Knowledge-Driven Architecture Composition: Assisting the System Integrator to Reuse Integration Knowledge [*]

Fabian Burzlaff[0000-0003-0632-5933] (✉) and Christian Bartelt[0000-0003-0426-6714]

Institute for Enterprise Systems (InES), University of Mannheim, 68131 Mannheim, Germany
`burzlaff@es.uni-mannheim.de`
`bartelt@es.uni-mannheim.de`

Abstract. Semantic interoperability for web services is still a problem. Although decentralized solutions such as describing the integration context with a formal mapping language or using a web service description language exist, practitioners rely on implementing software adapters manually. For IoT and Web of Things systems, current scientific solutions fall short as changing them, once defined, requires strenuous effort. However, devices and thus, their interfaces change often in this class of system. This paper tackles the barrier of high formalization effort for mappings between required and provided interfaces. Therefore, we apply and evaluate a novel integration method for web service choreography. Our empirical experiment shows that this method lowers the integration time and number of errors by assisting the system integrator to reuse integration knowledge from previous integration cases.

Keywords: Knowledge-Driven Architecture Composition · Web Service Integration · Reuse.

1 Introduction

In the universe of IoT, there will not exist one distinct standard for each use-case [1]. The agreement process 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's 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

[*] This work has been developed in the project BIoTope (Research Grant Number 01lS18079C) and is funded by the German Ministry of Education and Research (BMBF).
The final authenticated publication is available online at `https://doi.org/10.1007/978-3-030-74296-6_23`

descriptions and interface mappings. These integration contexts are defined with a closed-wold assumption in mind and relate to the concept of service choreography. However, if an unforeseen integration case comes up, no automated service integration occurs as the composition model is assumed to be complete. In dynamically changing IoT environments, this results in a high formalization effort that does not yield the desired benefits as structural and behavioral interface mappings are assumed to be stable once they are defined. High formalization effort such as adapting interface service descriptions, adjusting the underlying ontology or resolving reasoning errors rather increases than decreases integration effort over time. Furthermore, formalizing possible integration contexts ahead to put them into inventory increases the specification effort even more as they may not be used [2].
Within this gap, a novel integration method called knowledge-driven architecture composition (KDAC) has been suggested [3]. This method does not aim at a stable composition model of the desired domain. In contrast to existing bottom-up solution proposals, it refrains from formalizing integration contexts in a big-bang manner at system design time. Instead, the approach explicitly allows for interface mappings that are formalized incrementally and are thus incomplete. Interface mappings are only written in a machine-understandable way if a concrete integration case is present.
In this paper, we evaluate this so far conceptual method for web service composition. Therefore, we design an empirical experiment and build up the necessary tooling infrastructure. The method and tooling may assist the system integrator in reusing existing integration knowledge and lowering the required implementation effort. However, it is unclear if formalizing integration knowledge and implementing a software adapter in the beginning results in lower integration effort due to integration knowledge reuse over time.

2 Background for Applied Approach

The goal of KDAC is to assist the system integrator in generating software adapters automatically. The leverage of the method is to make integration knowledge reusable and reason about interface mappings [3]. Therefore, interface mappings must be stored in a machine-understandable way and made publicly available. These mappings must respect the semantic interoperability of services (e.g., REST). Semantic interoperability ensures that data exchanges between a provided and a required service make sense – that the requester and provider have a common understanding of the meaning of services and data [7]. Semantic interoperability in distributed systems is mainly achieved by establishing semantic correspondences (i.e., mappings) between vocabularies of different sources [1, 14].
From an engineering perspective, software adaptability (e.g., service choreography) can be 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) [15]. 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 using predefined composition models (e.g., as known from component-based software engineering). 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 (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 standards) and other knowledge-based bottom-up methods (e.g., describing an integration context using ontologies), KDAC explicitly allows for incomplete integration knowledge at all times.

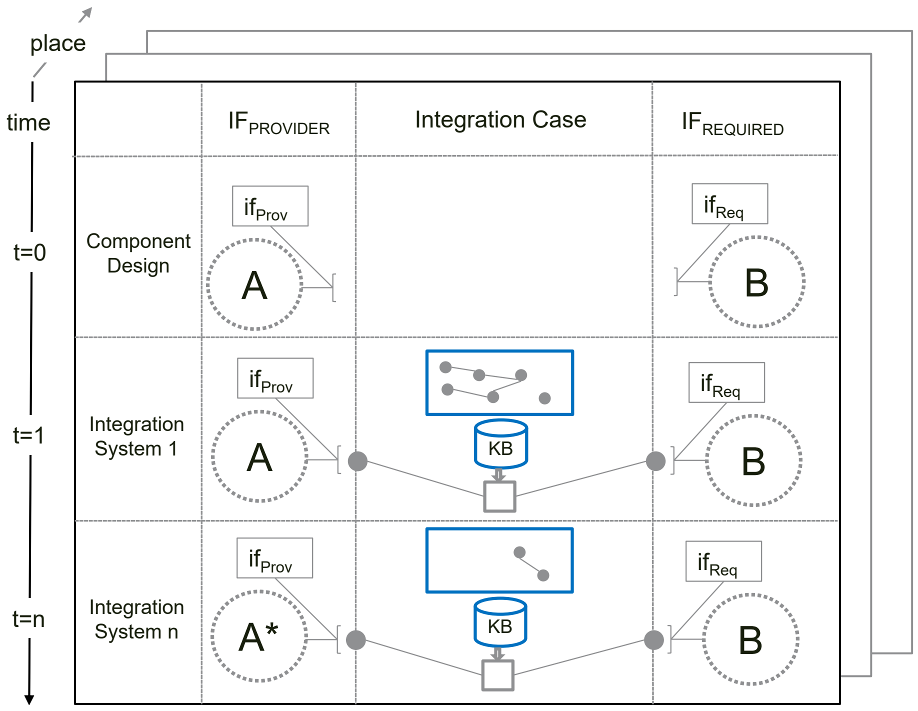


Fig. 1: Knowledge-Driven Architecture Composition [adapted from [3]]

Evolution: In the beginning, the human-in-the-loop principle applies as the underlying knowledge base is empty (see KB in Fig. 1). 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). 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 reusing mappings and reasoning principles (see dots and lines at t=n in Fig. 1).

Emergent: Although integration knowledge is incomplete, automated integration 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 two reasoning principles which are transitive relationships and inverse mappings. Moreover, we can integrate unforeseen component replacements without human anticipation.

In contrast, detecting semantic interoperability for ad-hoc integration cases using a software adapter pattern is always a manual task.

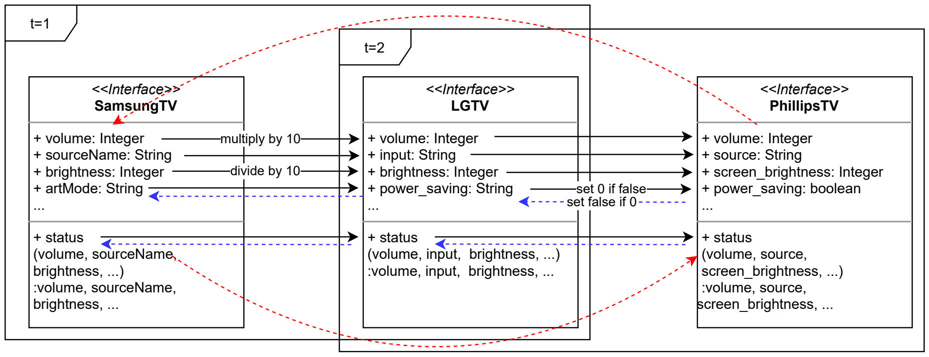


Fig. 2: Reasoning Example

2.1 Integration Knowledge Reuse Example

Assume for component A the interface of a Samsung TV and for component B the interface of an LG TV (see Fig. 2). At t=1, the method "status" and its input and output parameters are mapped. A formalized mapping function can include an attribute 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 for the Samsung and LG interface, all mappings have to be created manually by the system integrator. At t=n, these mapping functions can be reused for the same integration case or for the inverse integration case (i.e., LG TV is substituted by the Samsung TV). Furthermore, we can also reuse formalized mappings for extensions of already seen interfaces (i.e., indicated by component A* in Fig. 1).

For a transitive mapping chain, assume another integration from LG TV to a Philips TV at t=2 (see Fig. 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 function within the chain Philips TV $\leftrightarrow$ LG TV $\leftrightarrow$ Samsung TV. Hence, as soon as the system integrator selects the required and provided interfaces based on the available components, a software adapter can be (partially) generated.
However, integration knowledge is always incomplete, as not all methods offered by all available devices and their possible combinations are formalized or can be derived.

3 Evaluation Design and Results

In this experiment, we illustrate and test the end-to-end application of the proposed method for web services. The participants have to work in two environments. This also allows for editing mappings within the mapping and coding environment. The central evaluation goal is to compare implementing software adapters, generating software adapters without reasoning principles, and generating software adapters with reasoning principles. Thereby we allow reusing mapping functions between attributes and methods (see Fig. 2). As an effort indicator, we measure the integration time and the number of mapping errors and discuss problems during software adapter implementation.

3.1 Evaluation Setup

Challenge: It is unclear how KDAC can assist the system integrator during software adapter (SA) implementation. Especially, the additional time to formalize mappings should result in a working software adapter.
Experiment Design: We empirically compare the software adapter implementation method against KDAC applying a within-subject design [12, 17]. SA represents implementing a software adapter. KDAC is split up in generating software adapter without any mappings stored in the knowledge base (variant 1 – no reasoning) and with mappings stored in the knowledge (variant 2 – reasoning). Hence, we can compare the mapping time and errors made by the system integrator and, if any, made by the reasoning algorithms. Thereby we focus on creating correct mappings according to their semantic interoperability.
Therefore, three to five integration cases have been assigned to four students each week (i.e., 16th October 2020 until 16th December 2020).
Participants: The students study 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.
Experiment Scope: As we are interested in the method's performance and not in the underlying technology, we chose a technology stack that can be utilized by both methods (i.e., SA and KDAC). Regarding the underyling KDCA method, we focus on empty knowledge-bases and high formalization effort in the beginning (i.e., t=0 and t=1 within Fig. 1). We do not evaluate how reasoning

principles and the underlying method perform on a large knowledge base (i.e., t=0 in Fig. 1).

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

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

Hypothesis: The independent variable is the engineering method. The dependent variables are integration time and component interaction correctness. We suspect that the component interaction correctness and integration time is highest using the KDAC method (see Fig. 1).

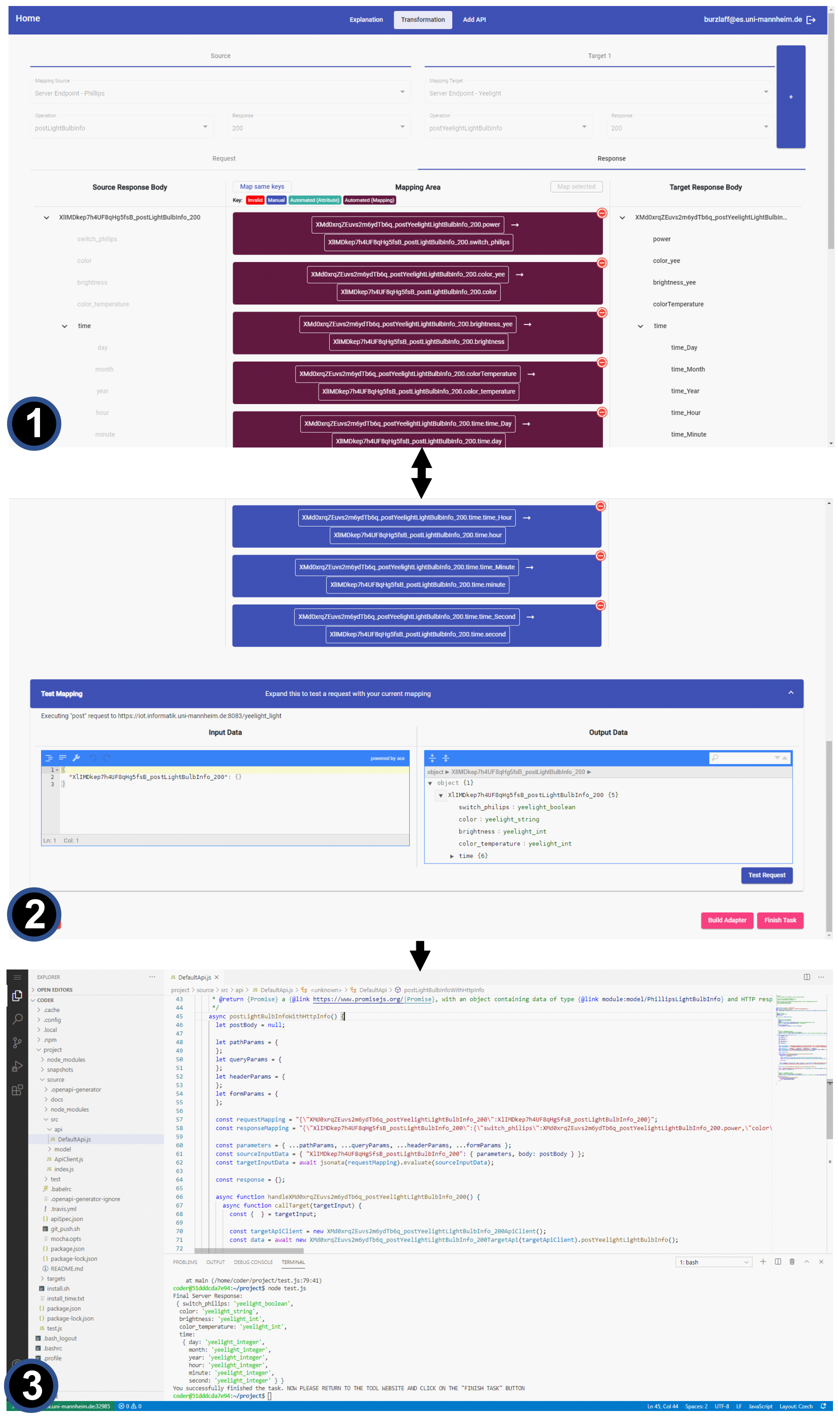
Technology Stack: We rely on the HTTP/JSON component model using POST and GET service calls. For specifying mapping functions in a declarative way, we use JSONata [9], and for implementing the software adapter, we use the Visual Studio Code Web IDE. For implementing the software adapter, we choose NodeJS. A project setup script is provided so that the participants can 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/`). Here, OpenAPI refers to a syntactical description of service instances (i.e., device abstraction) that does not support any relationship to a machine-readable or machine-understandable domain standard (e.g., URL links to an ontology). Then, service instances from the OpenAPI specifications have been mocked.

3.2 Evaluation Execution Process

The leitmotif for the students is 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 precisely 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.

A student took the role of a System Integrator. Six measurement runs using three different use cases were executed autonomously by the students. The integration contexts are illumination, music, and television (see running example).



Mapping Source: Server Endpoint - Phillips
Operation: postLightBulbInfo
Response: 200
Mapping Target: Server Endpoint - Yeelight
Operation: postYeelightLightBulbInfo
Response: 200

XMd0xrqZEuvs2m6ydTb6q_postYeelightLightBulbInfo_200.power → XlIMDkep7h4UF8qHg5fsB_postLightBulbInfo_200.switch_philips
XMd0xrqZEuvs2m6ydTb6q_postYeelightLightBulbInfo_200.color_yee → XlIMDkep7h4UF8qHg5fsB_postLightBulbInfo_200.color
XMd0xrqZEuvs2m6ydTb6q_postYeelightLightBulbInfo_200.brightness_yee → XlIMDkep7h4UF8qHg5fsB_postLightBulbInfo_200.brightness
XMd0xrqZEuvs2m6ydTb6q_postYeelightLightBulbInfo_200.colorTemperature → XlIMDkep7h4UF8qHg5fsB_postLightBulbInfo_200.color_temperature
XMd0xrqZEuvs2m6ydTb6q_postYeelightLightBulbInfo_200.time.time_Day → XlIMDkep7h4UF8qHg5fsB_postLightBulbInfo_200.time.day
XMd0xrqZEuvs2m6ydTb6q_postYeelightLightBulbInfo_200.time.time_Hour → XlIMDkep7h4UF8qHg5fsB_postLightBulbInfo_200.time.hour
XMd0xrqZEuvs2m6ydTb6q_postYeelightLightBulbInfo_200.time.time_Minute → XlIMDkep7h4UF8qHg5fsB_postLightBulbInfo_200.time.minute
XMd0xrqZEuvs2m6ydTb6q_postYeelightLightBulbInfo_200.time.time_Second → XlIMDkep7h4UF8qHg5fsB_postLightBulbInfo_200.time.second

Executing "post" request to https://iot.informatik.uni-mannheim.de:8083/yeelight_light

```
{
  "XlIMDkep7h4UF8qHg5fsB_postLightBulbInfo_200": {}
}
```

```
object ► XlIMDkep7h4UF8qHg5fsB_postLightBulbInfo_200 ►
▼ object {1}
  ▼ XlIMDkep7h4UF8qHg5fsB_postLightBulbInfo_200 {5}
      switch_philips : yeelight_boolean
      color : yeelight_string
      brightness : yeelight_int
      color_temperature : yeelight_int
    ► time {6}
```

```
   * @return {Promise} a {@link https://www.promisejs.org/|Promise}, with an object containing data of type {@link module:model/PhillipsLightBulbInfo} and HTTP resp
   */
  async postLightBulbInfoWithHttpInfo() {
    let postBody = null;

    let pathParams = {
    };
    let queryParams = {
    };
    let headerParams = {
    };
    let formParams = {
    };

    const requestMapping = "{\"XMd0xrqZEuvs2m6ydTb6q_postYeelightLightBulbInfo_200\":XlIMDkep7h4UF8qHg5fsB_postLightBulbInfo_200}";
    const responseMapping = "{\"XlIMDkep7h4UF8qHg5fsB_postLightBulbInfo_200\":{\"switch_philips\":XMd0xrqZEuvs2m6ydTb6q_postYeelightLightBulbInfo_200.power,\"color\

    const parameters = { ...pathParams, ...queryParams, ...headerParams, ...formParams };
    const sourceInputData = { "XlIMDkep7h4UF8qHg5fsB_postLightBulbInfo_200": { parameters, body: postBody } };
    const targetInputData = await jsonata(requestMapping).evaluate(sourceInputData);

    const response = {};

    async function handleXMd0xrqZEuvs2m6ydTb6q_postYeelightLightBulbInfo_200() {
      async function callTarget(targetInput) {
        const {  } = targetInput;

        const targetApiClient = new XMd0xrqZEuvs2m6ydTb6q_postYeelightLightBulbInfo_200ApiClient();
        const data = await new XMd0xrqZEuvs2m6ydTb6q_postYeelightLightBulbInfo_200TargetApi(targetApiClient).postYeelightLightBulbInfo();
```

```
    at main (/home/coder/project/test.js:79:41)
coder@51dddcda7e94:~/project$ 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',
     year: '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@51dddcda7e94:~/project$
```

Fig. 3: Evaluation Steps

The use case can be summarized as "As a mobile application user, I want to control all available devices in my current room by only using one application". 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 are selected, then no mappings can be shown at any time. Hence, the students could only continue to generate the software adapter and start implementing.
KDAC: In both variants, a mapping can be tested before generating the software adapter within the KDAC tool (i.e., perform a request to a provided service instance). This can be done as soon as all request attributes from the provided interface and all response attributes from the required interface are mapped.
A search over all stored mappings is performed when a mapping source and mapping target are selected. All computed mapping functions for the source (i.e., required) and target (i.e., provided) interfaces are automatically inserted and visualized in the KDAC tool. They can be edited at any time during the evaluation process. All attributes from the selected provided and the required interfaces can be used within one mapping function.

The students performed the following steps (see Fig. 3): First, they selected a task from the task overview. Second, the type of tasks determines how mappings are populated. If the tool is only used to generate the software adapter project, no mappings are populated, and students can only generate the project. If the tool is used to create mappings between operations, the students can inspect and specify mappings. This view also symbolizes the first variant of the KDAC method. If there are already mappings within the knowledge base, then the reasoning principles are applied and populated within the mapping view. Inferred mappings are annotated with a green or merlot color (see 1 in Fig. 3) and manually inserted mappings are annotated with a blue color (see 2 in Fig. 3). In the first KDAC variant, only manual mappings are available. In the second KDAC variant, manual and inferred mappings are available. Only when all calls succeed mappings are stored within the knowledge base, and the students may proceed to login into the Web IDE. This mandatory reliability feature ensures semantic interoperability.
Last, the students must resolve all dependencies in the underlying Node.js environment by executing an install script and provide their username and password for authentication towards the knowledge base. If the tool is only used to generate the adapter skeleton, then the method that should contain the actual transformations had to be implemented. Suppose the tool was used to formalize mappings or mappings have been computed based on the reasoning principles. In that case, these mappings are inserted into the software adapter code that had to be implemented (see 3 in Fig. 3).
Finally, the students can check anytime by executing a test script if their operationalized mappings are correct according to the test criterion (see 3 Fig. 3).

If this is the case, then a corresponding message is printed on the terminal, and the students end the task by switching back to the KDAC tool and click the finish task button (see 2 in Fig. 3).

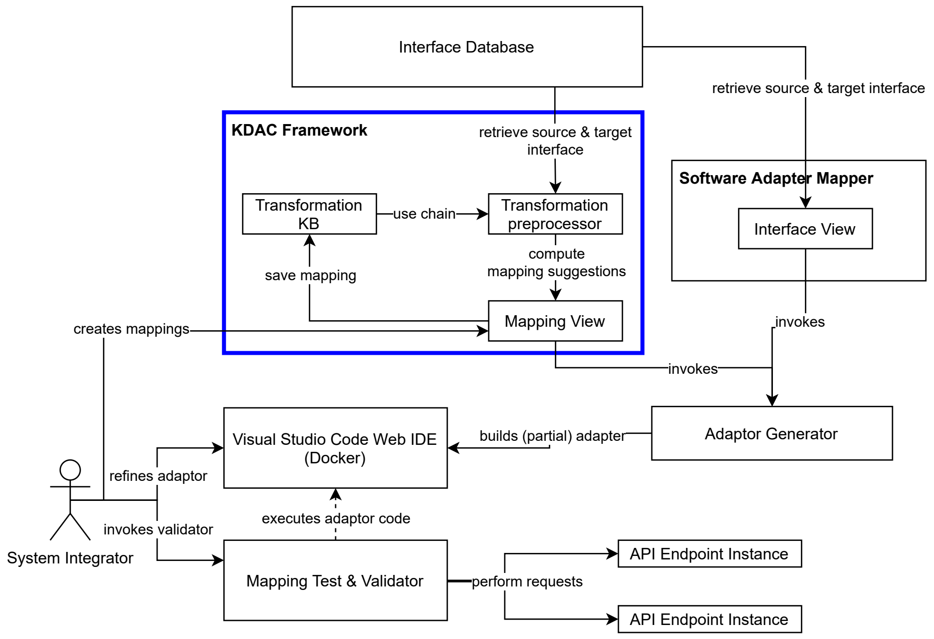


Fig. 4: High-Level System Architecture for Evaluation Setup

3.3 Implementation

The overall KDAC framework is built up of three parts responsible for generating interface mappings. In addition, a generic *Mapping Test & Validator* for testing the created mappings was implemented (see Fig. 4). Depending on the task type, a preprocessor might be applied. Their duty is to populate the *Mapping View* with automatically created suggestions of mapping functions. In the case of the first variant of the KDAC method (i.e., no reasoning), the web-tool only provides a graphical user interface for specifying mappings with JSNOata. The web-tool is used to generate the software adapter project skeleton so that both approaches are as similar as possible. Hence, the first and second variant only differentiate in whether existing mappings are evaluated or not. In the case of the second variant of the KDAC method, the *Transformation preprocessor* is invoked. It first tries to find a transitive mapping chain between the selected source and target interface using a breadth-first search on the *Transformation KB*. Once such a chain is identified, the preprocessor recursively applies the mappings stored in JSONata to each other, producing a final mapping from the source to the target interface (i.e., POST Samsung → POST LG). This is done for both, request and response data.

3.4 Results

We captured 108 one-to-one interface integration tasks to validate our hypothesis. There are 9 integration tasks for the Software Adapter Implementation method, 9 for the first KDAC variant and 9 for the second KDAC variant. Each integration task has been repeated 4 times during the evaluation period. It was made sure that one student did not work on a similar or identical integration task during a period of three weeks.

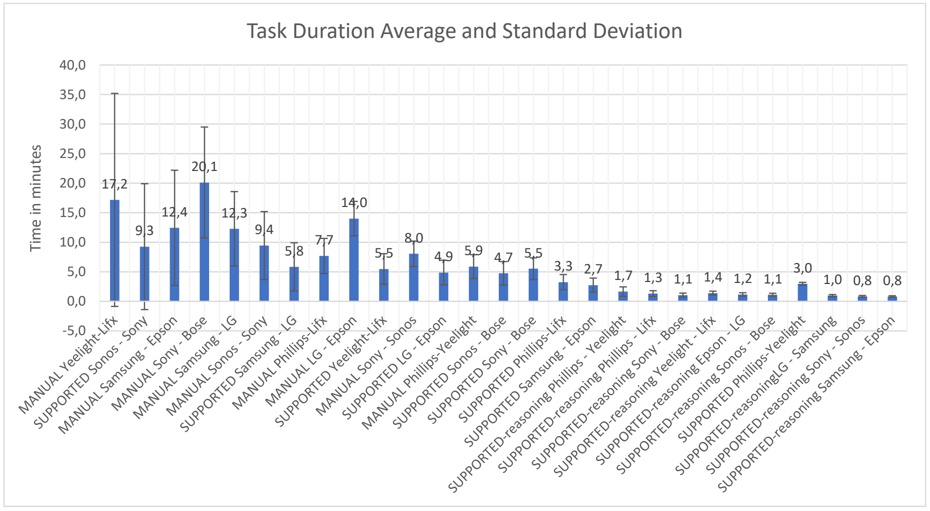
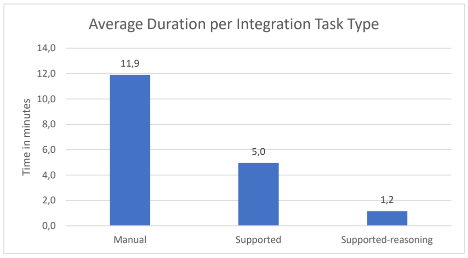


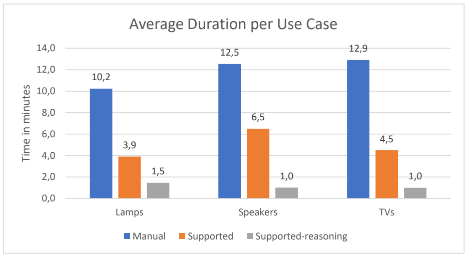
Fig. 5: Average and Standard Deviation for all integration tasks

Fig. 5 outlines the duration average for all integration tasks. An integration task involved ten to 16 attributes that had to be mapped. The integration time is measured in minutes, and the description of each task involves the integration task type. Here, "MANUAL" corresponds to only using the tool (see Fig. 3) as a software adapter generation environment where all mapping logic has to be implemented in the generated adapter project. "SUPPORTED" relates to the first variant of the KDAC method, where mappings between interfaces are defined using JSONata. Last, "SUPPORTED-reasoning" is the second variant of the KDAC method with reasoning and integration knowledge reuse. The devices from Sony, Bose, and Sonos are speakers, Yeelight, Lifx, and Philips are lamps, and Epson, LG, and Samsung are TVs.
Overall, the average time needed for constructing a working software adapter is the highest for implementing software adapters and the lowest when mappings

can be reused. Furthermore, the manual task's standard deviation is higher compared to the second variant of the KDAC method. This is mainly because of the presence or absence of errors during code writing. The number of attributes does not seem to directly affect the average integration time as the highest value of 20.1 minutes had 13 attributes to be mapped. The first integration task with 16 attributes scored an average duration of 14 minutes.

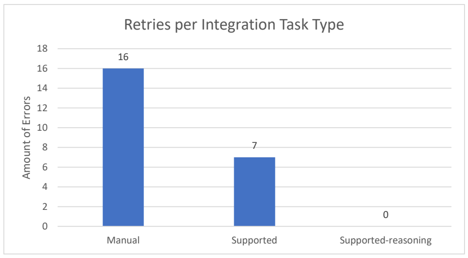


(a) Average Integration Duration per Method

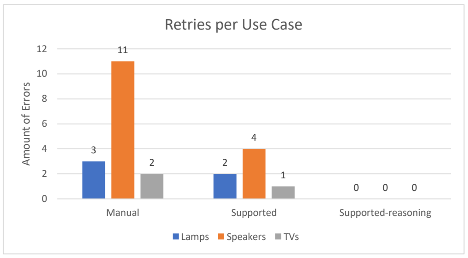


(b) Average Integration Duration per Use Case

Fig. 6: Metric Integration time



(a) Average Retries per Task Type



(b) Average Retries per Use Case

Fig. 7: Metric Errors

Fig. 6a illustrates the average integration time per engineering method and Fig. 6b the average integration time per use case. On average, the participants need 11.9 minutes to implement a software adapter, 5.0 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 tasks type, the same number of attributes had to be mapped (i.e., 117 attributes in total). For the three use cases, this equality does 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 attributes in total) lasted

10.2 minutes, speakers (117 attributes in total) lasted 12.5 minutes, and TVs (141 attributes in total) lasted 12.9 minutes. The average integration times are highest for the manual integration task types and lowest for the second variant of the KDAC method.
Fig. 7a and Fig. 7b 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 the errors made is 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 if no automation is involved (e.g., as for 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 are: 1) Missing or wrong attributes in the result 2) result object is undefined 3) result object is empty 4) attribute hierarchy was ignored 5) attribute 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 attribute. No errors have been made for the second variant of the KDAC method.
Error resolving strategies for all methods include the usage of logging functionality offered by the IDE. Regarding the manual method, this allowed for identifying attributes 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 have been mainly resolved by adjusting the inserted JSONata mapping strings directly in the software adapter. However, this case can be traced back to a non-use of the Mapping Test & Validator (see Fig. 4) as no incorrect mappings should be stored in the knowledge base.
We suspected that the component interaction correctness and integration time is highest using the KDAC method. Based on the data collected, we can summarize that the second variant of the KDAC method has the highest component interaction correctness (i.e., no errors made), and the integration time is lowest using the second variant of the KDAC method as well. However, the first variant of the KDAC method involved some errors. Nevertheless, a low number of the student population and the applicability of reasoning principles allow for improvement.

3.5 Threats to 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 measure the results for using the graphical tool without reuse and reasoning functionality (i.e., KDAC variant 1).

Consequently, we can 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 KDAC in addition to implementing the software adapter pays off. This is mainly due to the challenge of collecting realistic engineering data over time.
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 are randomly assigned to the students, but it is made sure that no student works on the same integration task in subsequent measurement runs.
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 is made sure during OpenAPI interface description selection that mappings could be chained 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. However, the tool also supports one-to-many mappings. Nevertheless, manual mappings are inevitable if there are multiple paths from a source to a target interface within the knowledge base.
The evaluation focuses on the engineering method. Hence, different technologies might have produced other results. We assume that more complex interface descriptions (e.g. using stateful services) would slow our approach down.
Last, there exists a learning curve by the students for all use cases. The first integration contexts worked on (i.e., lamps) have a higher standard variation than the later use case (i.e., TVs). This learning curve applied to all students and all task types as they had no prior experience in implementing software adapters or using the KDAC. Here, no experience can be measured more precisely than some experience.

4 Related Work

There are four different research streams that deal with semantic service interoperability for various system classes (e.g., web services, interactive systems, or embedded systems) based on interface mappings [4]. These are symbolic artificial intelligence [11], component-based software development [5], software architecture [2, 16], and web services [6, 8] .
For web service composition approaches with an explicit semantic layer, the following approaches are related to KDAC. Bennaceur et al. [2] present a fully automatable approach that achieves interoperability through semantics-based technologies. Their approach uses a domain-specific ontology, already annotated services based in SAWSDL, and model-checking techniques to generate correct-by-construction mediators automatically. They target the run time phase and minimize additional specification effort by using reasoning principles.
Khodadadi et al. [10] 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 service integration as services are annotated incrementally.
Kovatsch et al. [13] 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 is a potential obstacle to applying their approach out-of-the-box.
Like KDAC, all approaches describe the integration context, such as in our example (see section 2.1) in a decentralized manner. Hence, no global standard is used by any of the approaches. However, Kovatsch et al. [13] and Bennaceur et al. [2] assume that their decentralized integration context is complete (i.e., contains also all needed interface mappings for future cases). If a change occurs, updating these mappings requires substantial effort. Here, KDAC allows for incomplete mappings that can be easily edited. Khodadadi et al. [10] also support incompleteness by incrementally annotating data JSON data. However, they provide no leverage to support mapping creation as they only focus on creating interface descriptions. This means that only identical integration contexts can be solved. Here, KDAC offers reasoning principles to integrate also unseen integration cases.

5 Conclusion

Semantic interoperability for web services is still a problem for IoT and Web of Things systems. In this paper, we lower the formalization effort for web services and their integration context by applying and evaluating an integration method that makes use-case specific integration knowledge reusable. Therefore, we performed an empirical experiment that compares manual software adapter implementation with the knowledge-driven integration method. Our results suggest that, over time, reusing incrementally formalized integration knowledge is indeed faster than implementing software adapters manually without any integration knowledge reuse. In the future, we plan to extend the mapping language used to cover other domains that do not only rely on the HATEOAS principle for web services (e.g., cyber-physical systems).

References

1. Barnaghi, P., Wang, W., Henson, C., Taylor, K.: Semantics for the internet of things: early progress and back to the future. International Journal on Semantic Web and Information Systems (IJSWIS) **8**(1), 1–21 (2012)
2. Bennaceur, A., Issarny, V.: Automated Synthesis of Mediators to Support Component Interoperability. IEEE Transactions on Software Engineering **41**(3), 221–240 (Mar 2015). https://doi.org/10.1109/TSE.2014.2364844
3. Burzlaff, F., Bartelt, C.: 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). pp. 108–111. IEEE (2017)

4. Burzlaff, F., Wilken, N., Bartelt, C., Stuckenschmidt, H.: Semantic Interoperability Methods for Smart Service Systems: A Survey. IEEE Transactions on Engineering Management pp. 1–15 (2019). https://doi.org/10.1109/TEM.2019.2922103
5. Chang, H., Mariani, L., Pezze, M.: In-field healing of integration problems with COTS components. In: 2009 IEEE 31st International Conference on Software Engineering. pp. 166–176. IEEE (2009)
6. Garriga, M., Mateos, C., Flores, A., Cechich, A., Zunino, A.: RESTful service composition at a glance: A survey. Journal of Network and Computer Applications **60**, 32–53 (2016), publisher: Elsevier
7. Heiler, S.: Semantic interoperability. ACM Computing Surveys (CSUR) **27**(2), 271–273 (1995)
8. Jara, A.J., Olivieri, A.C., Bocchi, Y., Jung, M., Kastner, W., Skarmeta, A.F.: 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 **10**(2-3), 244–272 (2014)
9. JSONata: Json query and transformation language, `https://jsonata.org/[retrieved:2020.10.29]`
10. Khodadadi, F., Sinnott, R.O.: A semantic-aware framework for service definition and discovery in the internet of things using coap. Procedia computer science **113**, 146–153 (2017)
11. Klusch, M., Kapahnke, P., Zinnikus, I.: SAWSDL-MX2: A Machine-Learning Approach for Integrating Semantic Web Service Matchmaking Variants. In: 2009 IEEE International Conference on Web Services. pp. 335–342 (Jul 2009). https://doi.org/10.1109/ICWS.2009.76
12. Ko, A.J., Latoza, T.D., Burnett, M.M.: A practical guide to controlled experiments of software engineering tools with human participants. Empirical Software Engineering **20**(1), 110–141 (2015)
13. Kovatsch, M., Hassan, Y.N., Mayer, S.: 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). pp. 54–61. IEEE (2015)
14. Noy, N.F., Doan, A., Halevy, A.Y.: Semantic integration. AI magazine **26**(1), 7–7 (2005)
15. Rausch, A., Bartelt, C., Herold, S., Klus, H., Niebuhr, D.: From Software Systems to Complex Software Ecosystems: Model- and Constraint-Based Engineering of Ecosystems. In: Münch, J., Schmid, K. (eds.) Perspectives on the Future of Software Engineering: Essays in Honor of Dieter Rombach, pp. 61–80. Springer, Berlin, Heidelberg (2013)
16. Spalazzese, R., Inverardi, P.: Mediating Connector Patterns for Components Interoperability. In: Babar, M.A., Gorton, I. (eds.) Software Architecture. pp. 335–343. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2010)
17. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)