Towards Transforming OpenAPI Specified Web Services into Planning Domain Definition Language Actions for Automatic Web Service Composition

Christian Schindler, Christoph Knieke, Andreas Rausch, Eric Douglas Nyakam Chiadjeu
Technische Universität Clausthal
Institute for Software and Systems Engineering
Clausthal-Zellerfeld, Germany
Email: {christian.schindler, christoph.knieke, andreas.rausch, eric.douglas.nyakam.chiadjeu}@tu-clausthal.de

***Abstract*—Today, more and more highly complex Internet Of Things (IoT) ecosystems are emerging that can no longer be centrally designed and controlled, but must self-adapt to new environments and user requirements. An approach to achieve this self-adaptation are so called emergent software service platforms that must be able to react continuously at runtime to changes in the runtime environment, such as changes in user requirements or the addition/removal of software services. Therefore, all software service compositions are created only at runtime of the platform on the basis of the automatically detected user requirements. In a previous work, we introduced our vision of a software architecture for such an emergent software service platform. A core part of such a platform is the service composition mechanism. In this paper, we present a set of rules that can be used to transform web services specified in OpenAPI into Planning Domain Definition Language (PDDL) actions. This allows web service compositions to be performed automatically with common PDDL solvers.**

***Index Terms*—Web Service Composition, AI Planning, PDDL, OpenAPI, Emergent Systems.**

I. Introduction

IoT ecosystems are complex system conglomerates of autonomous and interacting individual systems that are adaptive as a whole because they exhibit a special capacity for adaptation [1]. Analogous to their model in nature, such ecosystems can only be vital in the long term if a balance of needs and interests is achieved with regard to data, services and processes. The different life cycles of the individual systems, whose behavior and interactions change over time, are also decisive here. These changes (e.g., in data models / domain ontologies) cannot usually be centrally pre-planned, but result from independent processes and decisions within and outside the IoT ecosystem [2]. In addition, it is becoming increasingly apparent that industry-wide standardization efforts are too slow and inflexible when it comes to the speed of innovation in IoT components. This poses major challenges to traditional approaches and technologies for self-adaptive systems, as these systems rely on automated (semantic) interpretation of data and functions in system composition.

Many of the currently realized mechanisms for self-adaptation are based on maximizing the interface couplings of the participating components of an IoT ecosystem or similar demand-independent criteria (see, e.g., [3], [4]). This results in self-functioning systems at runtime, but in many cases these systems are not able to address the changing functional and non-functional needs of the users. In particular, the same system guarantees are not useful or necessary for all applications on a platform; rather, they are highly application- or user-specific. For this reason, the rules governing self-adaptation within IoT ecosystems in the future must be increasingly introduced into the platform by the actual service demanders. Intervention by the platform operator, on the other hand, should be limited to moderating the communication and composition of services and components.

An emergent software platform must be able to react continuously at runtime to changes in the runtime environment, such as the addition/removal of software services or changes in user requirements [5]. Therefore, all software service compositions emerge only at runtime of the platform on the basis of automatically detected user requirements. In this paper, this property is defined as “emergence” in the context of a software platform. More formally, we define emergence as follows: A software platform is called emergent if the platform is able to automatically and dynamically compose the available software services into a higher-value software service in response to a triggering event. The emergent behavior of the platform is not predefined at design time and cannot be predicted by individual software services [5].

In a previous paper [5], we already proposed a vision of an architecture as a so-called Emergent Software Service 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.

A challenging task in the Emergent Software Service Platform is the web service composition mechanism. A common way to describe web services is using OpenAPI [6]. The OpenAPI specification defines an open and vendor-neutral description format for Application Programming Interface

 ISBN: 978-1-68558-047-6

(API) services. In particular, OpenAPI can be used to describe, develop, test, and document REST-compliant APIs. In the paper at hand we present an approach to transform web services specified in OpenAPI into PDDL [7] actions. The advantage of this approach is that web service compositions can be performed with common PDDL solvers (e.g., ENHSP [8]).

The paper is organized as follows: Section II gives an overview on the related work. In Section III, we introduce our architecture vision for emergent service composition, as well as the PDDL language. Our approach is proposed in Section IV. Finally, Section V concludes and gives an outlook on future work.

II. Related Work

Extensive research has been done on the topic of semantic integration of service interfaces in the last two decades. In particular, work in the area of semantic integration of web services [9]–[11], as well as in the area of dynamically adaptive systems in the IoT and Industry 4.0 environment should be mentioned here [12]–[14]. Unfortunately, until today, the supply of components and platforms that allow semantics-based networking of industrial distributed service systems falls far short of the initially high goals and expectations or only covers the data layer [15] [16].

As emergence offers the opportunity to take advantage of the composability of individual software components, we will comment on previous works in the field of service composition dealing with the translation of web service composition problem into planning problem, or the integration of web services.

PORSCE II [17] is one of the first semantic composition systems for web services with the particularity that it takes advantage of semantic information to improve the planning as well as the composition of software components. The realization of this feature is based on an external domain ontology coupled with OWL-S [17]. In addition, PORSCE II also provides a mechanism to replace services that fail during composition.

iServe [18], on the other hand, is not directly concerned with the composition of Web Services, but describes a new and open platform for publishing web services to better support their discovery and use. It provides a common vocabulary that can be used to describe different services in such a way that they can be found automatically by machines and their functionality is independent of the form used for the original web service description. An immediate advantage is that through iServe a large set of different Web Services can be discovered and integrated for composition.

To sum up, iServe offers an open platform for publishing Web services, but does not provide composition of Web services. PORSCE II, on the other hand, addresses planning and composition but requires an additional description of Web services in OWL-S. In contrast, our approach leverages the widely used and standardized description of Web services in OpenAPI which can thus be used immediately with less effort.

III. Fundamentals

Dynamic adaptive software systems in an IoT-based environment can be designed from reusable software components [19], e.g., as proposed in the DAiSI component model [20], 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.

A. Architecture

Figure 1 illustrates the architecture of our emergent platform as introduced in a previous work [5]. The architecture consists of five major parts (see letters A-E in Figure 1) which are either associated to the run-time or design-time part of the platform and will be briefly explained in the following:

Run-Time: The emergent platform as a whole interacts with users to determine formal user requirements through interactions and monitoring (A). The Domain is a central part of the architecture, as it is the foundational vocabulary to express user requirements and the foundation for a semantic description of the software components. The formal user requirement is passed on to the composition mechanism (B) to compose a sequence of Software Service Descriptions to fulfill the expected system behavior demanded by the formal user requirement. The composed sequence of components is forwarded to the Execution Engine (C). The responsibility of this component is to call Service Instances that "use/implement" the given software components of the composed sequence, to incorporate necessary user feedback into the execution, and to return process results to the user (E).

Design-Time: The Service Descriptions are developed and maintained by a service integrator at design-time. All those Service Descriptions that are allowed to be used in the composition are registered in the Service Registry (D).

B. Self-adaptive Composition Mechanism

The Self-adaptive Composition Mechanism has to compose services provided in the IoT ecosystem to achieve a given higher-level goal. This includes determining which services are needed and how these services can be invoked sequentially to achieve the goal. Computing an appropriate composition is formulated as a planning problem where the user's requirement is the goal and the service descriptions of the IoT ecosystem are possible actions that can be taken.

In the overall architecture, the Self-Adaptive Composition Mechanism has the following interfaces to other components: it has an interface from the User Requirements Handler, from which the target of an end user is provided. From the Service Registry, the component can query all currently available Service Descriptions and compute the composition based on them. Another interface is responsible for passing the calculated composition to the Execution Engine.

 ISBN: 978-1-68558-047-6

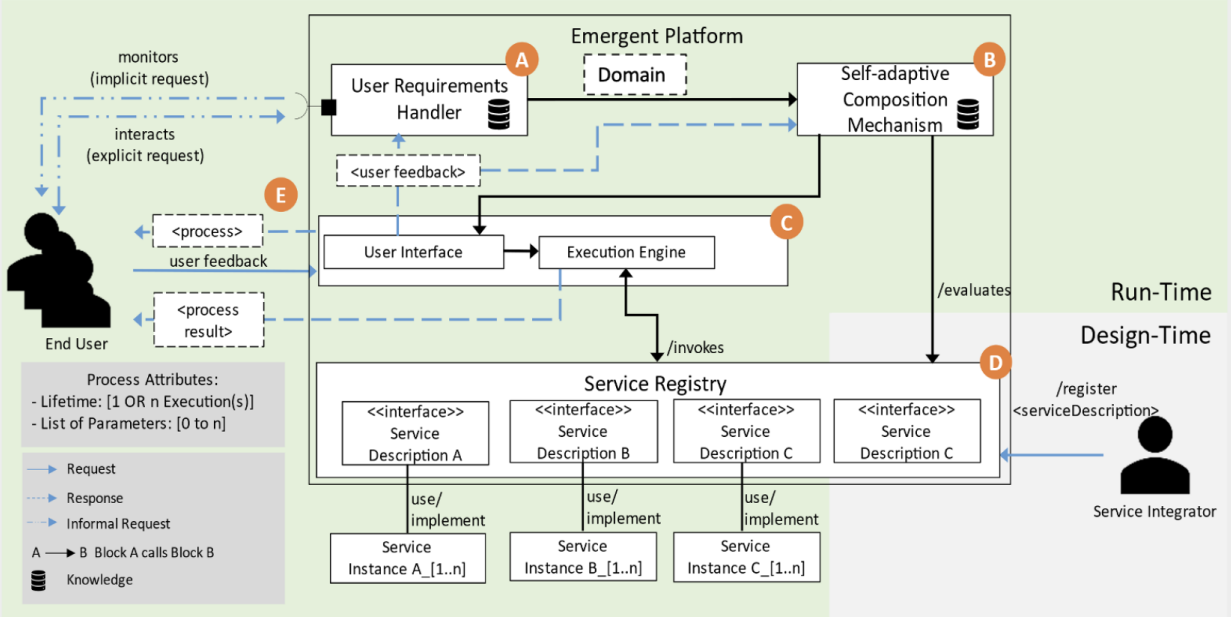

Fig. 1. Architecture of the emergent software service platform

C. Planning Domain Definition Language

PDDL is a standard encoding language for "classical" planning.

The components of PDDL files are:

- *Requirements:* Defining levels of abstraction in the language, temporal, probabilistic effects, etc.
- *Types:* Sets of the things of interest in the world
- *Objects:* Instances of types
- *Predicates:* Properties representing a state or condition in the world that can be true or false
- *Predicates:* Facts about objects that can be true or false
- *Initial state of the world*: Before starting the planning process
- *Goal:* Properties of the world true in goal states and achieved after the planning process
- *Actions/Operators:* Ways of changing states of the world and going from the initial state to goal states

Predicates are used to define the initial state of a planning problem, as well as to represent the goal state that the planner is trying to achieve. In addition, predicates can be used to describe properties of objects, relationships between objects, and other conditions that are relevant to the planning problem.

A planning task in PDDL is specified in two text files: A *domain file* for requirements, types, predicates and actions. A *problem file* for objects, initial state and goal specification.

PDDL plans from a state S_0 (initial state) to a S_z (target state). S_0 is our initial state which can be empty. S_z is the higher order requirement that can be expressed in terms of domain concepts. A is a set of actions available to the platform. The planner calculates a valid path (sequence of $a_i \in A$) from S_0 to S_z. The sequence of A_i can be called one after the other to reach the state S_z.

An important point in the computed sequence of A_i is that at the appropriate point in the execution sequence, all the information required at that time, which is needed as parameters, is available. These can either be given in the S_0 or obtained by executing an $a_j \in A$ with $(j < i)$.

IV. Rule-based Web Service Composition Approach

A. Motivating example

With a motivating example, we want to show the expressive power that the PDDL brings to the problem of web service composition, having the respective description of such planning tools needed to perform their planning. We want to give an example covering the Domain, available PDDL actions representing Service Descriptions, a user requirement needing to be composed of multiple available Service Descriptions, and finally the composed sequence of Service Descriptions a planing tool (such as the one we use) can compute.

The example reflects some core interactions of components of the architecture shown in Figure 1. Starting with the User Requirement Handler (A) giving the formal request (Figure 4) to the Self-Adaptive Composition Mechanism (B). The computed sequence of Service Description is shown in Figure 3. The example also shows the available Service Descriptions contained in the Service Registry (D) in Figure 3 and the Platforms Domain. The example does not cover the user interaction with the platform, neither the executing part (C). These aspects are addressed in a further paper [21] and shown together with the implementation of the platform and an application scenario. The example is in the application domain of a parking lot, offering to book a parking spot, and offering to get a car wash. The PDDL domain in this example (Figure 2) defines the types parkingid, reservationnr and bookedservice. The second part of the domain are the

 ISBN: 978-1-68558-047-6

```
(:types
    parkingid
    reservationnr
    bookedservice
)
(:predicates
    (parkplatz
        ?p - parkingid)
    (bookedparking
        ?p - parkingid
        ?r - reservationnr)
    (bookedcarwash
        ?p - parkingid
        ?g - bookedservice)
)
```

Fig. 2. PDDL Domain

predicates, which are defined similar to first order logic with an identifying name followed by the typed arguments.

Figure 2 contains the available actions given in the scope of the example, referring to types and predicates from the domain. Parameters are objects that will be passed to actions in the plan. The predicates given in an actions precondition need to evaluate to true for the passed objects to be able to be scheduled by the planning tool. The effects on the other hand are then enforced by the planner. The assignment of truth values to the predicates in the effects essentially allow the planning tool to finally reach the goal state of a planing problem. Figure 4 and 5 give a requirement and the respective calculated composition. The syntax of those is no plain PDDL, as this is the payload format of the platform itself. The requirement nevertheless describes available objects and their types and defines a goal to be reached. This goal's value is a PDDL conform String. An additional information that could be provided is an initial state (of the world) so the planner does not start from en empty state for the composition of actions.

The composition in Figure 5 also contains the objects originally received and in addition the composition. The composition is a sequence of actions, with valid assignments of available objects to the actions parameters, so that by following the sequence the original goal state is reached.

Coming back to the idea of treating web services as actions that can be executed in order, planning tools (e.g., ENHSP [8])) offer a good foundation on finding suitable action sequences to reach a given goal. In addition to the calculation of a sequence, the plan also contains objects that need to be passed to the actions in a given step of the plan to reach the goal state. To use such an approach for the web service composition part of this platform we want to emphasize the benefit of having the ability to derive a PDDL domain and actions reflecting the web services, so a composition can be calculated. As the manual translation can be cumbersome and error prune, we will give a set of rules on how a domain and actions similar to the ones in the example can be derived from given web service descriptions available in an OpenAPI specification.

```
(:action get_available-parkingspot
    :parameters (?p - parkingid)
    :precondition ()
    :effect (parkplatz ?p)
)
(:action post_book-parking
    :parameters (
        ?p - parkingid
        ?r - reservationnr)
    :precondition (parkplatz ?p)
    :effect (bookedparking ?p ?r)
)
(:action post_book-carwash
    :parameters (
        ?p - parkingid
        ?r - reservationnr
        ?l - bookedservice)
    :precondition (bookedparking ?p ?r)
    :effect (bookedcarwash ?p ?l)
)
```

Fig. 3. Available Service Descriptions

```
{ "environment": [
  {"type": "parkingid", "name": "p1"},
  {"type": "reservationnr", "name": "r1"},
  {"type": "bookedservice", "name": "g1"}
  ], "init": [],
  "goal": "(and (bookedparking p1 r1) (bookedcarwash
       p1 g1))"
}
```

Fig. 4. Requirenment

B. PDDL Actions

An action has three important sections that are required for planning. These are Parameters, Preconditions and Effects. Parameters are the objects that are passed to the action. These objects are used to check preconditions and to apply effects. The preconditions allow the planner to determine if an action can be scheduled and the effect allows the planner to determine if the action helps in getting towards the target state.

A web service, on the other hand, is not described in the same way. Here, the inputs and outputs are essentially described, along with interface details, such as Hypertext Transfer Protocol methods, content types, and response codes, which play a minor role for our approach.

```
{ "composition": [
  {"name": "get_available-parkingspot",
   "params": ["p1"]},
  {"name": "post_book-parking",
   "params": ["p1","r1"]},
  {"name": "post_book-carwash",
   "params": ["p1", "r1", "g1"]}
  ], "environment": [
    {"name": "p1", "type": "parkingid"},
    {"name": "r1", "type": "reservationnr"},
    {"name": "g1", "type": "bookedservice"}
]}
```

Fig. 5. Composed Sequence of Service Descriptions

The task now is to find suitable assignments of the inputs and outputs of the web services to the described sections of the PDDL actions, so that planning can be based on them.

C. Approach

The focus of this paper is on how to allow the semantical integration of available Web Services Descriptions into the platform's Service Registry to enable the platform's internal processing (calculation of compositions and incorporation into the platform domain to express user requirements).

We have defined multiple rules for the transformation of OpenAPI specifications into PDDL actions and the corresponding PDDL domain, to enable planning tools to use the actions and domain to plan according to provided problems. Figure 7 shows the algorithm on how to transform given paths of an OpenAPI specification into PDDL actions. We distinguish between primitives and more complex types (*object*) in the schema defined in the OpenAPI specification. Primitives in the sense of this approach are key-value pairs with a simple data type (e.g., string, number or boolean). Elements of type *object* are treated differently in our rules. To identify a given data type of an object in the algorithm, we introduce *isPrimitive()* and *isObjectType()*.

Rule Creation of an action: We create an action for each method of each path in the given specification. For each method of each path, we create an action (lines 4, 5, and 6 in the figure). The name of the action is a concatenation of the path and the currently handled method (e.g., *get_status*) for the path *status* and the method *GET*. Line 25 collects all the actions. To omit the risk of duplicates, it would be feasible to also incorporate a unique identifier, which is not part of this pseudo code, but would be attached in line 6.

Rule Precondition collection: To schedule an action into a plan, it is necessary to gather the preconditions required for the action to be executed successfully. To collect the preconditions, we process the requestBody of the OpenAPI specification. The idea behind this is that the requestBody also needs to be provided to the web service to be used, meaning the information needs to be present prior to the actual call. In the algorithm (Figure 7) this is shown in lines 8 to 15. The *objects()* in the algorithm collects and returns (recursive) all objects contained in the context of the calling element of the OpenAPI specification. Starting in line 8 we iterate over all objects and check if they are of type *object* (line 9). If this is the case a new precondition is found and added (line 10). In the next step all the child objects get processed, to find the precondition's parameters. To find them we again process all the contained objects recursive (line 11) and check if they are primitive (line 12) and add the parameter to the precondition in line 14. For simplification, we do not distinguish different contentTypes that can be offered by a requestBody (we only handle one).

Rule Effect collection: Similar to the collection of preconditions, we collect the effects, but by looking at the response specified in the OpenAPI specification. We focus on the good case responses (e.g., 200 or 201) which is not mentioned specifically in the algorithm. The underlying idea is that the response reflects what the web service has done. This is reflected for example by retrieved objects from the backend. The function block in the algorithm is line 17 to 23 and following the same logic than the collection of preconditions, with the mentioned difference of interacting over the response instead of the request body. A limitation of this transformation approach is if something happens while executing the web service which is not reflected by the response, this is not automatically covered by the actions. It is possible to extend the actions if more knowledge about the functionality of the web service is available (e.g., by experts) having a deeper understanding of the service and the correlation of its input and output.

Rule Parameter collection: The third building block of PDDL actions are the parameters. In comparison to the preconditions and effects, which are predicates, the parameters are typed variables, with types from the PDDL domain. There is a relationship between the parameters (of the action) and the predicates used in the preconditions and effects. In the algorithm the preconditions are simultaneously collected with the primitive child objects of preconditions (line 13) and the primitive child objects of effects (line 22).

Rule Creation of the PDDL Domain: The second algorithm given in Figure 8 describes how to derive the PDDL domain from OpenAPI specification. We iterate over all objects of the requestBody and the response, but this time we do not distinguish where the objects originate, as this does not matter for the domain. Primitive objects are transformed into domain types and objects of type *object* with their parameters are transformed into predicates in the domain. Iterating over the objects is done starting in line 7. Primitive types are identified in line 14 and added to the domain types in line 15. *Objects* are identified in line 8. Predicates get created in line 9. Lines 10, 11, and 12 collect the primitive parameters belonging to the domain predicate. This is the same logic, which is applied to collecting the preconditions and effects (compare Figure 5, lines 11, 12 and, 14). Lines 13 and 15 add the predicates and types to the domain.

D. Example transformation of an OpenAPI specified service

The following section gives a concrete example of how an OpenAPI-specified web service is transformed into a PDDL action and the corresponding domain. Figure 2 lists the OpenAPI specification (left-hand side) with the resulting PDDL action and domain (right-hand side). The highlighted parts visualize the corresponding parts in both formats, with the applied rules assigned. The grey box indicates a legend on the highlighting to the specific rules applied. We have one path with one method (*post*) transformed into a single action (orange box highlighting). The preconditions of the action are highlighted in green. On the right side, we see the resulting precondition *parkingspot* with its parameter *p*. On the left-hand side, we see that this is derived from the schema of the requestBody (and the according schema elements that are referred to by the requestBody). The same applies to the effect

 ISBN: 978-1-68558-047-6

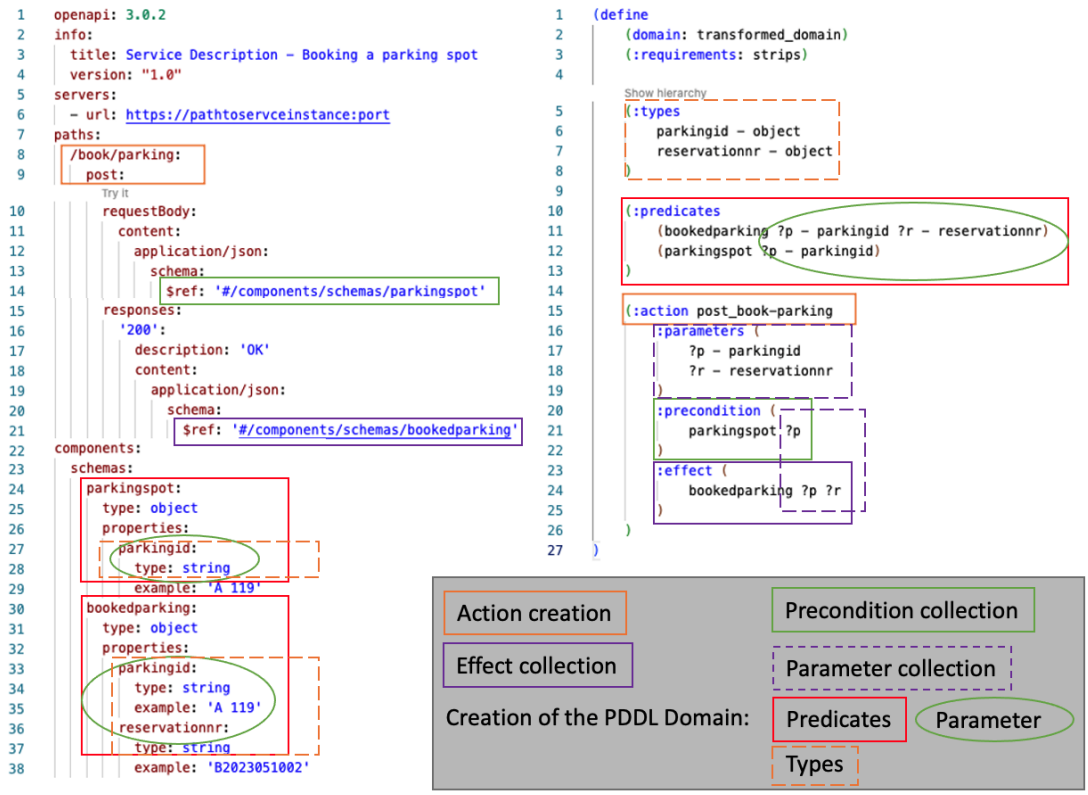

Fig. 6. Example REST API to PDDL

```
oas: OpenAPI
ps: PDDL

for path in oas.paths:
 for m in path.methods:
  action <- path.m + "_" + path.name.toLowecase()
  // Preconditions
  for object in m.requestBody.content.schema.objects():
   if object.isTypeObject():
    precondition.name <- object.name
    for childObject in object.objects():
     if childObject.isPrimitive():
      action.parameters <- childObject
      precondition.parameters <- childObject
    action.preconditions <- precondition
  // Effects
  for object in m.response.content.schema.objects():
   if object.isTypeObject():
    effect.name <- object.name
    for childObject in object.objects():
     if childObject.isPrimitive():
      action.parameters <- childObject
      effect.parameters <- childObject
    action.effects <- effect
  ps.actions <- action
```

Fig. 7. Rule Action Preconditions

highlighted in purple. In the upper part of the right-hand side, we see the definition of the domain types (orange dashed highlighting), the predicates (red highlighting), and their parameters (green ellipse) with their respective counterparts in the schema to which requestBody and the response refer.

```
oas: OpenAPI
ps: PDDL

for path in oas.paths:
 for m in path.methods:
  for context in { m.requestBody, m.response}:
   for object in context.content.schema.objects():
    if object.isTypeObject():
     predicate <- object.name.toLowercase()
     for childObject in object.objects():
      if childObject.isPrimitive():
       predicate.parameters <- childObject
     ps.domain.predicates <- predicate
    else if object.isPrimitiveType():
     ps.domain.types <- object.name.toLowercase()
```

Fig. 8. Rule PDDL Domain

V. CONCLUSION AND FUTURE WORK

In this paper, we have shown how PDDL can be used to compose web services to fulfill a more complex user request than any of the given services are capable of satisfying in its own way. We have given an example on how such a request for a given domain and given actions can look like. We have also described the differences in an OpenAPI specified web service and an action specified in PDDL. We have defined a set of transformation rules and described the pseudo code.

Nevertheless, the presented approach still has some limitations. As the OpenAPI specification does not explicitly describe coherence between the objects in the requestBody and

 ISBN: 978-1-68558-047-6

the response, the usage of given parameters in an action, and their occurrence in preconditions and effects massively affect the semantic meaning of an action. A possible quality gate is that an expert can ensure the quality and the correct semantics, editing the parameters, used parameters in preconditions and effects, if this collected information is not correct. A second limitation is the focus on specific response codes such as 200 or 201, as this is often the expected good case of such a web service. Other status codes in the range of 400 or 500 are not taken into account. This could easily be incorporated by also iterating over all given status codes per response, creating actions for each.

As future work, we want to investigate how the service integrator can be supported by the rules and the transformation (compare Figure 1, bottom right). Taking into account the stated limitations of the current approach, it is worth noting that a service integrator can offer supplementary knowledge beyond what is included in the OpenAPI specification.

ACKNOWLEDGMENT

This work was funded by the German Federal Ministry of Education and Research (Research Grant: 01IS18079, Project: BIoTope).

REFERENCES

[1] P. Pradeep, S. Krishnamoorthy, and A. V. Vasilakos, "A holistic approach to a context-aware IoT ecosystem with Adaptive Ubiquitous Middleware," *Pervasive and Mobile Computing*, vol. 72, 2021.

[2] M. Zdravković *et al.*, "Domain framework for implementation of open IoT ecosystems," *International Journal of Production Research*, vol. 56, no. 7, pp. 2552–2569, 2018.

[3] R. Rouvoy *et al.*, "Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments," *Software engineering for self-adaptive systems*, pp. 164–182, 2009.

[4] I. Corredor, J. F. Martínez, M. S. Familiar, and L. López, "Knowledge-aware and service-oriented middleware for deploying pervasive services," *Journal of Network and Computer Applications*, vol. 35, no. 2, pp. 562–576, 2012.

[5] N. Wilken *et al.*, "Dynamic adaptive system composition driven by emergence in an iot based environment: Architecture and challenges," in *Proceedings of the 12th International Conference on Adaptive and Self-Adaptive Systems and Applications*. IARIA Press, 2020, pp. 25–29.

[6] D. Sferruzza, J. Rocheteau, C. Attiogbé, and A. Lanoix, "Extending openapi 3.0 to build web services from their specification," in *International Conference on Web Information Systems and Technologies*, 2018.

[7] D. M. McDermott, "The 1998 AI planning systems competition," *AI magazine*, vol. 21, pp. 35–35, 2000.

[8] E. Scala, P. Haslum, S. Thiébaux, and M. Ramirez, "Interval-based relaxation for general numeric planning," in *ECAI 2016*. IOS Press, 2016, pp. 655–663.

[9] N. F. Noy, "Semantic integration: a survey of ontology-based approaches," *ACM Sigmod Record*, vol. 33, no. 4, pp. 65–70, 2004.

[10] H. Nacer and D. Aissani, "Semantic web services: Standards, applications, challenges and solutions," *Journal of Network and Computer Applications*, vol. 44, pp. 134–151, 2014.

[11] M. Klusch, P. Kapahnke, S. Schulte, F. Lecue, and A. Bernstein, "Semantic web service search: a brief survey," *KI-Künstliche Intelligenz*, vol. 30, pp. 139–147, 2016.

[12] M. D'Angelo, M. Caporuscio, and A. Napolitano, "Model-driven engineering of decentralized control in cyber-physical systems," in *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 2017, pp. 7–12.

[13] M. U. Iftikhar, G. S. Ramachandran, P. Bollansée, D. Weyns, and D. Hughes, "Deltaiot: A self-adaptive internet of things exemplar," in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2017, pp. 76–82.

[14] I. Crnkovic, I. Malavolta, H. Muccini, and M. Sharaf, "On the use of component-based principles and practices for architecting cyber-physical systems," in *2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*. IEEE, 2016, pp. 23–32.

[15] V. Bauer and A. Vetro', "Comparing reuse practices in two large software-producing companies," *Journal of Systems and Software*, vol. 117, pp. 545–582, 2016.

[16] P. Bonte *et al.*, "The massif platform: a modular and semantic platform for the development of flexible iot services," *Knowledge and Information Systems*, vol. 51, pp. 89–126, 2017.

[17] O. Hatzi, D. Vrakas, N. Bassiliades, D. Anagnostopoulos, and I. Vlahavas, "The PORSCE II framework: Using AI planning for automated semantic web service composition," *The Knowledge Engineering Review*, vol. 28, no. 2, pp. 137–156, 2013.

[18] C. Pedrinaci *et al.*, "iserve: a linked services publishing platform," in *CEUR workshop proceedings*, vol. 596, 2010, pp. 2–13.

[19] 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, pp. 2347–2376, 2015.

[20] 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 Press, 2017, pp. 37–42.

[21] C. Knieke *et al.*, "Emergent Software Service Platform and its Application in a Smart Mobility Setting," in *15th International Conference on Adaptive and Self-Adaptive Systems and Applications*, 2023, *to appear*.

 ISBN: 978-1-68558-047-6