Dynamic Adaptive System Composition Driven By Emergence in an IoT Based Environment: Architecture and Challenges

Nils Wilken*[1] Mohamed Toufik Ailane†[2] Christian Bartelt†[3] Fabian Burzlaff*[4] Christoph Knieke†[5]
Sebastian Lawrenz†[6] Andreas Rausch†[7] Arthur Strasser†[8]
*Institute for Enterprise Systems (InES), University of Mannheim
Schloss, 68131 Mannheim, Germany
†Institute for Software and Systems Engineering (ISSE), Clausthal University of Technology
Arnold-Sommerfeld-Straße 1, 38678 Clausthal-Zellerfeld, Germany
Email: [1]wilken@es.uni-mannheim.de [2]mohamed.toufik.ailane@tu-clausthal.de
[3]christian.bartelt@tu-clausthal.de [4]burzlaff@es.uni-mannheim.de
[5]christoph.knieke@tu-clausthal.de [6]sebastian.lawrenz@tu-clausthal.de
[7]andreas.rausch@tu-clausthal.de [8]arthur.strasser@tu-clausthal.de

Abstract— **Applications provided by software intensive systems in an Internet of Things environment offer new business opportunities from the industry. An application describes the expected behavior of the software system. Thereby, the steps of a business process (e.g., event booking) are determined using objects from the Internet of Things environment at run-time. The system behavior is determined at run-time and described as a composition of software components based on service descriptions. These software systems can be developed as so-called dynamic adaptive systems. Therefore, developers define a structure based on software components of the system for an application from the system context at design-time. Then, the selection of software components instances by the system takes place at run-time. However, an Internet of Things environment changes its state constantly over life time and thus, the required structure of a software system can not be defined at design-time. Hence, applications of a dynamic adaptive system must be determined at run-time. In this paper, we introduce our vision of an emergent platform as an architecture for the development of a dynamic adaptive system. Such a system is capable of determining an application and compose a service based process that fulfills this application at run-time. Furthermore, we provide two Internet of Things scenarios and describe challenges on the basis of the scenarios which need to be tackled to enable the implementation of our architecture.**

Keywords–Internet of Things; Dynamic Adaptive System; Service Interoperability; Emergence.

I. INTRODUCTION

The environment of today's software systems (e.g embedded systems and information systems) can consist of complex and powerful objects connected. Each object is running software and providing applications for customers over the internet. This environment is well-defined as the *Internet Of Things (IoT)* [1]. For example, the software running on these devices can be provided as software components from the domain of social events, the domain of transportation or the domain of home automation. Such an environment offers new business opportunities to component providers and to providers of business applications from the industry. From the technological point of view, a software system is then needed to enable providers to offer applications and enable customers to make use of these applications: We refer to such a software system as a *Platform Ecosystem* [2].

In the field of business to customer applications, IoT objects from an environment are used to provide business related applications. All necessary objects for an application are represented in a structure as *user requirements*. This structure can be described as steps of a business process, which are expected by customers and thus must be fulfilled by an application provided by the system. In this context, we refer to *formal user requirements* as an appropriate machine readable representation of user requirements. Hence, an *application* can be defined as a structure that describes a set of component-based software interfaces, which must fulfill the formal user requirements.

Consequently, the application and all required IoT components can not be predefined at design-time because an IoT environment changes its state continuously over life time [3]. Hence, a software system providing applications must automatically and dynamically conduct a composition in response to a state change. We refer to this as an *emergent property* of a software system in an IoT environment. The resulting behavior of the system is not predefined at design-time and not anticipated by individual components. As a consequence, a composition containing executable software components and their bindings for interfaces of a software system is neither known at development time, nor at deployment time. Its composition is changing over time considering changes of its environment.

Hence, each time its IoT environment changes, the software system must be maintained. However, in each maintenance phase additional costly steps must be conducted to find a suitable application from available services. To avoid these problems, a dynamic adaptive software system must be developed, such that it is capable to determine an application from available service at run-time using the state of the IoT environment at run-time.

Thus, we introduce our vision of an software architecture for the development of a dynamic adaptive system, which can fulfill the emergent property required for its applicability in an IoT based environment. The goal of this paper is to investigate if our proposed software architecture can be used by software engineers to develop the appropriate system in practise.

The remainder of this paper is structured as follows: In Section II, the vision architecture of the Emergent Platform and its building blocks are introduced. The activity booking and home automation examples are then used to describe how our Emergent Platform can be applied in the IoT environment

 ISBN: 978-1-61208-781-8

in Section III. In Section IV an investigation of challenges follows on the feasibility of approaches from research to discuss how a possible development path of our Emergent Platform can be realized.

II. Our Vision of an Architecture for an Emergent Platform Design

Dynamic adaptive software systems in an IoT based environment can be designed from reusable software components [1], e.g., as proposed in the DAiSI component model [4], which describes the structure and the behavior of the system. Therefore, software components and interfaces are used to describe the building blocks of the architecture. The behavior is described on the basis of a contract based approach. The contracts are used by the system to check required and provided interfaces of software components for semantic compatibility at run-time.

As introduced in Section I, the software components used in our scenario are service software components. A *service software component* is similar to the definition of a software component in component-based software development [5][6]. Here, a software component is a deployable and executable software entity. It defines required and provided interfaces. A software component is executed by calling functions from its implemented provided interfaces, whereas a *service instance* is defined as a software component which has been already deployed for execution by en execution environment.

An *interface* defines a set of actions, which is understood by the provider and the user of an interface [7]. An *action* contains a function name, a list of parameters that are inputs and an output. Thereby, the interpretation of a parameter from a service description is called *semantic* and thus are used to test semantic compatibility of software components. Furthermore, an interface can exist independently of a component which makes it possible to specify and test the system behavior without its concrete implementation. Hence, the syntax of an interface can be described by a service description language (e.g., WSDL defined by W3C).

In order to check the compatibility of a provided and required interface, the term contract is used [7]. A *contract* can be hierarchically classified on the syntactic, semantic, behavioral and non-functional level. A contract is instantiated by defining all necessary mappings between required and provided interfaces.

As already explained in Section I, the process of a dynamic adaptive system must fulfill the expected system behavior. The latter is described as a sequence of business process related steps. A *process* is a composition of service descriptions from many service software components to describe the developed system behavior, which can be executed by an execution engine.

After introducing the basic terms, we now show our vision of a software architecture for the development of a dynamic adaptive system. Figure 1 illustrates the architecture of our emergent platform. The architecture consists of six major parts (see letters A-E in Figure 1) and will be briefly explained in the following:

Run-Time. At first, the system determines formal user requirements from interactions with the end user of an IoT environment in (A). Next, an appropriate application needs to be identified for identified user requirements. Therefore, a set of implementation independent component based interfaces is calculated. In principle this calculation checks if a subset of service descriptions, which are useful to fulfill the formal user requirements from (A), can be found. In a next step, the software system must determine available and executable software components for an application by evaluating registered software components from (D) of the software system and by compositing them to a process (B). (B) then provides a set of software components for execution to (C). The expected system behavior is then fulfilled by the system behavior as an ongoing interaction (E) between user and system invoking software components (C).

Design-Time. In (D), all those service descriptions and software components are developed and maintained by a service integrator which could not be identified by (C) at run-time.

In the following subsections, we explain how the interplay between the blocks (A) to (E) of a dynamic adaptive system in an IoT based environment is achieved in detail. The numbering of the subsections refers to the letters A-E as used in Figure 1.

A. User Goal and Requirements Handling

The user requirements handler (A) (see Figure 1) is responsible for automatic elicitation and formalization of emerging user requirements. Basically, user requirements of a software system express *what* a user wants to achieve and partially also *how* this should be achieved by the help of the software system [8]. Thus, more formally, user requirements specify parts of the process (i.e., how) that should be executed to achieve one or several user goals (i.e., what). Where we consider a user goal to be defined by a distinct state of the operating environment of a software system. A software system is then considered to fulfill the user requirements when the execution of its underlying process eventually leads to an environmental state that corresponds to a set of specified user goals.

Usually, such user requirements are elicitated and formalized manually by requirements engineers during requirement-analysis-time and design-time of software systems. However, as already mentioned in Section I, a key aspect of IoT based execution environments is that user requirements do emerge during the run-time of software systems. As a consequence, the software system has to handle the challenges of eliciation and formalization of emerging user requirements automatically at run-time. In our proposed architecture, the user requirements might be communicated in two different ways: As an *explicit request* or as an *implicit request*. The difference between explicit and implicit user requests is characterized by different interaction patterns between user and platform. An explicit user request is actively formulated (e.g., in Natural Language (NL)) and forwarded to the platform by a user. Further, the user requirements handler might actively interact with the user (e.g., via a chatbot) to clarify possible ambigiouties in the request. In contrast, an implicit request is not actively provided by a user, but rather triggered implicitly by conclusions drawn from inference over constant monitoring of a user. Besides a request, the user requirements handler (A) takes a user profile as a second type of artifact as input. Such a user profile can capture any kind of additional information about a user like for example information about a personal profile, preferences, or interests. Therefore, in the case of an explicit user request, the user requirements handler (A) takes an actively formulated request and a user profile that captures additional information about the user as input. Based on this input, it tries to

 ISBN: 978-1-61208-781-8

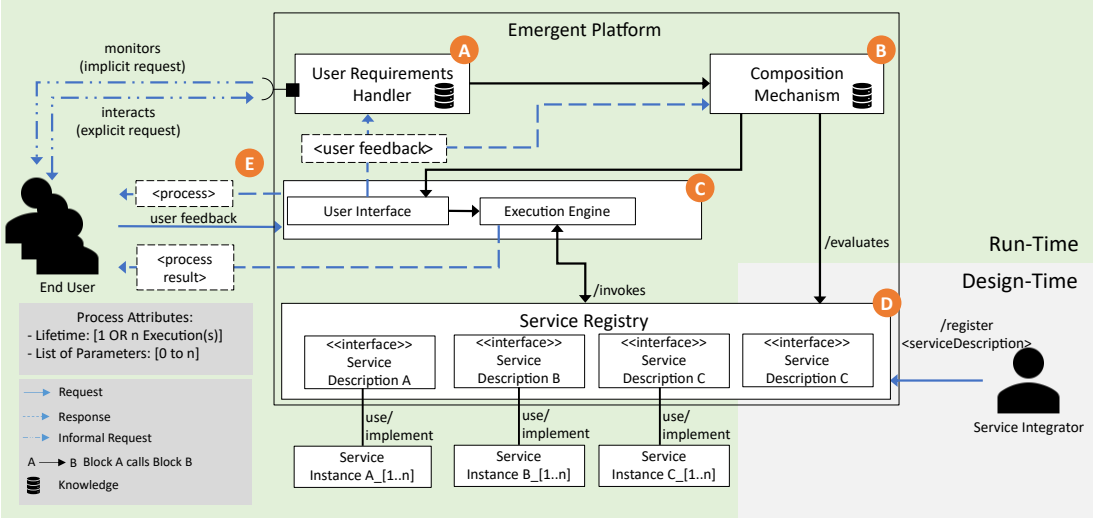

Figure 1. Our architecture to enable composition of software services based on emergence

automatically elicitate and formalize the formulated user goals and requirements. In this context, “formalized” means that the determined user requirements are encoded in a structured, machine-readable format (e.g., JSON). In contrast, in the case of an implicit user request, the user requirements handler (A) takes unstructured observational data of the end user and a user profile that captures additional information about the user as input. Then it tries to extract and formalize meaningful user requirements through continuous analysis of the data stream from user monitoring and creation of an implicit request when it is recognized that the user currently plans to achieve a supported user goal. In both cases, a formal domain model of the environment is needed that models important aspects of the operating environment of the software system (e.g., user goals, objects, possible user actions, etc.). As a next step, the service registry (D) component is checked to determine available and executable software components that are relevant for an application. Here, the main challenge is to match the identified user requirements from the domain model of the environment with service descriptions. As the composition mechanism should support an emergent property, the matching is performed by an algorithm at run-time. After determining an application for the identified user requirements, the user requirements handler forwards the service descriptions to the composition mechanism (B).

B. *Composition Mechanism*

The composition mechanism (B) is responsible on determining a process of a set of service instances that reflects the order by which services are supposed to be called, to fulfill an application which is provided by the user requirements handler (A). An evaluated service instance in (D) can represent one among many implementations for one service description. Consequently, different compositions can be identified by the mechanism to fulfill the same application. The decision of choosing the right composition is influenced by many factors. In our architecture we consider the user feedback (see Section II-E) and quality attributes (e.g., economic costs).
In this regard, we aim to enable new composition patterns that can emerge dynamically at run-time as illustrated in Figure 2. A pattern describes the structure of a process considering the factors mentioned above. The most optimal pattern is the pattern that achieves a high score meeting the factors after evaluation. As an example, Figure 2-B shows a chain pattern consisting of different providers and consumers as service instances. Yet, a different pattern (e.g., a tree pattern as shown in Figure 2-B) might be more optimal. For this, an evaluation mechanism that reasons over the factors is needed (as also mentioned in [9]). For these patterns to emerge, we review the description languages and discovery mechanism that can contribute not only to a pre-designed dynamic solution, but also enable these patterns to emerge and offer the possibility for the system to adopt one pattern. Hence, in order to define a robust composition mechanism, a unified formal description for both: (i) the service description of the interfaces (achieved by (B) in Figure 1), and (ii): the available software and component services (which will be invoked by (D) in Figure 1) are necessary.

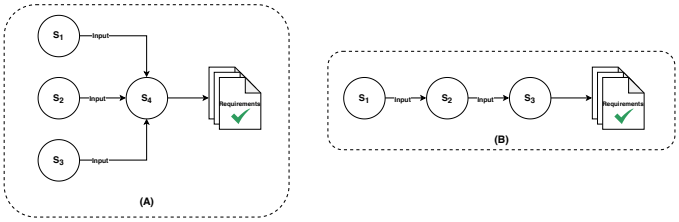

Figure 2. Service Composition Patterns: (A): Tree pattern, (B): Chain pattern

Fulfilling an application and ensuring a positive feedback of the end user can be seen as a problem of navigation in a three dimensional space as shown in Figure 3. The first axis consists of the *availability of services*, that are the available service instances or descriptions at a given point in time: The more services are available, the more flexible is the composition mechanism towards different end user feedback. The second axis consists of the *environments*, each environment can be related to different applications that require a variety of available services, which has to be handled by the composition mechanism. This cross-environments composition is a key element in the emergent composition of services. Thirdly, the *application axis* consists of a set of applications that are present in one or

more environments, they may share the same service instances or reuse the same process. Hence, an emerging composition pattern is the result of a en efficient navigation through this space and thus, can not be provided at design-time, but rather emerges from the continuous interaction of the system with its environment (e.g., end user and IoT environment) at run-time. To evaluate a pattern composed from service instances for given subset of service descriptions, the results from three different evaluation mechanisms must be considered: The service registry block must evaluate the interface compatibility of the service instances. As shown in Figure 3, new applications can be identified after the execution of an emergent composition. Thus, the service descriptions used by service instances of the new application must be evaluated. If the interfaces are compatible (see in Subsection II-D) and the new application from the emergent composition can be semantically matched to formal user requirements, then a pattern can be considered for evaluation in the three dimensional space.

C. Execution Engine and User Interface

The execution engine (D) is responsible for executing the process generated by the composition mechanism (B). The user interface (E) is a front-end component that is accessed by the end user to execute the generated process. As user interface design and process execution are not part of our main research focus, we rely on state-of-the-art technologies. In general, the following steps are performed:
An ordered set of service descriptions (i.e., a process) which has been identified by the composition mechanism (B) is displayed to the user. All services contained within the set of services descriptions are available or compatible to available service instances.
Next, the user interface collects feedback from the user and forwards this feedback to the user requirements handler (A) and the composition mechanism (B). Components (A) and (B) use the feedback to improve their performance for future requests and compositions. Afterwards, when positive user feedback is received, the execution engine invokes the respective service instances by using the needed interaction style.
Finally, values and messages as returned by the provided service instances are passed from the service registry (D) to the execution engine (C).

D. Service Integration in the Service Registry

The service registry (D) answers the questions which service instances and descriptions are available and which service instances fulfill a request. Hence, the service registry stores a set of service descriptions and manages the location as well as the interaction style with the service instances. Here, the execution engine (C) invokes available services during process execution. Now, the question becomes how service descriptions and service instances are managed within the platform and by whom.

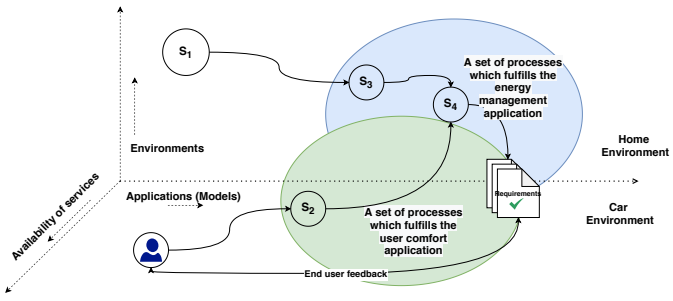

Figure 3. Composition space as a Cartesian plane,

Service descriptions may or may not conform to standards, common vocabularies or namespaces. Hence, the semantic level is unrestricted. However, the composition mechanisms is based on a service description language which specifies the syntactical level so that the composition mechanism (B) is able to process available service descriptions. A service integrator is responsible to describe service instances with a service description language. Service descriptions contain domain-specific parameters that are mapped to other service descriptions. The service integrator has a technical role within the platform team (e.g., no third parties). One service description can be fulfilled by multiple service instances (see "[1..n]" in Figure 1). The following steps are performed during system design-time (i.e., every time the system is maintained):
The input to the service registry are service descriptions that should be registered.
Next, the service integrator must decide whether to create a new service description or provide a compatibility contract from a provided (e.g., new service instance) to a required Service Description (e.g., already available in the platform). These mappings are called vertical mappings. Furthermore, service descriptions can be mapped to existing service descriptions for establishing a relationship between outputs of a service that may serve as an input to another service. These mappings are called horizontal mappings.
Finally, the service registry contains all service descriptions and associated service instances which potentially can be accessed by the platform. Furthermore, all possible mappings found by the service integrator to translate similar service descriptions into each other are specified. This can be done based on services or on a process produced by the composition mechanism (B).

E. Process Result and Feedback

A given process is only accepted based on the feedback of the end user. In addition to the evaluation of processes, the end user feedback is necessary to determine if the process can be accepted for execution by the execution engine. For example, the user can not afford the quoted price for booking an event and consequently decline a step of a process. Hence, a user feedback mechanism has to be defined and deployed and a validation process is required. Hence, a service instance, service description or the whole composed process can be declined by the end user. Based on the end user feedback, new composition patterns are pushed to emerge by the composition mechanism (see Section II-B). Once, a composition pattern is accepted, the execution engine can then execute the chosen process. In addition, the user requirements handler can use the feedback to evaluate whether the recognized requirements matched the real requirements of the user.

III. Application Scenario

In order to get a better understanding of emergence, this section will present two use cases and discuss their emergent behavior. The use cases are related to different application environments.

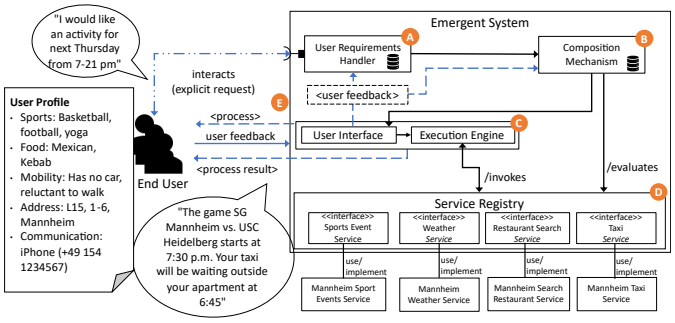

Figure 4. This scenario illustrates how the taxi service is provided in addition to the the booking service.

A. Automated Event Booking

The automated event booking can support the end user to manage his spare time. Therefore, an end user provides a profile to the software system: For example, favorite sports, food and public transport (e.g., because he does not own a car). Figure 4 illustrates the interaction between the end user and our proposed emergent platform.

The user requirements handler (A) starts to identify possible formal user requirements from the profile and from the message explicitly provided by the end user: These can be basketball, football and yoga as sport activities. Moreover, it identifies the time schedule and the location of the end users spare time. As a next step, applications in form of a set of service descriptions from the service registry (D) are determined which are semantically equivalent to the identified formal user requirements.

In this scenario, a set of possible applications can be identified: One can consist of the service descriptions Sports Event Service and Taxi Service. Another application can consist of Restaurant Search Service in addition to, the same service descriptions as in the first application.

The Sports Event Service can provide different sport events in a specific area, such as football and basketball games. The Weather Service provides the current temperature and the weather forecast. The Restaurant Search Service provides different restaurants, including the type of kitchen (Spanish fast food, etc.) in a specific area. And the Taxi Services enables booking a taxi from taxi companies such as Uber.

The set of applications is then provided to the composition mechanism (B). Based on the available service instances, the composition mechanism evaluates possible composition patterns in order to determine a process. Since the end users residence is Mannheim, the mechanism determines a set of services instances regarding their availability: Figure 4 constitutes the service instances used in the service registry (D), which can be used by (B) to compose a process. The process is then provided to the front-end software component (C) and is executed by the execution engine (C). As a result, the software system provides process steps to the end user which enable the user to book a football event and a taxi for transportation to the event-location (E). If the end user accepts this event, the platform will book the tickets and the taxi for him, if not it will try to evaluate another activity as an emerged composition: For example, a dinner in a Mexican restaurant. The emergence is represented by the behavior of the software system driven by the feedback from the end user and the IoT environment.

B. Home Automation

The smart home automation is responsible for managing energy flows of a house. Its goal is to manage the temperature and controlling the alarm system as economic efficient as possible for the end user. Unlike the first use case the user request is implicit, thus the smart home automation is observing the end user. The alarm should be turned off and the room temperature adjusted to a comfortable level, when the end user arrives at home: The user requirements handler (A) observes the actions and preferences of the end user and, based on the available services (see Figure 5), evaluates the best subset of service instance for composition to meet the user preferences and goals (B). The execution engine starts to execute the process provided by (B), after the end user acknowledge the process with the user interface (C) by providing his feedback (E). The process is executed as follows: It uses the actual GPS position from the user. When the end user getting closer to the house, then the home automation starts the climate control and turns off the alarm. Furthermore, the automation manages the energy flow in the house. For example, if the weather conditions are good, then the climate control can be powered by energy from the solar system of the home. This can be achieved by a balanced energy consumption strategy in cooperation urban energy suppliers.

IV. Challenges

A. Regarding User Goal and Requirements Handling

Specifying accurate and correct user requirements is crucial to ensure that a software system, which is developed from these user requirements, will be useful to the end users. As already highlighted in Section II, currently these tasks are handled manually by requirements engineers that specify the user requirements at design-time of a software system in cooperation with the end users. Over the past decades several different frameworks for requirements engineering were proposed and researched [8]. One popular framework is the KAOS framework [10], which is a multi-paradigm goal oriented modelling framework. Another example is the agent-oriented modelling framework i* [11], which is built around concepts such as goals, abilities, beliefs and commitments, and is the base of the Tropos methodology [12] which allows for validation by model-checking. However, all these frameworks include much manual work of requirements engineers during design-time of a software system. This contradicts to an important envisioned ability of the proposed platform, which is the automatic eliciation and formalization of user requirements

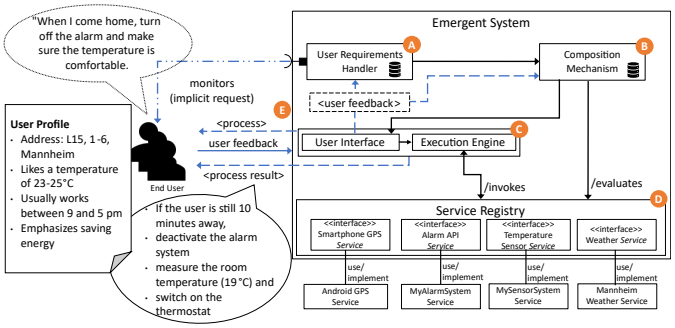

Figure 5. This scenario illustrates how the different services are provided in addition to the the alarm service.

that emerge during run-time of a software system. Hence, the following two challenges are relevant in the context of the user requirements handler (A):

How can user requirements be elicitated automatically from unstructured data? Along the lines of the distinction between explicit- and implicit user requirements, this challenge can be further subdivided into two sub-challenges. Both cases represent significant challenges that require different methods to be solved. In the case of an explicit request (like the example described in Section III-A) untrained end users are usually not able to express their requirements in a structured, machine-readable format, but rather describe their goals in NL. This is a major challenge, as descriptions in NL are commonly incomplete and ambiguous [13]. The same holds for an implicit request, where user goals are not even explicitly described, but are rather hidden in unstructured observational data that is generated through continuous monitoring of the end users. In the context of the example scenario described in Section III-B, an implicit request could be created when the system recognizes that the end user currently has the goal to go home from the past observed changes of the environment (e.g., change of user location). Hence, to achieve automatic elicitation of user requirements, the proposed platform has to be able to extract the relevant user goals and requirements from an unstructured input data format (e.g., NL or observational data), in order to use them for the composition of a useful software service.
Over the past decade first efforts towards automatic extraction of user requirements from explicit user interaction appeared. For example, van Rooijen et al. [14] try to automatically generate user requirements from sets of input examples that are created by the end users in the form of sequence diagrams. As an extension to this approach, van Rooijen et al. [15] try to learn such sequence diagrams directly from NL descriptions. Recently another approach uses a chatbot to refine user requirements that were extracted from NL input in a feedback communication cycle with the end users [16]. Some of these approaches might also be applicable in the context of the platform architecture proposed in this paper. In contrast, there has not been done much work yet towards the direction of automatically recognizing user requirements from unstructured observational data in the field of software systems engineering. However, the topic of automatically recognizing user goals and plans from observational data is a long standing research area in the Artificial Intelligence (AI) community. First works in this area appeared in the mid 80s and early 90s [17][18]. Several prominent recent approaches formulate the goal recognition problem as a planning problem which can be solved by the use of classical AI planners [19][20]. Another approach that adopts machine learning techniques, was proposed by Zeng et al. [21]. They use inverse reinforcement learning to model human behaviour to recognize goals in a dynamic local network interdiction environment. We envision that some of these approaches might be useful for automatic requirement elicitation in the context of implicit user interaction.

How can user requirements be formalized? Once the relevant user goals and requirements were extracted, they have to be transformed to a formal, machine-readable format so that the composition mechanism (B) can handle them. In literature there exist several approaches to formalize user requirements like for example state machines, activity diagrams, or sequence-diagrams [14][22]. Another possibility would be to use a Domain Specific Language (DSL) to formally encode the elicitated user requirements. Besides these formats that focus on behavioral aspects, there are also some approaches that focus on the structural aspects of software systems, like class diagrams or entity relationship models [22]. However, as the structure of the composed software component process is considered to be an emergent property, which emerges during composition, of the proposed platform, these kinds of models might be to restrictive regarding the structure of composed services. Hence, it is not clear which kind of formalization format is suited best for the application in the context of the proposed emergent platform architecture.
In addition, also the transformation which is required to translate the elicitated requirements to such a format is challenging, because it requires a mechanism that is able to interpret the semantic meaning of them. To achieve this some kind of domain model (e.g., a domain ontology) that encodes relevant semantic information about the environment is required. Overall, the formalization format and transformation mechanism that are used have to provide a level of semantic, behavioral, and structural information that on the one hand is sufficient to compose meaningful software component processes from it, but on the other hand does not restrict the degrees of freedom of the composition mechanism (B) in a way that its emergent behaviour is harmed or prohibited.

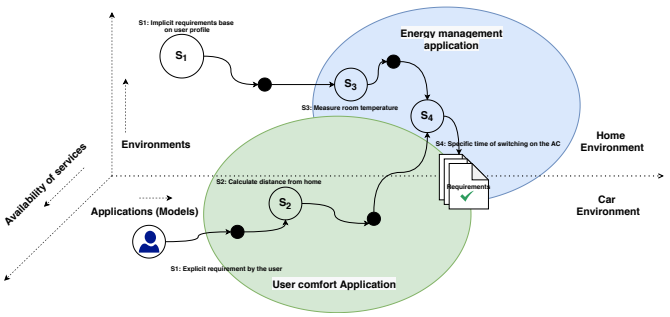

Figure 6. Composition space navigated using Markov Decision Process (MDP)

B. Regarding Composition Mechanism

As introduced in Section III, several application types can be identified in the home automation use case. Two examples are, energy management and user comfort applications (see Figure 6). The composition is governed by certain quality attributes of available service instances to fulfill the two possible concrete applications described in Section III and navigate through the three dimensional space (as introduced in Subsection II-B). Similar research has been made in this regard such as multi-agent solutions [23] and genetic algorithms [24]. The two challenges are to construct a suitable structure for the three dimensional problem space and and to evaluate emerged composition patterns. In order to enable a composition pattern to emerge, two main questions need to be answered:

How to compose? One fundamental question that rises when speaking of service composition is how the composition component is able to compose the relevant service instances.

As the composition mechanism is responsible to compose a process that fulfills an application, discovering the instances in the service registry (D) is a main challenge. In our suggested

solution, which is provided later in this section, we consider all possible compositions and evaluate each composition if it fulfills a given application or not. This leads us to the second question:

How to evaluate a given composition? After composing a set of service instances, the validity and optimality of the obtained process has to be evaluated. We also consider this in our suggested solution later in this section by speaking of an optimal policy (the formal term for the evaluation function).
To answer the questions, we suggest to formulate the composition problem as a markovian problem [25]. A Markov Decision Process (MDP) is a mathematical tool that is used to solve decision problems, it consists of a 4-tuple *M = (S, A, P, R)*, such that:

- *S*: Is a set of the system states (State space)
- *A*: Is a set of actions that the system can act on based on the current state (Action space).
- *P*: Is the probability function that indicates the probability of transitioning to state s' when taking action a at state s (Pr($s'|s, a$).
- *R*: The expected immediate reward for taking action a at state s and transiting to state s' (r = R($s'|s, a$).

Similar to [9], a dynamic composition is realized using a reinforcement learning method, the main challenge of such an approach is the definition of the sets of states and actions. Furthermore, engineering a reward function for the expected reward based on an emergent composition is a challenging task. Once, the challenge of designing the problem as a MDP, the solution challenge is how to obtain a policy π that helps detect all emergent behaviors and provide an optimal one. As shown in Figure 6, a MDP will emerge based on navigating different environments, applications and available service instances and descriptions. The compatibility of composed services is referred to as *Semantic Interoperability* and explained in the Section IV-C). Finally, the emergent process that is found to be the most optimal is then provided to the end user. If the feedback of the end user shows that the emergent process is accepted, the process is executed by the execution engine. Otherwise, it is not executed.

C. Regarding Service Integration in the Service Registry

Semantic Interoperability and Semantic Integration is a research topic since the 90s [26][27]. The interoperability of a required and a provided service is ensured when their syntactic level (e.g., data types and representation), semantic level (e.g., range of parameters or pre- and postconditions) and behavioural level (e.g., constraints on the ordering of interactions sequence) can be mapped [7]. In the context of distributed systems such as the presented platform, interactions styles implemented by the communication protocols (e.g., client-server) and non-functional properties are added. In most systems, syntactic incompatibility cannot happen as IoT components support standard communication protocols like HTTP over JSON. Consequently, the compatibility challenge shifts to the semantic level. In a distributed system, this is commonly referred to as Semantic Interoperability [26].
Semantic Interoperability ensures that services and data exchanges 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 [26]. Semantic interoperability in distributed systems is mainly achieved by establishing semantic correspondences (i.e., mappings) between vocabularies of different sources [27]. The question how to achieve semantic interoperability in dynamic and adaptive systems is still actively researched. For instance, formal domain standards such as ontologies or Linked Open Data vocabularies quickly lose their claim for correctness. Linked Open Data sources are prone to errors and inconsistencies (due to a lack of quality control) and ontologies must be constantly monitored and updated due to the pervasiveness and volatility of the underlying IoT environments [28]. In the scope of the presented architecture, the following challenges are relevant:

How are service components integrated? IoT services can be combined with other applications and services to create complex, context-aware business services. Therefore, service integration can either be performed from top-down or bottom-up [29]. Bottom-up refers to whether pre-existing service interfaces are glued together and thus adapter 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.
From the viewpoint of distributed systems (i.e., semantic interoperability), flow-based composition tools are commonly used to model an automation process by plugging in service instances. A common modelling pattern for such tools is *If this then that but only if*. This pattern may be visualized by using graphical elements (e.g., IFTTT [30]) or textual templates (e.g., TASKER [31]). When a process is modelled, all of these platforms integrate the available service instances by implementing a software adapter that manages the communication between the service description used in the process and the services instance. Here, no service registry is used as all software adapters must be manually implemented by software engineers. Hence, the end user himself must choose 1) which of the available services satisfies his requirements (c.f. service registry (D)) and 2) how services may be composed (c.f. composition mechanism (B)). A modification of just passing data and services offered by the service instance to the user interface of a platform is to define a domain-specific model. For example, the smart home platform openHAB [32] defines its data properties in a device vendor-independent way. In fact, the top-down defined domain model must be used to map device specific characteristic during software adapter implementation. Hence, the service integrator interprets both, the provided device interfaces and the required domain-specific model. However, this integration knowledge is not stored in a searchable format. Furthermore, this kind of integration knowledge is hard to reuse as every platform designs its own domain model.
On the one hand, the proposed platform in this paper should rely on heterogeneous and dynamically moving software components to provide emergent behaviour (i.e., vertical mappings between service instance and service description). On the other hand, the composition mechanism (B) requires a uniform representation of semantically interoperable service instances (i.e., horizontal mappings between service descriptions). Now, the question is whether the service integrator maps available service instances based on a top-down defined domain model or if a domain model of services is built up from the available service descriptions. In both ways, mappings are required. Hence, engineering approaches that can adapt and compensate

efficiently for a reconfiguration of service provisioning when context changes are necessary [28].

How can semantic interoperability between service components be achieved automatically? If mappings are required anyways when dealing with heterogeneous software components, the service integrator should be supported to create them. Current approaches such as INTER-IoT [33] heavily rely on the usage of domain ontologies. Therefore, a wide body of ontologies for different application domains can be used (see the website [34] for an overview). For example, Lov4Iot defines a subset of linked data vocabularies that are relevant within several domains that are affected by IoT (e.g., Home Automation or Wearables). When all service descriptions do refer to the same ontology concepts for a certain domain, then the ontologies can be queried for available mappings for needed vocabularies. As a consequence, generating a software adapter automatically is reduced to a technical challenge. However, most service instances do not provide semantic annotations and/or a machine-readable service description. Furthermore, it must be ensured that the domain ontology does contain all needed concepts so that the service integrator can map them based on a service instance. Hence, semantic annotations are critical for engineering integration knowledge in IoT.

From the viewpoint of artificial intelligence (i.e., semantic integration), ontologies are the state-of-the-art for storing the meaning of data. As IoT systems themselves offer data-driven service interfaces (i.e., mostly RESTful with no state or special behaviour), ontologies are often used in platform-based systems [33][35]. Naturally, the usage of an ontology is also possible within the proposed architecture.

However, what makes semantic data handling in IoT more challenging and fraught with technical difficulties is the scale of data generated by its corresponding resources, continuous changes in the state (and consequently description) of the resources and volatility of the IoT environments [28]. Hence, creating engineering tools that allow to cope with the increasing amount of IoT devices, their services and, most importantly, their combination by providing automation, reasoning and intelligence need to be designed. Furthermore, the question which service description language (e.g., OpenAPI or SA-WSDL) [36] and which mapping language is applicable by the service integrator and, more importantly, how semantic correspondences are captured during the lifecycle of the proposed platform must be answered.

To tackle these two challenges, a novel integration method called Knowledge-driven Architecture Composition that relies on an incremental formalization process for semantics can be used [37]. This method explicitly allows for incompleteness of integration knowledge and supports an evolutionary instead of an revolutionary definition of domain models (i.e., not based on fixed domain ontologies at system design-time). The novelty of this approach is formalizing semantic integration knowledge per use-case in a bottom-up manner. By focusing on integration knowledge instead of conforming to technological-oriented interface descriptions and standards, the method maximizes the impact for formalizing the semantic mappings as a concrete use-case must be present. Hence, formalization does only take place if it is specifically needed. The additional formalization effort in addition to implementing software adapters pays off as adapters for future devices can be generated (semi-) automatically through the usage of reasoning principles.

V. Conclusion and Future Work

We proposed a vision of an architecture as a so-called Emergent Platform. Using the architecture, a software engineer is able to develop a dynamic adaptive system which fulfills the emergent properties of an IoT based environment. The main building blocks of the architecture are separated into run-time and design-time. At run-time, the building blocks are capable to determine an application for user requirements, which emerge from the IoT environment based on available services. In a next step, the composition mechanism of the system provides the system behavior as a process which fulfills the expected behavior of an application. Thus, the mechanism to determine a process for an required application that emerged at run-time may minimize manual system maintenance. The feasibility of the architecture is exemplified with two use-cases from the home automation and event booking domain. In the future, we plan to evaluate the feasibility of identified techniques. Thus, our next step is a prototypical implementation including the technical basis for the composition mechanism.

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). The following partners were involved in this project: Universität Mannheim, Technische Universität Clausthal, Wolfsburg AG, StoneOne AG. Special thanks go to Mohamad Ibrahim, Abram Lawendy and Michael Pernpeintner for their valuable contribution to the project.

References

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," IEEE Communications Surveys Tutorials, vol. 17, no. 4, 2015, pp. 2347–2376.

[2] M. G. Jacobides, C. Cennamo, and A. Gawer, "Towards a theory of ecosystems," Strategic Management Journal, vol. 39, no. 8, 2018, pp. 2255–2276. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smj.2904 [retrieved: 2020.09.04]

[3] A. Bröring, S. K. Datta, and C. Bonnet, "A categorization of discovery technologies for the internet of things," in Proceedings of the 6th International Conference on the Internet of Things, ser. IoT'16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 131––139. [Online]. Available: https://doi.org/10.1145/2991561.2991570 [retrieved: 2020.09.04]

[4] K. Rehfeldt, M. Schindler, B. Fischer, and A. Rausch, "A component model for limited resource handling in adaptive systems," in ADAPTIVE 2017: The Ninth International Conference on Adaptive and Self-Adaptive Systems and Applications. IARIA, 2017, pp. 37–42.

[5] T. Vale et al., "Twenty-eight years of component-based software engineering," Journal of Systems and Software, vol. 111, 2016, pp. 128–148.

[6] C. Szyperski, Component Software: Beyond Object-Oriented Programming, 2nd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[7] 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.

[8] A. van Lamsweerde, "Requirements engineering in the year 00: a research perspective," in Proceedings of the 22nd international conference on Software engineering, ser. ICSE '00. Limerick, Ireland: Association for Computing Machinery, Jun. 2000, pp. 5–19. [Online]. Available: https://doi.org/10.1145/337180.337184 [retrieved: 2020.09.04]

 ISBN: 978-1-61208-781-8

[9] H. Wang et al., "Adaptive service composition based on reinforcement learning," in International conference on service-oriented computing. Springer, 2010, pp. 92–107.

[10] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," Sci. Comput. Program., vol. 20, no. 1–2, Apr. 1993, pp. 3—-50. [Online]. Available: https://doi.org/10.1016/0167-6423(93)90021-G [retrieved: 2020.09.04]

[11] E. Yu, "Towards modelling and reasoning support for early-phase requirements engineering," in Proceedings of ISRE '97: 3rd IEEE International Symposium on Requirements Engineering. IEEE, Jan. 1997, pp. 226–235.

[12] J. Castro, M. Kolp, and J. Mylopoulos, "Towards requirements-driven information systems engineering: the Tropos project," Information Systems, vol. 27, no. 6, Sep. 2002, pp. 365–389. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0306437902000121 [retrieved: 2020.09.04]

[13] F. S. Bäumer and M. Geierhos, "How to deal with inaccurate service descriptions in on-the-fly computing: Open challenges," in Natural Language Processing and Information Systems, M. Silberztein, F. Atigui, E. Kornyshova, E. Métais, and F. Meziane, Eds. Cham: Springer International Publishing, 2018, pp. 509–513.

[14] L. Van Rooijen and H. Hamann, "Requirements specification-by-example using a multi-objective evolutionary algorithm," in 2016 IEEE 24th International Requirements Engineering Conference Workshops (REW). IEEE, 2016, pp. 3–9.

[15] L. van Rooijen et al., "From user demand to software service: Using machine learning to automate the requirements specification process," in 2017 IEEE 25th International Requirements Engineering Conference Workshops (REW). IEEE, 2017, pp. 379–385.

[16] E. Friesen, F. S. Bäumer, and M. Geierhos, "Cordula: Software requirements extraction utilizing chatbot as communication interface," in Joint Proceedings of REFSQ-2018 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 23rd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2018), vol. 2075. CEUR-WS.org, 2018.

[17] H. A. Kautz and J. F. Allen, "Generalized plan recognition," in Proceedings of the Fifth AAAI National Conference on Artificial Intelligence, ser. AAAI'86. AAAI Press, 1986, pp. 32—37.

[18] E. Charniak and R. P. Goldman, "A bayesian model of plan recognition," Artif. Intell., vol. 64, no. 1, Nov. 1993, pp. 53—79. [Online]. Available: https://doi.org/10.1016/0004-3702(93)90060-O [retrieved: 2020.09.04]

[19] M. Ramírez and H. Geffner, "Plan recognition as planning," in Proceedings of the 21st International Jont Conference on Artifical Intelligence, ser. IJCAI'09. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 1778—-1783.

[20] ——, "Probabilistic plan recognition using off-the-shelf classical planners," in Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, ser. AAAI'10. AAAI Press, 2010, pp. 1121––1126.

[21] Y. Zeng et al., "Inverse reinforcement learning based human behavior modeling for goal recognition in dynamic local network interdiction," 2018. [Online]. Available: https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/16162 [retrieved: 2020.09.04]

[22] T. Yue, L. C. Briand, and Y. Labiche, "A systematic review of transformation approaches between user requirements and analysis models," Requirements Engineering, vol. 16, no. 2, 2011, pp. 75–99.

[23] H. Wang, X. Wang, X. Hu, X. Zhang, and M. Gu, "A multi-agent reinforcement learning approach to dynamic service composition," Information Sciences, vol. 363, 2016, pp. 96–119.

[24] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, "An approach for qos-aware service composition based on genetic algorithms," in Proceedings of the 7th annual conference on Genetic and evolutionary computation. Association for Computing Machinery, 2005, pp. 1069–1075.

[25] M. L. Puterman, Markov decision processes: discrete stochastic dynamic programming. John Wiley & Sons, 2014.

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

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

[28] 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.

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

[30] IFTTT, "IFTTT." [Online]. Available: https://ifttt.com [retrieved: 2020.09.04]

[31] Tasker, "Tasker for Android." [Online]. Available: https://tasker.joaoapps.com/ [retrieved: 2020.09.04]

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

[33] M. Ganzha, M. Paprzycki, W. Pawłowski, P. Szmeja, and K. Wasielewska, "Semantic interoperability in the internet of things: An overview from the inter-iot perspective," Journal of Network and Computer Applications, vol. 81, 2017, pp. 111–124.

[34] "Linked Open Vocabularies for Internet of Things (LOV4IoT)." [Online]. Available: http://lov4iot.appspot.com/ [retrieved: 2020.09.04]

[35] A. Bröring et al., "The big iot api-semantically enabling iot interoperability," IEEE Pervasive Computing, vol. 17, no. 4, 2018, pp. 41–51.

[36] K. Kurniawan, F. J. Ekaputra, and P. R. Aryan, "Semantic service description and compositions: A systematic literature review," in 2018 2nd International Conference on Informatics and Computational Sciences (ICICoS). IEEE, 2018, pp. 1–6.

[37] 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, Apr. 2017, pp. 108–111.

 ISBN: 978-1-61208-781-8