database conference-Volume 17*, 2003, pp. 201–208.
- [72] A. Aamodt and E. Plaza, “Case-based reasoning: Foundational issues, methodological variations, and system approaches,” *AI communications*, vol. 7, no. 1, pp. 39–59, 1994.
- [73] H. Stuckenschmidt, J. Noessner, and F. Fallahi, “A study in user-centric data integration.” in *ICEIS (3)*, 2012, pp. 5–14.
- [74] F. M. Shipman III and R. McCall, “Supporting knowledge-base evolution with incremental formalization,” in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1994, pp. 285–291.
- [75] C. Bartelt, T. Fischer, D. Niebuhr, A. Rausch, F. Seidl, and M. Trapp, “Dynamic Integration of Heterogeneous Mobile Devices,” in *Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software*, ser. DEAS '05. New York, NY, USA: ACM, 2005, pp. 1–7. [Online]. Available: <http://doi.acm.org/10.1145/1083063.1083085>
- [76] H.-J. Happel, A. Korthaus, S. Seedorf, and P. Tomczyk, “Kontor: an ontology-enabled approach to software reuse,” in *In: Proc. Of The 18Th Int. Conf. On Software Engineering And Knowledge Engineering*. Citeseer, 2006.

Bibliography

- [77] H. Chang, L. Mariani, and M. Pezze, “In-field healing of integration problems with COTS components,” in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 166–176.
- [78] H. B. Pötter and A. Sztajnberg, “Adapting heterogeneous devices into an iot context-aware infrastructure,” in *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2016, pp. 64–74.
- [79] S. K. Datta, C. Bonnet, H. Baqa, M. Zhao, and F. Le-Gall, “Approach for semantic interoperability testing in internet of things,” in *2018 Global Internet of Things Summit (GIoTS)*. IEEE, 2018, pp. 1–6.
- [80] N. Nostro, R. Spalazzese, F. Di Giandomenico, and P. Inverardi, “Achieving functional and non functional interoperability through synthesized connectors,” *Journal of Systems and Software*, vol. 111, pp. 185–199, 2016.
- [81] T. Vitvar, A. Mocan, M. Kerrigan, M. Zaremba, M. Zaremba, M. Moran, E. Cimpian, T. Haselwanter, and D. Fensel, “Semantically-enabled service oriented architecture: concepts, technology and application,” *Service Oriented Computing and Applications*, vol. 1, no. 2, pp. 129–154, 2007.
- [82] H. Abukwaik, D. Taibi, and D. Rombach, “Interoperability-related architectural problems and solutions in information systems: A scoping study,” in *European Conference on Software Architecture*. Springer, 2014, pp. 308–323.
- [83] F. L. Keppmann, M. Maleshkova, and A. Harth, “Adaptable interfaces, interactions, and processing for linked data platform components,” in *Proceedings of the 13th International Conference on Semantic Systems*, 2017, pp. 41–48.
- [84] M. Jacoby, A. AntoniĆ, K. Kreiner, R. Łapacz, and J. Pielorz, “Semantic interoperability as key to iot platform federation,” in *International Workshop on Interoperability and Open-Source Solutions*. Springer, 2016, pp. 3–19.
- [85] S. Grimm, B. Motik, and C. Preist, “Matching semantic service descriptions with local closed-world reasoning,” in *European Semantic Web Conference*. Springer, 2006, pp. 575–589.
- [86] M. Garriga, C. Mateos, A. Flores, A. Cechich, and A. Zunino, “RESTful service composition at a glance: A survey,” *Journal of Network and Computer Applications*, vol. 60, pp. 32–53, 2016, publisher: Elsevier.
- [87] F. Khodadadi and R. O. Sinnott, “A semantic-aware framework for service definition and discovery in the internet of things using coap,” *Procedia computer science*, vol. 113, pp. 146–153, 2017.
- [88] C.-H. Cheng, T. Guelfirat, C. Messinger, J. O. Schmitt, M. Schnelte, and P. Weber, “Semantic degrees for industrie 4.0 engineering: deciding on the degree of semantic formalization to select appropriate technologies,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 1010–1013.

Bibliography

- [89] O. Kovalenko and J. Euzenat, "Semantic matching of engineering data structures," in *Semantic web technologies for intelligent engineering applications*. Springer, 2016, pp. 137–157.
- [90] S. Evdokimov, B. Fabian, S. Kunz, and N. Schoenemann, "Comparison of discovery service architectures for the internet of things," in *2010 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*. IEEE, 2010, pp. 237–244.
- [91] S. Chun, S. Seo, B. Oh, and K.-H. Lee, "Semantic description, discovery and integration for the internet of things," in *Proceedings of the 2015 IEEE 9th International Conference on Semantic Computing (IEEE ICSC 2015)*. IEEE, 2015, pp. 272–275.
- [92] C. Brooks, C. Jerad, H. Kim, E. A. Lee, M. Lohstroh, V. Nouvelletz, B. Osyk, and M. Weber, "A component architecture for the internet of things," *Proceedings of the IEEE*, vol. 106, no. 9, pp. 1527–1542, 2018.
- [93] J. E. Kim, G. Boulos, J. Yackovich, T. Barth, C. Beckel, and D. Mosse, "Seamless integration of heterogeneous devices and access control in smart homes," in *2012 Eighth International Conference on Intelligent Environments*. IEEE, 2012, pp. 206–213.
- [94] O. Uviase and G. Kotonya, "Iot architectural framework: connection and integration framework for iot systems," *arXiv preprint arXiv:1803.04780*, 2018.
- [95] J. Kiljander, A. D'elia, F. Morandi, P. Hyttinen, J. Takalo-Mattila, A. Ylisaukko-Oja, J.-P. Soininen, and T. S. Cinotti, "Semantic interoperability architecture for pervasive computing and internet of things," *IEEE access*, vol. 2, pp. 856–873, 2014.
- [96] I. Grangel-González, L. Halilaj, G. Coskun, S. Auer, D. Collarana, and M. Hoffmeister, "Towards a semantic administrative shell for industry 4.0 components," in *2016 IEEE Tenth International Conference on Semantic Computing (ICSC)*. IEEE, 2016, pp. 230–237.
- [97] I. Grangel-González, D. Collarana, L. Halilaj, S. Lohmann, C. Lange, M.-E. Vidal, and S. Auer, "Alligator: A deductive approach for the integration of industry 4.0 standards," in *European Knowledge Acquisition Workshop*. Springer, 2016, pp. 272–287.
- [98] I. Grangel-González, P. Baptista, L. Halilaj, S. Lohmann, M.-E. Vidal, C. Mader, and S. Auer, "The industry 4.0 standards landscape from a semantic integration perspective," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2017, pp. 1–8.
- [99] H. Koziolak, A. Burger, and J. Doppelhamer, "Self-commissioning industrial iot-systems in process automation: a reference architecture," in *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018, pp. 196–19609.
- [100] V. Jirkovský, M. Obitko, P. Kadera, and V. Mařík, "Toward plug&play cyber-physical system components," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 6, pp. 2803–2811, 2018.

Bibliography

- [101] M. Schleipen, S.-S. Gilani, T. Bischoff, and J. Pfrommer, “OPC UA & Industrie 4.0 - Enabling Technology with High Diversity and Variability,” *Procedia CIRP*, vol. 57, pp. 315–320, Jan. 2016. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S2212827116312094>
- [102] D. Manzaroli, L. Roffia, T. S. Cinotti, E. Ovaska, P. Azzoni, V. Nannini, and S. Matarozzi, “Smart-m3 and osgi: The interoperability platform,” in *The IEEE symposium on Computers and Communications*. IEEE, 2010, pp. 1053–1058.
- [103] H. Rahman and M. I. Hussain, “A comprehensive survey on semantic interoperability for internet of things: State-of-the-art and research challenges,” *Transactions on Emerging Telecommunications Technologies*, p. e3902, 2019.
- [104] J. Nilsson and F. Sandin, “Semantic interoperability in industry 4.0: Survey of recent developments and outlook,” in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE, 2018, pp. 127–132.
- [105] M. I. Ali, N. Ono, M. Kaysar, K. Griffin, and A. Mileo, “A semantic processing framework for iot-enabled communication systems,” in *International semantic web conference*. Springer, 2015, pp. 241–258.
- [106] A. Gyrard, C. Bonnet, K. Boudaoud, and M. Serrano, “Assisting iot projects and developers in designing interoperable semantic web of things applications,” in *2015 IEEE International Conference on Data Science and Data Intensive Systems*. IEEE, 2015, pp. 659–666.
- [107] P. Patel and D. Cassou, “Enabling high-level application development for the internet of things,” *Journal of Systems and Software*, vol. 103, pp. 62–84, 2015.
- [108] I. Grangel-González, L. Halilaj, M.-E. Vidal, O. Rana, S. Lohmann, S. Auer, and A. W. Müller, “Knowledge graphs for semantically integrating cyber-physical systems,” in *International Conference on Database and Expert Systems Applications*. Springer, 2018, pp. 184–199.
- [109] M. Kovatsch, Y. N. Hassan, and S. Mayer, “Practical semantics for the internet of things: Physical states, device mashups, and open questions,” in *2015 5th International Conference on the Internet of Things (IOT)*. IEEE, 2015, pp. 54–61.
- [110] A. M. Nagib and H. S. Hamza, “Sighted: A framework for semantic integration of heterogeneous sensor data on the internet of things,” in *ANT/SEIT*, 2016, pp. 529–536.
- [111] A. J. Mooij, “System integration by developing adapters using a database abstraction,” *Information and Software Technology*, vol. 55, no. 2, pp. 357–364, 2013.
- [112] S. Yang and R. Wei, “Tabdoc approach: An information fusion method to implement semantic interoperability between iot devices and users,” *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 1972–1986, 2018.

Bibliography

- [113] F. Prinz, M. Schoeffler, A. Lechler, and A. Verl, “A novel i4. 0-enabled engineering method and its evaluation,” *The International Journal of Advanced Manufacturing Technology*, vol. 102, no. 5-8, pp. 2245–2263, 2019.
- [114] OPC-UA, “OPC-UA,” 2021. [Online]. Available: <https://opcfoundation.org/developer-tools/specifications-unified-architecture>[retrieved:15.02.2021]
- [115] S. Schwichtenberg, C. Gerth, and G. Engels, “From open api to semantic specifications and code adapters,” in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 484–491.

A. List of Own Publications

1. Burzlaff, Fabian. "Knowledge-driven architecture composition." *INFORMATIK 2017* (2017).
2. Burzlaff, Fabian, and Christian Bartelt. "Knowledge-driven architecture composition: Case-based formalization of integration knowledge to enable automated component coupling." *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017..
3. Burzlaff, Fabian, and Christian Bartelt. "Towards automating service matching for manufacturing systems: Exemplifying knowledge-driven architecture composition." *Procedia CIRP 72* (2018): 707-713.
4. Burzlaff, Fabian, Christian Bartelt, and Heiner Stuckenschmidt. "Next steps in knowledge-driven architecture composition." *CEUR Workshop Proceedings*. Vol. 2191. No. Proceedings of the Conference "Lernen, Wissen, Daten, Analysen". RWTH, 2018.
5. Burzlaff, Fabian, Christian Bartelt, and Steffen Jacobs. "Executing model-based software development for embedded I4. 0 devices properly." *CEUR Workshop Proceedings*. Vol. 2060. RWTH, 2018.
6. Burzlaff, Fabian, and Christian Bartelt. "I4. 0-device integration: A qualitative analysis of methods and technologies utilized by system integrators: Implications for engineering future industrial internet of things system." *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2018.
7. Burzlaff, Fabian, and Christian Bartelt. "A conceptual architecture for enabling future self-adaptive service systems." *Proceedings of the 52nd Hawaii International Conference on System Sciences*. 2019.
8. Fabian, Burzlaff, Ackel Maurice, and Bartelt Christian. "A mapping language for IoT device descriptions." *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. IEEE, 2019..
9. Burzlaff, Fabian, et al. "Semantic interoperability methods for smart service systems: A survey." *IEEE Transactions on Engineering Management* (2019).
10. Burzlaff, Fabian, et al. "MergePoint: A graphical web-app for merging HTTP-endpoints and IoT-platform models." *Proceedings of the 53rd Hawaii International Conference on System Sciences*. 2020.

A. List of Own Publications

11. F. Burzlaff, S. Jacobs, and C. Bartelt, “Automated configuration in adaptive iot software ecosystems to reduce manual device integration effort: Application and evaluation of a novel engineering method.” ADAPTIVE 2020, 2020.
12. N. Wilken, M. Ailane, C. Bartelt, F. Burzlaff, C. Knieke, S. Lawrenz, A. Rausch and A. Strasser, “Dynamic Adaptive System Composition Driven By Emergence in an IoT Based Environment: Architecture and Challenges.” ADAPTIVE 2020, 2020.
13. F. Burzlaff and C. Bartelt, “Knowledge-driven Architecture Composition: Assisting the system integrator to reuse integration knowledge (to be published).” in International Conference on Web Engineering. Springer, 2021, p. tba

B. Usage Examples

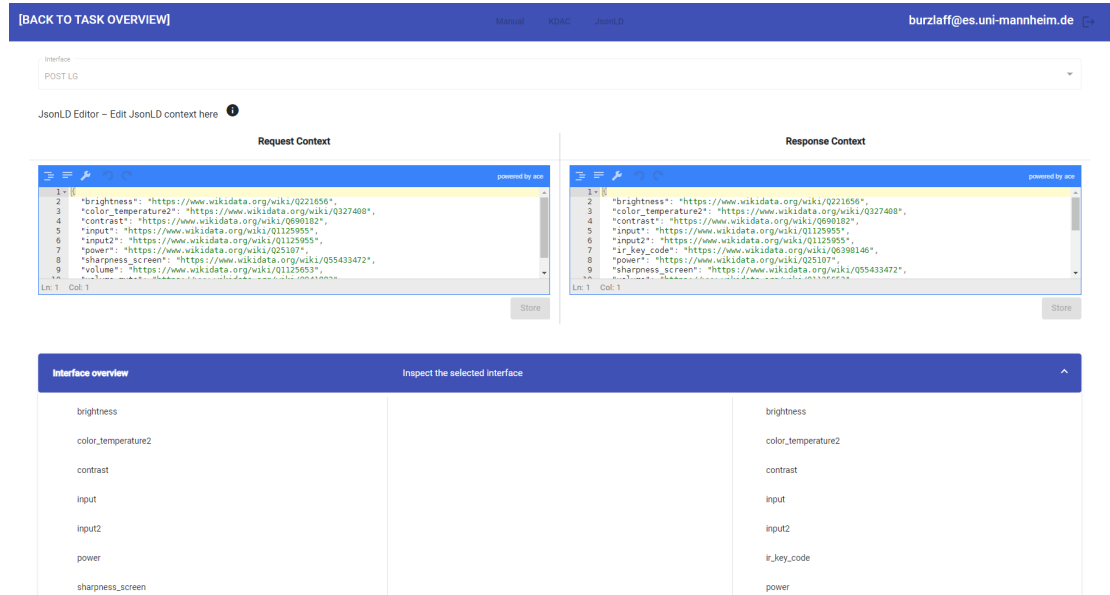


Figure B.1.: Eval 2 – Describing Context Based on An Ontology and JSON-LD

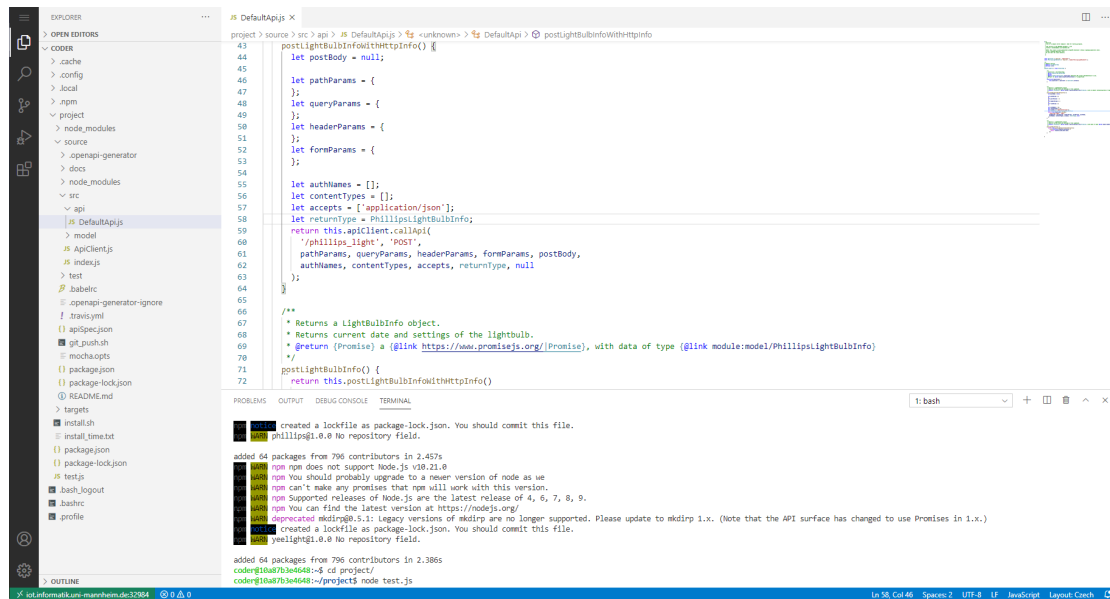
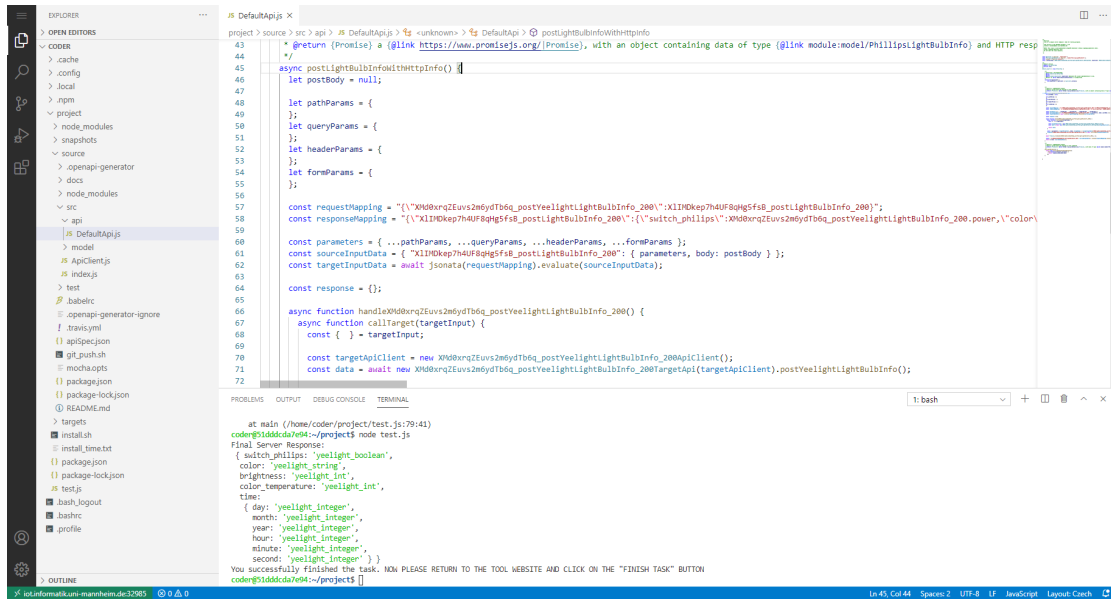


Figure B.2.: Eval 3 – Generated Software Adapter Project without Mappings

B. Usage Examples



```
project > source > src > api > DefaultApi.js > <unknown> > DefaultApi > postLightBulbInfoWithHttpInfo
43 * @return (Promise) a (@link https://www.promisejs.org/Promise), with an object containing data of type (@link module:model/PhilipsLightBulbInfo) and HTTP resp
44 *
45 async postLightBulbInfoWithHttpInfo() {
46   let postBody = null;
47
48   let pathParams = {
49     };
50   let queryParams = {
51     };
52   let headerParams = {
53     };
54   let formParams = {
55     };
56
57   const requestMapping = "{X11DKep74Uf8qH5fS8_postVeeLightBulbInfo_200}";
58   const responseMapping = "{X11DKep74Uf8qH5fS8_postLightBulbInfo_200}";
59
60   const parameters = { ...pathParams, ...queryParams, ...headerParams, ...formParams };
61   const sourceInputData = { "X11DKep74Uf8qH5fS8_postLightBulbInfo_200": { parameters, body: postBody } };
62   const targetInputData = await jsonata(requestMapping).evaluate(sourceInputData);
63
64   const response = {};
65
66   async function handleX11DKep74Uf8qH5fS8_postVeeLightBulbInfo_200() {
67     async function callTarget(targetInput) {
68       const {} = targetInput;
69
70       const targetApiClient = new X11DKep74Uf8qH5fS8_postVeeLightBulbInfo_200ApiClient();
71       const data = await new X11DKep74Uf8qH5fS8_postVeeLightBulbInfo_200TargetApi(targetApiClient).postVeeLightBulbInfo();
72
73     }
74
75     return callTarget(targetInput);
76   }
77
78   return callTarget(targetInput);
79 }
80
81 export default {
82   postLightBulbInfoWithHttpInfo: handleX11DKep74Uf8qH5fS8_postVeeLightBulbInfo_200
83 };
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

```
at main (/home/coder/projects/test-js:79:41)
coder@51d6d4d7e04:~/projects$ node test.js
Final Server Response:
{ switch_philips: 'yeelight_boolean',
  color: 'yeelight_string',
  brightness: 'yeelight_int',
  color_temperature: 'yeelight_int',
  time:
  { day: 'yeelight_integer',
    month: 'yeelight_integer',
    hour: 'yeelight_integer',
    minute: 'yeelight_integer',
    second: 'yeelight_integer' } }
You successfully finished the task. NOW PLEASE RETURN TO THE TOOL WEBSITE AND CLICK ON THE "FINISH TASK" BUTTON
coder@51d6d4d7e04:~/projects$
```

Figure B.3.: Eval 3 – Generated Software Adapter Project with Mappings

B.1. Link to Prototype

Additional information about the current state of the prototype can be found at the website <https://iot.informatik.uni-mannheim.de/>.

