Leveraging Model-Driven Technologies for JSON Artefacts: The Shipyard Case Study

Alessandro Colantoni, Antonio Garmendia,
Luca Berardinelli, Manuel Wimmer
Institute of Business Informatics - Software Engineering
Johannes Kepler University Linz
Linz, Austria
{firstname.lastname}@jku.at

Johannes Bräuer
Dynatrace Research
Linz, Austria
johannes.braeuer@dynatrace.com

***Abstract*—With JSON's increasing adoption, the need for structural constraints and validation capabilities led to JSON Schema, a dedicated *meta-language* to specify languages which are in turn used to validate JSON documents. Currently, the standardisation process of JSON Schema and the implementation of adequate tool support (e.g., validators and editors) are work in progress. However, the periodic issuing of newer JSON Schema drafts makes tool development challenging. Nevertheless, many JSON Schemas as language definitions exist, but JSON documents are still mostly edited in basic text-based editors.**

To tackle this challenge, we investigate in this paper how Model-Driven Engineering (MDE) methods for language engineering can help in this area. Instead of re-inventing the wheel of building up particular technologies directly for JSON, we study how the existing MDE infrastructures may be utilized for JSON. In particular, we present a bridge between the JSONware and Modelware technical spaces to exchange languages and documents. Based on this bridge, our approach supports language engineers, domain experts, and tool providers in editing, validating, and generating tool support with enhanced capabilities for JSON schemas and their documents. We evaluate our approach with Shipyard, a JSON Schema-based language for the workflow specification for Keptn, an open-source tool for DevOps automation of cloud-native applications. The results of the case study show that proper editors and language evolution support from MDE can be reused and, at the same time, the surface syntax of JSON is maintained.

***Index Terms*—JSON, JSON Schema, MDE, DevOps, Tool Interoperability**

I. Introduction

The JavaScript Object Notation (JSON) [1] was first introduced as a lightweight data-interchange format. However, because of its widespread use, especially among Web developers, it is nowadays applied in many application areas (e.g., declarative interfaces specifications, Rest API messaging, storing, etc.). With the considerable spread of the use of JSON, developers noticed the need to be aware of the structure of a document [2] and to have available validation mechanisms. In this sense, there were initiatives that emerged to mitigate this problem, such as OpenAPI [3] and RAML [4]. Nevertheless, these technologies were designed to standardize RESTful APIs; therefore these solutions are not applicable to other areas that make use of JSON documents.

To the best of our knowledge, the JSON Schema [5] initiative is the major approach that aims to provide a general-purpose language to validate JSON documents. It is a dedicated language to define the structure of JSON documents. Currently, the standardisation process of JSON Schema (by the IETF) is work-in-progress. Because of this process, newer specifications are released every six months. The continuous evolution of JSON Schema as draft standard in the ongoing standardisation process is of course important and required, but it also puts some challenges on developing tool support for editing and validating JSON documents [6] such as dealing with ambiguities in the specifications [7]. Therefore, JSON documents are still mostly edited in generic text-based editors.

In this paper, we tackle this problem from a Model-Driven Engineering (MDE) perspective. As such, JSON and JSON Schema are part of a new emerging technical space called JSONware [8] in addition to consolidated ones such as Modelware [9]. Instead of building up a tooling infrastructure from scratch for JSONware, we aim to reuse existing support from Modelware. We propose a semi-automatic bridging approach among JSONware and Modelware with the intent of improving the textual editing and validation of JSON documents via state-of-the-art MDE practices [10] based on Eclipse-based technologies (EMF, Xtext, OCL). At the same time, we aim at keeping the approach transparent to JSONware users by preserving the native JSON concrete syntax, and by this, compatibility with existing JSON-based tools [6]. We report in this paper the results of an industrial case study. Shipyard [11] is a JSON Schema-based language that is used to define workflows in the DevOps area. Shipyard is already available in several versions with a growing user base. Our results show that it is possible to derive editing and validation support based on model-driven technologies for JSON Schema-based languages and to benefit from the knowledge in the field of evolution concerns in language engineering such as the well-known metamodel/model co-evolution problem.

The rest of the paper is organized as follows. Section II introduces the background on JSON and JSON Schema as well as on MDE technologies used in this paper. Section III introduces our bridging approach between JSONware and Modelware. Section IV evaluates our approach with the Shipyard language. Section V discusses related work, and finally, Section VI concludes the paper sketching current limitations, potential improvements, and future research directions.

II. Background

The goal of this section is to provide a short introduction to JSON and JSON Schema, as part of the new emerging JSONware. Since we propose a bridge to Modelware, we briefly summarize the main building blocks of this technical space as well. Finally, we review the term technical space and provide the motivation for this work.

A. JSONware: JSON and JSON Schema

JSON [1] initially emerged as a lightweight and human-readable data serialization and messaging format for supporting information exchange. JSON can represent four primitive types, i.e., string, number, boolean, and null, and two structured types, i.e., objects (delimited by curly braces) and arrays (delimited by squared brackets). Two exemplary JSON artefacts are shown in Lst. 1 and Lst. 2.

Listing 1. A JSON schema example.

```
{
    "$schema": "http://json-schema.org/draft-07/schema#"
    "type":"object",
    "properties":{
        "name":{"type":"string"},
        "surname":{"type":"string"}
    },
    "additionalProperties":true
}
```

Listing 2. A JSON schema instance, i.e., JSON document, example.

```
{
    "name": "Alessandro",
    "surname": "Colantoni"
    "affiliation": {
        "universityName": "Johannes Kepler University",
        "city":"Linz"
    }
}
```

Data is stored in name/value pairs separated by commas, while curly braces hold objects and square brackets hold arrays. Several (un)marshallers are available[1], which parse and read/write memory representations of an object from/to JSON documents. As a plain textual artefact, a JSON document can be manually edited in any text editor.

Recently, the IETF is promoting a JSON Schema standard [12]. According to the IETF, JSON Schema is "*a JSON-based format for describing the structure of JSON data. JSON schema asserts what a JSON document must look like, ways to extract information from it, and how to interact with it.*" [13]. As a *schema* defines the data structure of JSON documents, the document is referred to as an *instance* of a given *schema*.

[1]cf. e.g., http://json.org.

For instance, Lst. 1 is a JSON Schema, while Lst. 2 is an instance of Lst. 1 (i.e., Lst. 2 conforms to Lst. 1).

The main goal of the JSON Schema standard is to provide users with a language to define constraints on JSON documents and tools for checking their conformance between schema instances and the corresponding schema [7]. In consequence, JSON schema allows defining languages that may benefit from dedicated tool support. Finally, with the term *metaschema*, the IETF refers to schemas against which other schemas can be validated. A metaschema is self-descriptive, i.e., validated against itself.

Being a *draft standard*, JSON Schema undergoes periodic revisions every six months. Each revision results in a new JSON Schema Draft including a *metaschema*. In this regard, the keyword $\$schema$ can be used to suitably link a schema to the corresponding metaschema. For example, Listing 1 shows a $\$schema$ declaration that refers to the metaschema issued by the Draft 7 revision.

B. Modelware: (Meta-)Modeling

The core of MDE includes the pillar concepts of *model*, *metamodel*, and *model transformation* [10].

Models in MDE are considered as machine-readable artefacts. *Metamodels* define the modeling concepts and their relationships and provide the intentional description of all possible models, which have to conform to the associated metamodel. From a language engineering perspective, a metamodel represents the *abstract syntax* of a modeling language. The standard metamodeling language defined by the OMG is the Meta Object Facility (MOF) [14]. The Eclipse Modeling Framework (EMF) [15] is its most prominent realization based on Ecore *metalanguage*. Metamodels define modeling languages in a purely conceptual way and are independent of any form of concrete representation. The *concrete syntax* of a language assigns graphical or textual elements to metamodel elements that can be understood by users and, possibly, edited through model editors [10]. Xtext, which is used later on in our bridge in order to deal with the textual concrete syntax of JSON, is a particular framework to define a text-based syntax for modeling languages. As models in MDE are considered as machine-readable artefacts, model transformations are applicable to transform them to different languages or to modify the models for particular purposes.

C. Technical Spaces

The term technical space (TS) [9], [16], [17] has been introduced in [18] as "*a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities*". TSs show a recurrent conceptual architecture of *metalayers*. Each metalayer defines the representation structure and a global typing system for the underlying level [18]. Typically, TSs are organized in three metalayers (metalanguage, language, and instance layers). Modelware, as explained before, fits these three metalayers when MOF/Ecore is used as the metalanguage. The same is true for JSON Schema-based languages.

TSs may differ w.r.t. offered capabilities [9] and related tool support. In this paper, we are particularly interested in editing and validation capabilities and the possibility to automatize the generations of supporting tools, i.e., editors and validators, for JSON artefacts by reusing the tools we have in Modelware. Therefore, we aim to shed some light on the differences and commonalities of the three metalayers of JSONware (focus on JSON Schema) and Modelware (focus on MOF/Ecore). Our hypothesis is that, if there is enough commonality, we are able to build tool support in Modelware for JSON Schema-based languages. With the help of the Shipyard case study, we aim to generate evidence if the hypothesis holds or not.

III. Approach

In this section, we present the requirements, an overview on our approach, a technical perspective, and finally, some information about our tool support.

A. Requirements

JSON Schema has been proposed as a schema language in response to the emerging need for validation to qualify a JSON document as an *instance* of a given *schema* . However, JSON Schema is going beyond its role as a schema language for defining data formats, as stated by the IETF [5] in each draft release [13], [19]. Indeed, it is going to be adopted also as *meta-language* for defining domain-specific languages (DSL), which, in turn, are used by domain experts to edit and validate documents (e.g., for configuring stage-based continuous delivery pipelines in DevOps environments [11] as is the case study of this paper, see Section IV).

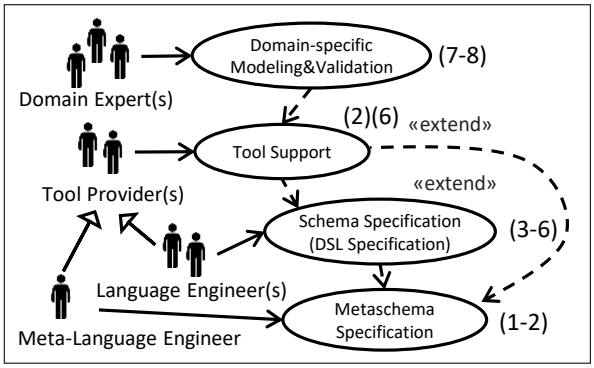

Fig. 1. Actors and use cases of our field of investigation.

Fig. 1 shows a use case diagram of actors and use cases, which depicts the requirements our approach has to fulfill:

- The *Metalanguage Engineer* defines and publishes metalanguages. In JSONware, this role is currently played by the IETF, the standardization body for JSON Schema, in charge of publishing the JSON *metaschemas*.
- The *Language Engineer* creates and manipulates languages. In JSONware, this is a new potential role. This role is providing JSON *schemas*, conforming to the JSON metaschemas, to specify new textual DSLs.
- The *Domain Expert* creates and manipulates JSON documents. In JSONware, those artefacts are schema instances conforming to a given schema.
- The *Tool Provider* creates tools to support activities in a given TS. As for any other TS, in JSONware, typical tools are parsers and editors for any JSON artefact[2]. The advent of JSON Schema is now demanding metalayer-specific tools, i.e., schema and schema-instance editors and validators to support language engineers and domain experts, respectively.

Next, we explain how our approach deals with these requirements.

B. Conceptual Architecture and Workflow

Our approach intends to support the aforementioned use cases (metaschema specification, schema specification, domain-specific modeling and validation, and tool support) as depicted in Fig. 1. For supporting these use cases, our approach follows two main principles. First, we aim at reusing MDE principles and practices to support the identified use cases. In particular, we choose the Eclipse Modeling Framework (EMF) [15] as the reference MDE framework due to its integration with the Xtext [20] language workbench for textual DSLs and the availability of Object Constraint Language (OCL) [21] for validation tasks. We also use this setting for building up the bridge between JSONware and Modelware by explicitly modeling JSON artefacts and utilizing model transformations. Second, we keep the full approach transparent for language engineers and domain experts from JSONware by preserving the standard JSON concrete syntax [1] in generated editors for schemas and their instances.

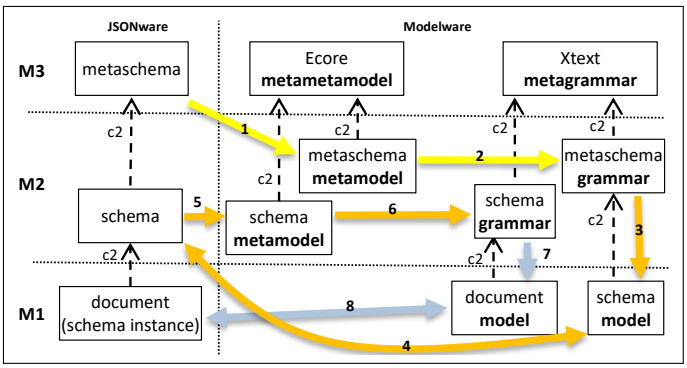

Fig. 2. General architecture: Artefacts and their relationships.

Fig. 2 depicts the overall architecture of the proposed approach. To visually qualify the nature of the involved artefacts, the conceptual architecture is vertically divided into two TSs, i.e., JSONware and Modelware, and horizontally split in the three *metalayers*, i.e., M3, M2, and M1. Directed conformance relationships (c2) pair artefacts across adjacent metalayers within the same TS.

[2]According to the IETF, metaschemas, schemas, and their instances are all JSON documents (.json).

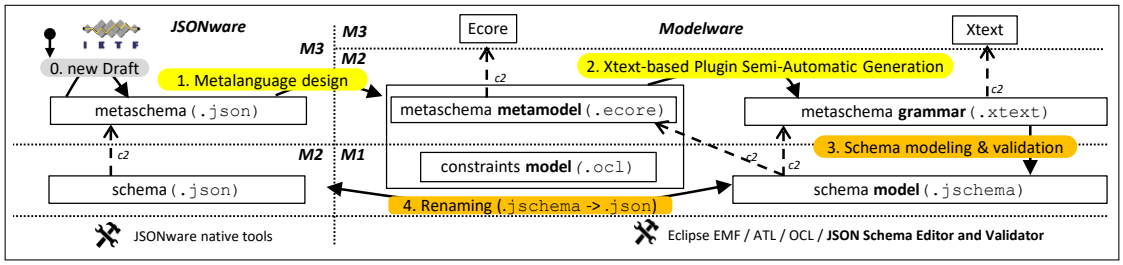

Fig. 3. Bridging approach: metamodeling support (M2) in JSONware (Steps 1 to 4).

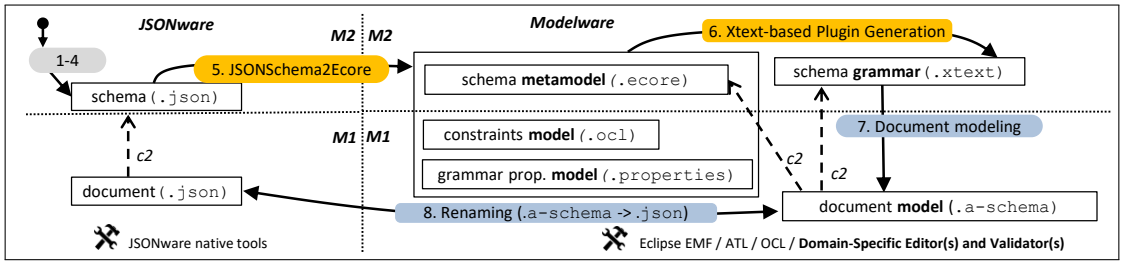

Fig. 4. Bridging approach: modeling support (M1) in JSONware (Steps 5 to 8).

The approach uses a workflow consisting of eight steps. In Fig. 1, such steps are grouped and related to corresponding use cases and user roles. Each step establishes relationships between pairs of artefacts bridging JSONware to Modelware (steps 1,4,5, and 8) and vice versa (steps 4 and 8). Together, the identified bridging steps realize two-way bridges for language engineers (steps 4) and domain users (step 8). As a result, the same schema (M2) and schema instance (M1) can be seamlessly manipulated within JSONware and Modelware.

First, a bootstrapping activity (steps 1-2) happens at level M3 when a new JSON metaschema is released by a metalanguage engineer. The main outcomes are (i) an Ecore metaschema metamodel, (ii) an Xtext metaschema grammar, and (iii) an Xtext-based textual editor for modeling and validating schema artefacts. The Ecore metamodeling activity is a manual step. The metaschema grammar and schema editor are automatically generated from the metaschema metamodel thanks to Xtext [20]. The involved steps are intended to be repeated in case of issue of a new JSON Schema Draft by the IETF (see Fig. 2).

Language engineers edit and validate schemas (steps 3-6). The schema metamodels, schema grammars, and schema instance editors can be automatically generated for each schema. Such schema can be edited in the previously generated schema editor or in any textual editor.

Finally, domain experts can edit and validate JSON documents as schema instances in dedicated textual modeling editors to accomplish domain-specific tasks (steps 7-8).

C. Technical Realization

A more detailed view on the bridging process is depicted by the two workflows shown in Figs. 3-4. For the sake of presentation, the bridging process has been graphically split into two sets of activities, completing the bridging at level M2 (steps 1-4, Fig. 3) and M1 (steps 5-8, Fig. 4), respectively.

Steps 0-1: The whole activity is triggered when a new JSON Schema Draft is published by the IETF (step 0), replacing the previous metaschema. The metalanguage engineer manually designs two artefacts, a metaschema metamodel in Ecore and validation constraints in OCL. Fig. 5 shows an excerpt of the metaschema metamodel based on JSON Schema Draft 7 [19][3]:

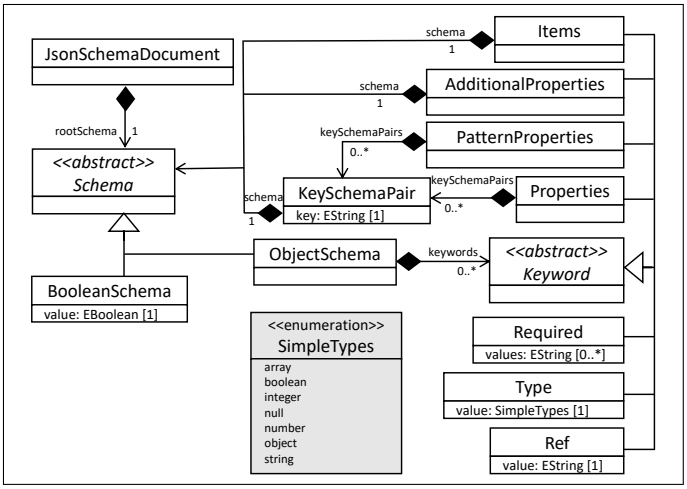

Fig. 5. Excerpt of the metaschema metamodel.

- A *JsonSchemaDocument* is a hierarchical collection of schemas. A root schema is the schema that comprises the entire JSON document. A *Schema* can be either a *BooleanSchema* or an *ObjectSchema*.
- As the name suggests, a BooleanSchema has only two valid instances: `true` and `false`. This type of schema is used

[3]The complete metamodel is available at [22].

for opening or closing a schema definition as we will explain in more detail later on.

- An ObjectSchema is a possibly empty, hierarchical collection of *Keywords*.

For each keyword defined in the JSON Schema draft 7 [19], a distinct metaclass has been created in our metamodel. Fig. 5 shows a minimal subset of seven Keywords (out of 46 defined in [19]), suitable for introducing the further steps and the case study (see Section IV). Some of them (*Items*, *AdditionalProperties*, *PatternProperties*, and *Properties*) allow the hierarchical structuring of a JsonSchemaDocument while others (*Required*, *Type*, and *Ref*) stops the nesting of ObjectSchemas. In particular:

- *Required* specifies keys that are mandatory.
- *Type* assigns a *SimpleType* to an ObjectSchema and, depending on the chosen SimpleType, a different subset of the keywords is allowed.
- *Ref* establishes references among Schemas owned by arbitrary JsonSchemaDocuments, following the companion JSON Pointer standard [23].
- *Items* assign a Schema to be used for validating the elements (i.e., the items) of an ObjectSchema of Type *array*.
- *Properties* defines explicitly structural features of ObjectSchemas. A KeySchemaPair consists of two elements: (i) a *key* attribute holds the property names that will appear in conforming schema documents, and (ii) a Schema defining the property structure.
- Similar to Properties, *PatternProperties* defines structural features but the key of the KeySchemaPair is provided as a regular expression. Every key matching the regular expression must conform to the Schema referenced by the KeySchemaPair.
- *AdditionalProperties* controls the handling of properties whose names are not defined via Properties and PatternProperties, thus greatly influencing the openness, and in consequence, the validation capabilities of schema documents. Three options may be used: (i) BooleanSchema is `false`, no properties will be allowed in addition to those explicitly declared; (ii) BooleanSchema is `true`, any well-formed JSON document can be added to a schema document[4]; (iii) an ObjectSchema is chosen, it will be used to validate such additional properties.

It is worth noting that the metaschema metamodel has been manually designed with the intent of (i) preserving the tree-based structure of JSON documents and string-based typing[5], and (ii) to ease the generation of the corresponding Xtext grammar (cf. steps 2 and 6).

Constraints of the metaschema metamodel have been manually specified in OCL. It includes constraints necessary to validate the values of Keywords, e.g., non-negative integers for `minLength`, `maxLength`, `minItems`, `maxItems`, `minProperties`, and `maxProperties`.

[4]The JSON Schema standard [19] assigns a BooleanSchema `true` to any ObjectSchema by default.

[5]For example, string-based attributes (e.g., *values* in Required) have been preferred and to EReferences (e.g., from Required to Properties).

Step 2: An Xtext grammar and a text-based metaschema editor are generated from the metaschema metamodel from step 1. The JSON Schema grammar, automatically generated by the Ecore2Xtext transformation provided by the Xtext framework, requires manual refactoring by the metalanguage engineer, which updates the production rules to support the standard JSON concrete syntax [1].

The Xtext-based editor can then be generated, providing advanced editing capabilities typically supported by an IDE (e.g., syntax colouring and basic content assist features). Moreover, the set of OCL metaschema constraints from step 1 can be integrated to enhance its validation capabilities.

Step 3: The language engineer is now equipped with a dedicated textual editor for JSON schema modeling (e.g., Fig. 10a and Fig. 10d) and validation (Fig. 10e and Fig. 10f). Thanks to Xtext, the text-based artefact can be parsed into an equivalent model-based representation . Thus, the language engineer can also benefit from typical model-driven language infrastructure capabilities such as syntax highlighting, auto-completion, and hooks for well-formedness rules, as the OCL constraints defined in step 1.

Step 4: In our approach, a JSON schema conforms to the metaschema definition in JSONware (i.e., metaschema.json) and to the metaschema grammar in Modelware (i.e., metaschema.ecore and metaschema.xtext). For this reason, switching between JSONware and Modelware becomes possible and the same document can be seamlessly edited and validated by tools available in both TSs. We will exploit this possibility in the next step.

Step 5: The schemas from step 4 can now be automatically translated into (i) a schema metamodel defined in Ecore, (ii) schema constraints in OCL, and (iii) configurations for tuning the generation of the schema grammar in step 6.

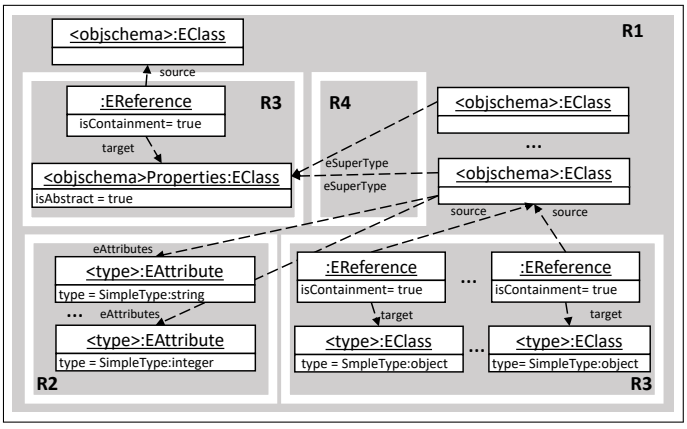

Fig. 6. Generic structure of EClass Object.

Step 5.1. Schema Metamodel Generation. A model transformation, implemented in ATL [24], translates a JSON schema artefact into a schema metamodel. The transformation rules are outlined in Table I (cf. Rules R0-R5), while Fig. 6 shows the excerpt of a generic schema metamodel in an object diagram as generated from the transformations rules.

TABLE I
MAPPING FROM METASCHEMA TO ECORE

Rule	Metaschema			Ecore		
R0	JSON Schema Document			EPackage		
R1	Schema		BooleanSchema = false	n.a.		
			BooleanSchema = true	JSONDocumentFragment (see Fig. 7)		
			ObjectSchema	EClass		
R2.1	Keyword	Type	string, boolean, integer, number	EAttribute	[1,1]	of containing ObjectSchema (R1), of type EString, EBoolean, EInt, EDouble, respectively
R2.2			null	EAttribute	[1,1]	of containing ObjectSchema (R1), of type NullEnum (see Fig. 7)
R2.3			array	EReference (containment)	[0,*]	from containing ObjectSchema (R1) to Items.schema (R1) (see Fig. 5)
R3		Type	object	EClass (self)	abstract=true	
				EReference (containment)	[0,*]	from containing ObjectSchema (R1) to self (R2)
R4.1		Ref		EReference (containment)	[1,1]	from containing ObjectSchema (R1) to referenced Schema (R1, via JSONPointer)
R4.2		Properties		ESuperType		from contained keySchemaPairs.schema (R1) to Type (R3)
R4.3		PatternProperties		for each contained KeySchemaPair do (R5)		
R4.4		AdditionalProperties		EAttribute	[1,1]	key::EString of contained schema (R1)
				ESuperType		from contained schema (R1) to Type (R3)
R5	KeySchemaPair			EAttribute	[1,1]	key::EString of contained schema (R1)
				ESuperType		from contained schema (R1) to Type (R3)

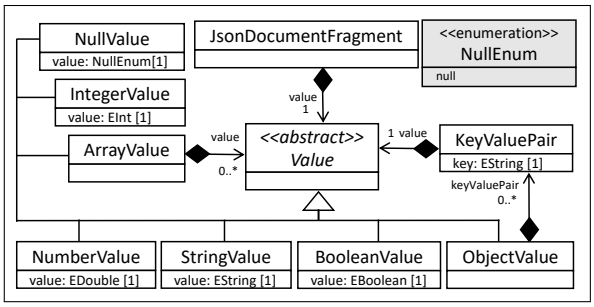

Fig. 7. JSONDocumentFragment.

A new EPackage is generated for each JSONSchemaDocument (R0). Continuing with R1, BooleanSchema `false` is ignored since it only forbids extensions, as required by any Ecore-based metamodel. BooleanSchema `true` allows arbitrary extensions of JSONSchemaDocuments with well-formed (i.e.,parsable) JSON fragments. Such fragments are out of the scope of the conformance relationship (c2) between M1 and M2 artefacts in Modelware and JSONware (see Fig. 4). However, they have to be syntactically validated (i.e., parsed) as well-formed JSON documents. For this reason, (i) the metamodel metaschema in Fig. 5 has been extended with the *JsonDocumentFragment* metaclass (see Fig. 7), and (ii) the BooleanSchema `true` has been mapped to JSONSchemaDocumentFragment.

A separate EClass is generated for each ObjectSchema (R1), while its structural features depend on the Keywords it contains. If properties are defined (Properties, PatternProperties, AdditionalProperties), a generic abstract Properties EClass is generated and extended by the corresponding EClasses (cf. schema references targeting the Schema metaclass in Fig. 5), if any (R3, R4.2, R4.3, R4.4). Such EClasses are further enriched by different EStructuralFeatures depending on the assigned SimpleTypes, i.e., (i) EAttributes (boolean, integer, null, number, string) (R2), and (ii) containment EReferences to a newly created EClass for object (R3) and array (R2.3).

Finally, a Ref generates a containment EReference to the Schema pointed by the JSON Pointer [23] string saved in the Ref's value (R4.1).

As an example outcome of the mapping, Fig. 8 shows the metamodel generated from the JSON Schema of Lst. 1.

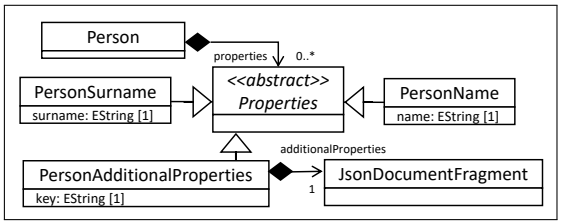

Fig. 8. Example of metamodel generated from Lst.1

The chosen mapping produces a high number of EClasses due to scalar types (i.e., string, boolean, number, integer, and null), which generate separate EClasses rather than EAttributes, for scalability reasons of the generated Xtext grammar (cf. step 6). Indeed, in each JSON schema, properties are unordered and a mapping to EAttributes in Ecore would eventually require the use of unordered groups in Xtext. As a consequence, Xtext would generate an alternative for all possible combinations for each group of JSON schema properties (i.e., n! for a group of n properties) [25]. Our mapping choice requires an a single alternative for each object's property (i.e., n for a group of n properties), thus scaling complexity from factorial to linear.

5.2 Constraints. Schema constraints are specified in OCL. Their generation is coupled with the schema metamodel generation by the ATL transformation invoking Xtend [26] methods. Constraints are generated for:

- The value of the EAttribute generated by R4.3 and R5 for PatternProperties matching the regular expressions.
- The Keyword Required[6] to check that each EClass, generated for an ObjectSchema (R1) declaring the keyword Required, contains at least a child (cf. R4.2, R4.3, R4).

5.3 Grammar Generation Configuration. A grammar properties model is generated as well. The artefact is required for defining the textual concrete syntax of the JSON Schema in-

[6]It does not appear in Table I because it has no effect on the generated Ecore.

stances. The metamodel is presented in Fig. 9 and is available in [27].

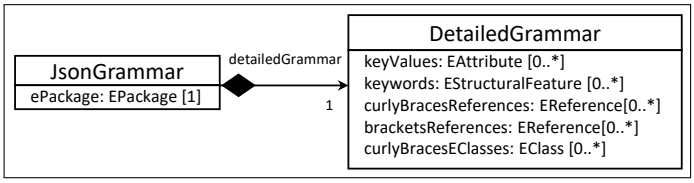

Fig. 9. Metamodel to specify the JSON grammar properties.

Indeed, the generation of a concrete JSON-based grammar from an Ecore metamodel has variation points. In particular, decisions have to be taken about:

- zero-to-many relationships, in order to manage them as array (included between brackets), or as name/value pairs owned by a JSON Object (included between curly braces).
- EAttribute's and EReference's names, whether they have to be managed as DSL keywords or not.
- EAttribute's value, whether it has to be considered the key of a key/value pair or not.
- EClass, whether it has to be considered a JSON Object (included between curly braces) or not.

The grammar properties models are automatically derived from the metaschema with an ATL transformation.

Step 6. Similarly to step 2, an Ecore artefact, in this case the metamodel for a given schema, is processed to generate an Xtext-based grammar for that particular schema. As in step 2, the production rules need to be refactored to support the standard JSON concrete syntax [1]. It is worth noting that, different from step 2, this grammar refactoring step has been automated [27]. We developed an extension of plugin to generate the Xtext Plugin from an Ecore, to take as input also the JSON grammar properties model in addition to the Ecore metamodel [27]. We developed a plugin to generate an Xtext JSON based grammar [27]. To do this, we adapted the default grammar provided by Xtext to generate a JSON grammar. This plugin takes as an input the JSON grammar properties model and also, the Ecore metamodel.

Step 7. Similar to step 3 for language engineers, domain experts are now equipped with a dedicated textual editor for editing and validating a JSON document conforming to a schema model (Fig. 4).

Step 8. Similar to step 4, JSON documents can now be treated as models conforming to a given schema available both in JSONware and Modelware. The bidirectional switch between JSONware and Modelware is possible and the JSON document can be seamlessly edited and validated by tools available in both TSs.

IV. Evaluation

In the following, we state two research questions (RQ) that are used to evaluate our proposed approach based on the Shipyard case study.

- **RQ1**: Is our approach capable of presenting the different versions of the Shipyard language initially defined with JSON Schema?
- **RQ2**: As the Shipyard language is subject to evolution, can we support the co-evolution of existing DevOps workflows?

Next, we introduce our case study: the Keptn project and the Shipyard DSL before we continue with the evaluation of our approach.

A. Case Description: Continuous Delivery with Keptn and Shipyard

Keptn is an open-source project[7] that provides a control-plane for orchestrating continuous delivery and operational processes activities. Continuous Delivery (CD) is part of the continuous-$*$ holistic endeavor towards Continuous-Software Engineering (CSE) [28]. In CD, delivery pipelines are executable software delivery workflows designed to automate the roll out of new software features and updates, useful in minimizing the amount of manual work needed to release or maintain software. Multiple problems may occur with delivery pipelines. Over time, pipelines become complicated because of mixed information about processes, target platforms, environments, and tools. Pipelines can be duplicated across different tools, which may require tool-specific changes resulting in maintainability problems. Finally, due to the lack of a clear separation of concerns, users with various roles (such as developers, DevOps experts, and site reliability engineers) may employ the pipeline for different purposes. As a result, delivery pipelines can quickly become an unmanageable legacy code artifact.

Keptn addresses this problem by separating the processes from the actual tooling. Thus, the definition of continuous delivery or operational processes manifests in a so-called *Shipyard* configuration. A Shipyard configuration is a JSON document declaring staged workflows as the basis for the orchestration conducted by Keptn.

With the advent of JSON Schema, a Shipyard schema is being developed, elevating it to the role of DSL. It demands proper tool support for language engineers and domain experts involved in the editing and validation of the Shipyard DSL versions and Shipyard configurations in Keptn, respectively.

B. Case Study Execution

In the following, we present two evaluation strategies to answer the aforementioned research questions by applying our approach for the Shipyard case: (i) we use our developed tool support to transform the Shipyard JSON Schema definitions in order to validate that our tool support is working as expected, and (ii) we inspect the generated Shipyard metamodels to reason about the Shipyard language and artefact (co-)evolution .

1) RQ1: Testing Shipyard DSL Versions: The Keptn project maintains its companion language Shipyard DSL in a public repository [11]. Fig. 11 reconstructs and intertwines the evolution of four versions of the Shipyard schema, with Keptn releases, and the executions of our approach steps.

[7]Started in January 2019 by the company Dynatrace and then donated to the Cloud-Native Computing Foundation (CNCF) in 2020.

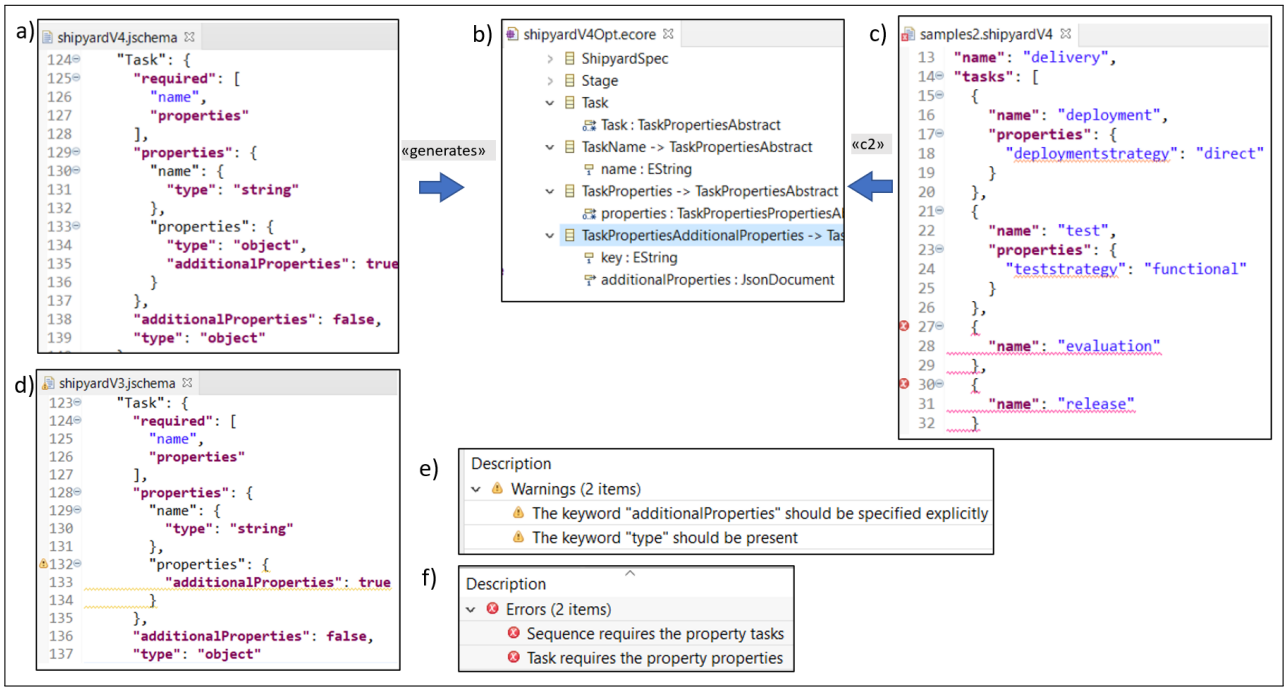

Fig. 10. (a) JSON Schema textual model excerpt of shipyardV4 (.jschema), (b) excerpt of generated Ecore for shipyardV4, (c) excerpt of Shipyard model (.shipyardV4) before last fix, (d) JSON Schema textual model excerpt of shipyardV3 (.jschema), (e) validation results for shipyardV3 (.jschema), (f) validation results of a Shipyard model (.shipyardV4).

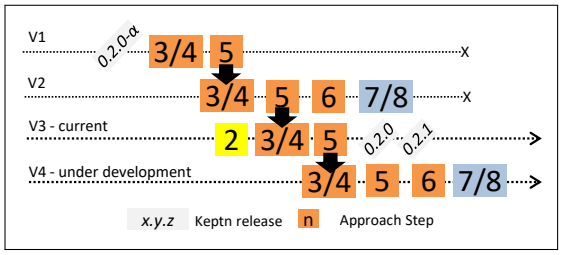

Fig. 11. Shipyard DSL evolution timeline.

In particular, at the time of writing, a total of four different schema versions have been collected including (i) two discontinued/archived one (v1 and v2), (ii) a current version shipped together with the latest release of Keptn (v3), and (iii) a version (v4) that is under development.

For the whole Shipyard schema evolution, we considered the JSON Schema Draft 7 [19] as the only reference metaschema in JSONware. Therefore, the bootstrapping steps, i.e., the metaschema design and the generation of the schema editor (steps 1 and 2 in Fig. 2) have been carried out by the authors as metalanguage engineers at the beginning of the evaluation activities. As a result, we obtained a stable schema editor to carry out schema development and validation activities (step 3), preserving by construction compatibility with existing JSON Schema tools (step 4).

We started to transform the Shipyard schema v1, shipped with Keptn 0.2.0α, into the corresponding Shipyard metamodel (step 5), which failed due to misuse of $ref (i.e., instance of Ref metaclass in Fig. 5), whose value, despite being valid w.r.t. the JSON Pointer standard [23], was pointing to missing definitions, causing the failure of R5 (missing target in EReference, see Table I)

We used step 3-4, to produce a new version of Shipyard schema (V2) to fix this issue and succeeded in generating the first Shipyard metamodel (step 5). The metamodel was manually inspected for potential inconsistencies. In particular, given the mappings in Table I, we were expecting a single, connected graph of EClasses. Instead, we realised that the root schema was disconnected from the rest of the metamodel. We revised the Shipyard schema (again, step 3-4) and we succeeded in generating the Ecore metamodel (step 5), Shipyard Xtext grammar, and editor (step 6) to provide model-based editing and validation support for Shipyard users (step 7-8, see Fig. 10).

Listing 3. Default values for Type and AdditionalProperties.

```
{
    "type" : ["array","object","integer","number",
              "null","string", "boolean"],
    ...
    "additionalProperties" : true
}
```

The V2 changes were included, together with other ones approved by the maintainers of Keptn, into V3, released with Keptn 0.2.0. We repeated the editing and validation activities with V3 (steps 3-4). In the meantime, we improved the validation capabilities of the generated tools by extending the set of OCL constraints with warnings. Such warnings detect missing schema keywords (i.e., instances of concrete subclasses of Keyword metaclass in Fig 5, step 2) like `type` and `additionalProperties` in any schema. If missing,

the chosen JSON Schema standard (i.e., Draft 7 [19]) assigns a default value to them (see Lst. 3), which has a great impact on applied validation rules, accepting any simple type values and any JSON document fragments in JSON documents, as Shipyard configurations are.

The warnings, detected on the Shipyard metamodel, caused new fixes, preventing the generation (and possible maintenance effort) of a new Shipyard editor and triggered a new, currently under development Shipyard version (V4), not yet shipped in any Keptn release. Currently, we are evolving V4 by checking which Shipyard keywords are required or simply optional.

Fig. 10 shows the current generated editors for JSON Schema Draft 7 and Shipyard V4, together with a tree-based representation of the internal representation based on Ecore EMF [15].

Answering RQ1: We are able to generate the metamodels/grammars from the existing Shipyard specifications, but it also turned out to require iterations to refine certain parts in JSON schemas which may not be required if they are "just" used as documentation. However, by this iterative process, our presented approach helps in improving language issues for future versions.

2) RQ2. Co-evolution Analysis: Shipyard is evolving over time together with Keptn as it is a cutting-edge technology. The coupled evolution (co-evolution) of model-driven artefacts is a typical common research topic for the MDE community. A classification of metamodel-level changes have been presented in [29] considering *non-breaking changes*, *breaking resolvable changes*, and *breaking non-resolvable changes* depending on their effect on the conformance relationships between M1 and M2 artefacts and possible (automated) repairing actions.

Thanks to the bridging approach towards Modelware, we aim at reusing this body of knowledge to tackle the co-evolution in JSONware in general, and for Shipyard case study in particular. In this respect, Table II and Table III list changes between two consecutive Shipyard versions, i.e., v2 vs. v3 and v3 vs. v4, respectively, comparing the artefacts automatically generated at step 5, i.e., the Shipyard Ecore metamodels and the companion OCL constraints artefacts. EMF Compare [30] and the Eclipse compare editor have been used to perform the comparison task.

In particular, we found the following results:

- V2 vs. V3: 10 new EClasses and 2 new OCL constraints have been added in V3, without changing existing metamodel elements and OCL constraints. The new OCL constraints apply to the newly added EClasses. No OCL constraints have been added to pre-existing EClasses. Thus, we classify them as non-breaking changes (↑).
- V3 vs. V4: 1 EClass and 2 OCL constraints have been removed. The removed EClass (SelectorMatchAdditionalProperties) was allowing arbitrary JSON fragments for the SelectorMatch schema object. Therefore, v3 Shipyard configurations containing such JSON fragments do not conform to v4. This change is classified as a breaking change (X), which is required to have a more precise definition of the language. Finally, the removal of OCL constraints, since it loosens the validation step, is considered as a non-breaking change by default (↑).

TABLE II
CLASSIFICATION OF CHANGES BETWEEN SHIPYARD V2 AND V3.

Added Ecore EClasses	
Selector	↑
SelectorPropertiesAbstract	↑
SelectorMatch	↑
SelectorMatchPropertiesAbstract	↑
SelectorMatchAdditionalProperties	↑
SelectorMatchPatternProperties0	↑
Trigger	↑
TriggerPropertiesAbstract	↑
TriggerEvent	↑
TriggerSelector	↑
Added OCL Constraints	
TriggerRequiredevent	↑
SelectorMatchPatternProperties0patternProperties0Regex	↑

TABLE III
CLASSIFICATION OF CHANGES BETWEEN SHIPYARD V3 AND V4.

Removed Ecore EClasses		**Removed OCL Constraints**	
SelectorMatchAdditionalProperties	X	SequenceRequiredtasks	↑
		TaskRequiredproperties	↑

Answering RQ2: By reusing knowledge from co-evolution research in MDE [29], the evolution of the Shipyard JSON schemas could be analyzed to reason about forward and backwards compatibility. Existing work from the MDE community has provided clear guidelines on how to classify the different changes happening in JSON schemas, however, in order to do so, it is also required to explicitly model aspects such as the openness of JSON schemas.

C. Threats to Validity

We now discuss threats that can affect the validity of our study.

1) Internal Validity: The current approach has been devised involving a still limited number of users. The authors played all the roles depicted in Fig. 1. As mitigation strategy, we aim at involving a larger community of language engineers (by applying the approach to several schemas used in different domains) and domain experts (starting from the involvement of the Keptn user community). We compared a limited set of related tools. Many other tools may provide dedicated features for JSON artefacts. Our intention, with this paper, is setting a baseline for dedicated tool support which may be further explored in the future.

2) External Validity: We cannot claim that the results of the case study can be generalized for other JSON-based DSLs as JSON Schema may be used for validating very large documents. For scalability assessment purposes, we will investigate larger schema documents as publicly available in the Schema Store [31].

V. RELATED WORK

Bridging TSs has a long tradition in the MDE research community. In the following, we summarize the most related work in this area with respect to the contribution of this paper.

In [32], Brunelliere et al. focus on the interoperability in a tool ecosystem providing model-driven bridges between two or more tools. By leveraging MDE principles and techniques, the authors express the internal structure of the tools as metamodels; map related concepts in the different metamodels; and finally, boost interoperability through model-to-model transformations. We used a similar approach to bridge JSONware and Modelware since we use Modelware as the major TS for representing the artefacts and performing the integration.

The work described in [33] proposes an approach and a case study for bridging technical spaces of different complexities. They consider the benefits of technical spaces bridging for tool collaboration instead of competition in order to promote synergies. While this is also our goal, we have to consider in our work completely different meta-languages involved and deal with concrete syntax concerns as well.

Previous work tackled the modernization of XML-based languages. For instance, in [34], we presented an approach to bridge the gap between XMLware, Modelware, and Grammarware, via the generation of Xtext-based editors from XSDs providing editor functionality, customization options for the textual concrete syntax style, and round-trip transformations enabling the exchange of data between the involved technical spaces. Our approach shown in this paper follows a similar spirit. We bridge the gap between JSONware and Modelware via the generation of Xtext-based editors from JSON metaschemas and schemas to edit JSON documents representing schemas and their instances, respectively. Thanks to the shared JSON concrete textual syntax among artefacts from JSONware, the round-trip transformation step is not required because the same JSON documents conform to the corresponding metalayer artefact in both JSONware and Modelware. Thus, we are augmenting the JSON artefacts with Modelware technologies instead of migrating them to a new TS. In the context of the XML-based language modernization work in [35], we proposed a model-based approach to specify reusable textual styles for domain-specific modeling languages. This approach can be investigated to define a JSON style to automatize the refactoring of the generated Xtext grammars for JSON-based DSLs as future work.

In [36], the authors implemented a JSONSchema to UML transformation tool. The tool represents a concrete attempt to bridge JSONware and Modelware for documentation purposes. In this context, the authors list several open challenges for mapping JSON Schema to UML. Since UML class diagrams are considered a superset of Ecore, we faced several challenges which have been described. Previous work of the same authors [2], [37] aim at resolving the problem of schemaless JSON documents which was especially critical before the emergence of JSON schema. Thus, they proposed to discover the implicit schema with an iterative process that generated a model, with an enrichment for every new JSON document to be analyzed.

In [38], the author collects and discusses requirements for the implementation of a model-driven editor for JSON documents and validates them against a given JSON Schema Draft. Tree-based and form-based views are provided for the edited schemas. In our work, we provide a TS-oriented approach to deal with JSON Schema drafts evolution and editor generation. Moreover, we aim at preserving the JSON textual notation as common concrete syntax across TS by suitably refactoring Xtext grammars. In future work, however, the provision of other kinds of editors may be possible based on our metamodel definitions and existing tool support in MDE such as EMF Forms or Sirius.

In [7], the authors propose a formal grammar for JSON Schema as an EBNF. The authors identified a fragment of the functionalities from the full official JSON schema language and expressed them as a core set of production rules with which all the remaining functionalities can be expressed. We have expressed the grammar by deriving it from an `Ecore` metamodel, and we plan to compare the two resulting grammars with the aim to improve and complete `metaschema.ecore` artefact in future work.

To sum up, while there have been several approaches for bridging technical spaces in the past, the bridge we propose in this paper is novel with respect to previous work as there was a lack of research in bridging JSONware with Modelware on all required meta-levels and putting a particular focus on maintaining the textual concrete syntax.

VI. Conclusion and Future Work

In this paper, we have presented a systematic approach to generate specific tooling for JSON artefacts by reusing technologies from Modelware. The Shipyard case study confirms that meaningful tool support can be established by exploiting a bridge between JSONware and Modelware.

However, we also foresee several future research lines that have to be tackled in order to gain even more benefits from Modelware for JSON artefacts. We are currently extending the mapping in order to support the combined schemas of JSON Schema such as allOf and anyOf. In addition, we are going to explore the possibility to add OCL constraints to validate JSON Schemas to ensure the correctness of the transformation to Ecore. For instance, this could be useful to warn the user about cases in which the transformation may fail such as it was the case with Shipyard V1. In the long term, we plan to offer an automatic generation of other kinds of grammars such as YAML by combining the presented approach with [35].

Acknowledgment

This work was partially funded by the EU Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 813884, the AIDOaRt project ECSEL Joint Undertaking (JU) under grant agreement No. 101007350, the Austrian Science Fund (P 30525-N31), and by the Austrian Research Promotion Agency (FFG), program ICT of the Future, project number 867535. The research also contributed to the ITEA3 BUMBLE project (18006).

References

[1] ECMA, "The JavaScript Object Notation (JSON) Data Interchange Format," https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf, 2021.

[2] J. L. Cánovas Izquierdo and J. Cabot, "Discovering Implicit Schemas in JSON Data," in *Proceedings of the 13th International Conference on Web Engineering (ICWE)*, ser. Lecture Notes in Computer Science, vol. 7977. Springer, 2013, pp. 68–83.

[3] "OpenAPI," https://www.openapis.org/, 2021, accessed: 2021-01-15.

[4] "RAML," https://raml.org/, 2021, accessed: 2021-01-15.

[5] "JSON Schema," http://json-schema.org/, 2021, accessed: 2021-01-15.

[6] "JSON Schema Implementations," https://json-schema.org/implementations.html, 2021, accessed: 2021-01-15.

[7] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of JSON Schema," in *Proceedings of WWW*, 2016, pp. 263–273.

[8] R. Lämmel, *Software languages: Syntax, semantics, and metaprogramming*. Springer, 2018.

[9] I. Kurtev, J. Bézivin, and M. Aksit, "Technological spaces: An Initial Appraisal," *Proceedings of CoopIS, DOA*, 2002.

[10] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice, Second Edition*, ser. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017.

[11] "Keptn: Cloud-native application life-cycle orchestration," https://keptn.sh/, 2021, accessed: 2021-04-30.

[12] M. Droettboom, "Understanding JSON Schema," https://json-schema.org/understanding-json-schema/UnderstandingJSONSchema.pdf, 2020.

[13] IETF, "JSON Schema Draft 2020-12," https://json-schema.org/draft/2020-12/json-schema-core.html, accessed: 2021-19-02.

[14] "Meta Object Facility," https://www.omg.org/mof/, 2021, accessed: 2021-04-30.

[15] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[16] J. Bézivin, "On the unification power of models," *Software & Systems Modeling*, vol. 4, no. 2, pp. 171–188, 2005.

[17] D. Djurić, D. Gašević, and V. Devedžić, "The tao of modeling spaces," *Journal of Object Technology*, vol. 5, no. 8, pp. 125–147, 2006.

[18] J. Bézivin, *Model Driven Engineering: An Emerging Technical Space*. Springer, 2006, pp. 36–64. [Online]. Available: https://doi.org/10.1007/11877028_2

[19] IETF, "JSON Schema Draft 7," https://json-schema.org/draft-07/json-schema-release-notes.html, accessed: 2021-19-02.

[20] "Xtext," https://www.eclipse.org/Xtext/, accessed: 2021-05-01.

[21] Object Management Group, "Object Constraint Language," https://www.omg.org/spec/OCL/2.4, accessed: 2021-05-01.

[22] "JSON Schema DSL," https://github.com/lowcomote/jsonschemadsl.parent/releases/tag/1.0.2, accessed: 2021-15-05.

[23] IETF, "RFC6901," https://tools.ietf.org/html/rfc6901, accessed: 2021-19-02.

[24] ATL, "Atlas Transformation Language," https://www.eclipse.org/atl/, accessed: 2021-05-01.

[25] M. Dalibor, N. Jansen, J. Kästle, B. Rumpe, D. Schmalzing, L. Wachtmeister, and A. Wortmann, "Mind the gap: lessons learned from translating grammars between MontiCore and Xtext," in *Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling (DSM@SPLASH)*. ACM, 2019, pp. 40–49.

[26] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[27] "JsonBasedXtextGrammar," https://github.com/jku-win-se/JsonBasedXtextGrammar, accessed: 2021-14-05.

[28] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," *Journal of Systems and Software*, vol. 123, pp. 176–189, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215001430

[29] B. Gruschko, D. Kolovos, and R. Paige, "Towards synchronizing models with evolving metamodels," in *Proceedings of the International Workshop on Model-Driven Software Evolution*, 2007.

[30] "EMF Compare," https://www.eclipse.org/emf/compare/, accessed: 2021-05-01.

[31] Repository of JSON Schemas, "Schema Store," https://www.schemastore.org/json/, accessed: 2021-05-01.

[32] H. Brunelière, J. Cabot, C. Clasen, F. Jouault, and J. Bézivin, "Towards Model Driven Tool Interoperability: Bridging Eclipse and Microsoft Modeling Tools," in *Proceedings of ECMFA*. Springer, 2010, pp. 32–47.

[33] A. Staikopoulos and B. Bordbar, "A metamodel refinement approach for bridging technical spaces, a case study," in *Proceedings of 4th MoDELS Workshop in Software Model Engineering (WiSME)*, 2005.

[34] P. Neubauer, A. Bergmayr, T. Mayerhofer, J. Troya, and M. Wimmer, "XMLText: from XML schema to xtext," in *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. ACM, 2015, pp. 71–76.

[35] P. Neubauer, R. Bill, D. S. Kolovos, R. F. Paige, and M. Wimmer, "Reusable Textual Notations for Domain-Specific Languages," in *Proceedings o fthe 19th International Workshop in OCL and Textual Modeling (OCl)*, ser. CEUR Workshop Proceedings, vol. 2513. CEUR-WS.org, 2019, pp. 67–80.

[36] J. L. Cánovas Izquierdo, "JSONSchema To UML: Tool to Generate UML diagrams from JSON Schema Definitions," https://modeling-languages.com/jsonschema-uml-tool-generate-diagrams-json/, 2018.

[37] J. L. Cánovas Izquierdo and J. Cabot, "JSONDiscoverer: Visualizing the schema lurking behind JSON documents," *Knowl. Based Syst.*, vol. 103, pp. 52–55, 2016.

[38] L. D. Köhler, "A Model-Driven JSON Editor," Master's thesis, Technische Universität München, 2017.