SN Computer Science (2022) 3:293
https://doi.org/10.1007/s42979-022-01170-5

ORIGINAL RESEARCH

LISSU: Continuous Monitoring of SOA Communication with Constraint-Based Validation

Johannes Theissen-Lipp[1,4] · **Moritz Kröger**[2] · **Benedikt Heinrichs**[3] · **Stefan Decker**[1,4]

Received: 13 January 2022 / Accepted: 19 April 2022 / Published online: 17 May 2022

Abstract

Service-oriented architectures (SOA) are becoming more widespread in the context of Industry 4.0, and their interface descriptions enable modular and scalable communication systems. Since syntactic checks such as data types are solved nowadays, the purpose of this work is to add semantic validation based on the idea of Semantic Web Services. This paper proposes *Lightweight Semantic Web Services for Units* (LISSU) and integrates promising concepts from the Semantic Web into SOA. We complement existing syntactic checks with semantic ones (e.g. for units), extend one-time initial checks with continuous monitoring, and include expressive constrain-based validations. LISSU can be integrated into any SOA and significantly increases the predictability of communications. Before components communicate, it checks their semantics via ontology URIs and automatically converts units if possible. Continuous monitoring at runtime extracts sent messages and guarantees flawless data quality via constraint-based validations. A real-world demonstrator setup in the manufacturing domain proves effectiveness and practicality. We present LISSU, which integrates concepts from the Semantic Web into SOA setups. It enables a wide range of semantic validations before and during communication, thereby increasing the quality and predictability of SOA communication.

Keywords Semantic web services · Data lifting · Service-oriented architecture · Semantic web

Introduction

In the last decades, the Semantic Web [1] has gained much popularity. Its extension called Semantic Web Services [2], or SWS in short, captures many types of information for services, such as data, metadata, properties, capabilities, interface, and pre- or post-conditions. Researchers proposed a variety of challenges and solutions related to these goals especially during its golden age starting around 2007. Most of the solutions were never properly used in practice because other challenges had to be solved there first. Service-oriented architectures (SOA) and its sub-field remote procedure calls (RPC) are promising application areas for SWS, but had to tackle issues with connectivity, data availability, and syntax validation at interfaces during that time. We argue on the one hand that the SOA research community has overcome most of these basic challenges and needs semantic information now. On the other hand, the design of SWS is too complex and needs to be simplified to be used in practice.

Our previous work [3] focused the area of SWS but in a smaller scope, aiming for practicable and feasible solutions. We particularly tackled real-world challenges in the

This article is part of the topical collection "Enterprise Information Systems" guest edited by Michal Smialek, Slimane Hammoudi, Alexander Brodsky and Joaquim Filipe.

✉ Johannes Theissen-Lipp
lipp@dbis.rwth-aachen.de

✉ Moritz Kröger
moritz.kroeger@llt.rwth-aachen.de

✉ Benedikt Heinrichs
heinrichs@itc.rwth-aachen.de

✉ Stefan Decker
decker@dbis.rwth-aachen.de

[1] Chair of Information Systems, RWTH Aachen University, Ahornstraße 55, 52074 Aachen, Germany

[2] Chair for Laser Technology, RWTH Aachen University, Steinbachstraße 15, 52074 Aachen, Germany

[3] IT Center, RWTH Aachen University, Seffenter Weg 23, 52074 Aachen, Germany

[4] Fraunhofer Institute for Applied Information Technology FIT, Schloss Birlinghoven 1, 53757 Sankt Augustin, Germany

engineering domain, where client and server have an a-priori semantic mismatch (e.g., different units). Since state-of-the-art solutions were not sufficient, we extended the syntax validations at service interfaces with machine-readable semantic ones and thereby made communication more predictable. We proposed a first version of *Lightweight Integration of Semantic Web Services for Units (LISSU)* as a backwards-compatible extension to RPC frameworks that, in addition to syntactic details like datatypes, allows configuration of semantic information including units. This extended validation workflow detects unit mismatches between client and server and even corrects these via automatic conversions if possible. LISSU finally provides an interoperable and consistently predictable communication among components, which we demonstrated in a real-world setup with machine components of a laser system.

This paper continuous our research on LISSU and extends it with novel approaches and implementations that tackle two newly discovered challenges: First, we found that it is insufficient to only validate the configurations of client and server during the initialization phase of communication, because their actual implementation might misbehave during runtime. We, therefore, extend our approach to a continuous monitoring of the actual SOA communication, which additionally verifies that sent messages comply to an expected schema. Second, implementations of real-world use-cases in the field show that LISSU's basic validation must be extended to a more sophisticated one, which covers advanced expressiveness requested by domain experts. This includes constraints on value ranges, complex datatypes, cardinalities, cross-attribute relations, and many more. We overcome this in our current work by expanding LISSU's basic URI-based validation with a proper constraint language that enables the needed expressiveness.

The remainder of our paper is structured as follows. The next section gives a motivating example for both semantic mismatch and rule-based validation and lastly defines our goals. "Related Work" investigates related work with a special focus on ontologies, SWS, validation and data lifting. "Proposed Approach" presents our base approach and our two novel additions, namely monitoring a SOA communication stream and using constraint-based validation, and "Implementation of the Proposed Framework" shows their implementation. We conduct an evaluation based on this setup in "Demonstrator Setup" via a demonstrator and finally conclude in "Conclusion and Future Work" with a summary of our work and possible future work in this field.

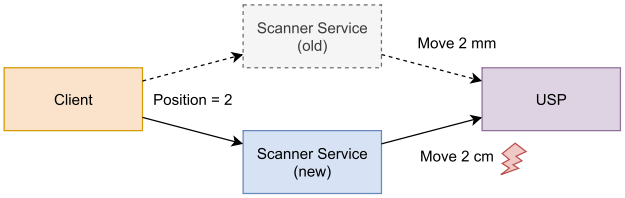

Fig. 1 Unit mismatch during a component swap at a USP laser system, which we observed in our previous work on LISSU [3]. The new server implementation internally uses different units and hence moves the laser for 2 centimeters instead of 2 millimeters

Motivating Example

This section motivates the extension of SOA with semantic capabilities based on SWS by presenting an example from ultrashort pulse (USP) laser system development and showcasing concrete challenges that we tackle in this work. This work is based on prior work [3] about unit mismatches and extends the idea of the validation by integrating a stream/client call validation framework.

Semantic Mismatch (Units)

Within the research project "Internet of Production" [4], Semantic Web experts and laser experts collaboratively build a USP laser system based on SOA. This includes to assemble machine parts from different manufacturers and subsequently achieve an interoperable communication between these. The very basic idea of USP could be referred to as "reverse 3D printing", which incrementally removes material with high amplitude laser pulses to form a final product. A USP laser is divided into multiple components such as scanner, movement system, and camera. Client applications (e.g., a central controller) call remote services on these components to configure the laser and execute actions. The syntax of service calls is validated with the help of Google's Protocol Buffers (Protobuf)[1], but semantic mismatches are not detected and, therefore, ignored.

A so-called scanner moves the laser to (x,y) coordinates based on received float values for `position.x` and `position.y`, but the interpretation of the units is left to the respective implementation of that service. Figure 1 illustrates a dangerous scenario of a component change: The former scanner hardware and its respective service interprets position values as *millimeter* and thus moves the laser to an x-position of 2 millimeters. A new component and its respective new service, however, could interpret the same data value as 2 centimeters and thus move the laser wrongly or even damage the product. Other examples from

[1] https://developers.google.com/protocol-buffers/.

that use-case are laser heating temperatures (e.g., Celsius vs. Fahrenheit) or laser speeds (e.g., millimeter per seconds vs. kilometer per hour).

The details of that real-world motivating example are the following. The currently existing project uses Google's implementation of RPC, called gRPC [5], for communication in the network. This allows the serialization of data with Protobuf, a Data Description Language made by Google [6]. Additionally, the system uses Bazel [7] as the utilized build tool to manage dependencies, build the source files and execute the code. The Protobuf files contain syntactical descriptions for all the relevant service and messages for the client server communication by strictly defining input and output parameters of service calls and the data types of these parameters in so called message definitions. In the process of utilizing gRPC, these files will then be compiled into new files in the desired programming language like Python or Java for the corresponding application, allowing the client and server to access the definitions made in Protobuf files. There is currently no way to properly include semantics like units into these definitions.

Developers in this research project have chosen comments and variable names in their Protobuf files as temporary solutions so that the units become visible to the reader. However, this approach is error-prone and not machine-readable. The above-mentioned motivating examples demonstrates the need to avoid semantic mismatches by properly defining and validating units at service interfaces.

Rule-Based Message Validation

While the detection of semantic mismatches in an industrial SOA Framework helps to prevent accidental errors when integrating new machine components into the framework it does not prevent runtime errors or malicious messages from being sent to the server. Typically, these errors can be prevented by formulating specific rule sets to validate the messages. An example of a hard runtime error is a MoveToPosition message, where the requested position is out of bounds of the scanner system. Currently the system detects these errors on the server side since it is physically not possible to move to that position. These system-specific rules, therefore, prevent the system from internal damage. While of course a domain-specific language for formulating these rules can be advantageous, the definition of softer runtime errors could hold much more potential. These soft runtime errors are typically producible but might not be preferable in a quality sense or long-term maintainability sense. These rules are very often "cross-platform" and could, therefore, be applied to multiple services implementing the same interface. Additionally, the formulation of these constraints is more complex than simple physical boundaries which raises the need for an expressive constraint language.

In the USP Production domain a typical example for one of these softer runtime errors would be ablation in the 0/0 Position of the scanner system. In this special point the laser angle of incidence is perpendicular to the workpiece surface. Laser light that get positioned on that point can, therefore, be reflected back inside the optical path of the system and damage the components due to overheating. [8] Another soft constraint lies in the combination of the laser power in Watt and the corresponding movement speed of the laser beam resulting in the delivered energy per area or short fluence. This constraint typically depends on the material used in the process since the melting point and the absorption rate changes. If the fluence is too low the material does not vaporize. If it is too high the material might rise in temperature, which leads to thermal deformations [9].

While these errors should be avoided during a USP ablation process, nothing inside the scanner component is limiting this behavior. Including this rule inside of the scanner microservice would also be disadvantageous since it would tie the very general component service "scanner system" which can be used in different manufacturing machines like Laser Powder Bed Fusion or Battery welding directly to the USP Process. At the moment these errors are mitigated by the experience of the machine operator who knows these rules by experience and, therefore, avoids them during the design process of the manufactured part and the setup of the machining process.

Requirements and Goals

The ultimate goal of this work is to overcome semantic mismatches and flawed messages in SOA frameworks by integrating Semantic Web components into these. We particularly aim to extend the existing syntax-only validation to also cover semantics and units, as well as the validation of specific message rules. Developers must be able to configure this information for both clients and servers to make service calls more predictable and errorfree. These need to automatically validate the given semantic information before initiating the actual communication. The definition of semantic message rules should also be extensible during the machine life cycle to allow further updates and changes depending on the state of the system (e.g., "material loaded" or "laser used"). This additional verification must be backwards-compatible, because SOA frameworks are usually distributed systems, and one developer can only modify some of the services. The message validation should also be injected during runtime into the process without interfering with the process itself. For monitoring purposes, a sampling strategy should be used while a possibility for blocking malformed messages should also be present. All interactions with the novel solution must be easily accessible for domain experts, which are typically non-experts in the Semantic Web. The

Table 1 Evaluation of relevant ontologies based on the requirements from "Requirements and Goals" with OM and QUDT ranked highest [3]. The symbols (– / o / +) represent a low / medium / high value of given metrics, respectively. Please note that this ranking is specific for our requirements and, therefore, not universal

Ontology	Relevance	Coverage	Features	Flexibility
SDO [13]	–	o	–	–
OntoSensor [15]	–	o	–	–
SSN [16]	o	+	+	o
OM [18, 19]	+	+	+	o
QUDT [20]	+	+	+	+
UCUM [13, 21]	+	o	+	o

manual effort should be minimized by automating as much as possible, e.g., validation or correction of used units. Following the Semantic Web best practices, we should reuse as much as possible, for example units from well-known ontologies. The proposed solution should be field-proven and easily adoptable to new use-cases.

Related Work

This section investigates and discusses related work in the areas of ontologies, semantic service descriptions, data lifting and mapping, and constrain-based validation. While the first two are based on our earlier work [3], the latter two extend it with completely new aspects.

Relevant Ontologies

A basic representation of units of measurement is already a big step forward. We argue that this is an optimal trade-off between semantic value and complexity overhead, and analyze relevant ontologies on sensors, units, and measures. Our analysis reuses existing concepts when possible, like proposed in [10]. We aim for ontology terms that we can reuse in a modular approach without unwanted side effects as described in [11]. Table 1 presents the metrics we applied and the final ranks for all ontologies we analyzed. The metrics are the following. Relevance determines how well the ontology scope matches our work, and coverage rates the applicability of the ontology's concepts to our needs. The metric evaluates available programmatic extensions such as unit conversions. Flexibility rates how well one can tailor the concepts from the ontology to specific applications, e.g., being able to build custom units with prefixes like "milli". Note that this rating is specific to our requirements and should not be interpreted as a universal ontology rating.

Sensor Ontologies

The article [12] reviews existing sensor data ontologies to decide if they can be reused for a manufacturing perception sensor ontology. The outcome of their work is a review and an ontology. Relevant ontologies mentioned are the Sensor Data Ontology (SDO) [13], which uses the Suggested Upper Merged Ontology (SUMO) [14], the OntoSensor ontology [15] and especially the Semantic Sensor Network (SSN) ontology [16]. The SSN ontology appears promising for our work and has multiple features: Data discovery and linking, device discovery and selection, provenance and diagnosis, and device operation, tasking, and programming [16]. It allows focusing on sensors, observed data, system, or feature and property. The SSN ontology describes sensor networks well but needs to be combined with other ontologies for describing units, like the ontology *Library for Quantity Kinds and Units* [17].

Unit Ontologies

The Ontology of Units of Measure (OM) [18, 19] allows descriptions of units with all their details and relations to each other and was developed during the development of the Ontology of Quantitative Research [22]. It is based on existing standards for units of measure such as [23] and contains units of measure, prefixes, quantities, measurement scales, measures, system of units, and dimensions. Any quantities are defined by a measurement scale, which is a mapping from categories and points to quantities. Units can then be further scaled with prefixes, making it easier to represent values for a base unit. The unit *millimetre* for example is defined as a prefixed unit with the unit *metre* and the prefix *milli*. Quantities and units have dimensions and systems of units for their organization. A system of units defines a set of base dimensions, which can then be used to express every other possible dimension as a combination of certain base units, like the International System of Units [24]. All other dimensions can then be computed from these base units.

The *Quantities, Units, Dimensions, and Types Ontology* (QUDT) [20] follows a similar approach and modularly builds on multiple ontologies. It covers fewer quantity kinds and units per application area than OM, but it allows more flexible conversion multipliers and offsets. The Unified Code for Units of Measure (UCUM) was published in [13, 21] covers practically all units used in science and engineering, while every unit has a unique identifier. It is possible to validate and convert datatypes via the UCUM web service https://ucum.nlm.nih.gov/ucum-service.html#conversion. UCUM unit codes are referenced in the above-mentioned QUDT ontology and supports these unit conversions. UCUM, however, specifies units and scales less comprehensive compared to OM.

We conclude that many ontologies for units in scientific applications exist. Following the rating depicted in Table 1, QUDT and OM are both promising for work due to their completeness and relevance. QUDT provides a mature SPARQL Protocol and RDF Query Language (SPARQL) integration and provides flexibility by linking to OM and UCUM via additional identifiers within the ontology. OM provides a more comprehensive structure and has potential to flexibly adapt to future requirement changes, while the SSN ontology primarily supports sensors and service calls instead of units.

Communication and Service Description

SWS [2], their organized peer-to-peer extensions [25], and similar approaches semantically enrich web services and provide machine-readable markups. Needed descriptions can be written in the Web Service Description Language, which in [26] was extended with Semantic Annotations. These annotations refer to ontologies and support lifting and lowering mappings between XML messages. A complex feature set including information, functional, behavioral, and non-functional semantics, however, complicates lightweight application, which we in fact aim for. Our approach is even more lightweight than the lightweight Semantic Web Service descriptions proposed in [27] and [28], which are also available as W3C submission [29], as they still include functional, non-functional, and behavioral semantics. [30] introduces a so-called descriptor that adds operation, link, non-functional, and service descriptions to RESTful services via an ontology-based approach.

RPCs allow remotely calling services and passing parameters [31]. Note that this concept was introduced nearly four decades ago but has become an active research topic again recently. Google offers gRPC [5], an open-source high performance RPC framework that has gained a lot of popularity. Benefits include high scalability, low latency distributed systems, and developed programming languages support. It is recommended to use Protobuf to describe the syntax of services' expected inputs and outputs [6]. One concrete example is an extension of a closed-source platform for deployment, integration, and orchestration digital services with semantic unit information [32]. Proposed features include data contextualization by enriching data with (semantic) information facilitating the understanding of the data and its context.

We summarize that the SWS was a very active research area between 2001 and around 2008 but lost some of its drive. We argue that this is mainly due to very ambitious goals that could not yet be applied in practice. Later developments based on RPC solved crucial basic problems and, therefore, the chance to properly combine these research fields is now.

Constraint-Based Validation

The concept of validation of entities based on given constraints is not a new problem and can be tackled in many ways. One option for validating data in the widely-used JSON format is using JSON Schema [33] to define constraints for specific key-value pairs. The relevant use-case here, however, goes beyond the standard constraint validation with the advent of using specific ontologies for describing units. Therefore, here a constraint-based validation method must understand and be able to work in an environment, which is largely dependent on RDF data.

The Shapes Constraint Language (SHACL) [34] is a W3C recommendation which fulfills the requirements of being able to validate RDF data and formulate requirements based on specific ontologies. It works by describing different concepts as so-called shapes and enabling constraints on the concepts itself and their properties. We, therefore, conclude that SHACL is a promising candidate for our task of constraint-based validation of our produced data.

Semantic Data Lifting

We understand the term (semantic) data lifting as a synonym to *uplifting*, which is the process of transforming data sources into triples in the Resource Description Framework (RDF) [1]. On the one hand, there is a wide range of data formats for communication (cf. "Motivating Example"), which mainly include binary formats and (semi-)structured ones such as JSON, CSV, or XML. On the other hand, there are expressive and convenient constraint languages such as SHACL, which work on RDF. This raises the need for a mapping from the different data formats to RDF, which we call data lifting. We evaluate the RDF Mapping Language (RML) [35, 36] as promising candidate as it allows expressing customized mapping rules from heterogeneous data structures and serializations to RDF. Its reference implementation *RMLMapper* is an open-source Java library[2] that execute RML rules via the command line. Please refer to their tool overview[3] for a list of processors, graphical user interfaces, wrappers, and supportive tools.

Proposed Approach

This section presents our concept to integrate ontology terms into a microservice-based SOA system to handle unit mismatches and continuously validate the retrieved data. That includes an extended architecture for SOA, novel semantic

[2] https://github.com/RMLio/rmlmapper-java.

[3] https://rml.io/tools/.

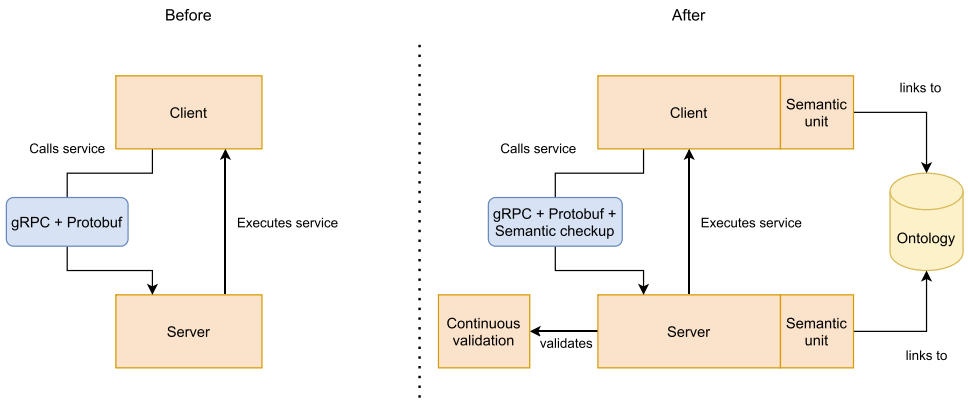

Fig. 2 The prior communication on the left only includes syntactic validation via Protobuf. Our contribution on the right adds novel semantic components to both client and server [3], and extends the validation workflow

configurations, and an extended communication workflow. Our approach is backwards-compatible and enables a modular system in which programmatic features and semantic descriptions (e.g., units) can be implemented in parallel.

Figure 2 depicts a current state-of-the-art setup building on gPRC on the left, which acts as communication technology and interface definition, e.g., stored in Protobuf files. In the baseline setup on the left, a client contacts a server with a service call, which triggers the server to validate the call's syntax and finally execute the service if its checks were successful. We overcome the lack of semantic validations presented in the previous sections by extending the SOA framework, as shown on the right side of Fig. 2. We add semantic units to both client and server, which link to ontologies and use ontology URIs to specify units in their configurations. Our extended pre-communication validation also includes a semantic check, which compares the ontology URIs from both communication partners, and only allow service execution if they match. Finally, the server is extended by continuous validation of the provided data based on defined constraints. The following sections present further details on the architecture and workflow.

Adding Lightweight Semantic Components to SOA Frameworks

This section explains more details on the components depicted in Fig. 2, namely the semantic unit configuration, ontology references, and novel semantic validations. "Initial Semantic Validation Workow in Detail" then gives even more details on the initial semantic validation workflow and "Continuous Semantic Validation Workow in Detail" then discusses the continuous semantic validation workflow.

As we plan to validate semantic information such as units between client and server, we add definitions to them, which map each entry of their interface definitions (e.g., gRPC) to respective units in form of ontology terms (URIs). The semantic units can later compare these URIs and only initiate communication if they match. Note that different

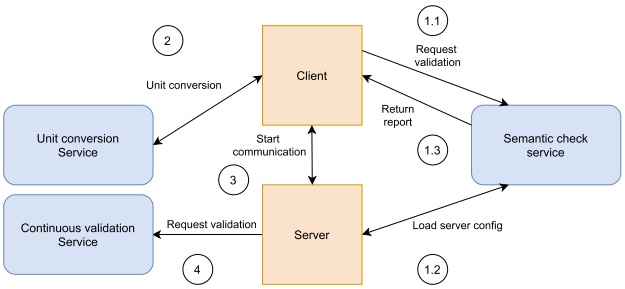

Fig. 3 SOA framework with communication parties named client and server for easier differentiation, and our augmenting services added as rounded rectangles. A client contacts up to three semantic services to make the subsequent communication more predictable. We particularly extend our previous work [3] with step four

other cases such as mismatches or partial matches exist. Our approach in this work uses publicly accessible URIs as ontology links, but one could easily adapt this design to custom local unit references, too.

We implement an additional step in the workflow of the SOA communication, which is illustrated in Fig. 3. A client initiating a service call first undergoes a semantic check, which compares the client's semantic specifications with the server's one. The client (1.1) requests validation at the semantic check service by transmitting its semantic configuration. That service (1.2) polls the respective configuration from the server, matches these and (1.3) returns a validation report to the client. The client subsequently interprets this report, for details please see the next section. If the report includes issues, the client needs to fix these or abort communication. It can, for instance, (2) contact a unit conversion service to correct unit mismatches. Additionally, (3) the client starts the actual communication with the server. Finally, (4) the server requests every n seconds validation of the currently sent data by a continuous validation service.

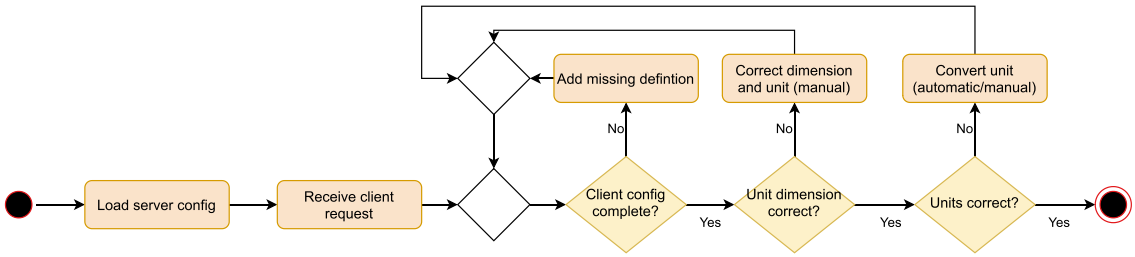

Fig. 4 Activity diagram of our proposed initial semantic validation workflow, which detects and recovers from semantic mismatches to finally establish a predictable communication [3]. Empty rhombuses merge alternative paths

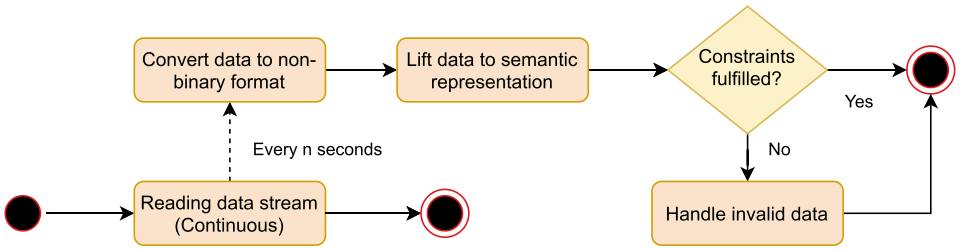

Fig. 5 Proposed continuous semantic validation workflow, which checks every n seconds whether the data stream contains valid data

Initial Semantic Validation Workflow in Detail

A client receives a validation report from the semantic check service while preparing communication. Depending on that report, one or more of the following four actions can be required at the client, as illustrated in Fig. 4:

- *Client configuration incomplete:* The provided configuration is missing properties required from the server. The client must correct it by adding missing definitions, usually manually.
- *Unit dimension mismatch:* Correct the dimensions and units used, and try again. Hard mismatch, e.g., *speed in m/s* used but *temperature in Celsius* expected. Usually fix manually.
- *Unit mismatch:* Used the correct dimension but the wrong unit, e.g., *m/s* instead of *km/h*. Use our proposed unit conversion service to automatically convert units.
- *All semantics match:* Start the communication.

This workflow detects all relevant possible issues w.r.t. semantic mismatches. While fatal issues such as incomplete configurations or dimension mismatches usually need to be solved manually, we can automatically resolve unit mismatches within the correct dimensions.

Continuous Semantic Validation Workflow in Detail

As illustrated in Fig. 5, a server sends a part of the continuously read data stream to the continuous validation service every n seconds. Since the targeted validation languages do not support the native SOA format, the data must be transformed by first representing it in a non-binary format and then lifting it to a semantic representation. This semantic representation can then be validated, and it is checked if the defined constraints are fulfilled. The approach proposes a handler implementation for when this is not the case, which can be represented by a logging action, for example.

Implementation of the Proposed Framework

In this section, we demonstrate our realization of the concept presented above. This includes justifying a selection of concrete tools based on our requirements. We then present in detail the implementation and evolution of two services from [3] and two novel aspects: First, an initial semantic validation service including a method to specify semantics at both client and server. Second, a unit conversion service that allows automatic correction for unit mismatches that only reside in the same dimension. Third, a continuous semantic validation service including a method to validate streaming data against given requirements after lifting heterogeneous (semi-)structured data. Finally, we suggest a user interaction and workflow with the new system.

Tool Selection Based on Requirements

As discussed in previous sections, there are multiple ontologies, data description formats, and additional software available. For our realization, we choose the following tools. We use Protocol Buffer and gRPC to model the interface

of services syntactically, and introduce semantic descriptions (e.g., for units) via additional JSON files stored at the client and server. Reasons for choosing JSON include its high expressiveness and increased human readability, plus its easy integration into many programming languages. We integrate Bazel to automate building and testing of these descriptions.

We select QUDT as the main ontology for the semantic representation of the units of measure, while keeping the support for OM. We do not require SSN or any of the other ontologies in our current use-case, since QUDT and OM already cover all necessary concepts. However, the semantic descriptions do link to additional ontologies for certain types of data, for example the representation of a UNIX timestamp. We chose QUDT, because it satisfies all relevant units of measure we need and offers a good support for unit conversions via SPARQL queries. This can be done using the Python library rdflib [37] or via other programming languages like Java or Go.

We use the JSON data format to export SOA messages from Protobuf. We choose RML to formulate data mappings to RDF, and use its reference implementation RMLMapper as Java library (cf. "Semantic Data Lifting") to execute these. Its command-line usage makes integration easy and modular. Automatic validation of generated RDF files can be achieved by any SHACL validator.

Adding a Semantic Service to Real-World Systems

This section introduces implementation details about the semantic check. The main idea is it to write client applications with the semantic check in mind. The client establishes a connection to the semantic server before it connects to the server it actually plans to communicate with. The semantic check receives the client's configuration together with the service call and then loads the server configuration file from the respective server. All the above-mentioned service calls and message types are defined through proto files while the client and server configuration can be found in JSON files.

The configuration of client and server have both the same unique format consisting of different levels. These levels can be seen in lst. 1. The first level is the name of the proto file which is being considered, an example for this would be *scannerservice*. The next level within the scanner service is then the name of the message. So far everything is structured like the corresponding proto file. The next and final level covers the variables within a message type with the variable values being URIs linking to the according ontology and describing the unit of measure belonging to that specific variable. In case of OM this would be the URI *om:millimetre*. In case of QUDT, however, we use *qudt:MilliM*, since this is the unique identifier within the Resource Description Framework (RDF) graph of QUDT and hence can be used for SPARQL queries to access more details about that specific unit.

We validate all relevant fields in the configuration files of the server and the client against each other and build our response to the client accordingly. We include a new proto file into the system for each implemented semantic capability, including the semantic check and the later explained unit conversion. The semantic check is a gRPC call and requires the contents of the config file as a string as input and then returns a response message consisting of multiple parts: A server response with the results of the check in form of a Boolean, a detailed description of what exactly went wrong in form of a string, and additional lists containing the previously assigned units and the desired correct units by the server.

Implementing a Unit Conversion Service

If the response of the semantic check includes a non-empty conversion list, the client proceeds by calling the unit conversion service. We implement the conversion service with QUDT units by querying the ontology to extract various properties of the units which were stored in the configuration files. While libraries offering unit conversion functionalities for OM are only available for Java and Python, we can query data directly from QUDT via SPARQL with many more programming languages, providing even more flexibility.

```
{
  "scannerservice": {
    "Position": {
      "x": "qudt:MilliM",
      "y": "qudt:MilliM"
    },
    "Time": {
      "timestamp": "wiki:Q14654"
    }
  }
}
```

Listing 1 Sample of a server configuration that adds semantic information via ontology URIs to Wikidata and QUDT, which we shortened to improve readability [3].

While the semantic check service only links to the utilized ontologies, the new unit conversion service extensively uses QUDT's structure. We know from the related work section that QUDT uses multiple concepts for the information it provides. For the unit conversion, the *conversionMultiplier* as well as the *conversionOffset* form the most important properties. A unit can only be converted from a specific source unit to a destination unit if the dimensions are the same. SI base units are used for the dimensions, like in OM. Especially the dimension property is valuable, by providing a mean to determine the base unit or to determine if we are dealing with a measure or scale. This ensures that conversions are both syntactically and semantically correct and avoids, for example, conversions from pounds (force) to kilograms (mass).

The unit conversion service takes as its input the lists that were returned with the semantic checks output. This includes for each list entry its identifiers, the source to convert from, and the destination to convert to. Additionally, the client sends its initial values for these variables. The query extracts the multiplier and offset of the previously assigned unit to its corresponding base, converts the value to the base unit and then converts the value from the base unit to the unit desired by the server. The conversion service finally returns the same input it got previously back to the client but this time with the converted values. The client can then proceed to a new semantic check or start the communication with the server.

Monitoring the Data Stream with a Continuous Validation Service

The server calls the continuous validation service every n seconds (can be freely configured). We implement the validation service by first proceeding with a conversion from the binary data to a JSON format. This is necessary so that in the next step, the data lifting can make the data readable for the validation step by semantically enriching it. The validation step then takes this data and checks the requirements. Based on the result, the error is pushed forward to an implemented handler, or the service returns positively. In the following, the data lifting and constraint-based validation are discussed in more detail.

Data Lifting

The data lifting in our scenario needs to transform Protobuf messages to RDF so that they can be validated via SHACL. Protobuf's built-in JsonFormatter[4] is capable of exporting Protobuf messages to JSON. Since the RMLMapper (cf. "Semantic Data Lifting") supports JSON input files, we apply it for our needs. A set of RML rules defines the respective mappings from the Protobuf messages to RDF, including valuable details about their unique identifiers (e.g., for units), properties, and structure. Note that one only needs to create an RML rule file once and then can apply it to every message from a given data stream. We integrate the RMLMapper into via command-line calls to automatically generate respective RDF files, which we then use for validation. An explicit call could look like `java -jar rmlmapper.jar -s turtle -m mapping.ttl -o output.ttl`, or one programmatically calls the respective method from a Java program using the same arguments.

Constraint-Based Validation

The constraint-based validation is set up by first establishing SHACL shapes, which describe the necessary requirements. These requirements can range and include the targeted QUDT unit quantity kinds like length for meter or kilometer and specific requirements for the values like ranges or filters but are not limited by that. In the validation step, the RDF data produced by the data lifting are combined with the QUDT ontology as a data graph and then validated against the described SHACL shapes. Such a validation method is present in many programming languages like Java, JavaScript, .NET or Python and can be, therefore, domain independent. On error, an implementable handler is called which in a simple case can just trigger a logging action for tracking the data quality but is not limited to that and could furthermore act as a firewall for stopping erroneous clients.

User Interaction and Workflow

Adding these new services adds a certain workflow to the system, when designing and implementing new applications. First, the system needs a server configuration for each server that is running available and on top of that every client that is being added to the system needs its own client configuration file. Whenever someone implements a new client, they also need to fill out their configuration file and have a section in their code where they load their own configuration and call the semantic check service with it. The normal client functionality is then be executed afterwards if no mismatches occurred. If there were errors, however, the client aborts communication and a developer needs to manually adjust the configuration file and its values based on the respective error message. Since our error messages show precise details, it is easy to track down the error and correct it accordingly. This process can be repeated until there are no errors or mismatches left. Since all errors are shown immediately all mismatches can be removed in one iteration.

[4] https://developers.google.com/protocol-buffers/docs/proto3#json.

Table 2 Real-world variables with their corresponding units in ontologies [3]

Variable name	Type	Unit of measure
preheatingTemp	double	qudt:DEG_C
laserSpeed	float	qudt:MilliM-PER-SEC
position.x/y	float	qudt:MilliM
shotTime	int64	wiki:Q14654

Demonstrator Setup

This section evaluates the differences to the previous state of the system after including the semantic components, the practical usability of the extended architecture, and its performance in form of a technical evaluation of the tools we used. The benefits will be explained on demo scenarios, showing the system's behavior before and after including semantic components. We finish this section with a conclusion regarding the benefits of semantically enriching such systems in general.

Implementation Evaluation

Above, we specified multiple requirements for extending SOA systems. The semantic description of utilized data was taken care of using ontologies like QUDT and linking to them in the configuration files. QUDT and OM sufficiently cover the support for units of measure. The only exception here are abstract values, for example the time values utilizing the UNIX stamp instead of regular time measurement. Hence, the first two points are already covered by the inclusion of the configuration files. The unit conversion is taken care of by adding the unit conversion service utilizing SPARQL queries on QUDT to the USP system. The new unit conversion service allows the conversion of units as long as source and destination are both in the same dimension. Table 2 shows the information that can be found for every variable after the inclusion of the semantic capabilities. The unit of the position values being millimeter can now be seen within the system.

In addition to these points, the human understanding of the system is also increased. Someone with access to the code will be able to much easier understand certain variables just by inspecting their definition, since everything is linked to a semantic description within an ontology. While this could have been covered in comments already, an ontology provides much more in-depth knowledge about certain concepts and even descriptive comments within the ontology and additional links to other concepts or even full ontologies. To conclude we can state that the above-mentioned requirements are fully satisfied by our implementation.

On the technical side of the implementation, however, there are consequences of our approach. A general first consequence of including semantics is the increase in data size that must be dealt with and slower processing time. Semantic data are included in text form and contains more data than just telling the system that a variable has a certain data type. A variable *temperature* does not just have a value such as *45* and the assigned datatype anymore but instead a link to an ontology in form of an URI, stored in multiple configuration files containing information about all the important variables across the system. The distribution and management of these configuration files, however, has still room for improvement as mentioned in the previous sections and can still change in the future. Even in the current form, however, the difference in execution time and used space is barely noticeable. Querying the ontology takes up most of the additional time and can be improved by adjusting the ontology to just our needs in the future. One final aspect of our technical evaluation is backwards-compatibility. Since the usage of the services for the semantic capabilities is not strictly required, one can still develop applications without using any semantic services.

Demonstrating Semantic Functionalities in Demo Scenarios

In the following, the results of applying the semantic functionalities in demo scenarios are presented in the context of the initial and continuous semantic validation workflow.

Initial Semantic Validation Workflow

The advantages of the extended system with the semantics can be emphasized if we directly compare the previous state of the system with the new one by creating and executing a client application with the old standard and one with the new workflow of going through a semantic check followed by an automatic unit conversion. For this reason, we create three client applications for a demonstration of the system capabilities. All three applications have the same goal of accessing the scanner service to call the *JumpToPosition* function, which orders the scanner to move the laser beam to the specified location on the scan field. For this purpose, the service takes a position argument consisting of x and y variables as input and returns the completion time when the execution of the service is finished. The existing server now has different understandings for the two components of this communication. The first component is the position argument with its two values having the unit *millimetre*, while the returned completion time is given in UNIX stamp. While all three applications eventually do exactly this, their behavior prior to the actual communication differs indeed.

Scenarios 1 and 2: Only syntax defined: The first and second demo applications immediately connect to the

Table 3 The scenarios compared to each other based on different standards for the communication in the system [3]. Especially note scenario 4 where LISSU prevents erroneous communication

Scenario	Expected	Baseline	LISSU
1 (y / -)	**warn**	**comm.**	**warn**
2 (n / -)	block	block	block
3 (y / y)	comm.	comm.	comm.
4 (y / n)	**block**	**comm.**	**block**

Bold values indicate improvements of LISSU compared to the baseline

scanner server and request the *JumpToPosition* service. The syntactic side is taken care of because of the strict definitions of the variables in the proto files but for any kind of semantic information the client must assume that the server uses the same specifications. In our demo scenario the client sends values with the knowledge of them being in *centimetre* and this then results in the server still interpreting it as a *millimetre* value and hence, moving the laser beam by a smaller amount than the user wanted.

Scenario 3: Syntax + matching semantics: The second application now uses our semantic components. This means the developer of the client application also provides a configuration file with their understanding of the variable semantic, which explicitly states their understanding of the positional arguments being understood as *centimetre* on the client side. Since now the first thing the client does is connecting to the semantic server and making a service call for the semantic check service, the difference in the semantic understanding will be spotted and the client notified to fix this. However, since the problem here is a positional argument, taking a length unit, we identify this specific problem within the configuration file as a problem that can be solved with the unit conversion service and hence, proceed by calling it with the position arguments as values to convert and then proceed with the communication to the scanner with the converted value. In this scenario, we did not just catch the error but instead also corrected it and proceeded with the execution of the initial goal without any problems. In this ideal scenario, the developer of the client application does not need much knowledge of the server-side semantics himself, instead they can just program their client by following the workflow proposed in the previous section.

Scenario 4: Syntax + mismatching semantics: The third scenario is created by a client application using the semantic components but with an incomplete client configuration file. Even in such a case the semantic check provides its advantages by telling the user what exactly is wrong with their configuration and what is missing, while the first case would not even realize the error and just proceed with the faulty values and cause a more vital error during the execution on the hardware, resulting in an error in the production.

Table 3 compares the different scenarios based on the standards of the system. The first and second are the previous state of the system where only the syntax was taken care of, and still work in every scenario, since our implementation does not change the Protobuf base of the system. However, we improve feedback by warning the user that only the syntax is validated, but there is no semantic information to validate. The third represents configurations where the system must check for both syntactic and semantic compatibility of client and server before initiating a communication. In the first case, we cannot give any information on this and start the communication with a risk of errors caused by a misunderstanding of the system semantics, while the second case manages to identify these semantics as correct and initiate the communication. The third case does give us information by identifying the semantics in the system as wrong and blocking the communication. The system should only initiate a client-server communication when they have a mutual understanding of the system. The results are here the same as in our approach, further proving the advantages of our implementation.

The semantic component mainly plays a role during two critical scenarios. The first one is when a newly written client application is introduced into the system. The developer of the application might have insufficient knowledge of the system, resulting in an incomplete configuration file or the usage of the wrong units for the arguments within the code. The semantic check would identify this and notify the developer. The second and more common scenario is a changed hardware component within the system. The complications of this were already explained with an example in the introduction section of this work. If we change the hardware of the scanner, we might have to write a new server code using a different server semantic, suiting the new hardware. A client code specifically written for the old server with no semantics included like our first demo application would fail in such a scenario but the second and third application would in the worst case at least catch the error and notify the user that something is wrong, and that the client must update its definition to satisfy the new system requirements coming with the new hardware.

To conclude this part of the evaluation, LISSU automatically avoids and corrects unit mismatches, and, therefore, leads to a more predictable communication in a SOA framework. Its backwards-compatibility allows a flexible integration even into large existing systems. Unclear cases, where only syntax but no semantic is defined, yield a warning but still operate. LISSU reports semantic mismatches between client and server to the calling client and prevents communication if needed. Although LISSU is completely backwards-compatible, we recommend applying it to all components of a SOA system to improve overall communication predictability.

Continuous Semantic Validation Workflow

The continuous semantic validation workflow first converts the binary format into a non-binary form for making the lifting possible. The resulting JSON file has a format which can be seen in lst. 2 and contains especially the points and the marking parameters, which all are implicitly defined as certain units. The mapping approach, therefore, accounts for the client configuration and implicit knowledge for lifting the JSON data to semantic data.

Finally, the validation part is evaluated, and the advantages of a continuous semantic validation workflow are shown. For this, we inspect different requirements provided by the use-case and evaluate if these are validated properly:

```
{
  "workPlanes": [
    {
      "vectorBlocks": [
        {
          "hatches3d": {
            "points": [
              0.585648,
              0.648475,
              -0.05
            ]
          },
          "markingParamsKey": 1
        }
      ],
      "numBlocks": 1
    }
  ],
  "jobMetaData": {
    "jobName": "Batsymbol_2mm"
  },
  "markingParamsMap": {
    "0": {},
    "1": {
      "laserSpeedInMmPerS": 100.0,
      "jumpSpeedInMmS": 1000.0,
      "jumpDelayInUs": 100.0,
      "laserOffDelayInUs": 100.0,
      "laserOnDelayInUs": 100.0
    }
  },
  "numWorkPlanes": 1
}
```

Listing 2 Sample of non-binary JSON data produced by our extraction.

A simple Java program based on the RMLMapper maps the non-binary JSON data using an RML definition, which is shown in lst. 3. The part shown is the mapping of the points which are mapped into x, y and z coordinates and are assigned the unit of millimeter in QUDT. The value itself is furthermore defined in QUDT as well, completing the lifting of the points. The other parts of the JSON are lifted in similar manner.

Value Ranges: It is useful to compare the value ranges of the provided data points. This is, therefore, described in lst. 4 as SHACL shapes and aims for a value range of -10 to 10.

```
:PointMap a rr:TriplesMap;
    rml:logicalSource :PointSource;
    rr:subjectMap [rr:class :Point; rr:termType rr:BlankNode];
    rr:predicateObjectMap
        [rr:predicate ex:x; rr:objectMap [rr:parentTriplesMap :PointXMap]],
        [rr:predicate ex:y; rr:objectMap [rr:parentTriplesMap :PointYMap]],
        [rr:predicate ex:z; rr:objectMap [rr:parentTriplesMap :PointZMap]].

:PointXMap a rr:TriplesMap;
    rml:logicalSource :PointSource;
    rr:subjectMap [rr:class qudt:Quantity; rr:termType rr:BlankNode];
    rr:predicateObjectMap
        [rr:predicate qudt:unit; rr:object unit:MilliM],
        [rr:predicate qudt:value; rr:objectMap [rml:reference "x";  rr:datatype xsd:decimal]].
```

Listing 3 Example excerpt of an RML mapping definition for lifting JSON data to RDF.

```
ex:Position
  a rdfs:Class, sh:NodeShape ;
  sh:property
    [
      sh:path ex:x ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:node ex:XMeterQuantity ;
    ] .

ex:XMeterQuantity
  a sh:NodeShape ;
  # Additionally references to quantitykind:Length
  sh:property
    [
      sh:path qudt:value ;
      sh:minExclusive -10 ;
      sh:maxExclusive 10 ;
      sh:datatype xsd:decimal ;
    ] .
```

Listing 4 Sample of shapes defining the constraint that x-values of positions must be within -10 and 10. Note that SHACL can model complex constraints including logical operators and cross-attribute relations, too.

Zero Checks: In a real-life scenario, there is the issue that the x and y coordinates of a point are not allowed to be zero. Therefore, the shapes in lst. 5 define this kind of constraint.

```
ex:Position
  a rdfs:Class, sh:NodeShape ;
  sh:not [
    sh:and (
      [
        sh:path ex:x ;
        sh:node ex:ZeroValue ;
      ]
      [
        sh:path ex:y ;
        sh:node ex:ZeroValue ;
      ]
    )
  ] .

ex:ZeroValue
  a sh:NodeShape ;
  sh:property
    [
      sh:path qudt:value ;
      sh:minInclusive 0 ;
      sh:maxInclusive 0 ;
      sh:datatype xsd:decimal ;
    ] .
```

Listing 5 Sample of shapes that fulfill the constraint that the x and y paths of a position cannot both have values equal to 0.

We perform time measurements for the above-mentioned full mapping scenario from practice, which yield the following results on a 64-bit Windows 10 Intel i7-8650U CPU @ 1.90GHz with 32 GB RAM. The experimental mapping file contains 18 triples maps that extract up to 24 different predicates like constructions of xyz-points or unit assignments. We conduct five runs for differently many input files and take measurements via Java's precise `System.nanoTime()` method. The median runtime for 101 ms for 1 file, 366 for 10 files, and 2196 for 100 files. The standard deviation is relatively high (360 / 278 / 830 ms), which we explain by Java's one-time class loading overhead, as the first runs always take much longer. Putting these numbers in the context of the real-world use-case, the mapping times between 22 and 101 ms need to be improved further. That is due to the high rate at which SOA messages in the USP use-case occur: JumpToPosition often occur at 1–100 Hz, but special products with many partial commands get over 1 kHz. Sensor reads possibly send around 100–300 Hz, while camera images and other sensors can even reach multiple kHz. We conclude from there time measurements that our current approach with periodic sampling is well-suited, but also hint that these execution times can be tuned in future work (e.g., by caching the mapping file).

Shopfloor Integration of the Proposed Continuous Semantic Validation System

The evaluated validation system can be used in multiple ways to ensure a high quality and error free production system. Depending on the "quality" of the defined rule set, multiple scenarios can be implemented. These scenarios are:

- Online monitoring of production data streams
- Creation of manufacturing firewalls for industrial services

Online monitoring of production data streams allows the continuous evaluation of the usage of a system. The traffic towards the scanning microservice is mirrored to the validation system and analyzed independently. The original system is not disturbed but a monitoring possibility is introduced. In case of the laser scanning system violations of specific rules can be tracked and counted. These analyses, therefore, build the foundation for dashboards or other real-time visualization systems which give the machine operator real-time feedback on the state of his process. More advanced systems which track the condition of the machine based on sophisticated rules could be implemented since a breach of a rule could lead to shorter expected lifetimes of components, for instance due to thermal stress.

The **creation of manufacturing firewalls for industrial services** takes the concept of continuous validation one step further and blocks request that do not pass the validation step. In this scenario every call to a service is analyzed and forwarded if succeed, therefore, putting additional requirements on the validation software. Especially computing time becomes a critical component since it directly influences the production speed. The quality of the validation rule set should be high to prevent machine stops that are unnecessary and, therefore, costly. The defined rules could for example be validated on example datasets to avoid false alerts.

The demonstrator discussed in this paper targets especially the first use-case in which a monitoring system is implemented that analyzes data streams with Semantic Web technologies regularly.

Conclusion and Future Work

The goal of this work was to handle semantic mismatches between services in SOA frameworks. These mismatches could occur during the initial set-up of the services or during runtime. We fist focused on unit mismatches, as these can already lead to critical results in practice during machine setup. We proposed LISSU, lightweight Semantic Web Services for units, which allows developers specify semantics (e.g., units) for their services via URI ontology references. In addition to existing syntactic validations, we added an initial semantic validation workflow that detects and corrects unit mismatches automatically. The correction can be done via an automatic unit conversion service that we built on top of the QUDT ontology in this work. The workflow was then extended to also allow formulating message constraints based on SHACL. These constraints built the foundation of a continuous semantic validation framework which allows the validation of individual messages send to a service. This workflow introduced a data lifting process from Protobuf messages to RDF.

We demonstrated our approach in a real-world use-case based on gRPC in the USP laser domain. Core findings are that our approach is backwards-compatible with existing gRPC and other SOA solutions, and adds an additional validation layer based on semantics. We thereby avoid semantic mismatches including unit mismatches and guarantee a more predictable communication in SOA setups. It is also possible to inject continuous semantic monitoring into existing message streams which can be formulated by SHACL. We also discussed possible integrations of LISSU into real-world manufacturing.

There are possibilities to extend our results. The distribution and management of configuration and constraints files could be improved. Using external tools would enable benefits including easier access to these files with possibly even a graphical user interface that assists finding and editing. Storing these files in databases would, however, require an adaption of the implementation regarding data access. Another possibility is to inject configurations and constraints into microservice orchestration systems like Kubernetes or Openshift. Furthermore, the generation of these configurations could be improved. Instead of manually creating the semantic configuration files, a generator could guide developers while creating these, and instantly validate these. Plus, one could add additional ontologies in the system, or even introduce new domain ontologies to also cover other semantic mismatches besides units. So far, we only utilize unit conversion capabilities, but current solutions offer more features that could be utilized. Finally, a more detailed evaluation of our constrained-based validation's performance with a focus on benefits vs. added computation complexity would further improve comparability with existing methods.

We conclude that LISSU provides a backwards-compatible semantic extension for SOA frameworks that is based on Semantic Web concepts and leads to a more predictable and stable communication during setup and runtime.

Funding Open Access funding enabled and organized by Projekt DEAL. This study was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2023 Internet of Production – 390621612.

References

1. Berners-Lee T, Hendler J, Lassila O. The Semantic Web. Sci Am. 2001;284(5):34–43.
2. McIlraith SA, Son TC, Zeng H. Semantic web services. IEEE Intell Syst. 2001;16(2):46–53.
3. Lipp J, Sakik S, Kröger M, Decker S. LISSU: Integrating Semantic Web Concepts into SOA Frameworks. In: Proceedings of the 23rd International Conference on Enterprise Information Systems - Volume 1: ICEIS, 2021; pp. 855–865. 10.5220/0010481408550865. INSTICC.
4. Pennekamp J, Glebke R, Henze M, Meisen T, Quix C, Hai R, Gleim L, Niemietz P, Rudack M, Knape S, et al. Towards an Infrastructure Enabling the Internet of Production. In: 2019 IEEE International Conference on Industrial Cyber Physical Systems (ICPS), 2019; pp. 31–37. IEEE.
5. Google: gRPC. https://grpc.io/ 2016.
6. Google: Protocol Buffers. https://developers.google.com/protocol-buffers/ 2015.
7. Google: Bazel. https://bazel.build/ 2015.
8. Toesko G, Dehnert C. Femtosecond laser optics combat pulse dispersion, color errors, and reflections. Pulse 1, 0 2016.
9. Byskov-Nielsen J, Savolainen J-M, Christensen MS, Balling P. Ultra-short pulse laser ablation of metals: threshold fluence, incubation coefficient and ablation rates. Appl Phys A. 2010;101(1):97–101.
10. Gyrard A, Serrano M, Atemezing GA. Semantic Web Methodologies, Best Practices and Ontology Engineering Applied to Internet of Things. In: 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT), 2015; pp. 412–417
11. Lipp J, Gleim L, Decker S. Towards reusability in the semantic web: decoupling naming, validation, and reasoning. In: 11th Workshop on Ontology Design and Patterns (WOP2020) Co-located with 19th International Semantic Web Conference (ISWC 2020), Virtual Conference, 2020; November 01–06, 2020.
12. Schlenoff C, Hong T, Liu C, Eastman R, Foufou S. A literature review of sensor ontologies for manufacturing applications, 2013; pp. 96–101. https://doi.org/10.1109/ROSE.2013.6698425.
13. Eid M, Liscano R, El Saddik A. A universal ontology for sensor networks data. In: 2007 IEEE International Conference on Computational Intelligence for Measurement Systems and Applications, 2007; pp. 59–62.
14. Niles I, Pease A. Towards a standard upper ontology. In: Proceedings of the International Conference on Formal Ontology in Information Systems-Vol. 2001, 2001; pp. 2–9.
15. Russomanno DJ, Kothari C, Thomas O. Sensor ontologies: from shallow to deep models. In: Proceedings of the Thirty-Seventh Southeastern Symposium on System Theory, 2005. SSST'05., 2005; pp. 107–112. IEEE.
16. Compton M, Barnaghi P, Bermudez L, García-Castro R, Corcho O, Cox S, Graybeal J, Hauswirth M, Henson C, Herzog A, et al. The SSN ontology of the W3C semantic sensor network incubator group. J Web Semant. 2012;17:25–32.
17. de Koning HP. Library for quantity kinds and units: schema, based on QUDV model OMG SysML(TM), Version 1.2. https://www.w3.org/2005/Incubator/ssn/ssnx/qu/qu 2005.
18. Rijgersberg H, Wigham MLI, Top J. How semantics can improve engineering processes: a case of units of measure and quantities. Adv Eng Info. 2011;25:276–87. https://doi.org/10.1016/j.aei.2010.07.008.
19. Rijgersberg H, van Assem M, Top J. Ontology of units of measure and related concepts. Semant Web. 2013;4(1):3–13.
20. QUDT: QUDT. http://www.qudt.org/ 2014.
21. Schadow G, McDonald CJ. The unified code for units of measure. Regenstrief Institute and UCUM Organization: Indianapolis, IN, USA; 2009.
22. Rijgersberg H, Top J, Meinders MBJ. Semantic support for quantitative research processes. IEEE Intell Syst. 2009;24(1):37–46. https://doi.org/10.1109/MIS.2009.17.
23. Taylor B. Guide for the use of the International System of Units (SI): The Metric System, 1995; DIANE Publishing.
24. Thompson A, Taylor BN. Guide for the Use of the International System of Units (SI). National Institute of Standards and Technology: Technical report; 2008.
25. Schlosser M, Sintek M, Decker S, Nejdl W. A scalable and ontology-based P2P infrastructure for semantic web services. In: Proceedings. Second International Conference on Peer-to-Peer Computing, 2002; pp. 104–111. IEEE.
26. Kopeckỳ J, Vitvar T, Bournez C, Farrell J. SAWSDL: semantic annotations for WSDL and XML schema. IEEE Internet Comput. 2007;11(6):60–7.
27. Fensel D, Facca FM, Simperl E, Toma I. Lightweight semantic web service descriptions, pp. 279–295. Springer, Berlin, Heidelberg; 2011. https://doi.org/10.1007/978-3-642-19193-0_12.
28. Roman D, Kopeckỳ 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.
29. Fensel D, Fischer F, Kopeckỳ J, Krummenacher R, Lambert D, Vitvar T. WSMO-Lite: lightweight semantic descriptions for services on the Web. W3C Member Submission August 2010.
30. Bennara M. Linked service integration on the semantic web. PhD thesis, Université de Lyon; 2019.
31. Birrell AD, Nelson BJ. Implementing remote procedure calls. ACM Trans Comput Syst. 1984;2(1):39–59. https://doi.org/10.1145/2080.357392.
32. Martín-Recuerda F, Walther D, Eisinger S, Moore G, Andersen P, Opdahl PO, Hella L Revisiting Ontologies of Units of Measure for Harmonising Quantity Values–A Use Case. In: International Semantic Web Conference, Springer; 2020, pp. 551–567.
33. JSON Schema: JSON Schema. https://json-schema.org/ 2022
34. Kontokostas D, Knublauch H. Shapes constraint language (SHACL). W3C recommendation, W3C July 2017. https://www.w3.org/TR/2017/REC-shacl-20170720/
35. Dimou A, Vander Sande M, Colpaert P, Verborgh R, Mannens E, Van de Walle R. RML: a generic language for integrated RDF mappings of heterogeneous data. In: Ldow 2014.
36. Dimou A, Vander Sande M, Colpaert P, Verborgh R, Mannens E, Van de Walle R. RDF Mapping Language (RML). W3C, Unofficial Draft 2020;15.
37. Krech, D.: RDFLib. https://rdflib.dev/ 2002.