MergePoint: A Graphical Web-App for merging HTTP-Endpoints and IoT-Platform Models

Fabian Burzlaff
University of Mannheim
burzlaff@es.uni-mannheim.de

Johannes Hammen
University of Mannheim
jhammen@mail.uni-mannheim.de

Benedikt Bongarth
University of Mannheim
bbongart@mail.uni-mannheim.de

Sven Grottker
University of Mannheim
sgrottke@mail.uni-mannheim.de

Christian Bartelt
University of Mannheim
bartelt@es.uni-mannheim.de

Abstract

More and more devices are connected to Internet of Things Platforms in various application domains. The resulting device integration effort is moderated by the concrete integration syntax and the technical abilities of the device integrator. Therefore, researchers from various communities have been investigating and designing component coupling architectures to achieve interoperability for more than 30 years. Emerging Smart Home scenarios challenges classical integration approaches as no single formal integration standard exists. In this paper we introduce a reference architecture called MergePoint that automates HTTP-Endpoint integration with smart home platforms such as openHAB in a plug-and-play manner. Based on a prototypical system implementation, our empirical evaluation demonstrates that average integration time can be reduced by 78% and average tool usability score is increased by 65% compared to textual integration approaches. MergePoint can serve as a reference implementation for practitioners that want to automate the integration between HTTP-Endpoints and IoT Platform Models.

1. Introduction

More and more devices are connected to Internet of Things (IoT) Platforms in various application domains [1]. The application cases range from Industry and Healthcare to Transportation and Smart Homes [2]. One key characteristic of IoT-Systems is their flexible architecture. In contrast to closed, classical systems (e.g. ERP Systems), these IoT-Systems can be efficiently extended during design and runtime. For example, use cases for smart home appliances such as "send warning message if stove or oven is not turned off" or "notification when washing cycle has completed" are currently trending [3].

Although smart appliances such as Amazons Alexa currently support more than 15'000 skills (=voice command), these skills must be manually developed by software engineers per device. Hence, interoperability across platforms and devices is still one of the main adoption barriers for consumers when avoiding vendor lock-in. To exploit the growing IoT-market (i.e. 153 billion US$ in 2023 [3]) the integration between IoT devices and platforms across company ecosystems, standards and markets will be key to success.

From a scientific viewpoint, integration challenges such as interface compatibility and (automated) software component coupling are in the scope of several research communities [4]–[6]. In general, relevant IoT architectures can be conceptually described as a four-layered architecture integrating a 1) sensing layer containing sensors and RFID tags 2) a networking layer providing basic transfer networks 3) a service layer to control system states according to user goals and 4) a platform interface layer accessed by the user and other systems. In most cases, such architectural styles are centralized platforms that connect distributed devices [7]. Furthermore, the most studied quality attributes of IoT architectures are scalability, security, interoperability and performance [7].

Engineering IoT systems involves several architectural decisions. For tackling the integration challenge for automated interface component coupling, there are 7 relevant design choices: 1) standardized vs. proprietary information models 2) raw vs. semantically annotated information 3) generic vs. domain-specific protocols 4) standardized vs. specific interfaces 5) common model vs. peer mapping for model interoperability 6) standardized vs. proprietary engineering data models and 7) desktop-based vs. web-based User Experience [8].

URI: https://hdl.handle.net/10125/64557
978-0-9981331-3-3

The contribution of this paper is the conceptualization and implementation of the web-based application called MergePoint. MergePoint automates the integration of self-describing HTTP interfaces including raw and proprietary information models between devices and platforms. By not relying on one data model from one manufacturer, the syntactical interface description is formalized using the openAPI specification standard and engineering knowledge is stored in a model. To achieve interoperability, a common intermediate mapping model is defined which can be used for smart home applications. Our proposed solution is evaluated within an empirical setting where we compare traditional, more expressive textual integration solutions with a graphical integration tool. On average, the system usability as suggested by Brooks [9], [10] increased from 29.25 to 85.25 (Scale 0-100) and integration time decreased from 27.51 to 5.9 minutes when using MergePoint.

The remainder of this paper is structured as follows: section 2 outlines the context of this work, section 3 introduces the reference architecture, section 4 describes MergePoint and section 5 presents our evaluation. Finally, section 6 concludes our work.

2. Background and Related Work

Context: From a software development point of view, the term "Internet of Things" can be defined as "co-engineered interacting networks of physical and computational components" [11]. Overall, six computational components can be identified. These are Identification, Sensing, Communication, Computation, Services and Semantics [2], [7]. Within smart home scenarios, a four-layered architecture can be regarded as a common model for structuring these components. Most major open-source smart home platforms such as openHAB, home-assistant or ioBroker pursue this architecture as these platforms are a centralized installation per household. Hence, the resulting layers are Sensing&Perception, Communication and Services, Application and UI&Interfaces (see Fig. 1).

Problem: Among others, one main problem for the widespread adoption of IoT-Platforms is massive scaling [12]. Within this problem space, interoperability between provided data offered by smart home devices and platform data models is one of the key challenges. Here a translation between IoT data models and Platform-Applications (e.g. automation rules) are necessary to fulfil desired user goals. As most smart home platforms already offer the underlying generic (e.g. HTTP, SOAP or TCP/IP) and domain-specific protocol adapters (e.g. MQTT, KNX or ZigBee), the integration challenge shifts from a syntactic-centric to a semantic-centric challenge. For instance, openHAB currently offers support for 1547 things, home-assistant provides 1395 thing adapters and ioBroker exhibits 276 interface adapters. Based on the assumption that future IoT-devices offer open interfaces (i.e. accessible for third parties), platforms must be equipped with a flexible logical data model. Consequently, different thing channels must be linked to platform model items (see red boxes in Fig. 1).

This mapping is trivial when all parties involved use the same standard. Each standard defines its own data model and must be supported by the platform (e.g. one home installation uses only products and platforms offered by Apples HomeKit).

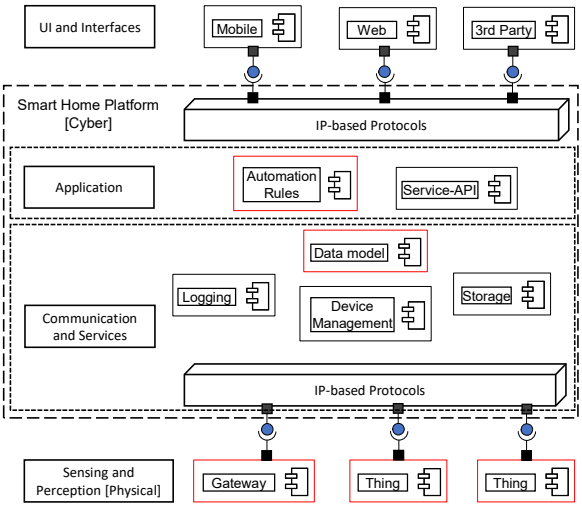


Fig. 1: Smart Home Platform Architecture

However, this may result in an undesired vendor lock-in. Hence, open-source smart home platforms propose their own data model where each IoT-device data model item must be integrated manually. This must be mainly done by the end-user. If end users formulate their own automation rules, the semantics of an IoT-device item is decided at integration time based on the application context. In contrast, standard-based integration solutions fix mappings already at component design time.

Example: Based on an adapted version of the well-known knowledge pyramid [13], the integration of a light bulb status is exemplified:

- Data: Value of "status" is *on, true active* or *1*
- Information: This means that the light bulb channel "status" acts as a switch and now the light bulb produces luminance
- Knowledge: Peter turns on the light when he´s playing the drums in his room
- Actionable Wisdom/Intelligence: Shut the windows in Peters room

Need: By assigning the data item value of "status" a type information offered by the smart home platform, the meaning is fixed per use case. If the item label is named differently (e.g. "condition"), the end user must reconfigure the assigned type. Such mappings must be currently defined manually by the end user and thus the user interface is subject to usability aspects. Here, the question arises how textual or graphical integration interfaces are perceived by the end user.

Solution Proposal: Facilitating this comparison, we propose MergePoint – a graphical user interface that allows for an easy mapping between IoT data models and platform models. Furthermore, the performance increases as integration knowledge in distributed smart home installations becomes reusable in an automated setting.

Related Work: Koziolek et al. just presented an OpenPnP (plug-and-play) reference architecture for the industrial IoT at ICSE 2019 Demonstrations Track. Krishna et al. presented a tool called IoT Composer within the same track that can compose and deploy IoT applications automatically. However, industrial IoT use cases differ significantly from smart home use cases [14]. Last, Platenius et al. published MatchBox, a configurable interface matching tool for component matching processes [15] and Bennaceur and Issarny implemented MICS (Mediator synthesis to Connect Components) for verifiable component mappings [16].

2.1. Scope Restrictions

The components of a formal mapping language can be categorized as syntax, semantics and pragmatic. Applying these categories on the adaptive DIKW pyramid pragmatics can be related to *Actionable Wisdom/Intelligence*. MergePoint and the accompanying engineering approach does not cover this aspect. Furthermore, contextual information (i.e. *Knowledge*) are necessary for determining the semantics of data but are also not in the scope of MergePoint. However, the context (e.g. automation rule) for determining the semantics during integration time can be described in forums, manuals or repositories. Consequently, only the semantical mappings (i.e. *Information*) between IoT device data and smart home platform model is formalized.

When formalizing the semantics of mappings between data models, a mapping language is required [8]. For example, Burzlaff and Bartelt [17] used a first-order logic language called OWL-DL. OWL-DL is a formal language with SHOIN expressivity [18]. Furthermore, there exist reasoners such as FaCT++ or Pellet that allow to infer new knowledge based on deductive reasoning (i.e. Conditional Statement && Antecedent → Consequence). Burzlaff and Bartelt conclude that the formalization effort when using OWL requires an advanced end-user skillset as declarative integration languages (e.g. SQL) behave differently compared to imperative languages (e.g. Java). We also believe that OWL exposes a poor end-user usability and exclude it from MergePoint. Consequently, this work is a subsequent work that focuses on the usability aspect of using formalized integration knowledge by decreasing language expressivity.

3. MergePoint Reference Architecture

From the viewpoint of a software developer, designing an interface for one IoT device is technically easy. For example, one can implement an OPC UA server by using a java-based reference implementation (e.g. Eclipse Milo). It is *easy* because the software developer picks a standardized communication protocol and designs the message content. The message content is mainly driven by his conceptual model.

When integrating multiple IoT devices from different vendors within a platform, data and service integration effort arise. For example, when integrating a light bulb providing a HTTP-Interface with predefined smart home platform types, the sub-route */status* does not syntactically match the platform type *Switch* although they refer (i.e. meaning) to the same semantic concept of a light-bulb state (see Fig. 2). The same circumstance applies for the sub-route */brightness* and the platform type *Dimmer*. Here, an existing automation rule may not be executable because these semantic relationships cannot be retrieved. The reason for this is that the engineering knowledge is not persisted during integration time. MergePoint closes this gap by attempting to answer the overall research question:

How can IoT device integration processes be automated by tools so that a high usability is achieved?

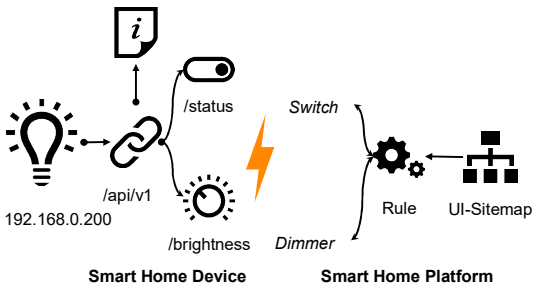


Fig. 2: Integration Example

Arguably, an intuitive approach for automating such integration challenges would be using domain-specific standards.

However, the creation of such standards requires time and thus cannot keep up with the fast-technical innovation cycles. This means, that IoT standards may be heavily adapted or that they do only provide a vocabulary. This vocabulary is then interpretable by a human, but not by a computer as conceptual links are not available. Hence, the usability of integration tools for storing these links are central to this work.

We are empirically investigating 1) if a textual or a graphical concrete syntax achieves a higher usability score and 2) how integration performance varies between both integration approaches.

In the upcoming section, we provide a static as well as a dynamic view of our proposed reference architecture. All technologies and standards used are open-source and can be implemented by using various libraries in a desired programming language. We will describe one concrete implementation of this reference architecture, MergePoint, in section 4.

3.1 Architectural Design Choices

Based on the proposed catalogue for implementing (industrial) IoT-platforms [8], we answer 7 (out of 13) relevant design choices for the main architectural layers in the physical and cyber-world (see Fig. 1):

[Layer: Sensing and Perception]
1. Standardized vs. proprietary information models: We mostly rely on unstandardized IoT device information models.
2. Raw vs. semantically annotated information models: As we are not relying on standards, semantic annotation tags that may point to an ontology are not used. Hence, we only raw information model data.
[Layer: Smart Home Platform]
3. Generic vs. domain-specific protocols: We strictly require generic protocols such as HTTP or OPC UA for communicating with edge devices. Hence, we do not support company-specific solutions. However, we are not restricting the communication paradigm (e.g. client/server or publish/subscribe)
4. Standardized vs. specific interfaces: We require REST-like interfaces for accessing the device APIs in a uniform way. For automating parsing of functions offered by APIs we furthermore require all devices to expose an openAPI specification file. Please note that openAPI only standardizes the hierarchy of path routes and not the meaning of data items.
5. Common model vs. peer mapping model: For achieving interoperability between device and platform information model, we rely on a peer mapping model. This is necessary as there exist no common information model that is supported by all available devices or every platform may have its own information model.
[Layer: UI and Interfaces of Engineering Tools]
6. Standardized vs. proprietary engineering data models: For formalizing mappings within our peer mapping model, we envision a domain-specific standard. To transfer integration information between platforms, we chose an open-accessible schema-free database.
7. Desktop-based vs. Web-based User Experience: Although desktop-based integration IDEs may offer more functionality and are more expressive [17], we chose a web-based integration frontend for mapping definition. The main reason for this lies within the overall usability goal of integration tools.

We are not supporting closed, domain-specific solutions such as Apple HomeKit, where interfaces are not accessible programmatically by third parties. Although this may affect overall system performance in a negative way, we gain technical interoperability when using generic communication protocols.

Furthermore, we are not concerned about device discovery and self-configuration mechanisms (e.g. network addresses or authentication credentials) as they may affect scalability aspects but are not directly related to usability aspects. Our evaluation targets are on-premise smart home platforms. Hence, aspects related to cloud communications (e.g. firewall friendly protocols) are not taken into consideration but are supported by most smart home platforms out-of-the-box.

Finally, we assume that the web-based integration tool is accessed by a device that can present all functions in a complete and concise way.

3.2 Static View

Fig. 3 shows an overview of all architectural elements from a logical point of view. The deployment nodes are NoSQL Database, Integration Service, Smart Home Device, Smart Home Platform and Integration Engineering Interface.

The **Integration Service** is the main backend component of our reference architecture. It includes functionality to parse the API specification which is based on the YAML syntax. The YAML syntax is an XML-like message format with well-known rules for making message parsing easy. Furthermore, the service contains the *Mapping Logic* component that is responsible for performing CRUD operations on the **NoSQL Database**. One can think of the result of the

mapping logic component as a recommender system that retrieves all relevant device-platform model mappings from the database and presents it to the end user. Mapping data is stored in a peer mapping model and must be transformed in a platform supported model when a HTTP binding is being generated. The **Integration Service** can support multiple platforms if the platform exposes a *Management API* where third-party systems can perform services like creating an automation rule or querying the platform type model (e.g. retrieve all devices of type *Switch*). Please note that the Device Discovery only can detect whether an IP address is answering a "ping"-command or not.

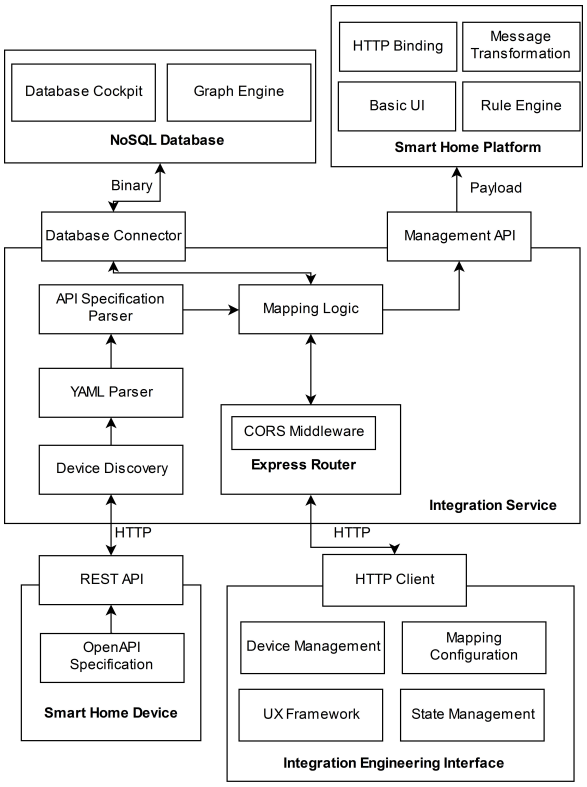


Fig. 3: Static View Reference Architecture

The frontend component of our reference architecture is the **Integration Engineering Interface**. This web-based user interface is accessed by the smart home platform owner and is exploited to perform the required integration tasks. These tasks will be presented in detail in section 3.3. Furthermore, the **Integration Engineering Interface** can render the *OpenAPI specification* of a **Smart Home Device**. Furthermore, all specific mappings between device and platform model can be transferred to the **Integration Service** via a *HTTP-Client*.

The last architectural component is the **Smart Home Platform**. It contains basic platform functionality as defined in Fig. 1. The **Integration Service** has some dependencies on the attached smart home platform. The Integration Service currently requires a *HTTP Binding* because mappings formalized in the frontend are only supporting devices that expose an openAPI specification. Although other bindings such as MQTT are technically feasible by most platforms, other endpoints than HTTP are not natively supported by the openAPI specification. Furthermore, it must be possible to parse Messages sent by the **Smart Home Device** (e.g. JSON messages). Lastly, a *Rule Engine* must be present that can access the *HTTP Binding*. The expressiveness of the rule formalization language must support update events (i.e. when a value is changed by the **Smart Home Device).** In most platforms a simple "If-This-Then-That" (IFTTT) language is available.

3.3 Dynamic View

The dynamic view of our reference architecture is influenced by the applied integration method. Here, we use the "Knowledge-driven Architecture Composition" (KDAC) approach [17], [19]). The KDAC approach can be described as an interface integration method where integration knowledge is formalized per use-case. This means, that a suitable knowledge base grows incrementally as only required component coupling knowledge is captured. This stands in stark contrast to classical integration methods from component-based software development [4] and automated web service composition [5]. Formal integration methods rely on heavy-weight formal standards such as OWL-S [20] and their domain-specific sub-ontologies (e.g. IoT-O [21]. However, such formal standards require expert knowledge and are hard to use for end users. Furthermore, the process to create such domain standards in practice is slow whereas the technological innovation cycles are expected to become faster and faster. Consequently, understanding and applying IoT standards may be more difficult in contrast to creating an adapter.

Applying the KDAC approach on the reference architecture results in the sequence diagram illustrated (see Fig. 4). Please note that there is only one swim-lane for the **Integration Service** and the **Database**. The reason for this is that only the Integration Service can access the Database and both components run on the same physical host. In general, the user has to 1) inspect all available device mappings 2) select and save mappings and 3) generate the respective platform-specific payloads for performing the integration. In 2) mappings can be either reused if they are present in the

database or mappings between smart home device and platform type model can be created or deleted. As a last note, the dynamic view shows that the **Integration Service** fetches platform types from its database (see Fig. 4).

This seems conceptually conflicting to supporting multiple platforms. However, this is only a technical circumstance as platform types can be queried by using the *Management API*. In addition, the schema-free NoSQL knowledge base can be easily manipulated in comparison to SQL- or OWL-based knowledge bases.

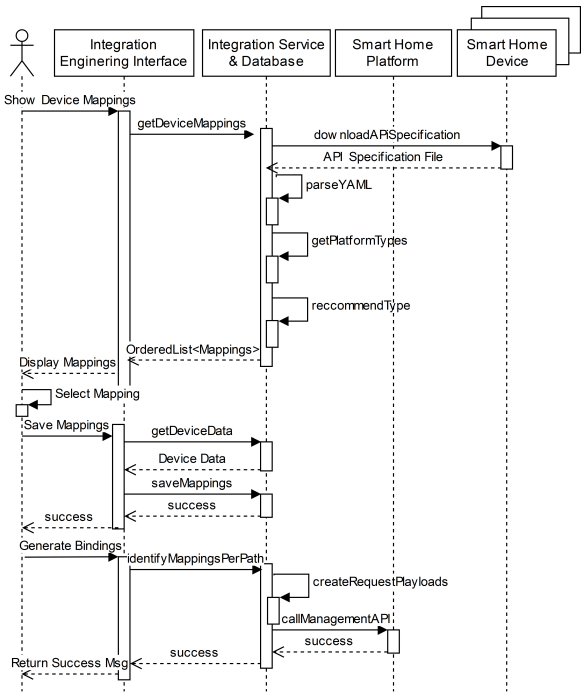


Fig. 4: Dynamic View Reference Architecture

In the context of the KDAC approach, one device-platform integration case performed by the end user is one formalization increment. Hence, one can apply the human-in-the-loop principle on the depicted dynamic view so that only required integration knowledge is formalized per use-case. Over time, the knowledge database contains more and more mappings. Consequently, the end user fades out of the loop and only approves recommendations made based on previous integrations. Reusable mappings in distributed integration settings over time require an easy to use integration user interface. This is the purpose of the upcoming implementation and evaluation chapters. Therefore, the amount of mappings that can be reused over a long period of time (i.e. full automation of thousands of devices) is not in the scope this work.

4. Implementation

By using the proposed reference architecture in practice, we focused on achieving the system characteristics for evaluating our research question. The beneficial system characteristics are end user usability and reduced integration time achieved by automating binding generation.

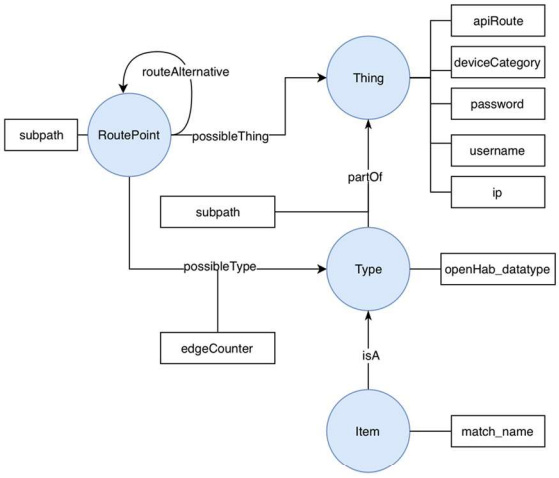


Fig. 5: Knowledge Base Schema

All components have been deployed on a dedicated *Linux Ubuntu Server 18 LTS*. As a **NoSQL database** we used *OrientDB*. For materializing the recommender feature, the data schema illustrated in Fig. 5 is used. For integrating **smart home devices**, two mock servers were implemented in Node.js and were equipped with *openAPI* specification files as an **API Specification language**. As a **smart home platform**, we selected *openHAB 2.4* as it offers most platform services over a HTTP API (i.e. *Management API*). OpenHAB relies on textual rules, items and sitemap files. Those files can be written by a JavaScript file writer to the respective directory as all logical components are deployed on the same operating system. Furthermore, openHAB offers a textual IFTTT rule engine and supports JSON Path Transformations for message handling send by the HTTP-based mock server. The **Integration Engineering Interface** UI communicates with the Integration Service using an *Axios HTTP Client.* The user interface is based on Vue.js which allows for the development of a single-page web application in combination with Vuex for state management across different UI components (see Fig. 6). The **Integration Service** backend application is based on the JavaScript runtime Node.js which was built on Google Chrome's V8 JavaScript engine. Node.js is built on an event-driven, non-blocking I/O model which ensures high throughput of requests and efficiency. Using the web-framework Express the backend provides REST

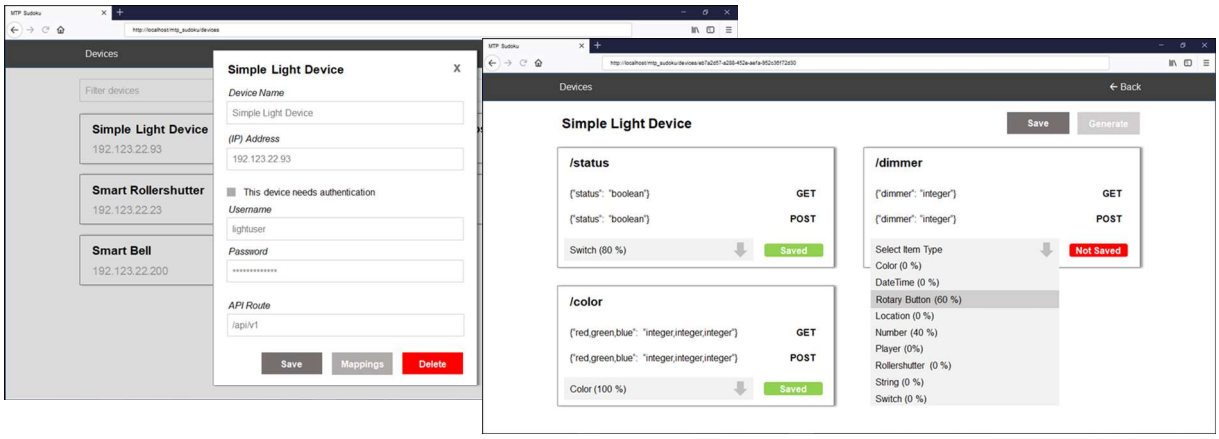


Fig. 6: UIs for Integration Engineering Interface

functions that can be called from the user interface. We only support one smart home platform.
However, this does not influence the evaluation results as most smart home platforms offer a user interface where end users can check whether the device integration was successful.

```
/***Item File for openHab***/
Switch EasyLight1_status "Get status of light (On / Off)"
{http="<[http://easyuser:Upd4t3d@localhost:
3000/api/v1/status:1000:JSONPATH($.status)]"}

/***Sitemap File for openHab***/
sitemap EasyLight1 label="ControlEasyLight1"
{Frame label="Controls" {Default item=EasyLight1_status}}

/***Rule File for openHab***/
rule "changeStatus_EasyLight1_status"
  when Item EasyLight1_status changed
  then
    var Boolean switchState
    if (EasyLight1.state == OFF) {
      switchState = false
    } else {
      switchState = true
 }
 var requestPayload = '{"status":' + switchState + '}'
 logInfo("Sending: ", requestPayload.toString)
 sendHttpPostRequest("http://easyuser:Upd4t3d@
 localhost:3000/api/v1/status", "application/json",
 requestPayload)
end
```

Fig. 7: openHABs' text-based Integration

5. Evaluation

For answering our core research question "How can IoT device integration processes be automated by tools so that a high tool usability is achieved?" we designed a user study which can be described as an action-case research [22]. Therefore, we adopted a within-subject evaluation design where the independent variable is the user interface and the dependent variables are the system usability score according to Brooks [9] and the integration time. Regarding the independent variable we compare the traditional text-based approach (see Fig. 7) with our graphical approach (see Fig. 6). The study participants were required to perform the following concrete integration task per input option:

Textual integration: 1) Map an offered smart home device sub-route to an openHAB type item 2) Create a rule that makes an HTTP POST request to the device as soon as the device state changes 3) Create a sitemap that displays all state changes as a simple UI component.

Graphical Integration: 1) Retrieve possible mappings for the device to be integrated 2) Show and adapted the mapping for the preferred device sub-route 3) Generate all necessary Bindings (i.e. files required by openHab depicted in Fig. 7).

All study participants performed both integration approaches for two sub-paths (i.e. status and brightness route of a light bulb). The order of treatment is randomly decided per candidate and all candidates where allowed to use openHABs debug console. All instructions and necessary parameters such as login credentials were given to participants before starting all measurements.

Furthermore, we facilitated Think-Aloud scenarios where one study *participant* is guided by one *facilitator* that may provide help on request. Every feedback and request is logged by a third person, the *log keeper* [23]. These roles were always fixed.

To generate a quantifiable hypothesis, we divided our core research question into two sub questions:

- RQ1: How do usability aspects differ for textual and graphical integration tools?
- RQ2: How many errors per minute are made during the respective integration sessions per minute?

For RQ1, we assume that the usability score is higher for the graphical integration interface. For RQ2, our hypothesis is that there are less errors when code generation is activated. Furthermore, we suppose that overall integration time decreases.

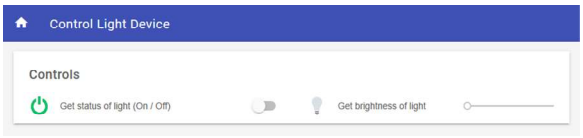
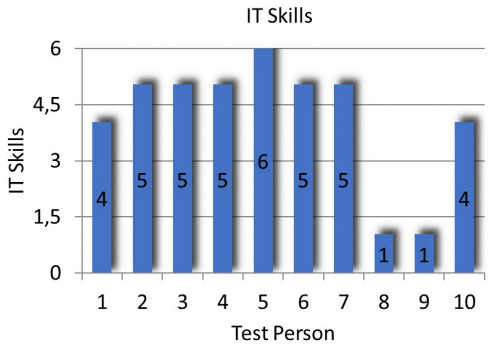


Fig. 8: OpenHAB Result Check

A device integration is successful if the device can be accessed by the openHAB UI and there is no error shown in the debug console when a state change is triggered (see Fig. 8). The evaluation operator also checked whether the mock IoT device did receive an update during each session.

5.1 Study participants characteristics

Based on an online self-assessment questionnaire, we collected the following study participants characteristics: Overall 10 persons participated in the evaluation. 6 persons are currently pursuing a PhD, 1 person pursues studies in a master and 3 persons in a bachelor program.



Graphic A: Overall IT Skillset of all participants

6 persons stated that they are majoring in business informatics and one person was from the fields of informatics, mechanical engineering, political sciences and business education. Overall, 6 persons had already heard of openHAB whereas 5 persons had theoretical knowledge, 2 persons had practical knowledge and 3 persons had no knowledge within the field of IoT. The overall IT Skillset by all participants achieved an average of 4.1 (see Graphic A).

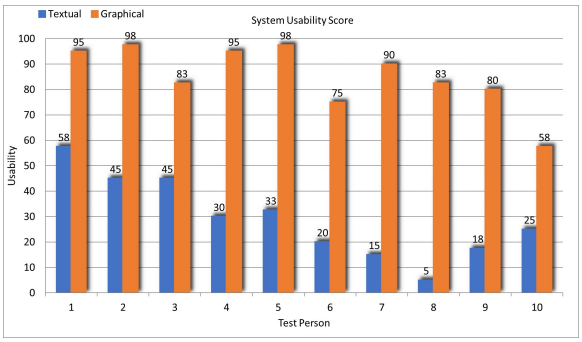
5.2 Usability

Usability was quantified by adapting the system usability score by Brooke [9]. After executing both integration approaches, the study participants where asked 8 questions concerning the usability of both tools. These are:

- I felt very safe using the system.
- I find the system unnecessarily complex (*)
- I find that there are too many inconsistencies in the system. (*)
- I find the system easy to use.
- I had to learn a lot of things before I could work with the system. (*)
- I think I would need more technical support to use the system. (*)
- I can imagine that most people learn to control the system quickly.
- I find that the various functions of the system are well integrated.

A question marked with an Asterix (*) are perceived as negative usability questions. We dismissed two questions as they were not relevant to our case.

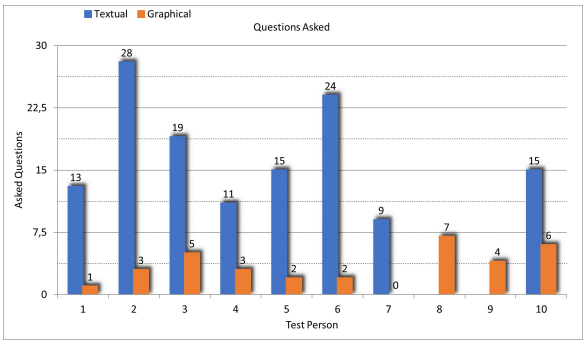
Each question could be answered based on a 6-point Likert scale ranging from 1 (strongly disagree) to 6 (strongly agree). This has been done to achieve a more fine-grained result. Please note, that the adaptations were also reflected in our score calculations. All values have been accumulated per persons and visualized in Graphic B.



Graphic B: Usability Score per Input Method

The average system utility score for the textual integration method is 29.25 and for the graphical integration method 85.25. Especially test persons with a non-technical background (person 8 and 9) performed well using the graphical tool. Brooke states that a system usability score over 68 can be considered as user friendly [9].

Regarding the Think-Aloud scenario [23], the amount of questions asked is visualized in Graphic C. Here the study participants where not actively asked to count questions themselves. This was done by the *log keeper* as a passive measurement. It can be noted that the amount of questions asked by the participants regarding the textual integration approach is higher compared to the graphical approach.

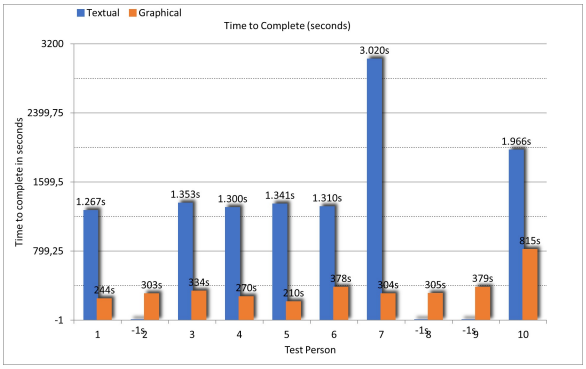


Graphic C: Question Asked per Input Method

Since, test person 8 and 9 (students of Business Education and Political Science) surrendered to process the test scenario with the textual approach immediately, no questions could be counted. Furthermore, most questions regarding the graphical approach have been asked by participants with a non-IT background (8-10 where test person 10 is a student from Mechanical Engineering). On average, 16.75 questions were asked for the textual and 3.3 questions for the graphical approach.

5.3 System Performance

For measuring overall system performance, we quantified the overall integration time per study participant as well as the amount of user errors. A user error is counted if and only if an error message was shown in the debug console or if the integration of the

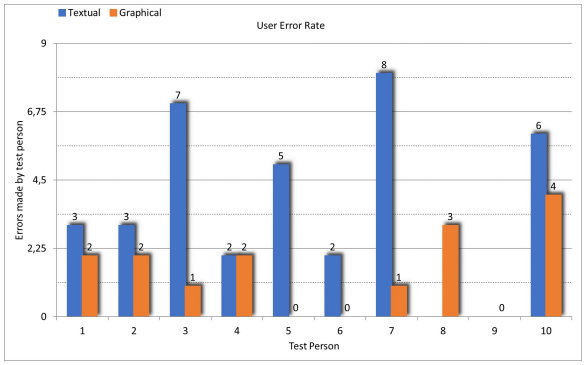


Graphic D: Integration Time per Input Method

device was not successful (i.e. not working via Basic UI). Please note that when integrating the second sub-path, some code fragments could be reused by copying and pasting using the textual approach.

On average, the integration of both sub-paths took 1651 seconds (27.51 minutes) for the textual approach and 354.2 seconds (5.9 minutes) for the graphical approach. If a test person was not able to solve the tasks, the time to complete is specified as 0 and is not included in the calculation of the average time. Overall, 3 persons did not finish the task.

On average, participants made 4.5 errors when using the textual approach and 1.5 errors using the graphical approach. Test persons 8 and 9 which surrendered using the textual approach were able to complete the graphical integration task (see Graphic E). To sum up, the error rate per minute is 0.164 for the textual and 0.254 for the graphical approach.



Graphic E: User Error Count per Input Method

For comparison, when a domain-expert performs both integration tasks, 337 seconds were spent using the textual and 48 seconds for the graphical tooling environment.

5.4 Result Interpretation

Based on our evaluation we can confirm the hypothesis for RQ1 as the overall usability increased. The hypothesis for RQ2 must be partially rejected as the error rate per minute is higher using the graphical approach although overall integration time was lower.

However, there are several threats to validity where some are listed in the following: Although our set-up can be considered as replicable due to using open-source technology exclusively, MergePoint only generates simple rules (see Fig. 7) and is not Turing-complete. Nevertheless, control and command use-cases can already be integrated (semi-) automatically.

Learning a new textual integration language is not an easy task. Therefore, we provided each study participant with a manual where relevant code snippets

for creating the openHAB files were illustrated. Although this introduces noise to the integration time, it reflects the reality how IoT software developer work. Furthermore, integration time is naturally lower when certain steps are automated. However, as the usability factor was our main object of investigation this influence can be tolerated.

6. Conclusion

Software development for IoT devices exposes new engineering challenges such as interface compatibility and semantic interoperability. In this work we presented MergePoint, a reference architecture for automating integration tasks by persisting integration knowledge incrementally. MergePoint can be beneficial for practitioners and researches. Practitioners can test and assess graphical, automated integration approaches for smart home platforms based on a reference architecture. Researchers can identify current conceptual pitfalls for future research in the area of software component coupling mechanisms.

In the future, we plan to support multiple IoT platforms and want to apply our tooling and integration approach in a distributed development setting. By doing so, we will be enabled to focus on engineering efficiency instead of tool usability.

Acknowledgement
This work was supported by the BMVI project xDataToGo (http://www.bmvi.de/goto?id=359354) under the support code 19F2048D.

7. References

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.
[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications," *IEEE Commun. Surv. Tutor.*, vol. 17, no. 4, pp. 2347–2376, Fourthquarter 2015.
[3] "Smart Home Report 2019," *Statista*. [Online]. Available: https://www.statista.com/study/42112/smart-home-report/. [Accessed: 07-Jun-2019].
[4] T. Vale, I. Crnkovic, E. S. de Almeida, P. A. da M. Silveira Neto, Y. C. Cavalcanti, and S. R. de L. Meira, "Twenty-eight years of component-based software engineering," *J. Syst. Softw.*, vol. 111, pp. 128–148, Jan. 2016.
[5] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, "Web services composition: A decade's overview," *Inf. Sci.*, vol. 280, pp. 218–238, Oct. 2014.
[6] A. Vakili and N. J. Navimipour, "Comprehensive and systematic review of the service composition mechanisms in the cloud environments," *J. Netw. Comput. Appl.*, vol. 81, pp. 24–36, Mar. 2017.
[7] H. Muccini and M. T. Moghaddam, "IoT Architectural Styles," in *Software Architecture*, 2018, pp. 68–85.
[8] S. Malakuti, T. Goldschmidt, and H. Koziolek, "A Catalogue of Architectural Decisions for Designing IIoT Systems," in *Software Architecture*, 2018, pp. 103–111.
[9] J. Brooke, "SUS: A Retrospective," *J Usability Stud.*, vol. 8, no. 2, pp. 29–40, Feb. 2013.
[10] P. W. Jordan, B. Thomas, I. L. McClelland, and B. Weerdmeester, *Usability Evaluation In Industry*. CRC Press, 1996.
[11] X. Larrucea, A. Combelles, J. Favaro, and K. Taneja, "Software Engineering for the Internet of Things," *IEEE Softw.*, vol. 34, no. 1, pp. 24–28, Jan. 2017.
[12] J. A. Stankovic, "Research Directions for the Internet of Things," *IEEE Internet Things J.*, vol. 1, no. 1, pp. 3–9, Feb. 2014.
[13] M. E. Jennex, "Big Data, the Internet of Things, and the Revised Knowledge Pyramid," *SIGMIS Database*, vol. 48, no. 4, pp. 69–79, Nov. 2017.
[14] F. Burzlaff and C. Bartelt, "I4.0-Device Integration: A Qualitative Analysis of Methods and Technologies Utilized by System Integrators: Implications for Enginering Future Industrial Internet of Things System," in *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2018, pp. 27–34.
[15] M. C. Platenius, W. Schäfer, 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*, New York, NY, USA, 2015, pp. 75–84.
[16] A. Bennaceur and V. Issarny, "Automated Synthesis of Mediators to Support Component Interoperability," *IEEE Trans. Softw. Eng.*, vol. 41, no. 3, pp. 221–240, Mar. 2015.
[17] F. Burzlaff and C. Bartelt, "A Conceptual Architecture for Enabling Future Self-Adaptive Service Systems," presented at the Proceedings of the 52nd Hawaii International Conference on System Sciences, 2019.
[18] "SHOIN." [Online]. Available: http://www.cs.man.ac.uk/~ezolin/dl/. [Accessed: 07-Jun-2019].
[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)*, 2017, pp. 108–111.
[20] "OWL Web Ontology Language for Services (OWL-S)." [Online]. Available: https://www.w3.org/Submission/2004/07/. [Accessed: 06-Jun-2019].
[21] N. Seydoux, K. Drira, N. Hernandez, and T. Monteil, "IoT-O, a Core-Domain IoT Ontology to Represent Connected Devices Networks," in *Knowledge Engineering and Knowledge Management*, 2016, pp. 561–576.
[22] G. Leroy, *Designing User Studies in Informatics*. Springer Science & Business Media, 2011.
[23] J. Nielsen, "Thinking aloud: The# 1 usability tool," *Nielsen Norman Group Online January*, vol. 16, 2012.