Automated Configuration in Adaptive IoT Software Ecosystems to Reduce Manual Device Integration Effort: Application and Evaluation of a Novel Engineering Method

Fabian Burzlaff
Institute for Enterprise Systems (InES)
University of Mannheim
Email: burzlaff@es.uni-mannheim.de

Steffen Jacobs
Institute for Enterprise Systems (InES)
University of Mannheim
Email: steffen.jacobs@hotmail.de

Christian Bartelt
Institute for Software and Systems Engineering (ISSE)
Clausthal University of Technology
Email: christian.bartelt@tu-clausthal.de

***Abstract*—Existing scientific approaches that integrate non-standardized software components automatically are seldom used in practice. Consequently, practitioners currently rely on standardization initiatives or they implement software adapters manually. However, standards quickly lose their claim for correctness as fast innovation cycles prohibit timeliness of machine-understandable domain standards for open and dynamically evolving software ecosystems. Although scientifically driven approaches can be applied in order to automatically generate software adapters reliably, they require a formal mapping specification for all possible integrations between a provided and a required interface at system design-time. In contrast, imprecise matching approaches based on service specifications can be applied at run-time but cannot produce reliable interface mappings. In this paper, we provide our first evaluation for a novel integration method that can integrate components automatically based on incomplete mapping knowledge. Although this method explicitly embraces manual integration efforts, we aim at achieving automatic adapter generation by making additionally formalized integration knowledge reusable. By storing integration knowledge only when a concrete use case is present, generated software adapters remain reliable. Using an empirical within-subject evaluation design, we quantify how reusing formal interface mappings can speed up integration tasks in an agile development setting. We expect the proposed method to be applied in adaptive software ecosystems that evolve in short innovation cycles such as the Internet-of-Things.**

Keywords–Semantic Interoperability; Knowledge Reuse; Software Component Compatibility; Engineering Methods

I. INTRODUCTION

Manual integration effort for open dependable software systems currently increases exponentially [1]. Although there exist standardization initiatives, semantic service interoperability is still a challenge [2] [3]. Software innovation cycles iterate faster each year and top-down standardization initiatives are too slow to keep up with the increasing rate of change. Furthermore, most standards pre-define a machine-readable (can be parsed) or machine-understandable (can be reasoned about) domain-specific vocabulary by relying on the assumption that syntactically identical words also expose the same meaning. Despite the fact that formalized and executable model checkers to ensure software component interoperability exist (e.g., AUTOSAR [4] for the automotive industry), standards describing a domain completely and unambiguously are currently not applied by other industries. Hence, independent software engineers implement point-to-point adapters without having the ability to reuse existing integration knowledge across software ecosystems. Standards quickly lose their claim for correctness as fast innovation cycles prohibit timeliness of domain knowledge for open and dynamically evolving software systems [5].

Formal approaches can couple heterogeneous software components in an automated way [6] [7]. Although selected semantic interface mappings can be deduced by exploiting reasoning, the applicability of such logic-based approaches is limited in practice. One key challenge such approaches face is the high specification effort needed upfront for open software ecosystems [6] [8]. Describing all interfaces for use cases based on fixed requirements and an increasing number of Internet-of-Things (IoT) devices is not practical for most open software systems [9]. From a platform owner viewpoint, it is not guaranteed that formalized integration knowledge derived from requirements is used during run time by the available components. This is mainly due to the circumstance that end-users express their requirements and use cases during run-time (e.g., by using *If-this-then-that* statements) and IoT device manufacturers cannot be forced to adhere to one service specification syntax and one domain-specific model. Formal mapping approaches can automatically generate highly reliable software adapters for closed and well-defined application contexts. They assume that integration knowledge is specified in an almost complete manner.

Service Matching Solutions, such as semantic interface matching approaches [10] [11] or fuzzy matching approaches [12] for comprehensive service specifications (e.g., MatchBox [2] [13]) allow for matching required against provided services based on their interfaces and/or behavioural description. Although these approaches can be applied in an open system context, they produce only probabilistic results. If no perfect match is found, then these approaches only serve as an assistant for manually selecting relevant interfaces. Furthermore, they may outsource knowledge engineering for a domain to an ontology which, like industrial standards, are prone to be outdated and/or contain inconsistencies [5]. Consequently, their matching result cannot be processed automatically and cannot be used reliably for adapter synthesis. Matching approaches abstract away from concrete use cases and device interfaces and are therefore widely applicable in open systems. Nonetheless, they do not produce reliable results as they mainly work on concrete syntax and only approximate service meanings in a given context. Practitioners require a method that maximizes interface mapping formalization effort in open software systems. Otherwise, similar software adapters are implemented all over again.

In this paper, we apply a novel integration method that formalizes interface mappings incrementally in the context of a home automation system. We evaluate the applicability of the method onto this system class by performing an empirical experiment to demonstrate how integration knowledge is

 ISBN: 978-1-61208-781-8

formalized, stored and reused by 15 students. In Section 2, we define all the required terms. Next, in Section 3 the novel integration method is introduced within the smart home context and in Section 4 we outline the experiment and its results. Finally, in Section 5 we conclude our work.

II. Context and Definitions

In order to explain the concepts of the novel integration method within an open IoT software ecosystem, the following definitions are used:

Software Component: A software component is a software element that conforms to a component model that can be independently deployed and that can be composed without modification according to a composition model [14]. Software component's actions can be accessed by a well-defined interface. An action contains a function name, a list of parameters that are inputs, and an output.

Interface Description Language: An Interface Description Language defines a set of actions in a programming-language independent way. Depending on the expressiveness of the language, the syntactic level (e.g., data types and representation), the semantic level (e.g., range of parameters or pre- and postconditions), and the behavioural level (e.g., constraints on the ordering of interactions sequence) can be described.

Component Integration: An interface description can be either used to express actions a component requires from its environment or to express actions that are provided to the environment. Hence, software components are compatible if there exists a contract between their interfaces that maps all necessary interface description elements (i.e., they can be integrated). In distributed systems (e.g., based on web services), this is similar to the concept of interoperability - or integration in a more colloquial style.

Semantic Interoperability: Semantic Interoperability ensures that services and data exchanged between a provided and a required interface makes sense - that the requester and provider have a common understanding of the "meaning" of services and data [15]. Semantic interoperability in distributed systems is mainly achieved by establishing semantic correspondences (i.e., mappings) between vocabularies of different (data) sources [16].

Software Ecosystems: A Software Ecosystem is a socio-technical system (e.g., it contains organizations, people, digital systems, and partnerships). Independent participants collaborate together to generate mutual benefits.

Adaptive Software Ecosystems : Adaptability in Software Ecosystems can be 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) [17]. The higher the degree of adaptability, the more the human is involved.

Open- vs. Closed-world models : A closed-world model of a system directly represents the system under study. This means that there is a functional relation between language expressions and the modelled world. Even when modelling is used to create a conceptual model of a domain, the represented knowledge is implicitly viewed as being complete. As software ecosystems have to be aware of constraints for ensuring adaptability and controllability, an open-world assumption is being assumed for this system class [17].

Top-Down vs. Bottom-Up System Engineering Approaches: In the context of web services, service compositions can be either performed from top-down or bottom-up [18]. Bottom-up refers to whether pre-existing service interfaces are composed and thus adapters may be required due to mismatches. Top-down refers to given a pre-existing composition model, which suitable interfaces need to be discovered and possibly adapted to fit into the composition.

Semantic Integration happens in a software ecosystem within the digital software system. This task is mostly performed by a System Integrator. Now, based on the assumption that Software Ecosystems are based on open-world models (see Definition II), bottom-up integration approaches for software adapter creation are required. Hence, a system integrator has to take the inter-dependencies between the models of all involved systems (e.g., IoT components and the platform models) into account. Consequently, integrating software components from bottom-up may be a valuable engineering approach, in contrast, to designing a machine-readable domain-model from top-down. Nevertheless, a formal underpinning is still required as it facilitates meaningful integration and transformation among models, which is needed for automation through tools.

A. Example

Assume that there exist multiple software component interfaces that were developed by independent software vendors. A semantic integration effort between provided and required software component interface exists when domain models are used by heterogeneous parties. This is mainly due to the circumstance that 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 domain S is imagined (e.g., *Open door in the living room*) and the respective mappings from the concrete syntax to a semantic domain element is identified. At component integration time, a mapping can be identified by a system integrator to semantically map two syntaxes each from one distinct vocabulary by providing e.g., a function "map(close,close)" (see Figure 1).

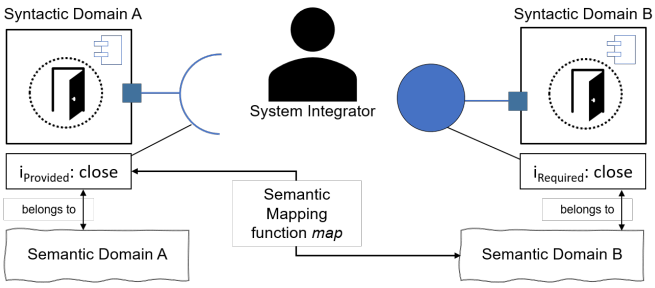

Figure 1. Semantic Interoperability Example for a Home Automation Platform

This mapping does not only take place on the syntactic but also on the semantic interface level. Depending on the use case (i.e., context), close can mean to *undo open* or *close in proximity*. This semantic integration knowledge is codified into the respective software adapter. Please note that this example is more related to being able to execute a corresponding automation process on various IoT devices (i.e., vertical integration). Another integration case is to connect home automation process elements in a horizontal way. For

 ISBN: 978-1-61208-781-8

instance, a wall switch defines a required interface to close a door.

III. Knowledge-driven Architecture Composition applied to House Automation Rules

In this section, we briefly introduce the novel integration method [19]. This method is one answer to the underlying research question *How can software components be semantically coupled in an automated way based on partially incomplete integration knowledge*? Then, we will mainly focus on how this method has been applied to integration tasks within a home automation platform.

A. Principles of Knowledge-driven Architecture Composition

As most IoT-related communication, the messages being sent within a smart home environment are predomenantliy data-driven (i.e., following the HATEOS principle) for REST (Representational State Transfer) services. Similar to most engineering approaches that aim at achieving semantic interoperability, this method also abstracts away from networking protocols (e.g., HTTP) and syntactic characteristics (e.g., JSON). Overall, the method aims at 1) storing mappings between two devices based on their interface definition using a declarative language and 2) to logically reason about these mappings. Finally, platform-specific software adapters can be automatically generated by reusing stored mappings or based on derived mappings. The method aims at finding all necessary mappings between a provided and required interface based on the following four principles:

- We do not require all component interface specifications and mappings to be present at system design-time but formalize them incrementally when a concrete use case is available
- If all mappings for one use case are present, then a software adapter can be generated in an automated way
- Manual implementation effort (e.g., software adapter) is explicitly allowed
- By storing and evaluating interface mappings for each use case, integration knowledge becomes reusable and decreases manual implementation effort

The set of all principles for semantic component integration can be subordinated under the term Knowledge-driven Architecture Composition (KDAC).

During the component design phase at time t=0, component provider, as well as requester can design their service interfaces without using a machine-readable interface description language that contains semantic data annotations based on a machine-understandable domain model (see Figure 2). Furthermore, a home automation process is created which uses software component B.

Assume at time t=1, an integration is necessary as a new device is present and the home automation process should not be changed. In addition to writing an individual software adapter to connect component model A (see Syntactic Domain A in Figure 1) to the component model of B (see Syntactic Domain B in Figure 1), the system integrator adds declarative mappings between both data vocabularies. This is illustrated in the rectangles above the KB in Figure 2 where a node represents a word from a vocabulary and an edge represents a mapping. These mappings are stored in a knowledge base KB (e.g., specifying *map* functions from the running example).

Over time, various other components are integrated as well (i.e., indicated by frames in Figure 3) and new mappings are added. For example, software component A* (i.e., another door from another manufacturer in Figure 1) uses the same domain model as software component A. Hence, the stored mappings for component A can be reused automatically. Now assume that there exists no common domain model between component A and A*. However, there exist mappings between components A and C (not shown in Figure 2) and from component C to A*. Due to the transitive relationship $A \leftrightarrow C \leftrightarrow A*$, the mappings from component $A \leftrightarrow A*$ can be calculated by logical reasoning.

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.

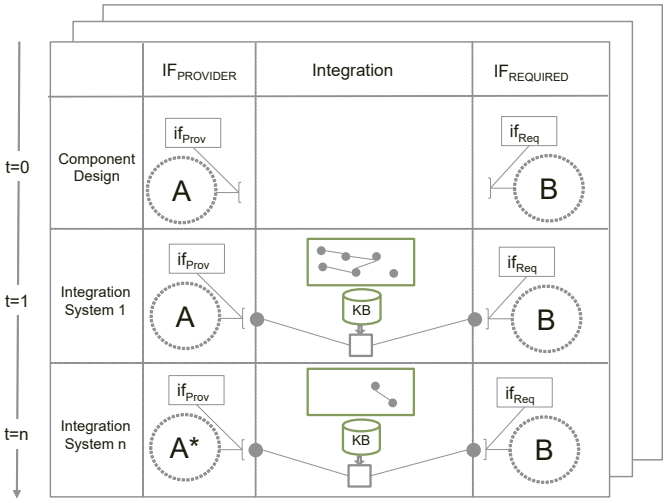

Figure 2. Knowledge-driven Architecture Composition Method

The novelty of this approach is the evolutionary process of formalizing integration knowledge without requiring a closed domain model from requirements. Rather, the focus lies on the incremental formalization of semantic mappings per use case. More details about the proposed method itself can be found in [19] [20].

B. Related Work

In general, there are two movements for solving syntax/semantic interoperability problems for independently developed components: 1) Interface Mapping Solutions [6] [7] as well as 2) Interface Matching Solutions [13].

Bennaceur et al. [6] [21] present MICS, a mapping approach that can infer interface mappings between operations and data of software components based on a formal component composition model. Their main motivation lies within the fact that other adapter synthesis approaches assume that all interface mappings are provided for all components at hand. Assuming that each application domain has its machine-understandable vocabulary, they use description logic properties (i.e., SHOIN(D) to identify implicit interface mappings (e.g., using subsumption). For modelling the behaviour of a software component interface, they used labelled transition

systems, which implement concepts of finite-state processes. In contrast to KDAC, this approach can deal with more complex interfaces (i.e., stateful services) but relies on the assumption that sub-models used in the service descriptions for each IoT component are fixed at component design-time.

When the semantic domain of an application domain is not formally specified, then matching solutions can be applied. For example, in the component-based software engineering community, configurable matching approaches such as MatchBox [13] are currently regarded as the state-of-the-art. Matchbox automates the re-configuration of individualized matching scenarios in service discovery scenarios. In this approach ontological signature matcher, condition matcher, privacy matcher and others can be combined automatically. The difference to KDAC lies within the automation property. Here, the matching approach always provides a probabilistic value for unseen automation rules. In contrast, mappings formalized within KDAC can be automated as a use-case with tested automation rule and device properties must be present (i.e., it can be applied in dependable systems such the presented home automation scenario). Other matching approaches for service-based components emerged in the web service community [10] [11]. For example, semantic web service descriptions such as SAWSDL and corresponding matching approaches such as SAWSDL-MX2 [10] can be applied similarly as code-based component matchers like MatchBox [13].

In practice, there is a tendency to solve interoperability problems by creating informal, machine-readable or machine-understandable standards. Such standards are created by multiple companies (e.g., OPC UA [22] or ZigBee [23]) or are dictated by a dominating market player (e.g., Apple Smart Home [24]). Some standardization such as IoT-Lite [25] also provide a machine-understandable domain models for device description. If one standardized vocabulary was used and interpreted by all device developers identically, then there would not be interoperability problems at the syntax/semantic interface. However, standards cannot cope with the increasing rate of change with software innovation cycles iterate faster each year in most application domains. If no single standard exists, then the system integrator must interpret the software component interface in the context of the automation rule. This currently means implementing similar imperative software adapters manually all over again (no domain model or multiple machine-readable domain models) or automated re-configuration is not feasible in a dependable way (no machine-understandable domain model). The latter problem is primarily tackled by KDAC.

C. Applying KDAC within Home Automation Platforms

To enable a system integrator to apply the KDAC method, the following infrastructure assumptions must hold for an IoT software ecosystem (see Figure 4)

- All software components must be already integrated at the technical and the syntactical level. This means, that the platforms must support all required communication protocols (e.g., MQTT or HTTP) as well as the syntactic payload definition (e.g., JSON or XML)
- The platform handles all calls/requests and the serialization/deserialization to/from the components
- A Formalization Editor must be able to retrieve all syntactically available device attributes provided by the connected software components. Furthermore, there must be a declarative language that can be used to formalize interface mappings
- The Formalization Editor must be connected to a knowledge base that is used to store, retrieve and evaluate available interface mappings

The main architectural components are the following:

Formalization Editor: For specifying interface mappings in a declarative style, we use JOLT [26]. JOLT is a JSON to JSON transformation library where the specification for the transformation itself is a JSON document. An example of a declarative JOLT specification can be seen in Figure 3. *Platform A*: We used openHAB [27] 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 that are offered by the IoT devices, a rule execution environment and a user interface to monitor the state of all devices.

Knowledge Base: The Knowledge Base stores formalized mappings in JOLT. Here, a graph-based structure is implemented where each node represents a data-channel and each edge represents a mapping specification. This allows 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 is connected to the platform by its interface. The required software adapter, that makes all data channels syntactically available within the platform, is provided by the platform.

User Interface for Formalization Editor: Automation Rules within the IoT context often follow the IFTTT (*If-This-Then-That*) structure. This structure also holds for openHAB. For example, the automation rule *Turn on Heating* contains a trigger, a condition and an action (*RuleBuilder* panel in Figure 3). Each rule part contains a drop-down menu where all available data channels provided by the connected devices are listed. A data-channel is called *Item* in openHAB and has a unique name, a description label and a type. For instance, the data-channel $Living\_Heating$ offered by a software component A has been used within the automation rule *Turn on heating*. Now, the rule should be executed in another environment where only the data-channel $Heating\_GF\_Living$ exists. Hence, the required interface $Living\_Heating$ by the automation rule must be integrated with the provided data-channel $Heating\_GF\_Living$. Now, the system integrator can map the data-channels $Heating\_GF\_Living$ provided by software component B to $Living\_Heating$. All currently provided data-channels are displayed in the *Integration-Items* perspective and all required items for one rule are displayed in the *Remote-Items* panel.

If no mapping for a data-channel exists, it must be created by the system integrator in addition to re-configuring 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 specification language and is the application of the *map* function as seen in the running example. This mapping is stored to the knowledge base.

If a mapping specification for a data-channel within the rule is available, then it is retrieved from the knowledge base,

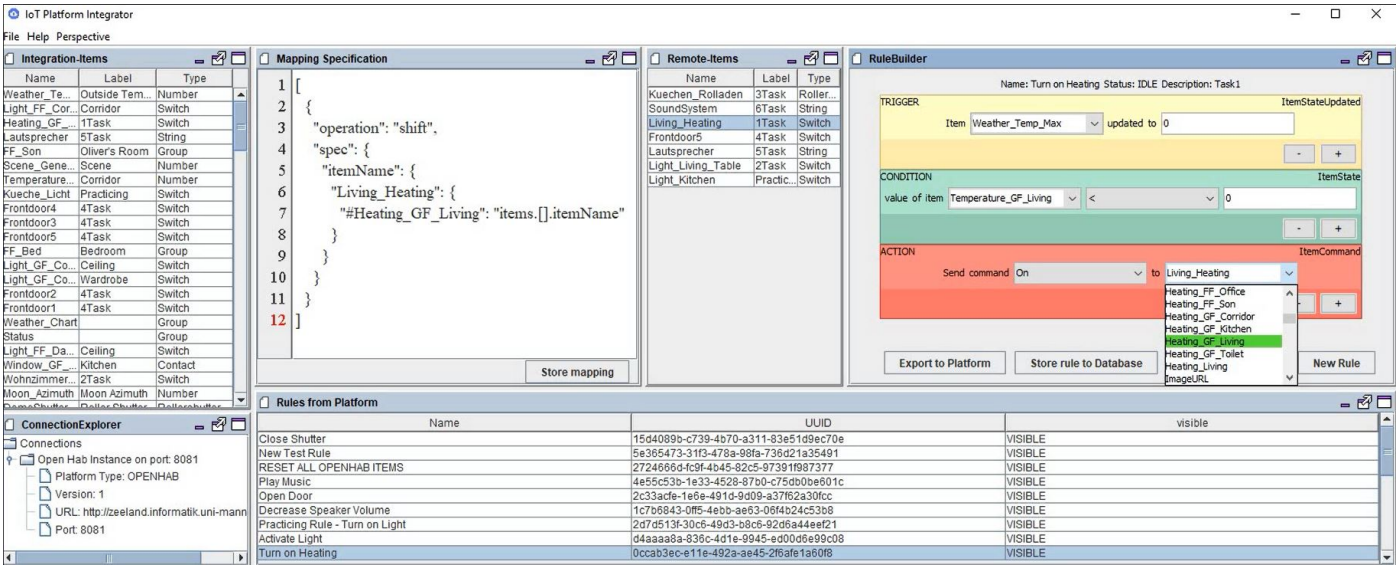

Figure 3. User Interface Formalization Editor

displayed (see *Mapping Specification* and selected *Remote-Items*). The availability of an interface mapping is indicated to the system integrator by a green background. If no specification or not all required specifications for an automation rule are found, then the system integrator has to create new mapping specifications and configure the system accordingly.

If all required mappings are available, the required configuration calls (cf. software adapter from the method) are generated and issued automatically to the connected platform. Finally, the system integrator can test the adapted rule by executing it and supervising the device status changes within the dashboard provided by openHAB (not displayed).

The main effect of applying the proposed method is helping to achieve automated adaptability in software ecosystems. Based on the provided definition II in the section Context and Definitions, this method focuses on *engineered adaptability* and *evolutionary adaptability*. Engineered Adaptability as the platform is now able to re-configure itself during run-time and execute context-sensitive automation rules. Evolutionary adaptability as new components can be integrated manually by the system integrator. Furthermore, human involvement is minimized over time as mapping specifications can be reused and/or generated by reasoning principles without having to rely on a predefined machine-understandable domain from requirements. As this method always requires a concrete use case, the formal specification effort gets controllable without losing the ability to automatically integrate new devices on-the-fly.

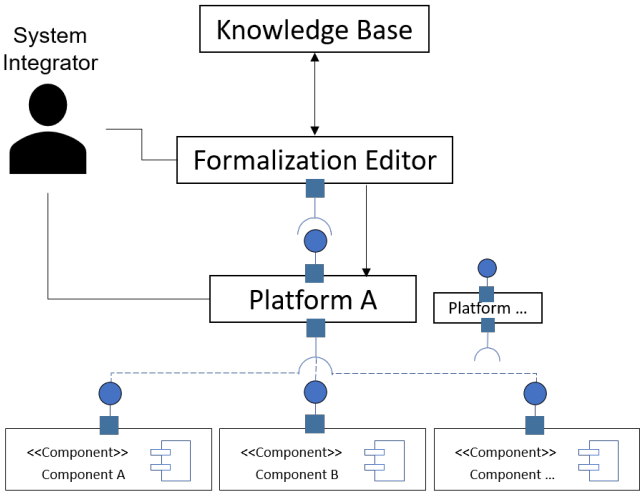

Figure 4. Logical System Architecture

IV. Evaluation

We designed a controlled experiment based on a within-subject evaluation design to validate our claim. Therefore, we used well-known design principles for empirical studies in software engineering [28] [29]. 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 re-configuration. Therefore, 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. Thus, in our study, the independent variable is determined by either using the conventional approach (i.e., re-configuring the platform each time a new IoT device enters the environment) or by using KDAC. The dependent variable is the required time for solving integration tasks [28].

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.

- 7 students are currently pursuing their master studies and 8 students their bachelor studies
- 4 out of 15 students own IoT devices and 2 out 15 have been already involved in IoT software development projects

 ISBN: 978-1-61208-781-8

- 4 out of 15 students have read about home automation platforms like openHAB [27] and 3 have already worked with IFTTT rules
- 5 out 15 students knew the IoT-related protocol MQTT [30]

Student Setup: Two groups of students were formed and competed against each other [31]. By assigning students to one group, it was made sure 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 manually re-configure the automation rule. The experimental group could reuse already specified mappings. However, if no mapping was found they had to re-configure the system (i.e., perform the task of the control group) and specify a mapping using JOLT in addition.

Evaluation Execution: The story line that was presented to the students is:
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 had to work 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) $Heating\_GF\_Living$

at the ground floor.
If the correct item is found, select (1) $Living\_Heating$ from the Remote Item Panel and create the mapping specification.

Hence, the automation rule was initially configured with the data-channel $Living\_Heating$, but the device that provided this data-channel was not available anymore. The other automation rules exposed a similar structure.
The participants were instructed that they had to follow the given order of data-channels replacements and then performed the following loop:

- *Configuration Time*: Configure next data-channel from the task and export it to the connected home automation platform by using the tool (see Figure 3).
  - The experimental group could also directly select a correct data-channel if a mapping has been retrieved from the knowledge-base (i.e., indicated by a green background). Hence, existing mappings are automatically evaluated and the necessary re-configuration calls to the platform are generated, but the students had to manually trigger their invocation. 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. Then, the respective device state icon was 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.
  - *Specification Time*: As soon as the correct item is found, then a mapping specification had to be created based on a template (experimental group only).

All created mapping specifications are stored in the knowledge base and are available to other students that have been assigned to the experimental group.

A. 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). Each task involved different amounts of item replacements and only task 1, 3 and 5 were equipped with mapping specifications for the experimental group. Overall, the following total times for all reuse rules have been measured (see Figure 5).

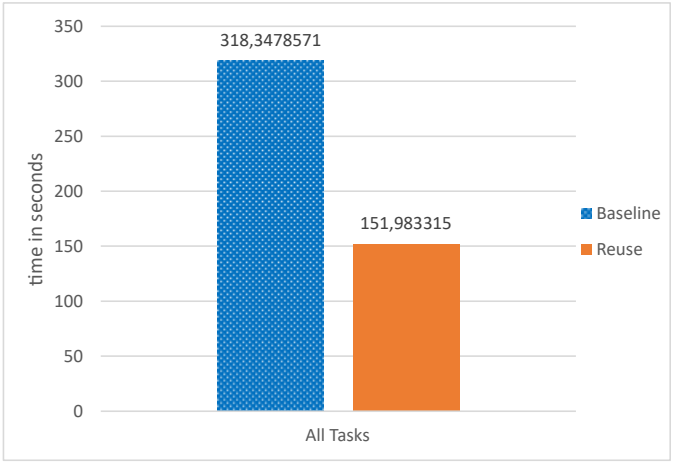

Figure 5. Reuse Tasks - Time Comparison for both groups

Therefore, we first calculate the total time in seconds for the rule using 3 for the control group and 4 for the experimental group. Here, *Y* returns the average time in seconds and *X* is an integer. Then, the sum of each rule result per group is calculated.

$$X = AmountOfReplacementOperationsPerRule \quad (1)$$

$$Y = \frac{TotalAmountOfReplacementOperations}{TotalTimeForAllRules} \quad (2)$$

$$TimeControlGroup = X * Y \quad (3)$$

$$TimeReuseGroup = \frac{(X-1) * Y + ReuseTime}{X} \quad (4)$$

Table I 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). *Reuse* (also *Configuration* and *Testing Time* translates to a task (i.e., one

of 1,3 or 5) where a mapping specification was found and evaluated so that one correct data-channels is available (experimental group only). Overall, *Specification Time* occurred only once to a 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.

TABLE I. INTEGRATION TIME IN SECONDS PER DATA-CHANNEL (AVERAGE : VARIANC)

	Baseline	Specification	Reuse
Rule 1	42 : 12	57 : 30	38 : 22
Rule 2	38 : 10	59 : 26	0 : 0
Rule 3	32 : 9	54 : 29	25 : 13
Rule 4	31 : 10	43 : 18	0 : 0
Rule 5	31 : 10	40 : 20	30 : 16
Rule 6	34 : 16	51 : 19	0 : 0

If a green data-channel (i.e., indicated by a green background as depicted in Figure 3) was present, measured *Reuse* times also refer to the sum of *Configuration Time* and *Testing Time* for the experimental group. However, the difference between the control group is 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 exist a mapping from data channel 1 to 4. Then, the control group must perform 3 replacement operations (i.e., 1-2, 2-3, 3-4) and the experimental group must perform 1 replacement operation (i.e., 1-4).

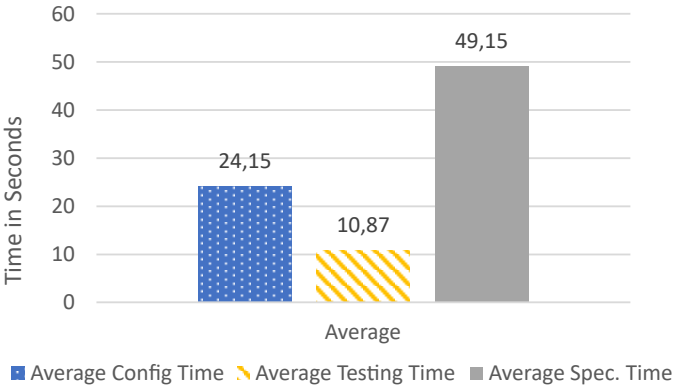

Figure 6. Average Participant Performance

Overall, 211 item replacement operations have been measured (i.e., Average Config Time) and have been tested within the home automation environment (i.e., Average Testing Time). For specification tasks, we measured 54 runs. On average, configuration time lasted 24 s, testing time lasted 11 s and specification time 49 s (see Figure 6).

Figure 5 suggests that the experimental group (i.e., reuse) is faster than the control group (i.e., baseline). Hence, our initial claim to outline the applicability of KDAC within an IoT software ecosystem is fulfilled. However, the point in time where specification effort pays off can only be estimated (e.g., based on the metrics in Table I).

B. Threats to Validity

There are several threats to internal and external validity. *Internal Validity*: Our evaluation targets the causal relationship between either re-configuring the home automation platform each time a new IoT device enters the system or by using KDAC (independent variable) and time (dependent variable). However, the presented results provide one 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 an early applicability of data-channels mappings. In large-scale engineering projects, it is unclear when and how often such mappings can be actually 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.

V. Conclusion and Future Work

Interoperability without modifying software component interfaces is needed in today's open and adaptive IoT software ecosystems. In this paper, we applied and evaluated a novel integration method called Knowledge-driven Architecture Composition that explicitly allows for manual integration effort without sacrificing reliability. First, we propose an adapted architecture for a home automation platform. Second, we provide a tool and six use-cases that allow for specifying interface mappings for data-driven IoT devices and automation rules from bottom-up. By using the tool, created mappings can be stored in a knowledge base and are made reusable for future integration scenarios. By doing so, we do not lose the ability to re-configure adaptive home automation platforms in an automated way. Formalization effort becomes controllable as a concrete use case must be present. There is no need to specify all possible interface mappings completely at design-time. Third and last, we evaluate the novel incremental, bottom-up integration method with 15 students. We could show that the data-channel mapping specifications created by the proposed method do speed up the overall reconfiguration of the home automation platform.

In the future, we plan to transfer our approach to other system classes such as HTTP-based micro-services in an enterprise setting. This will require more complex transformation functions and will enable a practical comparison to existing mapping approaches. Furthermore, we are currently working on incorporating reasoning services that can transitively infer extended mapping specifications by a sequential execution of mappings. By doing so, we expect to decrease the manual specification effort for the presented approach even further.

Acknowledgment

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).

References

[1] L. Baresi, E. Di Nitto, and C. Ghezzi, "Toward open-world software: Issues and challenges," Computer, vol. 39, no. 10, Oct. 2006, pp. 36–43.

[2] A. Broring et al., "The BIG IoT API - Semantically Enabling IoT Interoperability," IEEE Pervasive Computing, vol. 17, no. 4, Oct. 2018, pp. 41–51.

[3] F. Burzlaff, N. Wilken, C. Bartelt, and H. Stuckenschmidt, "Semantic Interoperability Methods for Smart Service Systems: A Survey," IEEE Transactions on Engineering Management, 2019, pp. 1–15.

[4] S. Bunzel, "AUTOSAR – the Standardized Software Architecture," Informatik-Spektrum, vol. 34, no. 1, Feb. 2011, pp. 79–83. [Online]. Available: http://link.springer.com/10.1007/s00287-010-0506-7 [retrieved: 2020.09.04]

[5] 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, 2012, pp. 1–21.

[6] 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 [retrieved: 2020.09.04]

[7] P. Inverardi and M. Tivoli, "Automatic Synthesis of Modular Connectors via Composition of Protocol Mediation Patterns," in Proceedings of the 2013 International Conference on Software Engineering, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 3–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486790 [retrieved: 2020.09.04]

[8] R. Spalazzese, "A Theory of Mediating Connectors to achieve Interoperability," Apr. 2011. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00660816 [retrieved: 2020.09.04]

[9] L. Cavallaro, E. Di Nitto, and M. Pradella, "An automatic approach to enable replacement of conversational services," in Service-Oriented Computing. Springer, 2009, pp. 159–174.

[10] 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.

[11] H. Dong, F. K. Hussain, and E. Chang, "Semantic Web Service matchmakers: state of the art and challenges," Concurrency and Computation: Practice and Experience, vol. 25, no. 7, 2013, pp. 961–988.

[12] M. C. Platenius, M. von Detten, S. Becker, W. Schäfer, and G. Engels, "A Survey of Fuzzy Service Matching Approaches in the Context of On-the-fly Computing," in Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering, ser. CBSE '13. New York, NY, USA: ACM, 2013, pp. 143–152. [Online]. Available: http://doi.acm.org/10.1145/2465449.2465454 [retrieved: 2020.09.04]

[13] M. C. Platenius, W. Schaefer, and S. Arifulina, "MatchBox: A Framework for Dynamic Configuration of Service Matching Processes," in Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering, ser. CBSE '15. New York, NY, USA: ACM, 2015, pp. 75–84, event-place: Montreal, QC, Canada. [Online]. Available: http://doi.acm.org/10.1145/2737166.2737174 [retrieved: 2020.09.04]

[14] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. Chaudron, "A classification framework for software component models," IEEE Transactions on Software Engineering, vol. 37, no. 5, 2010, pp. 593–615.

[15] S. Heiler, "Semantic interoperability," ACM Computing Surveys (CSUR), vol. 27, no. 2, 1995, pp. 271–273.

[16] N. F. Noy, A. Doan, and A. Y. Halevy, "Semantic integration," AI magazine, vol. 26, no. 1, 2005, pp. 7–7.

[17] 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.

[18] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, "Web services," in Web services. Springer, 2004, pp. 123–149.

[19] 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.

[20] F. Burzlaff, L. Adler, and C. Bartelt, "Towards automating service matching for manufacturing systems: Exemplifying knowledge-driven architecture composition," Procedia CIRP, vol. 72, 2018, pp. 707–713.

[21] A. Bennaceur and V. Issarny, "Automated Synthesis of Mediators to Support Component Interoperability," IEEE Transactions on Software Engineering, vol. 41, no. 3, Mar. 2015, pp. 221–240.

[22] 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, Jan. 2016, pp. 315–320. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2212827116312094 [retrieved: 2020.09.04]

[23] ZigBee Alliance, "ZigBee Specification," Sep. 2012. [Online]. Available: https://zigbee.org/zigbee-for-developers/network-specifications/ [retrieved: 2020.09.04]

[24] Apple Inc., "HomePod," 2019. [Online]. Available: https://www.apple.com/homepod/ [retrieved: 2020.09.04]

[25] "IoT-Lite Ontology." [Online]. Available: https://www.w3.org/Submission/2015/SUBM-iot-lite-20151126/ [retrieved: 2020.09.04]

[26] Jolt JSON to JSON transformation library written in Java, 2019. [Online]. Available: https://github.com/bazaarvoice/jolt [retrieved: 2020.09.04]

[27] openHAB Foundation e.V., "openHAB," 2019. [Online]. Available: https://www.openhab.org [retrieved: 2020.09.04]

[28] G. Leroy, Designing User Studies in Informatics. Springer Science & Business Media, Aug. 2011, google-Books-ID: IqR7M1h1yDQC.

[29] 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, Feb. 2015, pp. 110–141. [Online]. Available: https://doi.org/10.1007/s10664-013-9279-3 [retrieved: 2020.09.04]

[30] 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 [retrieved: 2020.09.04]

[31] D. Falessi et al., "Empirical software engineering experts on the use of students and professionals in experiments," Empirical Software Engineering, vol. 23, no. 1, Feb. 2018, pp. 452–489. [Online]. Available: https://doi.org/10.1007/s10664-017-9523-3 [retrieved: 2020.09.04]

 ISBN: 978-1-61208-781-8