applied sciences

MDPI

Article

JSON Schemas with Semantic Annotations Supporting Data Translation

Gonçalo Amaro [1], Filipe Moutinho [1,2,*], Rogério Campos-Rebelo [1,2,3], Julius Köpke [4] and Pedro Maló [1,2]

[1] Department of Electrical and Computer Engineering, NOVA School of Science and Technology, 2829-516 Caparica, Portugal; gg.amaro@campus.fct.unl.pt (G.A.); rcr@uninova.pt (R.C.-R.); pmm@fct.unl.pt (P.M.)
[2] UNINOVA-Instituto de Desenvolvimento de Novas Tecnologias, 2829-516 Caparica, Portugal
[3] Polytechnic Institute of Beja, School of Technology and Management, 7800-295 Beja, Portugal
[4] Department of Informatics Systems, University of Klagenfurt, 9020 Klagenfurt, Austria; julius.koepke@aau.at
* Correspondence: fcm@fct.unl.pt

Abstract: As service-oriented architectures are a solution for large distributed systems, interoperability between these systems, which are often heterogeneous, can be a challenge due to the different syntax and semantics of the exchanged messages or even different data interchange formats. This paper addresses the data interchange format and data interoperability issues between XML-based and JSON-based systems. It proposes novel annotation mechanisms to add semantic annotations and complement date values to JSON Schemas, enabling an interoperability approach for JSON-based systems that, until now, was only possible for XML-based systems. A set of algorithms supporting the translation from JSON Schema to XML Schema, JSON to XML, and XML to JSON is also proposed. These algorithms were implemented in an existing prototype tool, which now supports these systems' interoperability through semantic compatibility verification and the automatic generation of translators.

Keywords: arrowhead framework; interoperability; JSON schema; semantic annotations; message transformation; semantic and ontology reasoning; service-oriented architecture; translator automatic generation

Citation: Amaro, G.; Moutinho, F.; Campos-Rebelo, R.; Köpke, J.; Maló, P. JSON Schemas with Semantic Annotations Supporting Data Translation. *Appl. Sci.* **2021**, *11*, 11978. https://doi.org/10.3390/app112411978

Academic Editors: Pal Varga, Daniel Kozma and Felix Larrinaga

Received: 20 September 2021
Accepted: 7 December 2021
Published: 16 December 2021

1. Introduction

Industry 4.0 brings many advantages, and it has moved from fully human control-dependent systems to highly automated environments, where human interaction becomes less and less needed. Service-Oriented Architectures (SOA) are playing a major role since they allow for decentralized systems composed of interoperable systems that form a System of Systems (SoS). Therefore, interoperability between systems and their underlying services becomes the most important piece of the SoS. In a green-field approach, all of the systems can be built from scratch and interoperability can easily be achieved. However, such an approach is hardly possible since every machine or product has its own lifecycle, and manufacturing processes might need to be restructured based on existing devices and services [1].

Efficiency is a major aim in industry, and it certainly pays off to replace machines for new technologies or to restructure current processes to enhance processes or products. However, how does one integrate these new machines/new interoperable systems into the SoS that was previously built? The Arrowhead framework [2] facilitates the integration and orchestration of systems, whereas the framework knows which systems/service providers and consumers exist on the SOS, making it loosely coupled. The framework provides the systems with a late-binding, whereas a service consumer, upon requesting a service to the framework, does not know which service provider it is using until the framework couples them at runtime. However, the machines probably have different syntax on the messages

sent or even a different communication language (semantic and syntactical interoperability issues [3]). It can be a management subject to have all of these interoperability issues in mind for every new interaction, or it can be optimized to automatically translate every provider/consumer pair. However, how can these semantic and syntactical interoperability issues be solved automatically?

A solution for semantic and data interoperability has been proposed in [4,5], which consists of automatically generating a translator for different XML (Extensible Markup Language) speaking machines (provider-to-consumer communication) based on metadata inserted on their schemas in the form of semantic annotations. Semantic annotations are inserted following the SAWSDL (Semantic Annotations for Web Services Description Language) standard [6], using the extension attribute "modelReference" with annotation paths [7]. SAWSDL is a specification that allows for introducing semantic annotations into the WSDL or XML Schema (XSD), whereas annotation paths enhance their expressiveness. In [5], methods to group semantic annotations and to add complement data values were proposed to solve ambiguities and to provide additional data required by the consumers. Additionally, in [5], a tool named TAG-Tool was used to verify XML-based system compatibility as well as to automatically generate translators to support their interaction.

What if the messages that are sent and that can be received by machines, besides having different parameter names and/or structures (although having similar semantics), also have different data interchange formats, for example, JSON (JavaScript Object Notation) and XML; how does one solve this syntactical interoperability issue? Generic translations from JSON to XML and vice-versa do not ensure that provider messages are translated to messages that the consumers can understand. To ensure this, generating custom translators based on their schemas is required. To provide meta-data about JSON messages and XML messages, annotated JSON Schemas and XML Schemas should be used. Thus far, there is no standard or recommendation method for inserting semantic annotations or complement data values in JSON Schemas, such as in XML Schemas.

This paper has two main contributions: methods for adding semantic annotations and complement data values to JSON Schemas, and a set of algorithms that support data and meta-data translation. We propose an algorithm to translate annotated JSON Schemas to annotated XSDs and algorithms to convert from JSON to XML and from XML to JSON. These algorithms were implemented in the TAG-Tool [5], which now automatically generates translators to support the data translation between two systems that can use the same data format (both using JSON or XML) or not (provider using JSON/XML and consumer using XML/JSON).

This article is structured into nine sections. After the Introduction, the related work is presented in Section 2, followed by an introduction on the Arrowhead Framework and the original TAG-Tool (Section 3). For extending the expressiveness of legacy systems using JSON, a method for introducing semantic annotations and complement data values to JSON schemas is proposed in Section 4. A translation method for translating JSON Schemas to XML Schemas is presented in Section 5, and JSON to XML and XML to JSON translation methods are presented in Section 6. The previous version of the TAG-Tool was extended with the use of the proposed annotation mechanisms and translation methods, as presented in Section 7; the new TAG-Tool architecture was introduced, bringing data translation supported by JSON Schemas and automatic translator generation to support the translation of JSON to JSON, JSON to XML, and XML to JSON messages. The evaluation of the work is presented in Section 8, whilst the conclusions on the proposals are presented in Section 9.

2. Related Work

2.1. Background

Web Services [8,9] provide syntactic interoperability [3] and allow developers to easily build systems based on various existing services. Such services can be arbitrary applications, including IoT devices. However, due to heterogeneous message formats and

the lack of semantic interoperability [3], the integration of services is typically a manual task. Human developers infuse their knowledge with creating the correct mapping between applications and services in their code. In a setting such as the Arrowhead Framework, this is an unacceptable burden, and methods for the seeming less interoperation are required. A key feature is to automatically translate messages in the format provided by some service to the format required by some consumer (or vice versa). However, creating such translators requires knowledge about the semantics of the exchanged messages. The works in [4,5] present a method for providing the required semantics within the Arrowhead Framework by using lightweight ontology-based semantic annotations. There is a large body of research on the annotation of instance documents with approaches such as [10–13]. In contrast, we apply semantic annotations on the schema level. Schema-level annotations have the advantage that existing services (e.g., IoT devices such as temperature sensors) do not need to be changed. Instead, only annotated schemas for the message format need to be provided. However, the existing approach within the Arrowhead project [4,5] is restricted to XML and the XML Schema and does not yet support the annotation and translation of JSON [14]-based messages, which we tackle in this paper.

The annotation method for the XML Schema applied within the Arrowhead Framework is based on SAWSDL [6] and Annotation Paths [7,15–17]. SAWSDL is a W3C recommendation for the semantic annotation of Web Service Descriptions (WSDL [18]) and XML Schema [19]. SAWSDL allows us to include semantic annotations within the schema by using additional attributes for schema nodes. For mapping XML instance documents to ontologies, the two extension attributes lifting mappings and lowering mappings are used. Both refer to URIs containing scripts in arbitrary languages (typically XSLT or XQuery) that either translate the corresponding XML fragments of schema instances to a semantic representation such as RDF (lifting mapping) or that translate the semantic representation back to the XML representation (lowering mapping). This approach was proposed by the W3C for the enactment of services. It is very expressive, but it comes with a toll on scalability, maintainability, and ease of use for annotators. A more declarative and lightweight approach is provided by SAWSDL's ModelReference extension attribute. Model References connect XML Schema elements, types, or attribute declarations with named semantic concepts of some semantic model. Model References were proposed to assist Service Discovery and not primarily for the translation of messages or enactment of services. However, these lightweight annotations have several benefits: They can easily be applied by developers and provide benefits in terms of scalability and maintainability. Following a traditional matching and mapping workflow in the spirit of the Clio System [20], they can also be used for the generation of transformations scripts with existing mapping systems [21,22] based on the foundations of data exchange [23]. However, the restriction of annotations to named concepts of a reference ontology comes with a toll on expressiveness. It requires an ontology containing one concept for each (potentially) annotated schema element. Such an ontology unlikely exists and therefore needs to be created during annotation, leading to an additional burden for annotators. To overcome this issue, Annotation Paths [7,15–17] were proposed. When annotating a schema with Annotation Paths, path expressions are inserted as values for Model Reference attributes. These path expressions are composed of ontology concepts and properties of OWL [24] ontologies. Matching approaches for annotation paths [17,25] automatically translate the path expressions to complex description logic concepts and perform the matching based on ontological relations between these concepts. In [17] Annotation Path based matching was shown to provide superior matching quality compared with model references pointing to single concepts of a reference ontology and with traditional matching approaches such as [26–31].

Annotation paths were initially used as an additional source of knowledge for matching systems. Therefore, matchers such as [17,25] apply heuristics based on the structure of schemes or annotations to resolve ambiguities between semantically identical schema elements. Annotation paths can directly resolve a wide range of schema mismatches, as classified in [32]. However, mismatches requiring value transformations or the intro-

duction of additional values are resolved generically using so-called matching templates. Matching templates contain arbitrary data transformation functions with annotated inputs and outputs.

An extended variant of the original Annotation Paths [7], following the rules described in [33], was proposed for the Arrowhead Framework [5]. It allows for resolving typical mismatches between IoT devices solely on the annotation level without requiring annotation templates or heuristics. In particular, the following extensions were integrated:

Complement Data Values: In the IoT domain, often, specific static data values (e.g., the location of a sensor) are relevant for some consumers but are not included in the messages provided by the services. Such coverage mismatches [32] are addressed in [4,5] by providing these data within the annotated schema of the sensor.

Group identifiers: Ambiguities between schema elements with the same annotations are addressed in [4,5].

2.2. Schema-Level Annotations and Annotations of Web Services

Other declarative annotation methods for message schemas include works such as [34] and [35]. The work in [34] proposes semantic annotations for RDFs Schemas to generate lifting/lower mappings for the translation of RDF instances to/from instances of the reference Ontology. The structure of the annotations is similar to the basic structure of the Annotation Path. An external annotation method for DTDs rather than XML Schema is presented in [35].

Traditional XML-based Web Services use the SOAP protocol [8] and are based on the XML Stack. Message formats for inputs and outputs are defined using XML Schema. Descriptions about possible operations, inputs, and outputs can be provided using WSDL descriptions. These services provide machine-readable descriptions supporting automatic late-binding. However, the semantic interoperability issue is not solved. Therefore, methods for their semantic annotation were introduced with methods such as SAWSDL [6], OWL-S [36], and WSMO [37].

While XML-Based Web Services can provide arbitrary remote procedure calls, REST [9]-based Web Services are restricted to implementations following the REST principles. While in general not being restricted to specific message formats, REST services today typically use the JavaScript Object Notation [14] or short JSON as their message format. These services typically lack an explicit schema. Service descriptions are limited to human-readable HTML documentations. Since these documents are not machine-readable, various efforts for the semantic annotation of these documents using instance-level annotations [13] were made in approaches such as [38–42].

An alternative approach for solving semantic interoperability issues of services is to annotate JSON messages rather than message schemas or service descriptions. Such annotations can be realized with the JSON-LD [43] standard. JSON-LD documents can directly be translated to RDF and thus allow for seamless integration with the world of linked data. While JSON-LD should resolve many interoperability issues of Web Services in the future, it cannot easily be integrated into existing collaborations as addressed by the Arrowhead Framework, where existing and immutable services provided by IoT devices exchanging plain JSON or XML messages need to interoperate. However, interesting future work is the automatic translation of messages of these services to JSON-LD documents based on the proposed annotation method.

Finally, a third variant for adding semantics to REST-based services or APIs was proposed in [44]. It aims to describe inputs, outputs, preconditions, and effects (OWLS-Like IOPEs) in the form of specific JSON-LD documents. For defining inputs and outputs, a 1:1 relation between JSON keywords and ontology concepts or properties is assumed. Since this assumption is hardly realistic also, methods for mapping service-specific ontologies to other ontologies are proposed. However, the paper does not cover message transformations, and there is no indication that the approach can provide the required knowledge for the generation of transformation in the case of non 1:1 cases. The approach cannot be applied

to the Arrowhead framework as we seek for the inter-operation of all combinations of JSON and XML-based services.

To the best of our knowledge, there is no approach for the semantic annotation of message schemas for JSON messages with the appropriate expressiveness for supporting the automatic generation of translators between heterogeneous systems or services. Therefore, we propose an annotation method for JSON Schema based on an expressive and still light-weight annotation method for XML Schemas. We apply the approach in the Arrowhead Framework, supporting the successful interoperation of heterogeneous services no matter if they require JSON or XML messages.

3. Arrowhead Framework and TAG-Tool

Arrowhead Framework [2] is a SOA framework addressing IoT (Internet of Things)-based automation. It was created in the Arrowhead project [45] and continued being developed under other European projects as Productive4.0 project [46] and Arrowhead-Tools project [47] (ongoing project). Arrowhead framework supports the interaction between application services, supported by a set of core services included in the framework. These core services allow the Arrowhead Framework to support the exchange of information between application services. It helps a consumer find a provider that provides a particular service that the consumer needs. The main Core Services, also presented in Figure 1, are as follows:

- **Service Registry (SR)**, which allows a provider to register a certain service that becomes available in a framework;
- **Service Discovery (SD)**, which allows a consumer to find the available service instance(s);
- **Orchestration (ORC)**, which allows a consumer to find which service instance(s) it shall consume; and
- **Authorization (AUT)**, which permits the provider to verify that the consumer is allowed to consume the required service.

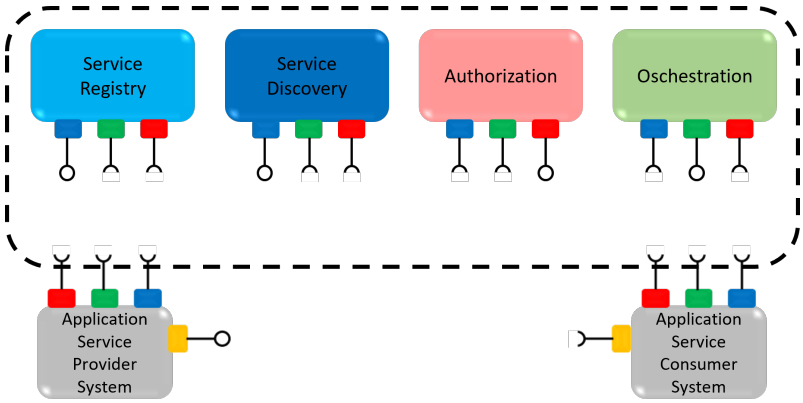

Figure 1. Arrowhead framework core services and application services.

A method in which a consumer and a provider can communicate through the Arrowhead framework can be described in the following six steps. A provider starts communicating with the Service Registry to register a service **(1)**. This communication makes that service available in the Arrowhead framework and informs Arrowhead that this provider provides that service. When a consumer needs a service, it communicates with Service Discovery **(2)**, asking if there are providers that offer that service. If it receives a positive response, the consumer can communicate with the orchestration **(3)**. Next, the orchestration analyzes all providers that offer the service and selects which one is better to offer the desired service to this specific consumer. The consumer already has all of the information that allows it to interact directly with the provider. Therefore, the consumer sends a direct request message to the provider's address **(4)** asking for the service. When the provider

receives the request from the consumer, it may send a message to the authorization asking if it can communicate with the consumer **(5)**. If yes, the provider answers the consumer request, sending a direct message to the consumer with the information related to the requested service **(6)**. All of this communication works properly if the consumer and the provider can understand each other. However, if they do not understand each other, then the direct exchange of information between them is impossible. In such cases, the orchestration would have to consider which systems manage to understand each other, or else, the consumer would have to ask the arrowhead framework for new providers until it finds one with which it could communicate.

The introduction of a translator in the communication between the provider and the consumer (Figure 2) may enable the interaction between heterogeneous systems, allowing them to understand each other. However, this solution presents a problem. It is necessary to provide translators between all consumer/provider pairs. This should not be accomplished in a tedious, cost-intensive, and error-prone manual task. To deal with this problem, an approach that supports the automatic generation of translators, based on their annotated metadata and a reference ontology, was proposed in [4,5]. This approach is presented in Figure 3. The algorithm analyzes the metadata from the consumer and the provider, semantically annotated with a reference ontology and—if possible—returns an automatically generated translator. A tool was developed (available online at http://gres.uninova.pt/tag/ accessed on 14 September 2021, based on this approach. The tag-tool is a semantic interoperability-focused tool that is capable of enabling the communication between semantically compatible and syntactically incompatible XML providers and consumers. It gives them syntactical interoperability through a generated XSLT translator, based on its metadata with semantic annotations to a reference ontology [4]. The execution of this tool is described in [5] and follows the architecture described in Figure 3.

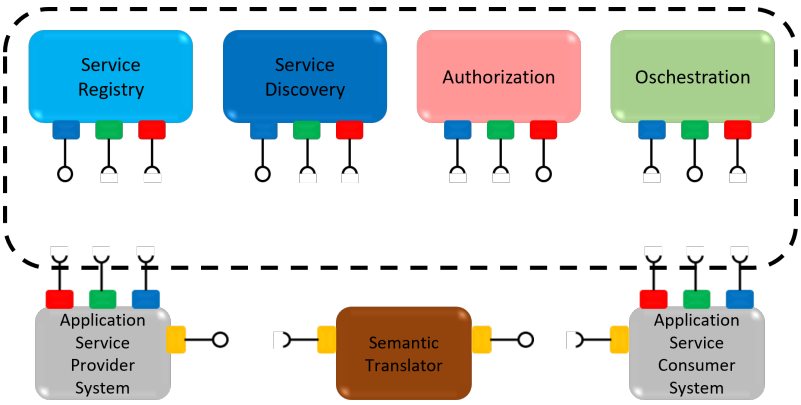

Figure 2. A translator supporting the interaction between heterogeneous applications services.

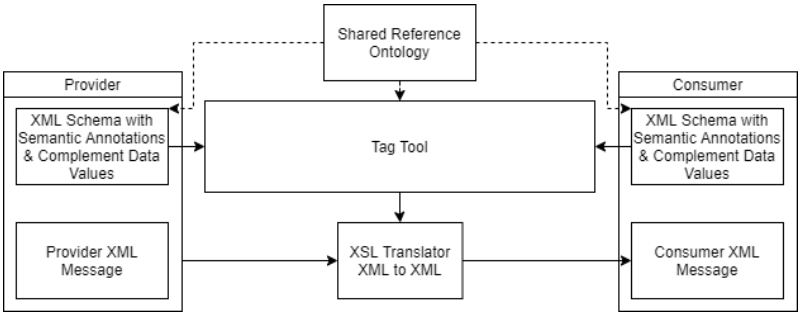

Figure 3. System architecture.

An up-to-date overview of the annotation rules that allows for a smooth behavior of this translation tool prototype was proposed in [33] and should be followed to ensure the success of provider–consumer translations.

Considering that systems (application services) often exchange JSON messages and that the TAG-Tool, as previously described, only supports interoperability between systems that exchange XML messages, our proposals enable the TAG-Tool to support the interaction between systems that exchange messages in these two different data formats (JSON and XML). The possibility to introduce semantic annotations in the JSON Schemas, which feeds the extended TAG-Tool, provides the required semantic information about the JSON messages that are sent and expected to be received to automatically generate the required message translators.

Given that the TAG-Tool already supported the translation between heterogeneous XML messages based on XML Schemas and that the goal was to support the translation between both XML and JSON heterogeneous messages, a set of translation algorithms are proposed in this paper. These algorithms enable translations from JSON Schema to XML Schema, JSON to XML, and XML to JSON. The JSON Schema to XML Schema translator was created considering the proposed annotation method for JSON Schemas and the annotated XML Schemas required by the TAG-Tool. The JSON to XML and XML to JSON translator algorithms were created to ensure that the translated messages are compliant with their schemas. For example, in many of the available XML to JSON translators, JSON values are always written within quotes since there is no reference to what the XML element datatype is, whereas the proposed algorithm generates translators based on the associated JSON Schemas, writing JSON values with or without quotes based on their data types, which are specified in the schemas.

It is important to note that our proposals (the annotation mechanism and algorithms) may have a broader application, beyond the TAG-Tool or Arrowhead Framework environment.

4. Proposed Annotation Mechanisms for JSON Schemas

JSON Schema aims to define the structure of JSON data [48], supporting its documentation and validation, among others. An example of JSON data (a JSON instance) is presented in Listing 1, and its schema is presented in Listing 2. Based on these listings, the information that can be extracted is the structure and data types of the document, and one might even suspect that the "units" property refers to a temperature unit from a temperature sensor "OutTempSens" because of our human semantic interpretation of the properties abbreviation. However, will a machine/system be able to interpret with such acuity?

Listing 1. An example of a JSON Instance.

```
{
    "datavalues": {
        "units": "Cel",
        "OutSenstemp": 23
    }
}
```

Listing 2. JSON Schema.

```
{
  "$schema": "http://json-schema.org/draft-07/schema",
  "type": "object",
  "properties": {
    "datavalues": {
      "type": "object",
      "properties": {
        "units": {
          "type": "string"
        },
        "OutSenstemp": {
          "type": "number"
        }
      }
    }
  }
}
```

Systems need domain knowledge and reasoning capabilities such as humans to meaningfully extract the data from the documents. For a correct interpretation of semantic information, a previous knowledge base needs to exist, exactly in the same way as for us, humans, who can access our knowledge and associate certain keywords with certain concepts. A way of giving this knowledge to the systems is by providing them with a reference ontology, and this ontology shall have all of the required concepts and properties. By sharing this reference ontology amongst all interoperable systems, it is possible to achieve common knowledge for the SOA. However, in addition to a shared understanding of the domain, meta-data mapping property names to ontology elements are also required since the message schemas are typically developed independently from potential ontologies or even the domain of the SOA. One way to have this metadata is with the introduction of semantic annotations that refer to the shared reference ontology. If, i.e., the "OutSenstemp" property is properly annotated, the system can access the ontology and identify it as an outdoor temperature sensor.

4.1. JSON Schema with Semantic Annotations

In this paper, we propose semantic annotation mechanisms to add semantic annotations to JSON Schemas. SAWSDL is a W3C recommendation for adding semantic annotations to WSDL and XML Schemas [6]; however, for JSON Schemas, there is no recommendation (or standard) for adding such annotations. There are open topics [49] on this subject, where the use of an attribute named "modelReference" is informally proposed, in analogy to SAWSDL, to introduce annotations pointing to semantic concepts. In this paper, we follow this route but we work out the required details for successfully annotating the schemas.

The semantic annotation can be inserted into the JSON schema by putting the semantic annotation in the property itself, with the keyword "modelReference". Since we aim to annotate data-carrying leaf-nodes, we support the annotation of the following JSON-Schema properties:

- "string";
- "integer";
- "number";
- "boolean";
- "array".

It was not considered the "object" property, since the annotation method is used for annotating data carrying nodes, similar to the previous works for XML Schema [4,5,33]. It was also not considered upper-level elements because this may lead to problems if the structure of the ontology and the schema differs. Therefore, the complete semantics is defined on the leaf level, using expressive annotation paths and group identifiers. This extension of

the JSON Schema is backward compatible since JSON validators should ignore unknown keywords, giving the freedom to define additional keywords for custom implementations.

In general, the property values of the "modelReference" keyword are not limited to a specific format. However, we propose the use of an extension of annotation paths as applied in the Arrowhead Framework [5]. An example of a JSON schema with the attribute "modelReference" is shown in Listing 3.

Listing 3. JSON Schema with semantic annotations.

```
{
  "$schema": "http://json-schema.org/draft-07/schema",
  "type": "object",
  "properties": {
    "datavalues": {
      "type": "object",
      "properties": {
        "units": {
          "type": "string",
          "modelReference": "/TemperatureSensor/hasTempUnits/TemperatureUnits"
        },
        "sensor1temp": {
          "type": "number",
          "modelReference": "/TemperatureSensor/hasDecValue"
        },
        "sensor2temp": {
          "type": "number",
          "modelReference": "/TemperatureSensor/hasDecValue"
        }
      }
    }
  }
}
```

4.2. JSON Schema with Complement Data Values

When two heterogeneous systems interact through JSON messages, the consumer not only has to understand the provider messages, which can be supported by the proposed semantic annotations, but also has to receive all of the required information. However, when these systems are not custom-built, the transmitted information may be insufficient. For example, when integrating a legacy temperature device (the provider system), which sends JSON messages only with temperature values and units, is required, a consumer system needs to know not only the temperature value but also the location of that temperature sensor.

Besides receiving all of the required information, the consumer needs to understand the provider's data values syntax. For example, a consumer expects to receive, as temperature units, a string with the value "Cel" (Celsius) or "Fahr" (Fahrenheit), but the provider uses a different syntax ("C" or "F").

To address the presented interoperability issues, in this paper, an annotation mechanism is also proposed to add complement data to JSON Schemas similar to the existing one [5] for XML Schema, enabling interactions between heterogeneous systems. We extend JSON Schemas with A3ST attribute names. A3ST is a proposal for adding complement data values to an existing XML Schema [4] that was extended in [5]. This information can be useful for facilitating compatibility between heterogeneous systems, broadening the range of systems/devices that can communicate with each other.

The proposal for inserting such annotations on a JSON Schema is by creating an a3st array at the JSON Schema root level, which will have as properties the following objects:

- **"data-property-value"** for adding complement data;
  - Property "property" is the semantic annotation of the OWL datatype property to be inserted;
  - Property "value" is the value of the property attribute;
- **"map-data-ind"** for mapping a concrete syntax which is used by a certain provider or consumer;

- Property “individual” identifies the individual of a concept;
- Property “mdi-id” defines an id for each element, it is a reference attribute;
- Keyword “mdi-ref” makes a reference for the "map-data-ind" element with the matching “mdi-id”.

A complete annotated schema for the instance in Listing 1, which is from a provider device, is shown in Listing 4. As exemplified before, the provider device only sends a temperature value and a string with the temperature units, whereas let us assume that a consumer system also needs location values as well as to understand the syntax of the units. Focusing on the temperature unit property, assuming that the provider sends a string with the value “Cel” (Celsius) or “Fah” (Fahrenheit) but the consumer expects a “C” or “F”, two things are needed: an annotation that relates the output syntax with the ontology meaning and a reference from the property to that annotation. As proposed, this annotation is a “map-data-ind” since it maps the syntax according to its ontology meaning. In lines 5 and 12 of Listing 4, two possible outcomes are shown (these could be as many as needed). The items of lines 5 and 12 have a property called “a3st:mdi-id”, of which the purpose is to be referenced later; have a property called “a3st:individual” pointing to the ontology individual, bringing a known term for all systems; and have a property called “a3st:node-value” which is the string that can be expected from the current system. While having an annotation describing the meaning of a string output, it needs to be related to the property that will send this value. In line 42, the keyword “mdi-ref” points to every possible outcome of that property (“units”). All of the links exist to interpret as many pre-set strings as the system might have. Regarding the location that is required by the consumer but not provided by the provider, the creation of an object is proposed to add complement data to the schema: “data-property-value”. As it is shown in lines 19 and 25 of Listing 4, these two objects describe the location coordinates of the system. By the use of the object property “a3st:property”, a semantic annotation to the property meaning can be inserted, and by the use of the property “a3st:value”, the value of the property is inserted, having a fully defined property and value.

Listing 4. Complete Annotated JSON Schema.

```
{
    "$schema": "http://json-schema.org/draft-07/schema",
    "a3st": [
        {
            "a3st:map-data-ind": {
                "a3st:mdi-id": "1",
                "a3st:individual": "Fahrenheit",
                "a3st:node-value": "F"
            }
        },
        {
            "a3st:map-data-ind": {
                "a3st:mdi-id": "2",
                "a3st:individual": "Celsius",
                "a3st:node-value": "C"
            }
        },
        {
            "a3st:data-property-value": {
                "a3st:property":"/TemperatureSensor/hasLocation/Latitude[hasLocUnits/
                        LocationUnits/hasIndividual /DD]/hasDecValue",
                "a3st:value": "38.659963"
            }
        },
        {
            "a3st:data-property-value": {
                "a3st:property":"/TemperatureSensor/hasLocation/Longitude[hasLocUnits/
                        LocationUnits/hasIndividual/DD]/hasDecValue",
                "a3st:value": "-9.203966"
            }
        }
    ],
    "properties": {
        "datavalues": {
            "type": "object",
            "required": [
                "units",
                "outSensTemp"
            ],
            "properties": {
                "units": {
                    "type": "string",
                    "modelReference": "/TemperatureSensor/hasTempUnits/TemperatureUnits",
                    "a3st:mdi-ref": "1;2"
                },
                "outSenstemp": {
                    "type": "number",
                    "modelReference": "/OutdoorTemperatureSensor/hasDecValue"
                }
            }
        }
    }
}
```

5. Translating JSON Schema to XML Schema

While there are various approaches for the translation of plain JSON Schemas to XML Schemas, such as [50], our architecture within the tag tool requires the translation of annotated JSON Schemas to annotated XML Schemas.

5.1. Mapping Data Types

There may be incompatibilities between JSON Schemas and XML Schemas regarding the data types. XSD's and JSON Schemas do not share all data types, with the latter being a more general, less detailed implementation. A datatype conversion mapping, as shown in Table 1, was established for making the translation possible. This table has the opposite direction of the table presented in [51], which proposes the translation from XSD to JSON Schema data types.

Table 1. JSON Schema to XML Schema data type compatibility rules.

JSON Schema	XML Schema
• String	• **String**
• integer	• Integer • Byte • **Long** • Short
• number	• Float • Decimal • **Double**
• boolean	• **Boolean**
• Array	• **maxOccurs = "Unbounded"**
• object	• **xs:complexType, xs:all**

JSON objects have a keyword that specifies the **required items** to be in the JSON instance. There is not a direct translation for this property to XSD since, in XML Schema, this is specified for each element as the minimum and the maximum number of occurrences. The translator must verify for each created element if it is present in the required array of the JSON Schema.

5.2. Algorithm

To support the translation of JSON Schemas, with the proposed semantic annotations and complement data values, to XML Schemas, the algorithm presented in Listing 5 is used.

For developing the algorithm, JSON schema characteristics were reviewed and resumed in the following assumptions, which should not limit the implementation in most applications:

1. There are no empty JSON objects, and they always contain JSON properties, which translates into an XSD complex type element with child elements of itself;
2. Any JSON property that does not have a "type" keyword or of which the property is not an object is considered a new XSD element. All keywords on this property are attributes of the XSD element.

Based on these assumptions, we developed an algorithm for translating JSON Schemas to XML Schemas. The pseudo-code of the algorithm is shown in Listing 5. The algorithm receives an annotated JSON Schema and the data type compatibility mapping shown in Table 1 as input and generates an annotated XML Schema as output. First, an empty XML-Schema 1.1 document with all required prefixes ("xs", "vc", "sawsdl", and "a3st") is created. Then, the function *schemaMaker*() recursively traverses the input JSON Schema and incrementally extends the output XML Schema. If the current JSON Schema node processed by *schemaMaker*() is of the type object, then a JSON Schema object declaration is recursively translated to an XML-Schema element containing a complex type. Otherwise, the current node is a data-carrying leaf-node and the output is generated based on the mapping table.

In particular, JSON Schema objects are translated as follows:

Based on Assumption 1, whenever a processed node is of type object, a new XML Schema element with the same name as the processed JSON node is created. Then, an anonymous XML Schema *complexType* element is appended as a child node of the new element. This can be seen from line 10 to line 20. In a next step, an XML Schema *ALL*

element is nested into the newly created complex type. *ALL* (rather than *SEQUENCE* or *CHOICE*) is used due to JSON Schemas' lack of validation of the order of elements within an object. If the currently processed JSON node is not required in a JSON Schema, a min-occurs property with the value "0" is added to the created XML-Schema Element. Note that JSON Schema and XML Schema follow different strategies for expressing required values. In XML Schema, the property is expressed using the min-occurs property of the element itself. In JSON Schema, the required properties are defined at the object level using an array of required properties within the current object. Therefore, the helper array *requiredArrayNew* is used for book-keeping purposes.

Finally, all of the properties of the currently processed node are traversed recursively. In all detail, data-carrying leaf-nodes are translated as follows:

Based on assumption 2, whenever a property does not have its type defined as an object, or even a "type" keyword, it is assumed to be a new XSD element and proceeds to create it (lines 22 to 49). It creates an XSD element with an element name equal to the JSON data-carrying leaf-node name, and for each keyword, it translates it into an XSD attribute, respecting the following:

- If the keyword is "modelReference", it creates an attribute with a name equal to "sawsdl:modelReference", of which the value will be the semantic annotation.
- If the keyword "type" has a value of "array", it will be translated according to the mapping table. In this case, there can be an "items" keyword at the same level with the array property data type; if this is the case, the content of the "items" keyword is analyzed and the datatype is translated according to the mapping table.
- Any other keyword is treated according to the mapping table in Table 1.

This is shown from lines 24 to 45, whereas the use of the mapping table, which is used for translating the "type" keyword with "string", "integer", "number", "boolean", "array", and "object", can be seen in line 43. After all attributes are created for the XSD element, the required array is inspected to evaluate the need for a minOccurs attribute with a value of 0. If the current property name is not in it, the minimum number of occurrences is set to 0.

A similar approach was taken for translating the complement data proposed in [5]; however, since the number of levels of this JSON data, as proposed in Section 4.2, is known, it is not needed to navigate through it recursively. If there are complement data in the JSON schema, the "annotation" and "appinfo" elements are added to the document so the data can start being added (lines 56 to 57). For each item of the JSON array with the complement data objects, an XSD element is created and completed with attributes or node values, according to the JSON object properties (lines 58 to 67). To insert these attributes, the JSON object on the item is looped, extracting and inserting the JSON properties one at a time as XSD attributes or node values. It is verified if the property name equals "a3st:node-value", and if so, this property value is directly inserted as the element node value; otherwise, an attribute is created with the property name and an attribute value is set with its value (line 60 to 66).

Listing 5. JSON Schema to XML Schema translator algorithm.

```
LOAD compatibility table
LOAD JSON Schema to JSON Object
CREATE empty XSD
CREATE and ADD to XSD the root element with namespaces XS, VC, SAWSDL and A3ST and schema version attribute
CALL schemaMaker (JSON Object, XSD root element, "root", NULL)
CALL a3stMaker (JSON Object, XSD root element)
SAVE xsd

FUNCTION schemaMaker (JSONnode, XSDnode, NODEname, JSONrequiredArray)
    IF JSONnode has "type" keyword AND "type" is "object" THEN
        CREATE and ADD to XSDnode an element with name NODEname
        CREATE and ADD to NODEname element an element with name "complexType"
        CREATE and ADD to "complexType" element an element with name "all"
        CREATE requiredArrayNew equal to NULL
        IF JSONnode has "required" keyword THEN
            ADD required properties to requiredArrayNew
        END IF
        FOREACH property on JSONnode
            CALL schemaMaker (property.node, "all" element, property.name, requiredArrayNew)
        END FOREACH
    ELSE
        CREATE and ADD to XSDnode an element with name NODEname
        FOREACH keyword in JSONnode
            IF keyword equals "modelReference" THEN
                CREATE and ADD to XSDnode an attribute with name "sawsdl:modelReference" and value
                equals to keyword
            ELSE IF keyword equals "items" THEN
                IF "type" is "object" THEN
                    CREATE and ADD to NODEname element an element with name "complexType"
                    CREATE and ADD to "complexType" element an element with name "all"
                    CREATE requiredArrayNew equal to NULL
                    IF JSONnode has "required" keyword THEN
                        ADD required properties to requiredArrayNew
                    END IF
                    FOREACH property on JSONnode
                        CALL schemaMaker (property.node, "all" element, property.name, requiredArrayNew)
                    END FOREACH
                ELSE
                    CREATE and ADD to XSDnode an attribute with name and value according to the mapping
                    of the "items" content keyword in the compatibility table
                END IF
            ELSE
                CREATE and ADD to XSDnode an attribute with name and value according to the mapping
                of the keyword in the compatibility table
            END IF
        END FOREACH
        IF requiredArray does not have the NODEname THEN
            CREATE and ADD to XSDnode an attribute with name "minOccurs" and value "0"
        END IF
    END IF
END FUNCTION

FUNCTION a3stMaker (JSONnode, XSDnode)
    FOREACH keyword in JSONnode
        IF keyword equals "a3st" THEN
            CREATE and ADD to XSDnode an element with name "annotation"
            CREATE and ADD to "annotation" element an element with name "appinfo"
            FOREACH item in a3st keyword array
                CREATE and ADD to appinfo element an element named with item name (new a3st element)
                FOREACH property in item
                    IF property name equals "a3st:node−value" THEN
                        ADD property value as the node value of the new a3st element
                    ELSE
                        CREATE and ADD to new a3st element an attribute with the property name and value
                    END IF
                END FOREACH
            END FOREACH
        END IF
    END FOREACH
END FUNCTION
```

A result of this algorithm applied to the JSON schema presented in Listing 4 is shown on Listing 6.

Listing 6. Translated XML Schema.

```
<?stringstylexml stringstyleversion="1.0" encoding="UTF-8" stringstylestandalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:a3st="http://gres.uninova.pt/a3st" xmlns:sawsdl="http://www.w3.org/ns/sawsdl">
  <xs:element name="root">
    <xs:complexType>
      <xs:all>
        <xs:element name="datavalues">
          <xs:complexType>
            <xs:all>
              <xs:element name="units" type="xs:string" sawsdl:modelReference="/TemperatureSensor/hasTempUnits/TemperatureUnits" a3st:mdi-ref="1;2"/>
              <xs:element name="outSensTemp" type="xs:double" sawsdl:modelReference="/TemperatureSensor/hasDecValue" />
            </xs:all>
          </xs:complexType>
        </xs:element>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:annotation>
    <xs:appinfo>
      <a3st:map-data-ind a3st:individual="Fahrenheit" a3st:mdi-id="1">F</a3st:map-data-ind>
      <a3st:map-data-ind a3st:individual="Celsius" a3st:mdi-id="2">C</a3st:map-data-ind>
      <a3st:data-property-value a3st:property="/TemperatureSensor/hasLocation/Latitude[hasLocUnits/LocationUnits/hasIndividual/DD]/
hasDecValue" a3st:value="38.659963"/>
      <a3st:data-property-value a3st:property="/TemperatureSensor/hasLocation/Longitude[hasLocUnits/LocationUnits/hasIndividual/DD]/
hasDecValue" a3st:value="-9.203966"/>
    </xs:appinfo>
  </xs:annotation>
</xs:schema>
```

6. Translating JSON to XML and XML to JSON

In this section, two algorithms are presented. The first algorithm translates JSON to XML with the support of an XSLT 3.0 function that generates a raw XML. The second algorithm, which translates XML to JSON, is dependent on the XML schema to ensure valid JSON data.

6.1. Translating JSON to XML

This algorithm supports the translation of JSON instances to XML instances based on the XSLT 3.0 built-in function JSON-to-XML [52]. Since this built-in method does not directly return XML documents compliant with the generated XML Schema (Section 5), we rewrite the outcome by applying an additional algorithm to it. An example output of the XSLT 3.0 built-in function is shown in Listing 7.

Listing 7. JSON-to-XML function result for Listing 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<map xmlns="http://www.w3.org/2005/xpath-functions">
    <map key="datavalues">
        <string key="units">Cel</string>
        <number key="OutSensTemp">23</number>
    </map>
</map>
```

The algorithm for rewriting the output of JSON-to-XML to a valid schema instance is shown in Listing 8. Since the goal of the translator is to be implemented in XSLT, the algorithm is oriented towards that logic. The JSON document is translated into XML format (as in the example presented in Listing 7) and stored in "XMLnode", as shown in line 1. This XML node will be the object of translation for this algorithm. Since it is an XSLT translator and the output method was chosen to be text for facilitating a dynamic translation, XML start tags and end tags are written on the go. Primarily, a root element start tag is written without any verification of the document and a builder function for the XML is called with the translated XML mapping as a parameter (lines 2 to 4). The createXML function is a recursive function that calls itself for every new node that has not been parsed.

The algorithm checks if the current node is an array. If it is not an array, a verification is performed to ensure that the current node is a valid one for translation (line 9). One example of a disposable node that can be in this category is the node on line 2 of Listing 7; this node is declaring a namespace that will not be used, so it has no useful content for our translation; therefore, the algorithm ignores it and calls the function with the child node (line 19). As soon as it is a valid node (with a "key" attribute), the process of translating an XML node is started: a start tag is written using as name the value of the "key" attribute of

the node (line 10): if the node has child elements, then the function is called once again for creating the child elements; if not, it is assumed that the node has a node value and its value is written as plain text after the start tag (line 11 to 16). After writing the node value or all child elements of the currentNode, the end tag is written (line 17). If the current node is an array, it can contain elements or complex type elements. The former is created from line 22 and the latter is from line 28. After finishing an XML element, if there are sibling elements to the currentNode, the function calls itself for its sibling node (line 36 to line 38).

Listing 8. JJSON to XML translator algorithm.

```
1  LOAD to XMLnode the result from JSON-to-XML function
2  CREATE empty XML
3  WRITE root element start tag
4  CALL createXML(XMLnode)
5  WRITE root element end tag
6
7  FUNCTION createXML(currentNode)
8      IF currentNode is not array THEN
9          IF currentNode has content THEN
10             Write start tag with name = attribute value of currentNode
11             IF currentNode has child elements THEN
12                 CALL createXML(currentNode.child)
13             END IF
14             IF currentNode has no child elements
15                 Write currentNode.nodeValue
16             END IF
17             Write end tag with name = attribute value of currentNode
18         ELSE IF currentNode has child elements THEN
19             CALL createXML(currentNode.child)
20         END IF
21     ELSE
22         IF array of objects THEN
23             FOREACH child of currentNode
24                 Write start tag with name = attribute value of currentNode
25                 CALL createXML(child)
26                 Write end tag with name = attribute value of currentNode
27             END FOREACH
28         ELSE
29             FOREACH child of currentNode
30                 Write start tag with name = attribute value of currentNode
31                 Write child.nodeValue
32                 Write end tag with name = attribute value of currentNode
33             END FOREACH
34         END IF
35     END IF
36     IF currentNode has next sibling THEN
37         CALL createXML(currentNode.nextSibling)
38     END IF
39 END FUNCTION
```

A result of this algorithm applied to the XML mapping (initial translation from JSON to XML) presented on Listing 7, which is originally the JSON instance in Listing 1, can be seen on Listing 9. This is valid according to the translated XML schema presented on Listing 6.

Listing 9. XML result of Listing 1.

```
1  <?xml version="1.0"?>
2  <root>
3      <datavalues>
4          <units>C</ssvalue>
5          <outSensTemp>14.3</outSensTemp>
6      </datavalues>
7  </root>
```

6.2. Translating XML to JSON

This section proposes an algorithm for the translation of an XML instance to a JSON instance using XSLT, based on the elements data types defined in its schema. To correctly output the data types of the JSON instance properties, the algorithm receives a translated JSON schema as an XSD to create a JSON instance compliant to the data types represented

in it, since XML instances do not carry data type definitions. The usage of the XML schema instead of the JSON schema is due to the accessibility in XSLT through Xpath expressions to the element data type.

In XML, the datatype of an element/attribute is only represented in its schema, with the XML node values being represented as they are, without any differentiation between strings, integers, floats, or any data type. In JSON, it is different; strings are represented within quotation marks while other data types are not. Without this extra information provided by the XML Schema, the translator has no way of knowing if the JSON property it is translating is a string or any other datatype. This will influence the validation of the JSON message according to its JSON schema; for example, it might be expecting a number as 14, and it receives one as "14", which will not be valid.

The proposed algorithm is presented in Listing 10. Similar to the algorithm of Section 6.1, this was developed for implementation in XSLT. Initially, the XML document root node is loaded and an empty JSON document is created, whereas the root brackets are opened and closed before and after a recursive function *createJSON*, which will extend the JSON document, is called (lines 2 to line 10). There are three possible scenarios for the XML document:

1. XML node is the root node of the document and has child elements;
2. XML node not root node of the document and has child elements;
3. XML node has no child elements.

In the presence of Scenario 1, it means that, in the translation, a new JSON object needs to be started (the JSON document was started on lines 4 and 12). It is important to remember that, in the translator of Section 5, a root element with name "root" is created to replicate the JSON unnamed root element. This is verified, and if a root element called "root" exists, the function *createJSON*() is called to its child node, eliminating the "root" element from the JSON instance in line 10. If it does not exist, then the XML node is translated to a JSON object, and then, the function *createJSON*() is called to its child node from line 5 to 9. Having the JSON document created, the *createJSON*() function traverses through every child of the current XML node recursively.

If multiple sibling XML nodes share the same name, it corresponds to a JSON array. In the algorithm, this is divided into three possibilities: the current node is the first element with the repeating name and a new JSON array needs to be started; it is neither the first nor the last element with the same name and is inserted in the middle of the array; or it is the last element with the same name and the JSON array needs to be closed. The XML elements can also be complex type elements (they have child elements), which translates to an array of objects. In the pseudo-code, from lines 15 to line 28, the three possible states of translating a JSON array are approached. On line 26, the JSON array is started, and on line 17, it is ended. Lines 27, 23 and 20 call the function *writeValueOrObject*(), which writes the array content. If the XML node has child elements, it translates it to an array of objects: the algorithm creates a new object and recursively calls the *createJSON*() function to build it (lines 48 to 51); otherwise, it evaluates the datatype of the element and writes it in the array (lines 51 to 54).

If no following or preceding siblings of the current node have similar names, then it is not an array. From lines 29 to 44, this situation is approached. If there are child elements to the current node, then a JSON object is initiated and the *createJSON*() function is called recursively for every child node of the current node (line 29 to 34). If there are no child elements to the current node, then it is an XML element that translates to a JSON data-carrying leaf node. In this case, the element name is written as a JSON property and the XSD is evaluated for the element data type, whereas the value is written accordingly (with or without quotes). If there are sibling nodes to the current node, then the *createJSON*() function is called recursively for the next sibling; if not, it closes the current JSON object (line 34 to 44).

The algorithm does not need to support the translation of XML attributes because the JSON schema to XML Schema translation does not generate attribute values in XML.

Listing 10. XML to JSON translator algorithm.

```
LOAD to XMLSchema the XML schema
LOAD to XMLnode the XML file
CREATE empty JSON
Open curly bracket
IF currentNode.name!="root" THEN
    Write element name and open curly bracket( "currentNode.name":{ )
    CALL createJSON(XMLnode.child, XMLSchema)
    Close curly bracket
ELSE
    CALL createJSON(XMLnode.child, XMLSchema)
END IF
Close curly bracket

FUNCTION createJSON(currentNode, XMLSchema)
    IF currentNode.name equals preceding sibiling and differs following sibiling THEN
        CALL writeValueOrObject(currentNode, XMLSchema)
        Close square brackets
        IF currentNode has following sibiling THEN
            Write a comma
            CALL createJSON (currentNode.followingSibiling)
        END IF
    ELSE IF currentNode.name equals preceding sibiling THEN
        CALL writeValueOrObject(currentNode, XMLSchema)
        Write a comma
    ELSE IF currentNode.name equals following sibiling THEN
        Write element name and open square bracket( "currentNode.name":[ )
        CALL writeValueOrObject(currentNode, XMLSchema)
        Write a comma
    ELSE IF currentNode has child elements THEN
            Write element name and open curly bracket( "currentNode.name":{ )
            FOREACH currentNode.child
                CALL createJSON (currentNode.child)
            END FOR
    ELSE IF currentNode has no child elements THEN
        Write element name
        Evaluate XMLSchema for element data type
        Write element node value in JSON with or without quotes according to data type
        IF currentNode has following siblings THEN
            Write a comma
            CALL createJSON (currentNode.followingSibiling)
        ELSE
            Close curly brackets
        END IF
    END IF
END~FUNCTION

FUNCTION writeValueOrObject(currentNode, XMLSchema)
    IF currentNode has child THEN
        Open curly brackets
        CALL createJSON(currentNode.child, XMLSchema)
    ELSE
        Evaluate XMLSchema for element data type
        Write element node value in JSON with or without quotes according to data type
    END IF
END FUNCTION
```

For demonstrating the results of this translator, we can use all of the previously given examples in reverse order. The original XML document is presented on Listing 9. This algorithm is applied to it while getting the data types presented on the translated XML schema of Listing 6, achieving the translated JSON instance in Listing 1, which is valid according to its JSON schema of Listing 4.

7. Extended TAG-Tool Supporting JSON

The goal of this work is to enable syntactical and semantic interoperability between systems that have different syntax and semantics in the transmitted messages as well as different data interchange formats based on schemas with semantic annotations and complement data values. To enable this possibility with systems that use JSON, the capabilities of the TAG-Tool, as described in Section 3, were extended. With the proposed annotation mechanisms for JSON schemas and respective translation algorithms, it becomes possible to extend the tool, achieving this goal.

The chosen approach to extending the TAG-Tool was to add modules to it, so that the additional work does not compromise in any way what already existed, keeping its

integrity and core functions. The tool already had the capability to generate XML to XML translators based on semantically annotated XML Schemas with complement data values, as the diagram in Figure 3 shows. The annotation mechanisms that exist for XML and are required in JSON for achieving semantic interoperability are proposed in Section 4; whereas the algorithms for translating such annotated JSON Schemas are described in Section 5. This translation algorithm was implemented in the tool, using Java language, translating JSON schemas to XML schemas before feeding them into the TAG-Tool generation engine. This engine receives two XML schemas as it was originally designed for generating an XML to XML translator based on them, namely an XSLT.

The automatically generated XSLT is not enough for translating JSON messages. If a service provider or consumer has JSON as data interchange format, the JSON message is translated to XML, so that the XSLT can be applied or the resulting XML message from the XSLT is translated to JSON, as required by the consumer. To ensure this, the TAG-Tool was extended to generate not only the mentioned XSLT but also the required JSON to XML translator and/or XML to JSON translator. The JSON to XML translator is returned as proposed in Section 6.1, if the service provider system communicates in JSON. The XML to JSON translator is returned if a service consumer system communicates in JSON. In Section 6.2, the XML to JSON translator evaluates the translated XML Schema in a quest for obtaining the correct data types of a certain JSON property; however, to unbind this translator from any schema, the TAG-Tool now generates and returns to the consumers custom translators that no longer evaluate the schemas, since they have all of the correct data types stored for an accurate translation to JSON.

The extended TAG-Tool, supporting JSON systems interaction, is presented in Figure 4. The green blocks are the implementation of the algorithm presented in Listing 5, and the blue modules are the new translators returned by the tool.

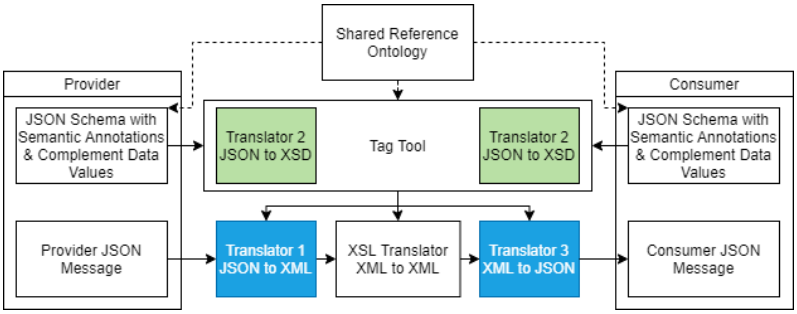

Figure 4. Extended TAG-Tool supporting JSON system interactions.

The flowchart of Figure 5 illustrates how the tool works. As described above and shown below, the tool verifies if the provider or consumer sends data in JSON; if so, it translates the schemas to XSD and feeds it to the TAG-Tool, and if not, it simply feeds the schemas to the tool. After evaluating the compatibility, if it is compatible, it returns the generated translator and required XML to JSON and/or JSON to XML translator. If the consumer expects JSON data, then the tool needs to generate a custom translator for it based on the data types presented on its schema. The extended TAG-Tool is available online at http://gres.uninova.pt/tag/ accessed on 14 September 2021.

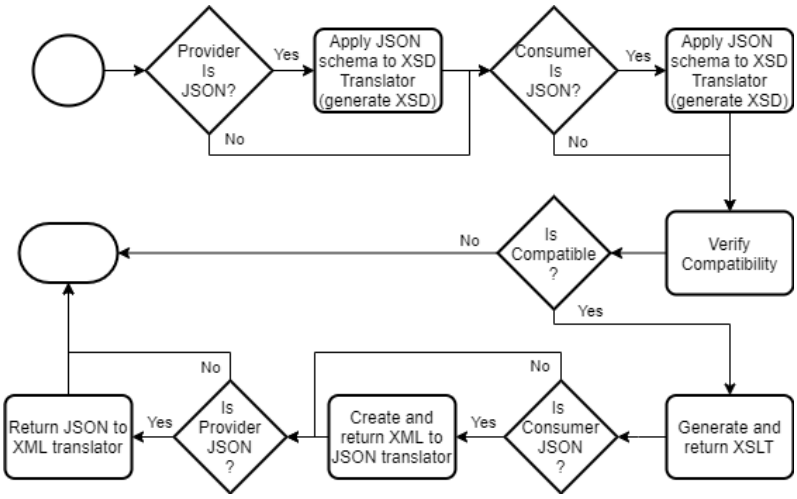

Figure 5. Extended TAG-Tool flowchart.

8. Validation

To validate our proposals and the extended TAG-Tool, several schemas and messages, both in JSON and XML, were used. Eleven JSON schemas with semantic annotations, as proposed in Section 4, were created. Additionally, 11 XML schemas that were used in [5] to validate the previous version of the TAG-Tool were also used. In both sets of 11 schemas, 6 were from providers and 5 were from consumers.

The extended tool was initially successfully tested with the XML schemas to ensure that its previous behavior was not affected. From all of the possible combinations of providers with consumers, the TAG-Tool generated translators (XSLTs) in 14 out of 30 (6×5), as expected due to compatibility constraints (described in [5]).

The following 30 tests were performed with the 11 semantically annotated JSON schemas. The extended TAG-Tool was able to properly identify the JSON schemas, to translate them into XML schemas, and to use them in the generation engine to produce the desired translators. As desired, the tool verified the compatibility in the 30 tests and generated the translators in 14 of them. These translators include not only the XML to XML but also the JSON to XML and the custom XML to JSON translators.

Next, 60 tests were made with one JSON schema and one XML schema, selected from the 11 JSON schemas and 11 XML schemas. As expected, the tool verified the compatibility in all tests and generated the translators in 28 of them.

It is important to note that performance evaluations have not been not conducted. However, it is expected to have reasonable performance due to the following reasons: (1) Translation scripts are generated during systems set-up, before the first message exchange between two systems. This is the most expensive operation and is realized by the TAG-Tool. A performance evaluation of the TAG-Tool is out of the scope of this paper. Based on our observations on matching annotated schemas, the matching time is strongly dominated by the size and complexity of the reference ontology and not by the inserted annotations concepts. A major problem in OWL reasoning is the poor scalability with regard to the A-Box. However, following the approach proposed in this paper, no instance data of documents need to be represented on the ontology level. (2) Once the scripts are generated, all later interactions are based on the generated scripts. These scripts have no references to the ontology and can typically produce their result in a single traversal of the documents. Therefore, it is expected good scalability.

The used JSON schemas, XML schemas, JSON messages, and XML messages as well as the generated XSLTs, are available online at http://gres.uninova.pt/tag/validation/json/ accessed on 14 September 2021.

9. Conclusions

In this paper, we proposed a method for the semantic annotation of JSON Schemas based on our previous work on the annotation of XML Schemas [5]. The approach aims to support the automatic generation of message translators as well as to detect the semantic compatibility between consumers and providers based on the annotations. We also introduced algorithms for the automatic translation of annotated JSON Schemas to annotated XML Schemas and of XML instance documents to JSON instance documents and vice-versa. The algorithms were implemented in the TAG-Tool developed for the Arrowhead Framework, which now supports the seamless interoperation of consumers and providers, no matter if they expect messages in XML or JSON format. Our experiments showed that, in all expected cases, a successful communication between a large set of heterogeneous XML and JSON-based consumer and provider pairs was made possible via the generated translators.

As future work, we intend to support additional data formats and the semi-automatic annotation of Schemas. Other data formats will be considered, such as RDF (Resource Description Framework), JSON-LD (JSON for Linking Data), and CSV (Comma-Separated Values), increasing systems interoperability. Machine learning techniques will be considered to support the semi-automatic annotation of schemas, speeding up the annotation task that currently must be performed manually.

Author Contributions: Conceptualization, G.A., F.M., R.C.-R., J.K. and P.M.; funding acquisition, P.M.; investigation, G.A., F.M., R.C.-R. and J.K.; methodology, G.A., F.M. and R.C.-R.; software, G.A. and F.M.; supervision, F.M. and P.M.; validation, G.A. and F.M.; writing—original draft, G.A., F.M., R.C.-R. and J.K.; writing—review and editing, G.A., F.M., J.K. and P.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by EU ECSEL Joint Undertaking (JU) under grant agreement n° 826452 (project Arrowhead Tools).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Kozma, D.; Varga, P.; Larrinaga, F. System of Systems Lifecycle Management—A New Concept Based on Process Engineering Methodologies. *Appl. Sci.* **2021**, *11*, 3386. [CrossRef]
2. Delsing, J. *Iot Automation: Arrowhead Framework*; CRC Press: Boca Raton, FL, USA, 2017.
3. Van Der Veer, H.; Wiles, A. *Achieving Technical Interoperability: The ETSI Approach*; European Telecommunications Standards Institute: Sophia Antipolis, France, 2008; p. 29.
4. Moutinho, F.; Paiva, L.; Maló, P.; Gomes, L. Semantic annotation of data in schemas to support data translations. In the Proceedings of IECON (Industrial Electronics Conference), Florence, Italy, 23–26 October 2016; pp. 5283–5288. [CrossRef]
5. Moutinho, F.; Paiva, L.; Köpke, J.; Malo, P. Extended Semantic Annotations for Generating Translators in the Arrowhead Framework. *IEEE Trans. Ind. Inf.* **2018**, *14*, 2760–2769. [CrossRef]
6. Semantic Annotations for WSDL and XML Schema. 2007. Available online: https://www.w3.org/TR/sawsdl/ (accessed on 20 August 2021).
7. Köpke, J.; Eder, J. Semantic Annotation of XML-Schema for Document Transformations. In *Proceedings of the OTM'10 Workshops*; LNCS; Springer: Berlin/Heidelberg, Germany, 2010; Volume 6428, pp. 219–228.
8. Lafon, Y.; Mitra, N. SOAP Version 1.2 Part 0: Primer (Second Edition). Technical Report, W3C. 2007. Available online: https://www.w3.org/TR/2007/REC-soap12-part0-20070427/ (accessed on 20 August 2021).
9. Fielding, R.T. Architectural Styles and the Design of Network-Based Software Architectures. Ph.D. Thesis, University of California, Irvine, CA, USA, 2000.
10. Sporny, M.; Birbeck, M.; Herman, I.; Adida, B. RDFa 1.1 Primer—Third Edition. W3C Note, W3C. 2015. Available online: http://www.w3.org/TR/2015/NOTE-rdfa-primer-20150317/ (accessed on 20 August 2021).
11. Khare, R.; Çelik, T. Microformats: A pragmatic path to the semantic web. In Proceedings of the 15th International Conference on World Wide Web, Edinburgh, UK, 23–26 May 2006; pp. 865–866.
12. Ciccarese, P.; Young, B.; Sanderson, R. Web Annotation Data Model. Candidate Recommendation, W3C. 2016. Available online: https://www.w3.org/TR/2016/CR-annotation-model-20161122/ (accessed on 20 August 2021).
13. Bizer, C.; Eckert, K.; Meusel, R.; Mühleisen, H.; Schuhmacher, M.; Völker, J. Deployment of RDFa, Microdata, and Microformats on the Web—A Quantitative Analysis. In *The Semantic Web—ISWC 2013*; Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N., Welty, C., Janowicz, K., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; pp. 17–32.

14. Bray, T. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259. 2017 Available online: https://datatracker.ietf.org/doc/html/rfc8259 (accessed on 20 August 2021).
15. Köpke, J.; Eder, J. Logical Invalidations of Semantic Annotations. In *Advanced Information Systems Engineering: 24th International Conference, CAiSE 2012, Gdansk, Poland, 25–29 June 2012*; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7328, pp. 144–159.
16. Köpke, J.; Eder, J.; Joham, D. Towards Path-Based Semantic Annotation for Web Service Discovery. In I*nformation Systems Engineering in Complex Environments: CAiSE Forum 2014, Thessaloniki, Greece, 16–20 June 2014*; Selected Extended Papers; Springer International Publishing: Cham, Switzerland, 2015; Volume 204, pp. 133–147.
17. Köpke, J. Annotation paths for matching XML-Schemas. *Data Knowl. Eng.* **2019**, *122*, 25–54. [CrossRef]
18. Booth, D.; Liu, K. Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. W3C Recommendation, W3C. 2007. Available online: https://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/ (accessed on 20 August 2021).
19. Mendelsohn, N.; Thompson, H.; Sperberg-McQueen, M.; Maloney, M.; Gao, S.; Beech, D. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. W3C Recommendation, W3C. 2012. Available online: https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/ (accessed on 20 August 2021).
20. Popa, L.; Velegrakis, Y.; Hernández, M.A.; Miller, R.J.; Fagin, R. Translating web data. In Proceedings of the 28th International Conference on Very Large Data Bases, VLDB Endowment, Hong Kong, China, 20–23 August 2002; pp. 598–609.
21. Mecca, G.; Papotti, P. Schema Mapping and Data Exchange Tools: Time for the Golden Age. *IT Inf. Technol.* **2012**, *54*, 105–113. [CrossRef]
22. Marnette, B.; Mecca, G.; Papotti, P.; Raunich, S.; Santoro, D. ++Spicy: An OpenSource Tool for Second-Generation Schema Mapping and Data Exchange. *VLDB Endow.* **2011**, *4*, 1438–1441. [CrossRef]
23. Fagin, R.; Kolaitis, P.G.; Miller, R.J.; Popa, L. Data exchange: Semantics and query answering. *Theor. Comput. Sci.* **2005**, *336*, 89–124. [CrossRef]
24. OWL Working Group. OWL 2 Web Ontology Language: Document Overview; W3C Recommendation. 27 October 2009. Available online: http://www.w3.org/TR/owl2-overview/ (accessed on 20 August 2021).
25. Szymczak, M.; Köpke, J. Matching Methods for Semantic Annotation-based XML Document Transformations. In *New Developments in Fuzzy Sets, Intuitionistic Fuzzy Sets, Generalized Nets and Related Topics*; SRI-PAS: Warsaw, Poland, 2012; Volume 2, pp. 297–308.
26. Do, H.H.; Rahm, E. COMA: A system for flexible combination of schema matching approaches. In Proceedings of the 28th International Conference on Very Large Data Bases. VLDB Endowment, Hong Kong, China, 20–23 August 2002; pp. 610–621.
27. Aumueller, D.; Do, H.H.; Massmann, S.; Rahm, E. Schema and ontology matching with COMA++. In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, Baltimore, ML, USA, 14–16 June 2005, pp. 906–908.
28. Madhavan, J.; Bernstein, P.A.; Rahm, E. Generic Schema Matching with Cupid. In *Proceedings of the 27th International Conference on Very Large Data Bases*; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2001; pp. 49–58.
29. Nikovski, D.; Esenther, A.; Ye, X.; Shiba, M.; Takayama, S. Matcher Composition Methods for Automatic Schema Matching. In *Enterprise Information Systems: 14th International Conference, ICEIS 2012, Wroclaw, Poland, 28 June–1 July 2012*; Revised Selected Papers; Springer: Berlin/Heidelberg, Germany, 2013; Volume 204, pp. 108–123.
30. Kim, J.; Peng, Y.; Ivezic, N.; Shin, J. An optimization approach for semantic-based XML Schema matching. *Int. J. Trade Econ. Financ.* **2011**, *2*, 78. [CrossRef]
31. Thang, H.Q.; Nam, V.S. XML Schema Automatic Matching Solution. *Int. J. Electr. Comput. Syst. Eng.* **2010**, *4*, 68–74.
32. Missikoff, M.; Schiappelli, F.; Taglino, F. A controlled language for semantic annotation and interoperability in e-business applications. In *Proceedings of the Workshop on Semantic Integration*; CEUR-WS: Aachen, Germany , 2003; pp. 1–6.
33. Campos-Rebelo, R.; Moutinho, F.; Paiva, L.; Malo, P. Annotation Rules for XML Schemas with Grouped Semantic Annotations. In the Proceedings of IECON (Industrial Electronics Conference), Lisbon, Portugal, 14-17 October 2019; pp. 5469–5474. [CrossRef]
34. Vujasinovic, M.; Ivezic, N.; Kulvatunyou, B.; Barkmeyer, E.; Missikoff, M.; Taglino, F.; Marjanovic, Z.; Miletic, I. Semantic mediation for standard-based B2B interoperability. *Internet Comput. IEEE* **2010**, *14*, 52 –63. [CrossRef]
35. Pankowski, T.; Udalowski, R. Generating schema mappings based on annotations in a P2P data integration system. In Proceedings of the iiWAS'2010—The 12th International Conference on Information Integration and Web-based Applications and Services, Paris, France, 8–10 November 2010; pp. 687–691.
36. Martin, D.; Burstein, M.; McDermott, D.; McIlraith, S.; Paolucci, M.; Sycara, K.; McGuinness, D.L.; Sirin, E.; Srinivasan, N. Bringing semantics to web services with OWL-S. *World Wide Web* **2007**, *10*, 243–277. [CrossRef]
37. Vitvar, T.; Kopeckỳ, J.; Viskova, J.; Fensel, D. Wsmo-lite annotations for web services. In *European Semantic Web Conference*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 674–689.
38. Kopecky, J; Vitvar, T; Fensel, D; Gomadam, K *CMS WG Deliverable D12V0.1 HRESTS & MICROWSMO*; Technical Report; STI International: Innsbruck, Austria, 2008.
39. Kopecký, J.; Gomadam, K.; Vitvar, T. hRESTS: An HTML Microformat for Describing RESTful Web Services. In Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, Sydney, Australia, 9–12 December 2008; Volume 1, pp. 619–625. [CrossRef]
40. Sheth, A.P.; Gomadam, K.; Lathem, J. SA-REST: Semantically interoperable and easier-to-use services and mashups. *IEEE Internet Comput.* **2007**, *11*, 91–94. [CrossRef]

41. Alarcon, R.; Saffie, R.; Bravo, N.; Cabello, J. REST Web Service Description for Graph-Based Service Discovery. In *Proceedings of the 15th International Conference on Engineering the Web in the Big Data Era—Volume 9114*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 461–478. [CrossRef]
42. Roman, D.; Kopecký, J.; Vitvar, T.; Domingue, J.; Fensel, D. WSMO-Lite and hRESTS: Lightweight semantic annotations for Web services and RESTful APIs. *J. Web Semant.* **2015**, *31*, 39–58. [CrossRef]
43. Champin, P.A.; Kellogg, G.; Longley, D. JSON-LD 1.1. W3C Recommendation, W3C. 2020. Available online: https://www.w3.org/TR/2020/REC-json-ld11-20200716/ (accessed on 20 August 2021).
44. Wang, X.; Sun, Q.; Liang, J. JSON-LD Based Web API Semantic Annotation Considering Distributed Knowledge. *IEEE Access* **2020**, *8*, 197203–197221. [CrossRef]
45. Arrowhead—Ahead of the Future. Available online: http://www.arrowheadproject.eu/ (accessed on 14 May 2019).
46. Productive 4.0—A European Co-Funded Innovation and Lighthouse Project on Digital Industry. Available online: https://productive40.eu/ (accessed on 19 May 2019).
47. Arrowhead-Tools. Available online: https://www.arrowhead.eu/arrowheadtools (accessed on 30 May 2019).
48. JSON Schema: A Media Type for Describing JSON Documents. Available online: https://json-schema.org/draft/2020-12/json-schema-core.html (accessed on 20 August 2021).
49. Adding Semantic Annotations to JSON Schema Issue #13. Available online: https://github.com/json-schema-org/json-schema-vocabularies/issues/13 (accessed on 19 May 2019).
50. Convert JSON Schema to XSD Online—ConvertSimple.com. Available online: https://www.convertsimple.com/convert-json-schema-to-xsd/ (accessed on 14 September 2021).
51. Nogatz, F.; Frühwirth, T. From XML Schema to JSON Schema—Comparison and Translation with Constraint Handling Rules. Bachelor's Thesis, University of Ulm, Ulm, Germany, 2013.
52. XSL Transformations (XSLT) Version 3.0—Funtion JSON-to-XML. Available online: https://www.w3.org/TR/xslt-30/#func-json-to-xml (accessed on 24 June 2019).