Generating Visual Programming Blocks based on Semantics in W3C Thing Descriptions

Michael Freund[1,*], Justus Fries[1], Thomas Wehr[1] and Andreas Harth[1,2]

[1]Fraunhofer Institute for Integrated Circuits IIS, Nürnberg, Germany
[2]Friedrich-Alexander-Universität Erlangen-Nürnberg, Nürnberg, Germany

Abstract

We present mappings that leverage the semantic information in W3C Thing Descriptions (TDs) to generate structure definitions and code generator functions for visual programming blocks. In addition, we use link following to discover and consume related TDs. This approach extends device support in low-code IoT environments. Our implementation generates blocks and code generators in $O(n)$, where n is the number of interaction affordances in a consumed TD, and follows links to Thing Descriptions in $O(n + m)$, where n is the number of TDs and m the number of links to follow. Specifically, our implementation can discover 35 TDs with 128 interaction affordances and generate blocks and code in less than 200 ms, which is considered acceptable for interactive user interfaces.

Keywords

Web of Things, Block Generation, Low Code

1. Introduction

Constrained devices are used in industrial and consumer applications to gather information about the environment and act on the physical world as needed [1, 2]. The World Wide Web Consortium's (W3C) Web of Things (WoT) [3] specification attempts to simplify the interaction with constrained devices by using a semantic description of the device's metadata and available interaction affordances, which are organized into three categories *properties*, *actions*, and *events*. The W3C recommendations for the WoT typically refer to a constrained device as a Thing, and to a semantic interface description as a Thing Description (TD) [4]. On the one hand, experts and programmers can use libraries that implement the WoT Scripting API [5], such as `node-wot`[1] for the JavaScript programming language, to interact with Things. On the other hand, simple graphical tools are needed to empower everyday users without programming expertise to interact with WoT devices. Towards this goal, we developed and introduced an easy-to-use tool called BLAST [6] for browser-based graphical program creation and execution targeting WoT environments.

SWoCoT'23: 1st International Workshop on Semantic Web on Constrained Things, 28th May 2023, Hersonissos, Greece, co-located with 20th Extended Semantic Web Conference (ESWC 2023)
*Corresponding author.
michael.freund@iis.fraunhofer.de (M. Freund)
00000-0003-1601-9331 (M. Freund); 0000-0003-3433-7245 (J. Fries); 0000-0002-0678-5019 (T. Wehr); 0000-0002-0702-510X (A. Harth)

[1]https://github.com/eclipse/thingweb.node-wot

BLAST and other block-based visual programming environments (VPEs) have the limitation that the available graphical programming elements require a structural block definition that describes the layout of the block, and an associated source code generator function that generates predefined code when a block is dragged into the graphical workspace. In a WoT context, this means that all available interaction affordances of a Thing require a separate block and source code definition. Currently, all blocks and code generators must be designed and implemented by hand, even if the interface of a Thing is already described by a semantic Thing Description. The high manual design effort for blocks significantly limits the number of supported devices in VPEs.

Since Thing Descriptions are implemented with machine readability in mind, we propose an algorithm that maps property keywords of a TD to programming blocks and corresponding source code generator functions, minimizing the manual implementation effort required to integrate various graphical programming elements. Such an algorithm can improve the flexibility of VPEs and allow users to interact with arbitrary constrained devices that are described by a TD.

The main challenges in automating block generation include defining mappings between semantics in TDs and block/code generators, processing links to related TDs, and handling variations in the semantic richness of TDs.

In this work, we present a prototypical implementation of such an algorithm. To design the algorithm, we analyzed the structure of a Thing Description, created a mapping of mandatory and optional property keywords to block structures and code generators, and included functionality to recursively follow links to consume and generate blocks for multiple related and directly linked Thing Descriptions. We also implement and evaluate the algorithm as a proof-of-concept. The contributions of our work are

- the definition of mappings between the semantic information in Thing Descriptions and block structures with code generators,
- the implementation of a block and code generation algorithm with device discovery using link following,
- and an analysis of the performance of our proposed algorithms.

2. Related Work

Block generation for IoT devices and hardware has already been explored by Culic *et al.* [7]. The algorithm presented uses Intel's libmraa C/C++ library and generates blocks by parsing the comments used to describe functions. The generated block structure definitions and code generator functions are stored in a centralized database. In contrast to this approach, we focus on the W3C Thing Description, which is already a well-structured semantic abstraction layer to the API of an IoT device. Our approach also excludes the creation of a centralized database to discover new devices and download previously created block and code definitions. Rather, users can consume TDs to create blocks and code generators locally, and discover new devices using link following.

Wang *et al.* present a block editor called HolonCraft for smart home environments [8]. HolonCraft uses a semantics-based block and code generation algorithm built on the Holon

Ontology. Users of HolonCraft can create a Holon-compliant document of a device and upload it to a server where the block definitions and code generator functions are created. Both, the generated blocks and the corresponding code are sent back to the client, where users can start building programs. Contrary to this approach, we build on and reuse the W3C Thing Description Ontology, create block and code generators directly in a web application without the need for a server, and use link following to discover related TDs.

In Node-RED, a package[2] can be used to enable support for WoT devices by consuming a TD, dragging a predefined interaction node into the workspace, and manually selecting the name of the desired interaction affordance. Device discovery is not supported. Compared to this approach, we generate blocks for all available interaction affordances and users can select the block that performs the desired operation and discover devices by following links.

3. From a Thing Description to Blocks and Code

A Thing Description is a document in JSON-LD [9] format, a serialization of RDF. A JSON-LD document uses property keywords that map to ontology vocabulary terms via a context. The context is identified by a URI and specified via the *@context* property keyword. A TD uses property keywords defined in the Thing Description Ontology[3] (td), the Hypermedia Controls Ontology[4] (hctl), and the Dublin Core Metadata Schema[5] (dct), for instance, *title* maps to td:title, and *op* maps to hctl:hasOperationType. As long as the context is known, a property keyword can be easily mapped to a property of an ontology. In the following parts of this paper we will use the property keywords of TDs (e.g. title) defined in the WoT JSON-LD context[6], instead of the ontology properties (e.g. td:title), to be consistent with the examples presented in the WoT recommendations.

To develop an algorithm that generates blocks and code generator functions based on the semantic information in a TD, we must first define mappings. These mappings focus on the property keywords used in the TD information model, which includes both mandatory and optional property keywords. The optional elements introduce uncertainty into the level of semantic detail present in a consumed TD. As a result, the layout and features of generated blocks will vary depending on the information available. Examples of the aforementioned TD property keywords include the interaction affordances of a device, a human-readable title, and security definitions.

The generated blocks and the resulting program structure in a VPE should follow the structure and interaction abstraction level used in the WoT Scripting API as closely as possible, to ease the transition to text-based implementations for users already familiar with the graphical environment [10]. The general abstraction structure for interacting with a Thing using the Scripting API consists of two distinct phases. In the first phase, the creation phase, a TD is consumed and a Thing object is created based on the information it contains. In the second phase, the interaction phase, the Thing object can be used to interact with the Thing by calling functions

[2] https://flows.nodered.org/node/node-red-contrib-web-of-things
[3] https://www.w3.org/2019/wot/td
[4] https://www.w3.org/2019/wot/hypermedia
[5] http://purl.org/dc/terms/
[6] https://www.w3.org/2022/wot/td/v1.1

Table 1
Mapping of Thing Vocabulary to Thing Blocks.

Block Property	TD Property Keyword	Additional Notes
Block Name	titles or title	If available
Block Color	-	Flexible to choose
Block Output	-	Output type 'thing'
Tooltip	description(s), version, modified, created	If available
Help URI	support	If available

created from its described interaction affordances. The relevant functions for basic interaction with a Thing are **readProperty**, **writeProperty**, **invokeAction**, and **subscribeEvent**. More advanced interaction functions[7] are not covered by this work, but can be mapped in a similar way.

For instance, to read a property affordance of a consumed Thing using the JavaScript implementation of the Scripting API, the following snippet of source code can be used.

```
thing.readProperty('status');
```

Where `thing` is the Thing object, `readProperty` is the interaction function, and `status` is the semantic name of the property affordance to read. From the WoT Scripting API structure, we can deduce that a consumed TD should result in a Thing object block to specify the Thing, and various interaction blocks corresponding to available affordances.

In the following subsections, we analyze the mandatory and optional vocabulary terms and create mappings for blocks and code generators based on the different parts of a TD, namely the Thing-specific part needed in the Thing object block in section. 3.1, the three interaction affordances needed in the different interaction blocks in sections 3.2, 3.3 and 3.4, and the link part used for link following in section 3.5. A TD of a lamp is depicted in Listing 1, which is used as a running example throughout the paper.

3.1. Mapping of the Thing specific Vocabulary

The Thing's metadata in a TD contains four mandatory property keywords, these are *@context*, *title*, *security*, and *securityDefinitions*. All other property keywords of TDs are optional, which means that the contained information can only be used in the block generator if a value is provided or a default value is defined. A sample of a TD containing the relevant Thing-specific property keywords is shown in Listing 1 from line 1 to line 16.

We use the information contained in the Thing-specific part of a TD to create a block representing the consumed Thing object, to graphically specify the Thing to interact with. An example of a generated Thing block with tooltip based on the introduced lamp TD is shown in Fig. 1. The mapping between the information in a Thing block and the Thing-specific part of a TD is shown in Table 1.

[7] https://www.w3.org/TR/wot-thing-description11/#form

```
{
 "@context": "https://www.w3.org/2022/wot/td/v1.1",
 "@type": "Thing",
 "id": "urn:dev:ops:32473-WoT-Thing-1234",
 "title": "LampThing",
 "titles": { "en": "LampThing", "de": "LampenDing",},
 "description": "A lamp",
 "descriptions": { "en": "A lamp", "de": "Eine Lampe"},
 "version": "1.0",
 "created": "2020-10-10T17:00:00Z",
 "modified": "2022-10-10T17:00:00Z",
 "support": "https://example.org/lamp",
 "links": [{"href": "http://example.com/related-td", "type": "application/td+json"}]
 "securityDefinitions": {
        "basic_sc": {"scheme": "basic", "in": "header"} },
 "security": ["basic_sc"],
 "properties": {
   "status": {
     "title": "status",
     "titles": {"en": "status", "de": "Zustand"},
     "description": "Read the status of the lamp",
     "descriptions": {"en": "Read the status of the thing",  "de": "Auslesen des Lampenzustands"},
     "type": "string",
     "forms": [...]
   } },
 "actions": {
   "toggle": {
     "title": "toggle",
     "titles": {"en": "toggle", "de": "umschalten"},
     "description": "Toggle current lamp status",
     "descriptions": {"en": "Toggle current lamp status", "de": "Umschalten des aktuellen
         Lampenstatus"},
     "output": {"type": "string"},
     "forms": [...],
    } },
 "events": {
   "overheating": {
     "title": "overheating",
     "titles": {"en": "overheating", "de": "Ueberhitzung"},
     "description": "An overheating event of the lamp",
     "descriptions": {"en": "An overheating event of the lamp", "de": "Ein Ueberhitzungs Event der
         Lampe"},
     "data": {"type": "string"},
     "forms": [...],
   } },
}
```

Listing 1: Sample Thing Description of a lamp with metadata and three interaction affordances.

(a) Thing Block with en-US locale

(b) Thing Block with de-DE locale

Figure 1: Generated Thing block and tooltip based on Thing Description of Listing 1. Note the user chosen name, 'lamp' in English and 'Lampe' in German.

The *title* and *description* entries can be available in different languages. If this optional information is provided, the language of the Thing block name (e.g., LampThing) and the tooltip can be adapted to the language of the users (e.g., LampenDing). Further optional information provided by *version*, *created*, and *modified* is also added to the tooltip of the generated block. The help URI of the Thing block is set to the URI provided by the *support* field if contained in the consumed TD, otherwise no help URI is set. The color of the block cannot be derived from the TD, but must be chosen manually to match the overall theme of the VPE. The output of the block is defined as type *thing*, which allows type checking in combination with other blocks. The generated Thing block must also be associated with the Thing object specified and generated by the WoT Scripting API to allow interaction with the Thing. The source code generated by the Thing block returns the blockId, which is used to identify a specific block and its associated Thing. The *@context* and *id* entries are currently not reflected in the generated Thing block.

In addition to the Things metadata, security definitions are also provided in the Thing specific part. Based on the mandatory entries for *security* and *securityDefinitions*, we can infer the need for an additional block to set up the security configuration, e.g., to enter a username and password in case of basic HTTP authentication. The layout of the generated security block depends on the security definitions, such as *NoSecurityScheme*, where no block needs to be generated, or *BasicSecurityScheme*, which generates a block with three inputs for a username, its password, and for the Thing that requires the security information.

3.2. Mapping of Property Affordance Vocabulary

The information provided in the property affordance specific part (see Listing 1, lines 17 to 25) is used to generate property affordance blocks. Property affordances are a subclass of the interaction affordance class and come in two operation types *readProperty* and *writeProperty*. Each of the two property affordance interactions requires a different block structure and code generator, because a *readProperty* operation has an output and no input, whereas a *writeProperty* operation has an input and no output.

In the property affordance section of a TD, only the *forms* field is mandatory, which limits the information that is always available for block creation. However, as described above, TDs have default values defined. These are the property keywords *readOnly*, *writeOnly*, *contentType*, and *op* indicating the operation. The information contained in the default values can also be used for block generation if no other data is provided. In order to create a meaningful and

Table 2
Mapping of Property Affordance Vocabulary to Property Blocks.

Block Property	TD Property Keyword	Additional Notes
Block Name	titles, title, property affordance name, op	If available
Block Color	op	Dependent on op
Block Output	op, type	Only if op is read
Block Input	op, type, enum	Only if op is write
Tooltip	description(s), default	If available

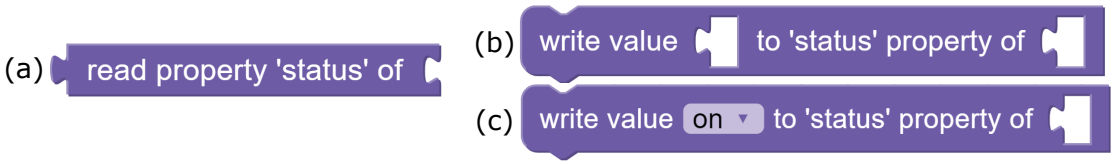

Figure 2: Layout of possible property affordance blocks, starting with read property affordance (a), write property affordance (b), and write property affordance with *enum* (c). The running example would generate (a).

easy-to-use block, we have also defined additional default values for some block properties in case the optional information is not available. The defined mappings are shown in Table 2.

The name of a block must indicate its capabilities. To achieve a clear and unique name when generating a block, a combination of the performed operation (e.g., *readProperty*) and the title or name of the target property affordance can be used. This approach results in block names, such as ”read property ’status’ of” or ”write value {X} to property ’status’ of”.

The color of a block plays an important role in VPEs, indicating the block category to users [11]. The color of a generated block therefore depends on the interaction affordance and should match the theme of already existing blocks in the same category.

The layout, inputs, and outputs of a generated property affordance block depend on the operation specified by *op*, and, if available, on the *enum* property keyword, which specifies an array of allowed values.

The *readProperty* block consists of an output for the read data, an input for a Thing block, and the source code to read a property affordance and return the decoded value. The *writeProperty* block has two possible layouts, depending on the *enum* property keyword of the consumed TD. The first variant, without *enum*, has no output, but an input for a value to send and an input for a Thing block. The second variant, with *enum*, has the same layout except that the value input which is replaced by a drop-down menu containing all the allowed values specified by the enum array. The source code for both *writeProperty* blocks performs a write operation on the specified property affordance. The three resulting blocks are shown in the following Fig. 2.

The *type* property keyword in a TD contains important datatype information about the input or output data, which is used for type checking. Type-checking restricts the connection of the generated block to other existing blocks. Blocks can only be connected to each other if the output datatype of one block matches the input datatype of another block. Since the *type*

Table 3
Mapping of Action Affordance Vocabulary to Action Blocks.

Block Property	TD Property Keyword	Additional Notes
Block Name	titles, title, action affordance name, op	If available
Block Color	op	Dependent on op
Block Output	output, type	If output provided
Block Input	input, type	If input provided
Tooltip	description(s), default	If available

property keyword is optional, not all generated blocks have type checking.

The tooltip of the generated property block is based on either the optional localized description, the general description, or on a default sentence. The default tooltip sentence is generated from the two mandatory property keywords *op*, *deviceName* and *propertyName*, resulting in the template sentence "{part of op} the {propertyName} property of {deviceName}" which, for instance, can lead to "read the status property of LampThing" followed by a supported Thing block.

3.3. Mapping of Action Affordance Vocabulary

Action blocks are generated from the action affordance section of a TD (see Listing 1, lines 26-34). An action affordance is a subclass of the interaction affordance class, with only the mandatory *forms* property keyword, but in combination with the default values listed in section 3.2, meaningful blocks are generated.

The mappings between the information contained in a TD and the generated blocks are similar to the mappings for property affordances. The main difference is the structure of the generated blocks. An *invokeAction* block can generally have four different layouts, since actions can only have one input, only one output, both, or neither. Each of the four possible block variants comes with different source code generator functions. For instance, if an action only has one output and no input, the corresponding source code will invoke the action and decode the received value. The four possible block structures are shown in Fig. 3 and the overall action affordance mapping is displayed in Table 3.

The name of the four action block variations is generated from the title or the name of the action, resulting either in "invoke {actionName} action of" for the no input and no output case, and the one output and no input case, or in "invoke {actionName} action with value {X} of" for the one input and no output case, and the one input and one output case.

The color of the generated action block must be set to the color of the action category already in use and the tooltip of the block is either the description in the browser language, the default description, or the default phrase "Invoke {actionName} action of {deviceName}" resulting, for instance, in "Invoke toggle action of LampThing", which wil be used if no other optional information is provided. All inputs and outputs of the generated action blocks are typed if possible, depending on the information available in the consumed TD.

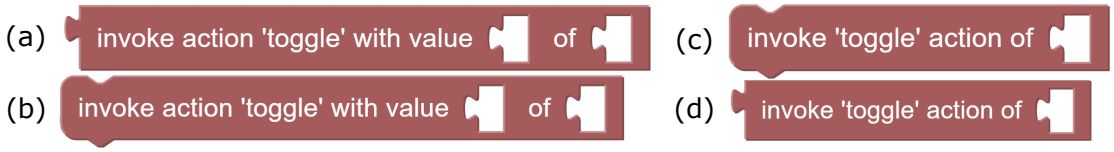

Figure 3: Layout of generated action affordance blocks, starting with a block with an input and an output (a), a block with an input and no output (b), a block with no input and no output (c), and a block with no input but an output (d). The running example would generate (d).

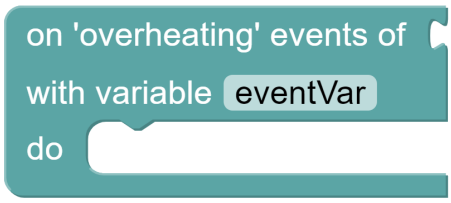

Figure 4: Layout of the generated event affordance block with statement input, using the running TD example.

Table 4
Mapping of Event Affordance Vocabulary to Event Blocks.

Block Property	TD Property Keyword	Additional Notes
Block Name	titles, title, event affordance name, op	If available
Block Color	op	Dependent on op
Tooltip	descrioption(s), default	If available

3.4. Mapping of Event Affordance Vocabulary

Event affordances are the last of the three subclasses of the interaction affordance class. As a result of this subclass relationship, only the *forms* property keyword is mandatory and the same default values as for property and action affordances are available. An example of the event affordance part of a TD can be seen in Listing 1 from lines 37 to 45.

The structure of generated event blocks differs from the structure of the other introduced blocks since event blocks consist of statement inputs instead of the value inputs used by property and action blocks. A statement input allows users to add programming blocks within an event block, as shown in Fig. 4. When a subscribed event is triggered, the received event data is passed to the event block as a variable, and the enclosed blocks are executed. The information provided by the optional *data* property keyword can be used to add a type to the received event data. The mappings for event affordances are shown in table 4.

The name and color of the event block are generated, as with the property and event affordances, using the title in the browser language, the default title if available, or simply the name of the event. The block color should match already existing event blocks.

3.5. Link Following Vocabulary

A fundamental aspect of Web browsing is the concept of link following in hypertext, which enables users or a user agent to find and explore related Web resources by dereferencing URIs. The same concept can be applied to the Web of Things by using the *links* property keyword in a TD to link to related TDs using standard Web technologies. By allowing the generator algorithm to follow available links in a consumed TD, users can consume multiple devices at once, such as a light bulb TD and the corresponding light switch TD or all the TDs of WoT Things contained in a given room. Direct links from a Thing Description to other TDs can be achieved by using the *links* property keyword and specifying a link target via the mandatory *href* property keyword. Additional semantic information can be provided by using the two optional property keywords *type* and *rel* to define the content type of the target and the link relation type.

A crawler algorithm can use the optional *type* property keyword to check if the link target is a TD, or if no optional semantic information is provided by the TD use a HTTP HEAD request to check the content type of the link target.

```
function addConsumedDevice(deviceName,td):
 generateThingBlock(deviceName, td.description, td);
 generateThingCode(deviceName);
 genSecurityBlock(deviceName,td);
 genSecurityCode(deviceName,td):

 for [propertyName, operation] of td.properties:
   if operation == 'readproperty':
     genReadPropBlock(propertyName,deviceName);
     genReadPropCode(propertyName,deviceName);
   if operation == 'writeproperty':
     genWritePropBlock(propertyName,deviceName,td);
     genWritePropCode(propertyName,deviceName,td);

function genReadPropBlock(propertyName,deviceName,td):
 langTag = getLanguage();
 blockName = generateBlockName(td,langTag);
 new Block[${deviceName}_readPropBlock_${propertyName}] = {
   ValueInput('thing').setCheck('Thing').append(`read property "${propertyName}" of`);
   setOutput(true, td.properties.type || null);
   setColour(255);
   setTooltip(generateToolTip()); }

function genReadPropCode(propertyName,deviceName):
 new Code[${deviceName}_readPropBlock_${propertyName}] = function(block){
    blockId = getInputBlock(block);
    thing = getBlockById(blockId).thing;
    return "await (await thing.readProperty(propertyName)).value()"; }
```

Listing 2: Pseudocode of generator algorithm for ThingBlock and property affordances.

4. Mapping Algorithm

To demonstrate the theoretical approaches presented in the previous section, we implemented a proof-of-concept using JavaScript and the defined mappings to generate blocks and source code based on discovered and consumed TDs. The structure of the generated blocks is in the JavaScript block format of Google's Blockly[8] library, and the generated code is based on node-wot. We implemented the algorithm as a standalone application for benchmarking purposes.

The pseudocode in Listing 2 outlines the algorithm for generating ThingBlocks and corresponding readProperty affordances. It begins with the *addConsumedDevice* function (Lines 1-13), which parses the input Thing Description to identify generatable blocks and code, and calls the appropriate generator functions for the ThingBlock, the security block, and the various affordance blocks. The *genReadPropBlock* algorithm from Lines 15–22 contains the function for creating the block structure of readProperty affordances, and the *genReadPropCode* function (Lines 24 - 28) contains the source code generator function for the blocks.

The algorithm also consists of a simple crawler that relies on the focused crawling technique [12] to search only for TD documents. The crawler searches within an initial Thing Description for links to other TDs, either by finding links with the specified content type `application/td+json` or by following links without specified content type using an HTTP HEAD request to check if the response object is a TD. The crawler algorithm then follows the found links, consumes the TDs at the target address and searches recursively for more links within the newly consumed TDs. Users can disable link following or limit the search depth when providing an initial TD URI to the generator algorithm.

While the current implementation of the algorithm is fully functional, it has several limitations. These include the missing ability to load and save created block definitions, no support for protocols other than HTTP, and limited crawler functionality. In particular, only links described by the *links* property keyword of the TD Ontology can be followed.

5. Performance Evaluation

In this section we present an evaluation of the implemented algorithm. First, we analyze the performance of the block and code generator, followed by an analysis of the TD crawler. Finally, we evaluate the performance of both algorithms in combination.

All empirical timing measurements were performed on consumer hardware (i7-10610U, 16 GB RAM) running Windows 10 21H2 using node.js and the command `performance.now()`[9] which provides millisecond time resolution.

5.1. Performance of Block and Code Generator

To determine the relationship between the size of the input TD and the performance of the generator algorithm, we analyzed the time complexity by focusing on the code responsible for

[8] https://developers.google.com/blockly/guides/create-custom-blocks/define-blocks
[9] https://nodejs.org/api/perf_hooks.html#performancenow

generating the property affordance blocks introduced in Listing 2, since the steps required for actions and events are largely similar.

The algorithm is composed of three functions, the *genReadPropBlock* function, the *genReadPropCode* function, and the *addConsumedDevice* function. The latter function calls the two generator functions. The *genReadPropBlock* function sets the block name and creates the block object, resulting in a constant time complexity or $O(1)$. Similarly, the *genReadPropCode* function also has a constant time complexity or $O(1)$, as it does not iterate over any data structures, but rather performs a constant number of operations such as variable assignments. The *addConsumedDevice* function has a linear time complexity or $O(n)$, where n is the number of property affordances in the input TD since the function contains one `for` loop iterating over `td.properties`. All other operations contained in the *addConsumedDevice* function are executed in constant time. Overall, the time complexity of the shown algorithm can be seen as a combination of the time complexities of these three functions, resulting in $O(n \cdot (1 + 1)) = O(n)$. Since similar operations are performed for action and event affordances, the entire algorithm also has linear time complexity.

5.2. Performance of TD Crawler

To assess the performance of the TD crawler, we conducted a combined analysis of the algorithm structure and empirical measurements of the execution time while interacting with an increasing number of TDs.

The implemented crawler algorithm performs a breadth-first search (BFS) of Thing Descriptions, adding each newly discovered TD and its links to a queue and fetching all discovered links using asynchronous functionalities until no more unvisited links are found or until the maximum depth is reached. The execution time of the TD crawler based on BFS depends on the number of TDs n and links m in the searched link structure, and thus on the size and complexity of the TDs to be crawled. Therefore, the time complexity of the crawler algorithm is $O(n + m)$.

However, in a real-world application, the performance of the TD crawler depends on many factors, including network latency and bandwidth, and the use of techniques such as caching, parallelization, or asynchronous operations. Therefore, we also performed an empirical evaluation of the crawler using two types of TDs: TDs containing only one new link, forming a chain structure, and TDs containing two new links, forming a tree structure. The results are shown in Fig. 5.

5.3. Combined Performance of Both Algorithms

We use an express.js server to host Thing Descriptions and evaluate the performance of the combination of the two algorithms presented under real-world conditions. Our goal was to achieve a response time of no more than 200 ms, which is widely considered the maximum acceptable response time for interactive user interfaces [13]. To ensure controlled test conditions, we limited the algorithm to a maximum of 40 TDs, each containing only one link. To investigate the performance of the algorithm in different scenarios, we varied the number of interaction affordances in the discovered TDs and ran the experiment with 32, 128, and 512 affordances, respectively. The results of the experiments are shown in Fig. 6.

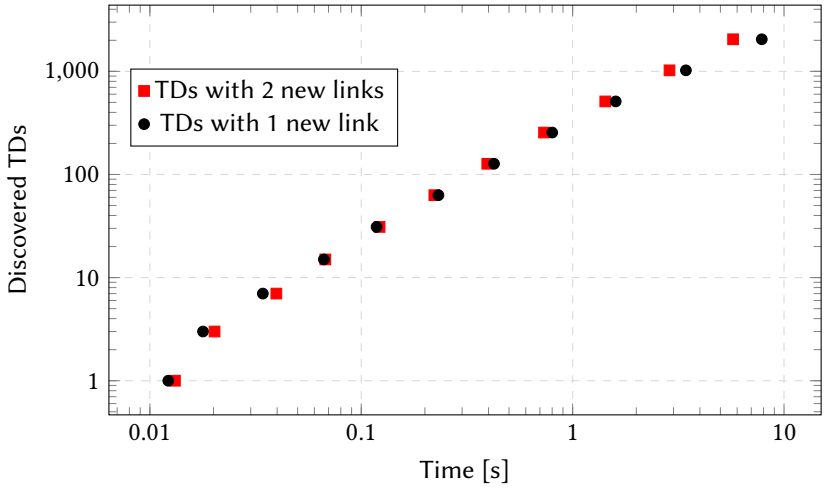

Figure 5: Average runtime of 10 runs in seconds, depending on discovered TDs.

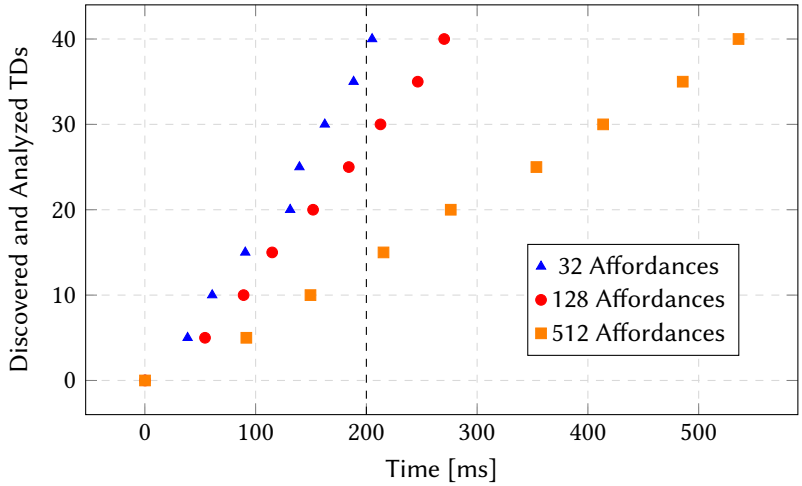

Figure 6: Average time to discover TDs and generate blocks and code from 10 runs in milliseconds with varying number of interaction affordances.

The algorithm performs well within the target response time of 200 ms, as users can use it, for example, to discover 35 Thing Descriptions with 128 interaction affordances in less than 200 ms. This is especially relevant considering that the average number of IoT devices in US households is 22, according to a 2022 market study by Deloitte[10]. Thus, our algorithm can be a solution for discovering and generating visual blocks and code in a typical smart home environment.

[10]https://www2.deloitte.com/us/en/insights/industry/telecommunications/connectivity-mobile-trends-survey.html

6. Conclusion and Future Work

To improve the extensibility of VPEs, we introduced mappings to generate various block structure definitions and code generator functions based on the semantics provided by W3C Thing Descriptions. We implemented a proof-of-concept algorithm based on JavaScript and the node-wot library. Through our evaluation of the algorithms, we found that the block and code generator has an upper bound of $O(n)$, where n is the number of interaction affordances in a TD, while the crawler has an upper bound of $O(n + m)$, where n is the number of TDs and m the number of links. When used in combination, the algorithms were able to discover and generate blocks and code for 35 TDs with 128 interaction affordances each in less than 200 ms, which is sufficient for a typical smart home environment and considered usable in interactive user interfaces. As a next step, we plan to further investigate the link-following concept in TDs and perform usability tests to evaluate the ease of use of our approach.

Acknowledgements

This work was funded by the the Bavarian State Ministry of Economic Affairs and Media, Energy and Technology through the AI-Nalyze project (grant no. DIK0134/01).

References

[1] C. Cimino, E. Negri, L. Fumagalli, Review of digital twin applications in manufacturing, Computers in Industry 113 (2019) 103130.

[2] A. Khanna, S. Kaur, Internet of things (iot), applications and challenges: a comprehensive review, Wireless Personal Communications 114 (2020) 1687–1762.

[3] 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: 2023-02-09.

[4] 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: 2023-02-09.

[5] Z. Kis, D. Peintner, C. Aguzzi, J. Hund, K. Nimura, Web of things (wot) scripting api, https://www.w3.org/TR/wot-scripting-api/, 2020. Accessed: 2023-02-09.

[6] M. Freund, T. Wehr, A. Harth, Blast: Block applications for things, in: The Semantic Web: ESWC 2022 Satellite Events: Hersonissos, Crete, Greece, May 29–June 2, 2022, Proceedings, Springer, 2022, pp. 68–72.

[7] I. Culic, A. Radovici, L. M. Vasilescu, Auto-generating google blockly visual programming elements for peripheral hardware, in: 2015 14th RoEduNet International Conference-Networking in Education and Research (RoEduNet NER), IEEE, 2015, pp. 94–98.

[8] Z. Wang, Y. Elkhatib, A. Elhabbash, Holoncraft–an architecture for dynamic construction of smart home workflows, in: 2022 9th International Conference on Future Internet of Things and Cloud (FiCloud), IEEE, 2022, pp. 213–220.

[9] G. Kellogg, P.-A. Champin, D. Longley, Json-ld 1.1, W3C Rec (2020).

[10] C. Kelleher, R. Pausch, Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers, ACM Computing Surveys 37 (2005) 83–137.
[11] E. Pasternak, R. Fenichel, A. N. Marshall, Tips for creating a block language with blockly, in: IEEE Blocks and Beyond Workshop, 2017, pp. 21–24.
[12] M. A. Kausar, V. Dhaka, S. K. Singh, Web crawler: a review, International Journal of Computer Applications 63 (2013) 31–36.
[13] R. B. Miller, Response time in man-computer conversational transactions, in: Proceedings of the December 9-11, 1968, fall joint computer conference, part I, 1968, pp. 267–277.