





LISSU: Integrating Semantic Web Concepts into SOA Frameworks

Johannes Lipp^{1,3}^a, Siyabend Sakik¹^b, Moritz Kröger²^c and Stefan Decker^{1,3}^d

¹Chair of Information Systems, RWTH Aachen University, Aachen, Germany

²Chair for Laser Technology LLT, RWTH Aachen University, Aachen, Germany

³Fraunhofer Institute for Applied Information Technology FIT, Sankt Augustin, Germany

Keywords: Semantic Web Services, Service-oriented Architecture, Semantic Web, Internet of Things.

Abstract: In recent years, microservice-based architectures have become the de-facto standard for cloud-native applications and enable modular and scalable systems. The lack of communication standards however complicates reliable information exchange. While syntactic checks like datatypes or ranges are mostly solved nowadays, semantic mismatches (e.g., different units) are still problematic. Semantic Web Services and their derivatives tackle this challenge but are mostly too ambitious for practical use. In this paper, we propose *Lightweight Semantic Web Services for Units* (LISSU) to support Semantic Web experts in their collaboration with domain experts. LISSU allows developers specify semantics for their services via URI ontology references, and automatically validates these before initiating communication. It automatically corrects unit mismatches via conversions whenever possible. A real-world demonstrator setup in the manufacturing domain proves that LISSU leads to a more predictable communication.


1 INTRODUCTION


In the last decades, the Semantic Web (Berners-Lee et al., 2001) has gained much popularity. Its extension called Semantic Web Services (McIlraith et al., 2001), 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.


We see our work in the area of SWS but have a smaller scope, because we aim for concrete and feasible solutions. Our goal is to resolve real-world problems in an example scenario from the engineering domain, in particular when client and server have a semantic mismatch (e.g., different units). State-of-the-art solutions do not solve such mismatches and developers try to tackle this via unstructured, human-readable data such as comments or documentations. Existing syntax validations at service interfaces must be extended with machine-readable semantic ones to make communication more predictable.


We propose 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. Our extended validation workflow detects unit mismatches between client and server and even corrects these via automatic conversions if possible. This provides an interoperable and consistently predictable communication among all components, which we prove in a real-world demonstrator setup with machine components of a laser system.

The remainder of our paper is structured as follows. The next section gives a motivating example and defines our goals. Section 3 investigates related

^a <https://orcid.org/0000-0002-2639-1949>

^b <https://orcid.org/0000-0002-5190-5753>

^c <https://orcid.org/0000-0001-7476-6226>

^d <https://orcid.org/0000-0001-6324-7164>

work with a special focus on ontologies and SWS. Section 4 presents our approach and Section 5 shows a real-world implementation. We conduct an evaluation based on this setup in Section 6 via a demonstrator and finally conclude in Section 7 with a summary of our work and possible future work in this field.

2 MOTIVATING EXAMPLE

This section motivates the extension of SOA with semantic capabilities based on SWS by presenting an example from USP laser system development and showcasing concrete challenges that we tackle in this work, particularly unit mismatches.

2.1 Semantic Mismatch (Units)

Within the research project "Internet of Production" (Pennekamp et al., 2019), Semantic Web experts and laser experts collaboratively build a ultrashort pulse (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 "revers 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 Protocol Buffers (Protobuf), 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 millimeter. A new component and its respective new service however could interpret the same data value as 2 centimeter and thus move the laser wrongly or even damage the product. Other examples from 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 gRPC (Google, 2016) for the communication

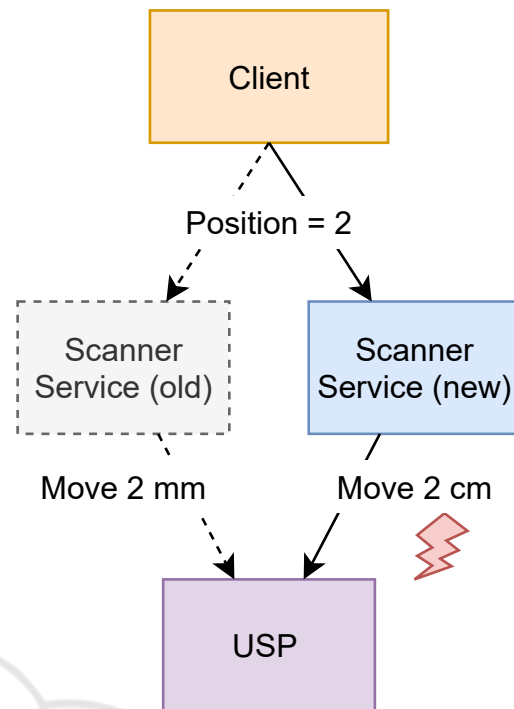


Figure 1: Unit mismatch we observed during a component swap at a USP laser system. The new server implementation internally uses different units and hence moves the laser for 2 centimeters instead of 2 millimeters.

in the network. This allows the serialization of data with Protobuf, a Data Description Language made by Google (Google, 2015b). Additionally, the system uses Bazel (Google, 2015a) 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 also not machine-readable. The above-mentioned motivating examples demonstrates the need to avoid semantic mismatches by properly defining and validating units at service interfaces.

2.2 Requirements and Goals

The ultimate goal of this work is to overcome semantic mismatches in SOA frameworks by integrating Semantic Web components into these. In particular, we aim to extend the existing syntax-only validation to also cover semantics and units in particular. Developers must be able to configure this information for both clients and servers to make service calls more predictable. These need to automatically validate the given semantic information before initiating the actual communication. This additional verification must be backwards-compatible, because SOA frameworks are usually distributed systems, and one developer can only modify some of the services. All interactions with the novel solution must be easily accessible for domain experts, which are typically non-experts in the Semantic Web. The 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.

3 RELATED WORK

This section investigates relevant ontologies and semantic enrichment including Semantic Web services.

3.1 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 (Gyrard et al., 2015). We aim for ontology terms that we can reuse in a modular approach without unwanted side effects as described in (Lipp et al., 2020). 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.

3.1.1 Sensor Ontologies

The article (Schlenoff et al., 2013) 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 also an ontology. Relevant ontologies mentioned are the Sensor Data Ontology (SDO) (Eid et al., 2007), which uses SUMO (Niles and Pease, 2001) as upper ontology, the OntoSensor ontology (Russomanno et al., 2005) and especially the Semantic Sensor Network (SSN) ontology (Compton et al., 2012). 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 (Compton et al., 2012). It allows to focus 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* (de Koning, 2005).

3.1.2 Unit Ontologies

The Ontology of Units of Measure (OM) (Rijgersberg et al., 2011; Rijgersberg et al., 2013) 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 (Rijgersberg et al., 2009). It is based on existing standards for units of measure such as (Taylor, 1995) 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 particular 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.

Table 1: Evaluation of relevant ontologies based on our requirements from Section 2.2 with OM and QUDT ranked highest. Note that the rating (- / o / +) is specific for our requirements and is not a universal ranking.

Ontology	Relevance	Coverage	Features	Flexibility
SDO	-	o	-	-
OntoSensor	-	o	-	-
SSN	o	+	+	o
OM	+	+	+	o
QUDT	+	+	+	+
UCUM	+	o	+	o

sion as a combination of certain base units, like the International System of Units (Thompson and Taylor, 2008). All other dimensions can then be computed from these base units.

The *Quantities, Units, Dimensions, and Types Ontology* (QUDT) (QUDT, 2014) 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 (Eid et al., 2007; Schadow and McDonald, 2009) 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 adopt to future requirement changes, while the SSN ontology primarily supports sensors and service calls instead of units.

3.2 Communication and Service Description

SWS (McIlraith et al., 2001), their organized peer-to-peer extensions (Schlosser et al., 2002), 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 (Kopecký et al., 2007) 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 (Fensel et al., 2011) and (Roman et al., 2015), which are also available as W3C submission (Fensel et al., 2010), as they still include functional,

non-functional and behavioral semantics. (Bennara, 2019) 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 (Birrell and Nelson, 1984). Note that this concept was introduced nearly four decades ago but has become an active research topic again recently. Google offers gRPC (Google, 2016), 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 (Google, 2015b).

One concrete example is an extension of a closed-source platform for deployment, integration, and orchestration digital services with semantic unit information (Martín-Recuerda et al., 2020). 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.

4 CONCEPT

This section presents our concept to integrate ontology terms into a microservice-based SOA system to handle unit mismatches. That includes an extended architecture for SOA, novel semantic 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 gRPC 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 Figure 2. We add semantic units to both client and server, which link to ontologies and use ontology URIs to spec-

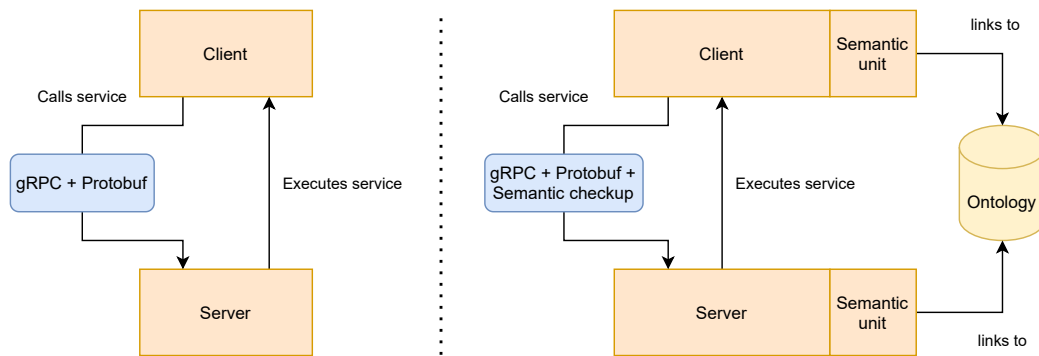


Figure 2: The prior communication on the left only includes a syntactic validation via Protobuf only. Our contribution on the right adds novel semantic components to both client and server, and upgrades the validation.

ify 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. The following sections present further details on the architecture and workflow.

4.1 Adding Lightweight Semantic Components to SOA Frameworks

This section explains more details on the components depicted in Figure 2, namely the semantic unit configuration, ontology references, and novel semantic validations. Section 4.2 then gives even more details on the 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 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.

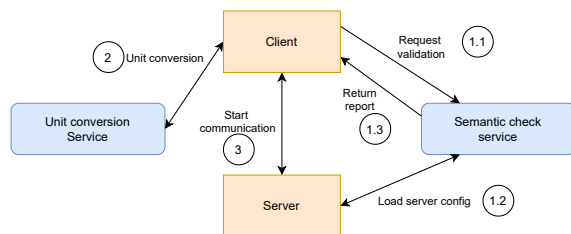


Figure 3: A client contacts up to two semantic services to make the subsequent communication more predictable.

We implement an additional step in the workflow of the SOA communication, which is illustrated in Fig-

ure 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. Finally, (3) the client starts the actual communication with the server.

4.2 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 actions can be required at the client, as illustrated in Figure 4.

- *Client Configuration Incomplete*: The provided configuration is missing properties required from the server. The client has to 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

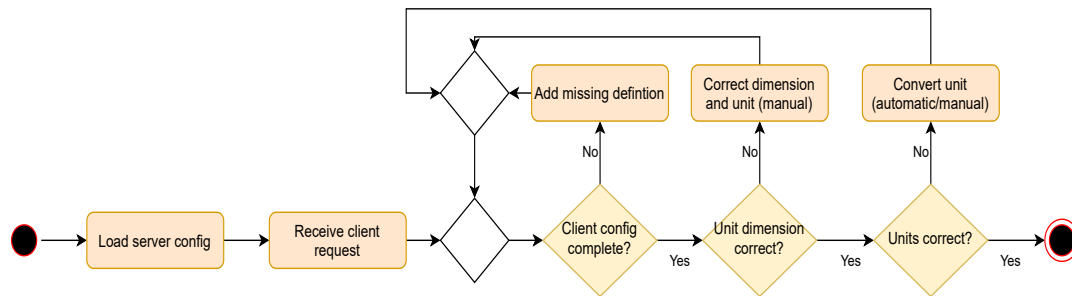


Figure 4: Proposed semantic validation workflow, which detects and recovers from semantic mismatches to finally establish a predictable communication.

incomplete configurations or dimension mismatches usually need to be solved manually, we can automatically resolve unit mismatches within the correct dimensions.

5 REALIZATION

In this section, we demonstrate our realization of the concept proposed above. This includes justifying a selection of concrete tools based on our requirements. We then present in detail the implementation of the two new components: First, the 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. Finally, we suggest a user interaction and workflow with the new system.

5.1 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. The reason for additionally using JSON was mostly because of higher expressiveness and increased human readability, plus 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

```

{
  "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.

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 by using the Python library rdflib (Krech, 2002) or via other programming languages like Java or Go.

5.2 Extending a Real-world System with a Service for Semantics

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 Listing 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 specifically for all the semantic capabilities implemented, 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 actually desired units by the server.

5.3 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, giving us even more flexibility.

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 pro-

viding 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.

5.4 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 the developer needs to adjust the configuration file and according values based on the displayed 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.

6 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 systems behavior before and after including the semantic

Table 2: Real-world variables with their corresponding units in ontologies.

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

components. We finish this section with a conclusion regarding the benefits of semantically enriching such systems in general.

6.1 Implementation Evaluation

Above we specified multiple requirements for the extended state of the USP system. The semantic description of the utilized data was taken care of by 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 consequence of including semantics is first of all the increase in data size that has to be dealt with and slower processing time. Semantic data is 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.

6.2 Demonstrating Semantic Functionalities in Demo Scenarios

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 in order 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 units *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 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 chapter.

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 has to 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 an misunderstanding of the system semantics, while the second case manages to identify these semantics as correct and initiate the communi-

Table 3: The scenarios compared to each other based on different standards for the communication in the system.

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

cation. 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 in order to satisfy the new system requirements coming with the new hardware.

To conclude 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.

7 CONCLUSION AND FUTURE WORK

The goal of this work was to handle semantic mismatches between services in a SOA framework. We focused on unit mismatches, as these can already lead to critical results in practice. 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 a semantic validation 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.

We demonstrate 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, but 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.

There are possibilities to extend our implementation. First of all the distribution and management of the configuration files could be improved. Using external tools here would come with multiple benefits including easier access to the configuration files with possibly even a graphical user interface providing means to find and edit the various configuration files in the system. Storing the configuration files in databases would, however, require an adaption of the implementation so far regarding loading and sending data within the system. Another possibility is to inject these configurations into microservice orchestration systems like Kubernetes or OpenShift.

Not only the management of the configuration files could be further improved, but also their generation. Instead of manually creating the semantic configuration files, a configuration generator could guide developers while creating these, and instantly validate their structure and completeness. Further improvements could use additional ontologies in the system, or even introduce 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.

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

ACKNOWLEDGEMENTS

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2023 Internet of Production – 390621612.

REFERENCES

- Bennara, M. (2019). *Linked Service Integration on the Semantic Web*. PhD thesis, Université de Lyon.
- Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. *Scientific American*, 284(5):34–43.
- Birrell, A. D. and Nelson, B. J. (1984). Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59.
- Compton, M., Barnaghi, P., Bermudez, L., García-Castro, R., Corcho, O., Cox, S., Graybeal, J., Hauswirth, M., Henson, C., Herzog, A., et al. (2012). The SSN Ontology of the W3C Semantic Sensor Network Incubator Group. *Journal of Web Semantics*, 17:25–32.
- de Koning, H. P. (2005). 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>.
- Eid, M., Liscano, R., and El Saddik, A. (2007). A Universal Ontology for Sensor Networks Data. In *2007 IEEE International Conference on Computational Intelligence for Measurement Systems and Applications*, pages 59–62.
- Fensel, D., Facca, F. M., Simperl, E., and Toma, I. (2011). *Lightweight Semantic Web Service Descriptions*, pages 279–295. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Fensel, D., Fischer, F., Kopecký, J., Krummenacher, R., Lambert, D., and Vitvar, T. (2010). WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web.
- Google (2015a). Bazel. <https://bazel.build/>.
- Google (2015b). Protocol Buffers. <https://developers.google.com/protocol-buffers/>.
- Google (2016). gRPC. <https://grpc.io/>.
- Gyrard, A., Serrano, M., and Atemez, G. A. (2015). 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)*, pages 412–417.
- Kopecký, J., Vitvar, T., Bournez, C., and Farrell, J. (2007). SAWSDL: Semantic Annotations for WSDL and XML Schema. *IEEE Internet Computing*, 11(6):60–67.
- Krech, D. (2002). RDFLib. <https://rdflib.dev/>.
- Lipp, J., Gleim, L., and Decker, S. (2020). 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, November 01-06, 2020*.

- Martín-Recuerda, F., Walther, D., Eisinger, S., Moore, G., Andersen, P., Opdahl, P.-O., and Hella, L. (2020). Revisiting Ontologies of Units of Measure for Harmonising Quantity Values—A Use Case. In *International Semantic Web Conference*, pages 551–567. Springer.
- McIlraith, S. A., Son, T. C., and Zeng, H. (2001). Semantic Web Services. *IEEE intelligent systems*, 16(2):46–53.
- Niles, I. and Pease, A. (2001). Towards a Standard Upper Ontology. In *Proceedings of the international conference on Formal Ontology in Information Systems—Volume 2001*, pages 2–9.
- Pennekamp, J., Glebke, R., Henze, M., Meisen, T., Quix, C., Hai, R., Gleim, L., Niemietz, P., Rudack, M., Knape, S., et al. (2019). Towards an Infrastructure Enabling the Internet of Production. In *2019 IEEE International Conference on Industrial Cyber Physical Systems (ICPS)*, pages 31–37. IEEE.
- QUDT (2014). QUDT. <http://www.qudt.org/>.
- Rijgersberg, H., Top, J., and Meinders, M. (2009). Semantic Support for Quantitative Research Processes. *IEEE Intelligent Systems*, 24(1):37–46.
- Rijgersberg, H., van Assem, M., and Top, J. (2013). Ontology of Units of Measure and Related Concepts. *Semant. Web*, 4(1):3–13.
- Rijgersberg, H., Wigham, M., and Top, J. (2011). How Semantics Can Improve Engineering Processes: A Case of Units of Measure and Quantities. *Advanced Engineering Informatics*, 25:276–287.
- Roman, D., Kopecký, J., Vitvar, T., Domingue, J., and Fensel, D. (2015). WSMO-Lite and hRESTS: Lightweight semantic annotations for Web services and RESTful APIs. *Journal of Web Semantics*, 31:39–58.
- Russomanno, D. J., Kothari, C., and Thomas, O. (2005). Sensor Ontologies: From Shallow to Deep Models. In *Proceedings of the Thirty-Seventh Southeastern Symposium on System Theory, 2005. SSST'05.*, pages 107–112. IEEE.
- Shadow, G. and McDonald, C. J. (2009). The Unified Code for Units of Measure. *Regenstrief Institute and UCUM Organization: Indianapolis, IN, USA*.
- Schlenoff, C., Hong, T., Liu, C., Eastman, R., and Foufou, S. (2013). A Literature Review of Sensor Ontologies for Manufacturing Applications. pages 96–101.
- Schlosser, M., Sintek, M., Decker, S., and Nejd, W. (2002). A Scalable and Ontology-based P2P Infrastructure for Semantic Web Services. In *Proceedings. Second International Conference on Peer-to-Peer Computing.*, pages 104–111. IEEE.
- Taylor, B. (1995). *Guide for the Use of the International System of Units (SI): The Metric System*. DIANE Publishing.
- Thompson, A. and Taylor, B. N. (2008). Guide for the Use of the International System of Units (SI). Technical report.