The SPA Ontology: Towards a Web of Things Ready for Robotic Agents

Michael Freund[1,*], Daniel Schraudner[2], Sebastian Schmid[2], Christoph Stade[1,3], Thomas Wehr[2] and Andreas Harth[1,2]

[1]Fraunhofer Institute for Integrated Circuits IIS, Nürnberg, Germany
[2]Friedrich-Alexander-Universität Erlangen-Nürnberg, Nürnberg, Germany
[3]Technische Hochschule Nürnberg Georg Simon Ohm, Nürnberg, Germany

Abstract

We present the Simple Planning Annotation (spa) ontology for modeling preconditions and effects of interaction affordances within Web of Things Thing Descriptions and propose a mapping to the Planning Domain Definition Language to enable robot-device interaction. We use encoded semantic knowledge to generate a planning problem that can be used within existing AI planning algorithms to dynamically plan interaction sequences to achieve specified goals without the extensive pre-programming traditionally required, as demonstrated by the validation of our approach through a prototypical implementation. The scalability evaluation shows that when the number of input Web of Things actions increases by a factor of 1,000, the runtime of the implemented prototype increases by about 14.3% and the memory consumption by about 9.4%, indicating vertical scalability.

Keywords

Actionable Knowledge Representation, Web of Things, AI planning

1. Introduction

Autonomous robotic agents can perform tasks by interacting and manipulating their environment either through direct physical interaction, such as fetch-and-place and wiping tasks [1], or through an IoT-aided approach where a robot acts as an actuated edge device and uses network requests to interact with and control other smart devices [2, 3]. The IoT-aided approach requires machine-to-machine communication, where robots must seamlessly interact with diverse and previously unknown Internet of Things (IoT) devices. The World Wide Web Consortium's (W3C) Web of Things (WoT) specification [4] supports such an approach by providing a uniform, machine-readable abstraction of an IoT device [5], the so-called Thing Description (TD) [6]. A TD is a semantic interface description structured using the Resource Description Framework (RDF) and typically serialized in JSON-LD format. TDs contain metadata about a device, such as location and environmental information, and available interaction affordances, which are grouped into three categories: *Properties*, which represent the internal state of a device, *Actions*,

AKR³: 1st International Workshop on Actionable Knowledge Representation and Reasoning for Robots, 27th May 2024, Hersonissos, Greece, co-located with 21st Extended Semantic Web Conference (ESWC 2024)
*Corresponding author.
michael.freund@iis.fraunhofer.de (M. Freund)
0000-0003-1601-9331 (M. Freund); 0000-0002-2660-676X (D. Schraudner); 0000-0002-5836-3029 (S. Schmid); 0009-0009-3973-5139 (C. Stade); 0000-0002-0678-5019 (T. Wehr); 0000-0002-0702-510X (A. Harth)

which allow invocation of tasks, and *Events*, which are used to model asynchronous data pushes. An IoT device described by a TD is called a *Thing* in the WoT context.

Robotic agents can parse TDs, but TDs lack information about preconditions and effects of interactions. This limitation prevents robotic agents from reasoning out the correct order of command execution using AI planning algorithms to achieve predefined goals, especially in unfamiliar environments. As a result, robotic agents rely on control programs that specify their interactions with other devices and outline step-by-step procedures for each task [7].

By incorporating a semantic description of the conditions required to invoke interactions and the effects of those interactions on devices into TDs, and by providing a mapping algorithm between the semantic information contained in TDs and the standardized Planning Domain Definition Language (PDDL) [8], robotic agents would be able to adapt to new environments without the need for exhaustive pre-programming for every possible scenario.

Previous approaches attempting to integrate information about sequential behavior, preconditions, and effects into TDs have primarily focused on interactions between resource-constrained devices that are unable to use planning algorithms. Therefore a pre-computed interaction plan has been annotated in TDs, which hinders use in dynamic scenarios. Other approaches have adopted centralized architectures where all environmental information is consolidated in a central knowledge repository. While this strategy is viable for mostly static smart home applications, it is not suitable for dynamic environments.

In this paper we present the spa ontology for modelling preconditions and effects of interaction affordances in TDs. Together with the ontology, we introduce a mapping between the semantic information contained in TDs into PDDL domain and problem definitions. The mapping allows standard AI planning algorithms to plan the interactions necessary for an autonomous robotic agent to achieve a given goal. We also provide a prototypical implementation of the mapping algorithm with link following [9] to discover TDs to validate our approach, and evaluated the scalability of the approach by systematically increasing the available WoT actions, i.e. the search space, and measuring runtime and memory consumption.

The contributions of our work are

- an ontology describing preconditions and effects in TDs,
- a mapping between semantic annotations in TDs and PDDL domain and problem statements,
- and a validation via a prototypical implementation of a mapping algorithm with an evaluation of the scalability of the proposed mapping approach.

2. Running Example

In the following sections, we will use the interaction of a robotic agent with two WoT Things to illustrate the introduced annotations and mappings. The setup consists of a controller capable of opening and closing an automatic door and an intelligent power supply that provides different voltage levels to the door. Specifically, the power supply must provide 5 volts to open the door and -5 volts to close the door. The goal for the robotic agent in the scenario is to open the automatic door. Figure 1 shows the setup.

Figure 1: Example scenario of an intelligent power supply that provides different voltage levels to an automatic door and a controller capable of opening and closing the automatic door. Available properties (P) and actions (A) are shown.

The power supply has interactions that allow users to read the current voltage output and to change the voltage output by adding a value between -10 and 10 to the current value. A TD of such a power supply following the W3C TD 1.1 standard, is shown in listing 1.

The automatic door controller can read the current status of the door and provide interactions to either open or close the door. But, for the door to open, two conditions must be met: the power supply must be set to output 5 volts and the door must be in the closed state. Conversely, to close the door, the power supply must be at -5 volts and the door must be in the open state. A TD of such an automatic door controller following the W3C TD 1.1 standard, is shown in listing 2.

However, both TDs lack information about the preconditions and effects of interactions. For example, an agent will currently not be able to determine that to open the door, the `VoltageOutput` property must be 5 volts, or that `changeVoltageByValue` changes the `VoltageOutput` property.

3. Related Work

Modelling sequential behavior and effects in TDs has been investigated before. Korkan *et al.* [10] introduced a *path* vocabulary. The vocabulary describes the sequential interactions required to change the current state to a desired goal state, thereby abstracting from the internal state machine of the device. The authors argue that using the path vocabulary in combination with a pre-computed order of interactions is suitable for interactions between resource-constrained devices because it simplifies processing. In contrast, we focus on the interaction between a robotic agent and resource-constrained devices, where the robotic agent, possessing greater computational power, is able to dynamically use planning algorithms during runtime without relying on pre-computed paths encoded in TDs.

Planning algorithms in combination with autonomous robotic agents have been used to tackle a variety of challenges, such as robot navigation [11]. The classical planning approaches in general try to solve planning problems where the objective is to transform a given initial state of a domain into a goal state by taking predefined actions. The Planning Domain Definition Language (PDDL) [8, 12] is a standardized way to model planning problems. PDDL models the initial state, the available actions, and the goal state by dividing the definitions into two parts: a general domain description, which includes the available actions along with their preconditions and effects, and a specific problem description, which includes concrete objects, the initial state,

Listing 1: Excerpt of an intelligent power supply TD that exposes an `outputVoltage` property and a `changeVoltageByValue` action.

```
"title": "PowerSupply",
...
"properties": {
  "outputVoltage": {
      "@id": "outputVoltage", "type": "integer",
      "readOnly": true,
      "forms": [{ "href": "http://ex.org/outputVoltage" }] }
},
"actions": {
  "changeVoltageByValue": {
      "input": {"@id": "changeVoltageInput", "type": "integer", "minimum": -10,
          "maximum": 10},
      "forms": [{ href: "http://ex.org/changeValue" }] }
}
```

Listing 2: Excerpt of an automatic door controller TD that exposes an `openState` property and `open` and `close` actions.

```
"title": "DoorController",
...
"properties": {
  "openState": {
      "@id": "doorOpenState", "type": "boolean",
      "readOnly": true,
      "forms": [{ "href": "http://ex.org/openState" }] }
},
"actions": {
  "open": {
      "forms": [{ href: "http://ex.org/open" }] },
  "close": {
      "forms": [{ href: "http://ex.org/turnOff" }] }
}
```

and a goal state [12].

The possible combination of TDs and PDDL has been considered before by Noura *et al.* [13]. The authors presented an architecture that enables end-user development for smart home WoT environments using PDDL. To generate the planning problem, a central knowledge repository is used that stores domain-relevant aspects in RDF. In addition, a context-sensing component monitors the state of devices, while the planner component translates information from the knowledge repository into PDDL definitions and generates a plan that is executed by an execution engine. In contrast to this centralized approach, which relies on a knowledge repository to store domain information and context-sensitive components to monitor device states, we use

Ontology Requirements Specification Document

Purpose
Assist autonomous robots in interacting with unfamiliar IoT device environments using the WoT abstraction covering WoT properties and WoT actions.

Scope
IoT, WoT, Autonomous Robots

Implementation Language
RDF Schema (RDFS), Web Ontology Language (OWL)

Intended Users
Autonomous robots, software agents, developers

Non-functional Requirements
NFR1: The ontology must be based on terminology from the WoT architecture.
NFR2: All ontology resources must be labeled and commented in English.

Functional Requirements
Multiple functional requirements have been identified, as seen in Table 1.

Figure 2: The Ontology Requirements Specification Document for the spa ontology.

only the TD annotated with precondition and effect information to dynamically transform TDs into a PDDL planning problem solvable by standard PDDL planners. Furthermore, our focus is on interactions between robotic agents and Things, rather than between non-professional users and Things.

4. Conditions and Effects in Thing Descriptions

To describe a mapping between interaction affordances in TDs and PDDL descriptions, additional semantic annotations for conditions and effects are required. Therefore, we introduce an ontology called Simple Planning Annotation with a preferred prefix of spa. In the TD we focus on the basic interaction affordances of reading a property, writing a property, and invoking an action. We developed spa using the Linked Open Terms (LOT) methodology [14]. LOT is a lightweight methodology for developing vocabularies and ontologies, based on lessons learned and best practices from several established ontology development guides, such as Ontology Development 101 [15] and the NeOn project [16]. LOT describes four steps *ontology requirements specification*, *ontology implementation*, *ontology publication*, and *ontology maintenance* to create an ontology. The key results from each step are described in the following subsections.

4.1. Ontology Requirements Specification

The first step of the LOT methodology is to produce an *ontology requirements specification document*, by defining potential use cases and deriving related competency questions.

Use case specification The use case specification activity outlines potential use cases of the spa ontology by listing concrete applications. During the creation of the ontology, we identified several potential use cases, which are documented online[1]. Below, we present the use case representing the running example.

Use Case: An autonomous robot wants to pass through an automatic door.

Description: A robotic agent operates in an environment that includes two IoT devices: an intelligent power supply and an automatic door controller, both controllable by requests. The devices are described by TDs, which turn them into Things. To interact with these Things, the robotic agent must analyze the available interaction affordances and evaluate the preconditions and effects of each interaction to achieve its goal of opening the door.

Actors: robotic agent, automatic door controller, intelligent power supply.

Flow: The robotic agent sets the goal of opening the door, discovers and consumes the TD of the automatic door controller, and finds a link to the TD of the intelligent power supply, which the agent also consumes. Next, the agent evaluates all affordances, associated preconditions, and effects, and maps them to a PDDL planning problem. A PDDL planner then solves the planning problem and returns the sequence of interactions required to achieve the goal.

Functional Ontological Requirements In addition to the general behavior-describing non-functional requirements, competency questions are utilized to specify document content-specific requirements. Some sample competency questions extracted from the use case presented are listed in Table 1.

The final ontology requirements specification document is shown in Figure 2.

Table 1
Identified Competency Questions

Competency Question
Does action X have any preconditions?
Which are the preconditions of action X?
What properties will be affected by performing action X?
How does performing action X affect properties Y?
What actions must be invoked to satisfy the precondition for action Y?

4.2. Ontology Implementation

The ontology model is described using RDFS vocabulary elements. To reuse definitions and make the way precondition terms are evaluated unambiguous, we base ontology operations on operations defined in the W3C's XPATH recommendation[2]. For instance, our ontology's `spa:NumericMultiply` is mapped to XPATH's `op:numeric-multiply`. Other mathematical expressions are based on OpenMath[3] definitions, such as `spa:LogicalAnd` or `spa:LogicalOr`. Finally, the qualities of our ontology in terms of structure, semantic representation, and interoperability were evaluated using the OntOlogy Pitfall Scanner! [17].

[1] https://github.com/wintechis/SimplePlanningAnnotation/tree/main/docs
[2] https://www.w3.org/TR/xpath-functions-31/
[3] https://openmath.org/

4.3. Ontology Publication and Maintenance

The ontology, available classes and properties are documented using WIDOCO [18] to create a human-readable online document[4]. As part of the maintenance step, we are making all resources publicly available on GitHub[5]. This will not only allow for community-driven refinement of the ontology through issue tracking, but will also serve as a place to publish feature extensions as new use cases are added.

The introduced spa ontology allows us to describe interaction preconditions and effects of the intelligent power supply and the automatic door controller, as can be seen in listings 3 and 4.

Listing 3: Excerpt of an intelligent power supply TD showing the precondtion and effect annotations using the spa ontology.

```
"title": "PowerSupply",
...
"actions": {
  "changeVoltageByValue": {
    "input": {"@id": "changeVoltageInput", "type": "integer", "minimum": -10, "maximum":
        10},
    "forms": [{ href: "http://ex.org/changeValue" }],
    "effect": [ {
      "assign": {
        "numeric-add": [ { "@id": `outputVoltage` }, { "@id": "changeVoltageInput" } ] },
        "to": { "@id": "outputVoltage" } } ] }
}
```

5. From Thing Description Annotations to PDDL

A PDDL planning problem consists of a domain description and a problem description [8]. In the following subsections we show how the annotations in TDs and the reading of properties to get the initial state of the environment can be used to generate both descriptions.

5.1. Domain Description Generation

A PDDL domain description provides generic information about all available object types, predicates, functions, and PDDL actions with preconditions and effects in a particular domain.

Mapping Object Types The first step in the mapping process is to identify object types. Object types defined in a PDDL domain description are types that can be used for objects in a PDDL problem statement. When transforming the annotations of a TD into a domain description, we create a `Thing` type for each discovered Thing. The `Thing` type allows the assignment of specific predicates, functions, and PDDL actions to a particular Thing, which is necessary in multi-Thing planning domains.

[4] https://paul.ti.rw.fau.de/~jo00defe/voc/spa#
[5] https://github.com/wintechis/SimplePlanningAnnotation

Listing 4: Excerpt of an automatic door controller TD showing the precondtion and effect annotations using the spa ontology.

```
"title": "DoorController",
...
"actions": {
  "open": {
    "forms": [{ href: "http://ex.org/open" }],
    "precondition": {
      "and": [
        {"numeric-equal": [{"@id": "outputVoltage"}, 5]},
        {"boolean-equal": [{"@id": "doorOpenState"}, false]}
       ] },
    "effect": [{
      "assign": true,
      "to": { "@id": "doorOpenState" } }]},
...
}
```

Mapping Predicates and Functions Predicates represent boolean values, while functions represent numeric values - specifically, integers or floats - that describe properties of specific objects or the domain itself. Predicates are created for all boolean properties that are exposed by WoT read or write property interactions, as well as for all annotated hidden properties that are not exposed. In addition, a predicate used as flag with the suffix *Read* is created for all read properties, indicating whether or not a WoT read property interaction was performed on that predicate. Functions are created in a similar way for all numeric properties exposed by WoT read or write property interactions, and for all annotated hidden properties that are not exposed. All generated predicates and functions are associated with the `Thing` type of the particular Thing.

Mapping Actions PDDL actions consist of *parameters*, which can be predicates, functions, and objects on which the PDDL action is performed. They have *preconditions*, which are logical expressions composed of predicates and functions that must be satisfied for the PDDL action to be performed. In addition, an *effect* describes the changes to predicates and functions after the PDDL action has been successfully executed. WoT read properties are translated into PDDL actions using the precondition annotated in TDs, and have the effect of setting the associated predicate indicating a read to true. WoT write properties and WoT actions are translated into PDDL actions in one of two ways, depending on the data type. If the data type is boolean, two actions are created with the precondition as annotated in the TD and with the effect of setting the associated predicate to either true or false. If the data type is a numeric value, for each available input value and annotated step size one PDDL action is created with the precondition as annotated in the TD and the effect of setting the numeric value function representing the WoT property to the input value.

Domain Description of Running Example The first step in applying the mapping to the running example is to generate two object types: a `Thing0` type representing the automatic

door controller, and a `Thing1` type representing the intelligent power supply. The second step is to generate predicates and functions. For the automatic door controller, we generate two predicates: one indicating if the `doorOpenState` has been read, and one representing the `doorOpenState`. For the intelligent power supply, we generate a predicate indicating if the `outputVoltage` has been read, and a function representing the current `outputVoltage` value. Finally, we generate the PDDL actions. For the automatic door controller, we generate PDDL actions to open and close the automatic door, and to read the `doorOpenState`. For the intelligent power supply, there are PDDL actions to change the current `outputVoltage` for each possible value of `changeVoltageInput`, and to read the `outputVoltage`. An excerpt of the generated domain description can be seen in listing 5.

Listing 5: Excerpt of the PDDL domain description based on the running example.

```
...
  (:types Thing0 Thing1)

  (:predicates
    (doorOpenState_Read ?thing0 - Thing0)
    (doorOpenState ?thing0 - Thing0)
    (outputVoltage_Read ?thing1 - Thing1) )

  (:functions
    (outputVoltage ?thing1 - Thing1) )

  (:action invokeAction_open
    :parameters( ?thing1 - Thing1 ?thing0 - Thing0)
    :precondition(and (= (outputVoltage ?thing1) 5) (not(doorOpenState ?thing0)))
    :effect(doorOpenState ?thing0) )
...
```

5.2. Problem Description Generation

A PDDL problem description contains the concrete objects, the initial state and values of PDDL predicates and functions, and a goal definition.

Mapping Objects Objects represent concrete elements that exist in the problem description and are associated with object types defined in the domain. For each Thing involved, an object with the name defined by the title annotated in the TD is created and associated with the corresponding `Thing` type.

Mapping Initial State and Values The initial state of predicates and the values of functions can be determined by reading the exposed WoT properties of the associated Things. If properties cannot be read directly, they must either be observed by another device, such as a sensor, or be predefined. We assume that WoT properties are static and do not change after being read.

Mapping the Goal The goal in a PDDL problem description is a logical expression, consisting of predicates and functions, that specify the desired goal state. The planner's task is to determine

a sequence of actions that will transition the system from its initial state to a state that satisfies the goal expression, indicating that a solution to the planning problem has been found. We assume that the goal expression is predefined.

Problem Description of Running Example In the PDDL problem description, the objects must first be instantiated: one object for the `DoorController` with type `Thing0`, and one for the `PowerSupply` with type `Thing1`. Next we need to set the initial states. The initial values are obtained by invoking WoT readProperty interactions, which results in `doorOpenState` being false and not read, and `outputVoltage` being `0` and not read. An excerpt of the generated problem description is depicted in listing 6.

Listing 6: Excerpt of the PDDL problem description based on the running example.

```
...
  (:objects
    DoorController - Thing0
    PowerSupply - Thing1 )

  (:init
    (not (doorOpenState DoorController))
    (not (doorOpenState_Read DoorController))
    (not (outputVoltage_Read PowerSupply))
    (= (outputVoltage PowerSupply) 0) )

  (:goal (doorOpenState DoorController))
...
```

6. Evaluation

We have created a prototype implementation of a Thing Description to PDDL (TD2P) processor[6] using the mappings introduced in section 5 in JavaScript. The implementation is based on the W3C WoT Scripting API reference implementation node-wot[7]. node-wot handles the consumption of TDs and the interaction with Things via the WoT abstraction. Additionally the TD2P processor uses a link following algorithm [9] to discover related TDs starting from an initial TD, allowing the creation of complex planning problems involving multiple Things. The TD2P processor consumes TDs and produces PDDL domain and problem descriptions as output. To solve the generated planning problem, a PDDL 2.1 compliant planner supporting fluents is required. The TD2P processor currently supports the ENHSP-19 planner [19], which means that the algorithm is able to parse the output of the planner and generate a WoT program. Our approach supports the mapping of boolean, integer, and number schemas used by the WoT architecture, but currently does not cover arrays, objects, or strings. Moreover, the focus is on

[6] https://github.com/FreuMi/TD2P
[7] https://github.com/eclipse-thingweb/node-wot

the basic WoT interactions for reading or writing properties and invoking actions, while more complex interactions such as events are not covered.

To validate our system, we implemented a simulation that covers the running example with a automatic door controller and an intelligent power supply. The source code of the simulation including a short demo video[8] can be found as an example in the GitHub repository of the TD2P processor. The simulation demonstrates that our approach allows the automatic generation of PDDL domain and problem description files usable in standard PDDL planners.

To evaluate the scalability of the system as the number of PDDL actions, and thus the search space, increases, we systematically expanded the number of WoT actions of the simulated devices in the running example. During the evaluation, other parameters such as the goal or the number of devices remained unchanged. We evaluated the TD2P processors runtime and memory consumption to create the PDDL domain and problem descriptions, and the total runtime and memory consumption to solve the planning problem.

The tests were conducted on consumer hardware equipped with an i5-10500H CPU and 16 GB of RAM running Ubuntu 22.04 LTS. Total runtime and peak memory usage were measured separately using the `time` command in Linux. To solve the planning problem we used the ENHSP-19 planner. All measurements were repeated five times.

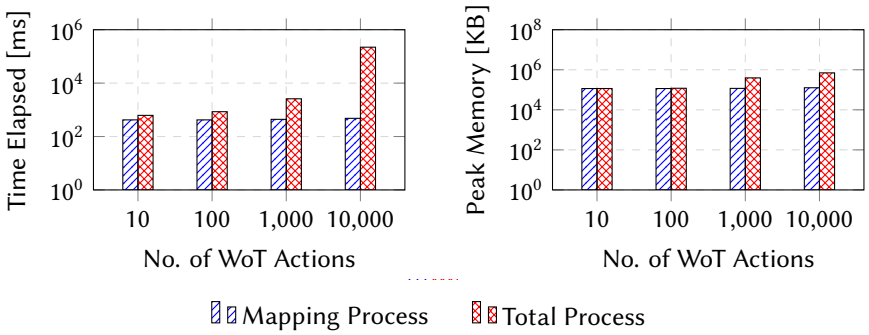

Figure 3: Average elapsed time (left) and average peak memory usage (right) for the TD2P processor (blue) and the total process (red) across five runs. Note that the vertical axes of the diagram are scaled logarithmically.

The resulting diagrams in Figure 3 show that for up to 100 WoT actions, the TD2P processor accounts for the majority of total processing time (between 51% and 67%) and total memory usage (between 96% and 99%). But beyond 1,000 WoT actions, the PDDL planner consumes most resources, and the share of the TD2P processor accounts for a minority of the total processing time (between 2% and 16%) and total memory consumption (between 18% and 29%).

As the number of WoT actions increases from 10 to 10,000 (an increase by a factor of 1,000, or about 99,900%), the runtime for the TD2P processor increases from 420 ms to 480 ms (an approximate increase of 14.3%), and the memory consumption increases from 115,532 KB to 126,400 KB (an approximate increase of 9.4%), indicating vertical scalability for the TD2P processor.

[8] https://github.com/FreuMi/TD2P/tree/main?tab=readme-ov-file#demo

7. Conclusion and Future Work

In this paper, we introduced the spa ontology that is used to annotate preconditions and effects of different WoT interactions in a TD. Additionally, we introduced a mapping between the annotations in TDs to PDDL domain and problem definitions, allowing robotic agents that have discovered a Thing to use established AI planning algorithms to dynamically evaluate interactions to achieve a given goal. As next steps, we plan to release the spa ontology as a full OWL ontology with OWL-restricted resources, focus on the correctness of manually scripted WoT programs, and extend the TD2P processor to cover more complex WoT interactions.

Acknowledgments

This work was funded by the Bayerisches Verbundforschungsprogramm (BayVFP) des Freistaates Bayern through the KIWI project (grant no. DIK0318/03).

References

[1] D. Leidner, W. Bejjani, A. Albu-Schäffer, M. Beetz, Robotic agents representing, reasoning, and executing wiping tasks for daily household chores, Autonomous Agents and Multi-Agent Systems (2016).

[2] B. Pradhan, D. Bharti, S. Chakravarty, et al., Internet of things and robotics in transforming current-day healthcare services, Journal of healthcare engineering 2021 (2021) 1–15.

[3] A. Roy Chowdhury, IoT and Robotics: a Synergy, PeerJ Preprints 5 (2017).

[4] M. Lagally, R. Matsukura, M. McCool, K. Toumura, K. Kajimoto, T. Kawaguchi, M. Kovatsch, Web of Things (WoT) Architecture 1.1, https://www.w3.org/TR/wot-architecture/, 2023. Accessed: 2024-01-31.

[5] L. A. Grieco, A. Rizzo, S. Colucci, et al., IoT-aided robotics applications: Technological implications, target domains and open issues, Computer Communications 54 (2014) 32–47.

[6] S. Kaebisch, M. McCool, E. Korkan, T. Kamiya, V. Charpenay, M. Kovatsch, Web of Things (WoT) Thing Description 1.1, https://www.w3.org/TR/wot-thing-description/, 2023. Accessed: 2024-01-31.

[7] A. Olivares-Alarcos, D. Beßler, A. Khamis, et al., A review and comparison of ontology-based approaches to robot autonomy, The Knowledge Engineering Review 34 (2019).

[8] C. Aeronautiques, A. Howe, C. Knoblock, et al., PDDL - The Planning Domain Definition Language, Technical Report, Tech. Rep. (1998).

[9] M. Freund, J. Fries, T. Wehr, A. Harth, Generating visual programming blocks based on semantics in W3C thing descriptions, in: Proceedings of the First International Workshop on Semantic Web on Constrained Things co-located with 20th Extended Semantic Web Conference, SWoCoT@ESWC 2023, Hersonissos, Greece, May 28, 2023, volume 3412 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023, pp. 1–15.

[10] E. Korkan, S. Kaebisch, M. Kovatsch, S. Steinhorst, Sequential behavioral modeling for scalable iot devices and systems, in: 2018 Forum on Specification & Design Languages (FDL), IEEE, 2018, pp. 5–16.

[11] Y.-q. Jiang, S.-q. Zhang, P. Khandelwal, et al., Task planning in robotics: an empirical comparison of pddl-and asp-based systems, Frontiers of Information Technology & Electronic Engineering 20 (2019) 363–373.
[12] M. Fox, D. Long, PDDL2.1: An extension to PDDL for expressing temporal planning domains, Journal of artificial intelligence research 20 (2003) 61–124.
[13] M. Noura, S. Heil, M. Gaedke, Growth: Goal-oriented end user development for web of things devices, in: Web Engineering: 18th International Conference, ICWE 2018, Cáceres, Spain, June 5-8, 2018, Proceedings 18, Springer, 2018, pp. 358–365.
[14] M. Poveda-Villalón, A. Fernández-Izquierdo, M. Fernández-López, R. García-Castro, LOT: An industrial oriented ontology engineering framework, Engineering Applications of Artificial Intelligence 111 (2022) 104755.
[15] N. Noy, Ontology development 101: A guide to creating your first ontology, 2001. URL: https://api.semanticscholar.org/CorpusID:500106, accessed: 2024-01-31.
[16] A. Gomez-Perez, M. C. Suárez-Figueroa, Neon methodology for building ontology networks: a scenario-based methodology (2009).
[17] M. Poveda-Villalón, A. Gómez-Pérez, M. C. Suárez-Figueroa, Oops!(ontology pitfall scanner!): An on-line tool for ontology evaluation, International Journal on Semantic Web and Information Systems (IJSWIS) 10 (2014) 7–34.
[18] D. Garijo, WIDOCO: A Wizard for Documenting Ontologies, in: International Semantic Web Conference, Springer, Cham, 2017, pp. 94–102.
[19] E. Scala, P. Haslum, S. Thiébaux, et al., Interval-based relaxation for general numeric planning, in: ECAI 2016, IOS Press, 2016, pp. 655–663.