applied sciences

MDPI

Article

Service Interface Translation. An Interoperability Approach

Cristina Paniagua

Embedded Intelligent Systems LAB. of Technology, Department of Computer Science, Electrical and Space Engineering, Luleå University, 971 87 Luleå, Sweden; cristina.paniagua@ltu.se

Abstract: Interoperability plays an important role in Industry 4.0. Interoperability in the engineering process allows the automation of the engineering phase, reducing the human effort involved and the associated engineering costs. It improves the quality of the engineering process and its overall efficiency. Nevertheless, the diversity of available standards, devices, and systems leads to great levels of heterogeneity and makes it difficult to achieve the aforementioned interoperability. As the lack of interoperability increases, a generic solution to the problem is increasingly demanded by the industry. This paper approaches the interoperability problem from a service interface perspective. A novel approach is presented to address service interface heterogeneity. The proposed solution is based on service interface translation, which is achieved via the generation of service interfaces. A new system, the consumer interface generator system, has been designed and implemented to generate interface instances to solve the interoperability mismatches between service consumers and providers at runtime. In this paper, the autonomous consumer interface generation process, the system architecture, and the generated interface instance are described. The proposed approach has been validated through practical experimentation, including the implementation of a system prototype and a testbed.

Keywords: SOA; interoperability; service interfaces; Industry 4.0; IoT; cyber physical systems

Citation: Paniagua, C. Service Interface Translation. An Interoperability Approach. *Appl. Sci.* **2021**, *11*, 11643. https://doi.org/10.3390/app112411643

Academic Editors: Javier García-Heras Carretero and José María Alvarez Rodríguez

Received: 22 October 2021
Accepted: 6 December 2021
Published: 8 December 2021

1. Introduction

The emergence of the Fourth Industrial Revolution, Industry 4.0, represents a major paradigm shift for manufacturing industry scenarios. Industry 4.0 focuses on the digitization and integration of all physical processes across the entire organization [1]. Decentralized manufacturing, far from the previous centrally controlled automation, enables smart factories to effectively handle growing complexities and increase production efficiency.

The Industrial Internet of Things (IIoT) and cyber-physical systems (CPSs) are attempts to make Industry 4.0 scalable, robust, flexible, and secure. The IIoT covers all aspects of networked intelligent manufacturing systems, including a wide range of application domains and user requirements. Nevertheless, to fully obtain the potential of these technologies, interoperability between heterogeneous systems is an essential requirement.

The diversity of available robots, IIoT devices and systems leads to great levels of heterogeneity and makes integration difficult. In addition, each solution provides its own IoT infrastructure, devices, APIs, and data formats, leading to interoperability issues. Interoperability can be understood as the seamless connection and communication of heterogeneous systems and software components. The lack of interoperability between platforms, systems, and applications is a serious problem that prevents the adoption and deployment of Industry 4.0. and translates into significant engineering costs.

Reducing the engineering time dedicated to the integration and connection of heterogeneous systems is thus a key task to increase productivity and efficiency. This task is even more important in regard to the integration of legacy systems and the collaboration between vertical domains.

Interoperability problems are particularly relevant for systems of systems (SoS). A SoS can be defined as a collection of independent systems and their interrelationships gathered together to obtain an emergent behavior greater than the sum of its individual parts [2]. Technology diversity in SoS is an indicator of innovation and added value. The associated fragmentation, in contrast, represents an obstacle for the IoT and SoS technologies to be adopted and has a negative impact on the market's solidity.

The evolution of globally connected markets indicates the need to reduce the level of isolation of different silos, increase the amount of shared information, and adopt SoS solutions that can seamlessly work together. Manyika et al. [3] estimated that 40% of the potential benefits of IoT and SoS solutions can be obtained with interoperability between systems, highlighting interoperability as a key SoS technology enabler.

As the lack of interoperability increases, a generic solution to the problem is increasingly demanded. The interoperability issues that industry faces have multiple causes. This paper approaches the problem from a service interface perspective. The presented solution relies on the assumption that the heterogeneity in the interfaces acts as a barrier during the integration of heterogeneous systems and legacy technologies; thus, service interfaces enable communication between systems. The aim of this study was to find a solution for the service interface translation between heterogeneous systems.

Automatic service interface translation across protocols, information models, and standards needs to be addressed to fulfill industry demands, such as flexible reconfigurations of production systems and optimizations across different models, standards, and legacy systems.

From the above, it can be observed that addressing service interface interoperability via interface translation will potentially benefit industries. The major contributions of this article are as follows:

1. A novel approach to address service interface heterogeneity, which is one of the causes of interoperability issues in industrial scenarios.
2. The proposed solution makes use of the autonomous generation of service interfaces to solve the mismatches between consumers and providers in runtime.
3. The approach consequently reduces engineering costs and the time associated with interoperability problems.
4. The system was validated with practical experimentation, including a prototype and testbed implementation.

The rest of this article is organized as follows. Section 2 discusses the related work. Sections 3 and 4 present the proposed approach and performance evaluation, respectively. Section 5 presents the final discussion. Finally, Section 6 concludes this article.

2. Related Work

Currently, interoperability mismatches are solved by developers who are forced to implement the appropriate interface manually. Hand-coded solutions require large amounts of time and effort, increasing costs. In addition, they can not ensure exhaustive results. There may be remaining mismatches after a solution's application, for example, due to omission of mapping or lack of runtime interoperability. To solve this interoperability problem to any scale and promote the adoption of the IIoT, it is necessary to find an autonomous solution capable of closing the mismatch gap without human intervention. Autonomous information translation is thus a necessity, but no automated industrial solutions exist for this problem.

Academic efforts have been made to structure the enterprise interoperability concepts and issues. An example of this is the Framework for Enterprise Interoperability (FEI) [4]. The framework has three dimensions (1) interoperability concerns at different enterprise levels (business, process, service, and data), (2) interoperability barriers (conceptual and organizational), and (3) interoperability approaches to overcome the barriers (integrated, unified, and federated).

Academia and industry are experimenting with different approaches, including those presented as follows. Despite these efforts, no clear preferable approach can be identified based on current research results.

2.1. Semantic Platforms and Ontologies

Semantic compatibility between heterogeneous systems is seen as an effective approach to addressing interoperability issues. This is intended to provide a uniform and agnostic representation of service entities and their relationships, which can be achieved via a common data model to semantically describe entities. In addition to the use of existing technologies from the semantic web (OWL, OWL-S, SPARQL), promising new approaches can be found in the literature. Bhardwaj et al. [5] proposed a semantic interoperability architecture for smart spaces that details an ontology model and interoperability interactions between smart objects.

Due to the heterogeneity of the data produced by IoT, several ontologies have been created to address interoperability issues among sensor data. Some well-known examples are the semantic sensor network (SSN) ontology [6] and the sensor, observation, sample, and actuator (SOSA) ontology [7]. A wide number of ontologies have been developed in recent years; however, ontologies ultimately simply move the interoperability mismatch to a higher level. Applications with different ontologies cannot understand each other.

Other approaches include the use of ontology alignment to avoid the need for a common ontology [8]. These approaches include mediation engines that use semantics to support relationships among services [9]. Such approaches are required to be fully consistent and maintained over time to provide a mechanism that truly relates concepts and vocabularies from different ontologies. Unfortunately, according to Ganzha et al. [10], these alignment tools are no longer maintained after 1 or 2 years. Moreover, the use of ontologies implies confidence in the interpretation of semantic meaning. Automated ontology alignment still provides unsatisfactory results: only 40% accuracy is obtained for multilanguage matching and even lower values (approximately 20%) for complex matching, according to the studies of Algergawy et al. [11].

2.2. Web of Things (WoT)

The World Wide Web Consortium (W3C) has developed the Web of Things (WoT) [12]. The WoT's major goal is to improve the interoperability and usability of the IoT. The proposed architecture consists of rich metadata that describe the interaction models and information exposed by applications, enabling IoT applications to share common semantics. The WoT architecture may play a key role in the future of IoT applications; however, it also has several limitations. WoT relies on semantic interpretation. Semantic interpretation can be a barrier to communication exchange since IoT devices need generic contextual knowledge to fully interpret the information from other devices, according to Novo et al. [13]. Some automatic solutions have been proposed to reduce this problem based on ontology matching [14]. Nevertheless, these solutions are unable to faithfully understand the environment in terms of related high-level information [11].

2.3. Middleware and Mediators

In an attempt to overcome interoperability problems, new middleware frameworks and mediators have been proposed. In [15], an automated synthesis of mediators for middleware-layer protocol interoperability in the IoT was presented. The interoperability between heterogeneous middleware protocols is challenging due to the difference in semantics and technology diversity. This approach relies on the identification of common abstract interaction types according to the interaction paradigms and their modeling into a DeX API and a connector model. Another powerful example is the GeoNis framework for interoperability [16,17], which is focused on electric power supply companies and acts as a core component of the interaction of distributed entities. Middleware is often applied to overcome mismatches in terms of computational devices, communication networks, and

operating systems. This approach is usually tailored to specific technologies and scenarios that lack scalability and flexibility.

2.4. Machine Learning and Artificial Intelligence

During the last decade, a new research stream inspired by human cognition has gained popularity. New approaches based on machine learning and artificial intelligence provide a promising future that can affect different fields, including interoperability challenges. Examples of the application of these technologies to the stated problem can be found in the literature. The work presented in [18] includes predictive machine learning algorithms and unsupervised self-evolving AI algorithms to address interoperability in the IoT environment. Nilsson et al. [19] proposed the basics of a novelty semantic translator based on machine learning. This approach is considered for many as the future tool to address interoperability problems. Machine learning is used as a main approach for the aforementioned examples but also can be included as a support mechanism to complement other approaches. This last option will be considered for future implementations of the interface translation. Currently, the major limitations of this approach are the low maturity level and the low accuracy, which are difficulties for its adoption in industrial scenarios.

2.5. Proposed Approach. Translators

The work presented by Derhamy et al. [20] opened a new viewpoint where the information interoperability between two systems can be addressed by means of translators. Their translator was focused exclusively on dynamic application protocol translation. In addition to protocol translators, semantic translators have been proposed to solve specific semantic interoperability issues. De et al. [21] presented a translation approach to translate automotive interface description models from Franca to AUTOSAR frameworks.

To fully close the mismatch gap between two systems, a complete service interface translation is needed. All the aspects defined in the service contract, both functional and nonfunctional requirements, have to be considered and addressed to overcome interoperability mismatches. Integration platforms have to provide mechanisms for dynamic and instant information translation across protocols, encodings, standards, ontologies, and semantics used by individual embedded systems.

This paper proposes an approach to address interoperability between service oriented architecture-based heterogeneous systems. It is based on the autonomous generation of a suitable consumer interface that allows the consumption of the provided service despite mismatches between the systems. The proposed approach not only provides autonomous generation but also a deployment mechanism to permit a complete service interface translation. The approach is presented in detail in the following sections.

3. Service Interface Translation

The work presented in this paper makes use of the Service oriented architecture (SOA) approach by generating service interfaces to increase the interoperability in heterogeneous SOA environments. SOA is a well-known and adopted architectural style based on the use of services as the main mean to exchange information [22]. It has been proposed to provide syntactic interoperability and interoperability across heterogeneous systems [23]. It is built on top of the network layer; hence, processed information and data are easily managed through services [24]. SOA has been selected as a reference architecture for this work due to its principles, such as encapsulation, composition, reuse, loose coupling, late binding, and maintainability. The proposed service interface translation has been designed following the aforementioned SOA principles. For this paper, microservices are considered a specialization of SOA [25]. Therefore, the presented approach may be also adopted in this variant or other SOA variants.

In SOA environments, systems communicate through services. Successful service communication is critical to achieving system interoperability. A mismatch can occur at different service levels: application protocol, encoding, semantics, and notation.

The first step to solve service interoperability issues is to define and identify the service mismatches. Service metadata are defined in the service contract; consequently, an accurate definition of the service contract is crucial for the identification of mismatches. The information defined in the service contract is required not only for the identification of mismatches but also for the autonomous generation of interface code.

3.1. Methodology

This work is framed into experimental computer science and engineering (ECSE) research, which can be broadly defined as "the building of, or the experimentation with or on, nontrivial hardware or software systems" [26]. Based on a top-down strategy, the research process has been divided into sub-problems, which have been individually characterized and addressed to achieve an overall solution. The sub-problems addressed in this paper are (1) identification of requirements; (2) payload translation design; (3) definition and analysis of the consumer interface generation system and the interface instance; (4) problem definition and generalization of the runtime deployment.

3.2. Interface Generation Requirements

This paper proposes a novel approach based on autonomous runtime consumer interface generation. The presented approach translates information exchanges between systems with heterogeneous interfaces. The approach is defined based on the following requirements: (1) autonomous, (2) runtime, and (3) service contract-based.

- Autonomous. To reduce the engineering effort, the generation needs to be autonomous. The generation must be performed without human intervention.
- Runtime. The interoperability problems that occur during operation time require more effort and cost to be resolved. The solution to interoperability mismatches at runtime allows the connection of heterogeneous systems to open new possibilities, including the integration of vertical domains without increasing the engineering cost.
- Service contract-based. The service contract is a key piece of the generation since the service metadata that compound the service contract is necessary for a complete generation and translation. The final objective is to provide a reliable lossless service interface translation.

3.3. Proposed Solution

In order to solve the interoperability issues at the service level, a new methodology has been designed. The proposed solution is based on the aforementioned requirements. The interface mismatch between two systems can be solved via the generation of an interface translator instance. The new interface bridges both interfaces, providing a secure and successful path for data exchange.

The generated instance (translator) has an interface capable of understanding the data from the consumer. When the provider response is received, it is translated and sent to the consumer. The translation algorithms are based on the service contract information of both systems.

The use of this intermediate translation instance allows seamless communication. Details about the translation, the generator system, and the generated interfaces are provided in the following sections.

3.4. Payload Translation Process

The payload translation is one of the key aspects of the interface translation. Payloads are defined as the actual information or message exchanged via services. Even though tools and libraries to translate specific aspects of the payload can be found (eg. Jackson Project), the difficulty increases when different aspects are considered at the same time. In order to find a generic solution to the aforementioned problem, the translation uses as an input the service contract description metadata. The service contract can be described via interface description languages. It provides a common ground for a variety of heterogeneous

technologies at the same time that describes all functional and non-functional aspects in detail [27].

The current payload translation includes four parts: (1) the encoding translation, (2) mapping of semantic terms, (3) payload structure and mapping, and (4) casting of data types. The encoding conversion can be performed using libraries to address encoding-to-encoding translation. Nevertheless, the number of libraries to use grows faster than the number of encodings. To address this issue, the proposed translation utilizes Java objects as an intermediate format. The generator thus generates Java classes to map the different elements for both payloads. This approach also allows for capturing complex payload structures with nested objects.

The mapping of the semantic terms and data types is performed at the same time. The generator uses the metadata defined in the interface description of both systems to match the key values of both payloads. Once the payload elements are matched, there is a conversion of the data type of each element according to the payload descriptions. To obtain an accurate translation, metadata about the payload elements are crucial. The definition of the metadata in the interface description document and the algorithm is currently in development. The initial implementation includes the use of the terms "variation" and "unit" for physical magnitudes. Each element in the payload description includes name, datatype, variation, and unit. The variation attribute is a generic term used to define the key value. An example is shown in Figure 1.

```
<element name="v" type="Float"
         variation="temperature" unit="Celsius"/>
```

Figure 1. Payload element description.

3.5. Consumer Interface Generation System

The system in charge of code generation is the consumer interface generation system, an SOA-based software system that bridges service exchanges between non-interoperable systems. The system has been implemented in the frame of the Eclipse Arrowhead Framework [28]. The Eclipse Arrowhead is an open-source framework designed to provide automation capabilities for SoS, such as real-time, security, safety, and engineering of automation functionalities. The framework established the local cloud concept to meet the aforementioned automation system requirements.

The generator generates and dynamically instantiates the code for the interface that serves as an adaptor between the consumer and provider. The input to the generator consists of the service contract descriptions of both the consumer and provider systems. The output of the generator is executable code designed to communicate with the core systems of the local cloud and to enable consumption of the service offered by the provider.

The purpose of this system is as follows:

- Solve interoperability problems at the service interface level between heterogeneous systems.
- Generate new autonomous and runtime consumer interface codes.
- Translate service interfaces.

Figure 2 represents the interaction between the consumer interface generator and other Arrowhead core systems. The generator communicates with the authorization system, the service and system registry and the orchestrator. The service provider and consumer are able to achieve successful communication via the new interface generated by the generator.

3.6. Generated Interface Instance

The generator or system responds to the orchestration system request and generates a new instance based on both service contracts (from the provider and the consumer). The new instance is formed in three parts, explained as follows. Figure 3 shows a block description of the generated interface.

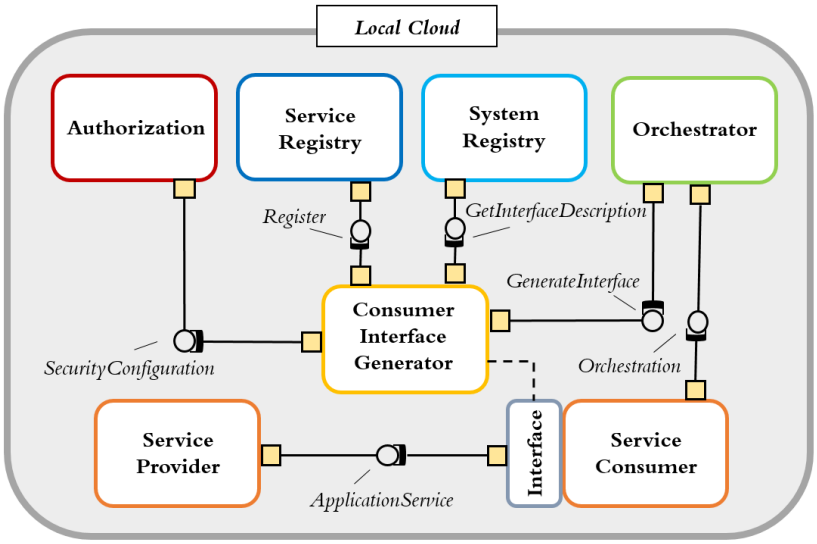

Figure 2. An example of an Arrowhead local cloud where consumer interface generation systems are used to generate a new interface between a service provider and a service consumer.

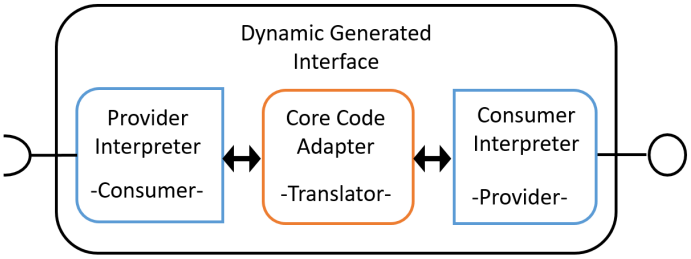

Figure 3. Block description of the internal parts of the dynamically generated interface.

- *Consumer Interpreter—Provider.* The new interface provides a service to the consumer using the metadata from the consumer service contract to match the interface of the system. The service is not public to any other system to avoid scalability and security problems; in other words, the service is local to the consumer device and is not registered in the service registry.
- *Core Code Adapter—Translator.* The core part of the interface is the service interface translator. It focuses on several aspects of the service interface of both systems, providing the code necessary to bridge them and transform the payloads.
- *Provider Interpreter—Consumer.* To complete the service interaction, a consumer piece of software is included in the instance, and it can be considered a client to some architectures. It consumes the service from the provider system using the translated message and sends back the response.

The service instance behaves like a black box between both systems. The service consumer initiates the consumption of the service via the generated interface without knowing of its existence. The instance is in charge of the translation between the system interfaces and proceeds with the consumption of the service using the correct service contract information. The provider answers the request normally, and finally, the interface translates and transmits the response back to the consumer.

3.7. System Architecture

The Interface Generator communicates with the other systems via services. One service is provided (*GenerateInterface*), and three services are consumed (*GetInterfaceDescription*, *SecurityConfiguration* and *ServiceRegistry*) to achieve code generation.

- *GenerateInterface.* This service is the trigger for code generation. It is consumed by the orchestration system. The orchestration system posts a request providing the service identifiers of the requesting consumer and the selected provider. This information is used by the consumer interface generator system to request the interface descriptions of both systems for the service to be delivered.
- *GetInterfaceDescription.* The interface generator consumes this service from the interface description database to request the interface descriptions. The preliminary location of the interface description database is the system registry. The system sends a get request using the system name.
- *SecurityConfiguration.* The security configuration for the new interface is requested via the *SecurityConfiguration service*, which the authorization system must provide.
- *ServiceRegistry.* This service is consumed to ensure that the interface generator system becomes accessible to other systems.

3.8. Runtime Consumer Interface Deployment

The last step to achieve an overall interoperability solution is the injection of the generated code into the consumer device. This step involves several challenges. In contrast to previous steps, the deployment of the code in an external device implies considering, among others, the hardware limitations and the security of the device. The most significant aspects are outlined in Table 1.

Table 1. Significant aspects of interface deployment.

Aspect	Definition
Hardware	Hardware characteristics and constraints must be considered at the time of injecting and executing the autonomously generated code. The generator system requires access to the Device Registry and the information about the device type, computational power, memory and operating system.
Security	Devices can include a variety of security measures to prevent threats and malware. These security measures must be considered, and the privileges or access policies have to be updated in order to inject and execute the code without introducing new vulnerabilities.
Generality	The heterogeneity of the devices needs to be considered. The deployment mechanism must be adaptable to the different device types and architectures. Devices may be from different vendors; consequently, they do not share a previous dedicated middleware installation to inject the code.

The autonomous updating of software is not a new concept. From smart devices to industrial equipment, new versions of the software have to be updated periodically. This situation has led to the investigation of autonomous and decentralized approaches to address the deployment of the new code. For example, Hu et al. [29] proposed a new approach based on blockchain to autonomously and securely update the firmware of IoT systems.

Nevertheless, software deployment solutions are designed for a specific device or system and rely on proprietary technologies. Devices are designed with the capabilities and tools to receive software updates. In contrast, in industrial scenarios, due to the heterogeneity in the devices and the variety of vendors, devices do not share a common technology that can be used for code deployment. Finding a generic solution for all types of devices can be seen as an impossible task. To facilitate deployment, two steps are suggested.

3.8.1. Identification of Hardware Requirements

The first step is the identification of the device type, which should include the computational power, memory, hardware type, security mechanism and operating system. This information can be used to determine whether the generation is feasible. Resource-constrained devices, for example, are less likely to execute new code unless they are adapted to their constraints. The device characteristics can also be used to determine if the performance will be sufficient to fulfill time constraints and execution requirements.

3.8.2. Execution of the Adequate Solution

Once the device requirements are analyzed and the injection of the code is considered feasible, the best solution for that specific device must be determined and executed. Different partial solutions can be considered for the generator system to cover the most common cases. To select and execute adequate solutions, access privileges and permissions must be considered and granted, which may cause security vulnerabilities. Therefore, the injection of the code has to be designed and implemented thoughtfully.

With regard to the work presented in this paper, the deployment and injection of code have not been completely analyzed, and consequently, a final approach has not been adopted. The attainment of a generic solution that can be used in different platforms is part of future research. Nevertheless, to take measures and evaluate the concept, a partial and temporal solution was implemented. Consumer devices are considered to have sufficient computational power to support Java and execute JAR files. To simplify the process, only two options have been considered: (1) the use of devices with UNIX-based operating systems or Microsoft Windows and (2) the use of devices that allow the use of Docker containers. Assuming complete trust in the system, permissions can be obtained, and the use of shell injection can be exploited for this purpose.

4. Performance Evaluation

The performance of the proposed architecture was evaluated in an experimental setting.

4.1. Use Case Scenario

A simple use case scenario was selected to test and evaluate the proposed service interface translation. The scenario comprised a weather station that acted as a provider of various services, a service consumer, mandatory Arrowhead core systems [30], and the consumer interface generator system. The weather station included six sensors: indoor and outdoor temperature, humidity, pressure, solar radiation, and wind speed sensors. To retrieve data from each sensor, several services were provided.

All systems were Arrowhead framework compliant. The systems ran within the frame of a unique local cloud. All interactions between systems were thus intracloud operations. Figure 4 describes the local cloud scenario.

The scenario was designed to illustrate the interface generator system's behavior and performance. Therefore, interoperability mismatches were introduced between the service consumer and provider. Consumer and provider service contracts differ in various aspects. Interoperability mismatches prevent successful communication, making service consumption impossible without the use of the generator.

The indoor temperature service was the service selected for the test and experiments. Table 2 describes the differences between the service consumer and provider for the indoor temperature service.

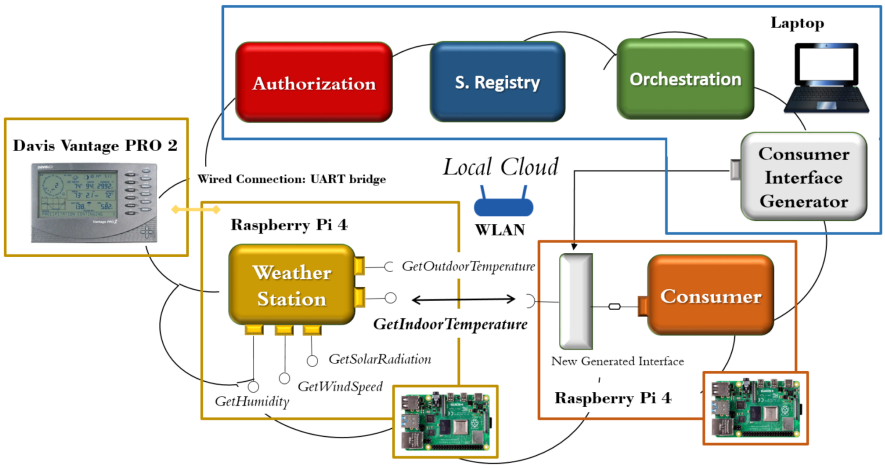

Figure 4. A case scenario that included the Davis Vantage Pro2 weather station, the weather station provider and consumer, the mandatory Arrowhead core system, and the consumer interface generator.

Table 2. Interface characteristics.

Characteristic	Service Provider	Service Consumer
Protocol	CoAP	HTTP
Encoding	JSON	XML or CBOR
Semantic	SenML	Customized
Temperature Data Type	Float	Integer

Mismatches in the data model and payload are of great relevance, and semantic interoperability is a major barrier in the current technological paradigm [31]. Consequently, the consumer and provider payloads for the indoor temperature service differ in different aspects to show the potential of the proposed solution. As shown in Figures 5 and 6, mismatches introduced between the two payloads can be found in the structure of the message, the encoding, the semantics used in the key values, and the data types. The payload described in Figure 5 makes use of JSON and SenML. The temperature service message includes the name (n), value (v), unit (u), and time (t). Figure 6 payload, on the other hand, includes XML and a natural language to describe the name, localization, and value.

```
{"n" :"indoortemperature",
  "v" : 23.3333,
  "u" : "Celsius",
  "t" : 22651124 }
```

Figure 5. Example of the payload sent by the service provider.

```
<RequestDTO_C0>
   <name> TemperatureService </name>
   <localization> Sweden </localization>
   <value>
      <temp> 23 </temp>
      <unit> Celsius </unit>
  </value>
</RequestDTO_C0>
```

Figure 6. Example of the payload expected by the consumer.

4.2. Testbed Implementation

Experimentation was conducted through a testbed. Davis Vantage Pro2 was the weather station device selected for the tests. The weather station was connected to a Raspberry Pi (Raspberry Pi 4 Model B) acting as a service provider in the Arrowhead local cloud. The connection between the Davis Vantage Pro2 and the Raspberry Pi was established via serial communication. The service consumer was executed in another Raspberry Pi. Finally, the local cloud mandatory core systems and the interface generator system operated on a laptop. The specifications of devices used in the testbed are shown in Table 3. All systems were connected via a local WLAN network.

Table 3. Testbed setup specifications.

Specifications	**Provider & Consumer (Raspberry Pi)**	**Core Systems & Interface Generator (Laptop)**
Memory	4 GB	16 GB
Processor	Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5 GHz	Intel(R) Core (TM) i7-7500U CPU @ 2.70 GHz
OS	Raspbian	Windows 10
Disk	16 GB	413 GB

The application systems (provider and consumer) were implemented in Java Spring (available: https://github.com/CristinaPaniagua/WeatherStation_Scenario, accessed on 6 December 2021) following the Arrowhead framework general recommendations. The consumer interface generator was also implemented as a Java Spring project (available: https://github.com/CristinaPaniagua/InterfaceTranslation-velocity, accessed on 6 December 2021) and acts as a support system in the local cloud. The generator is still a work-in-progress project, and the code is susceptible to change.

4.3. Metrics

The evaluation of the scenario was performed using the most representative times and sizes of the generated code. Figure 7 shows the sequence diagram of the interaction between the service consumer and provider making use of the consumer interface generator. The generation was triggered by the orchestration system. The consumer requested the provider endpoint to consume its service despite the interoperability mismatches between them. After the interface was generated, the consumer was able to communicate with the provider via the newly generated interface. The generation process was only necessary once. The blue diamonds in the figure indicate when the samples were taken. In Table 4, a comprehensive summary of the selected metrics is presented.

Table 4. Metrics used for the evaluation.

Notation	**Meaning**	**Calculations**
T_G	Time taken to generate the interface	$t_b - t_a$
T_B	Time taken to compile and built the interface project	$t_c - t_b$
T_I	Time taken to perform the service translation by the generated interface instance	$t_f - t_g$
T_S	Total time taken to consume the service	$t_g - t_d$

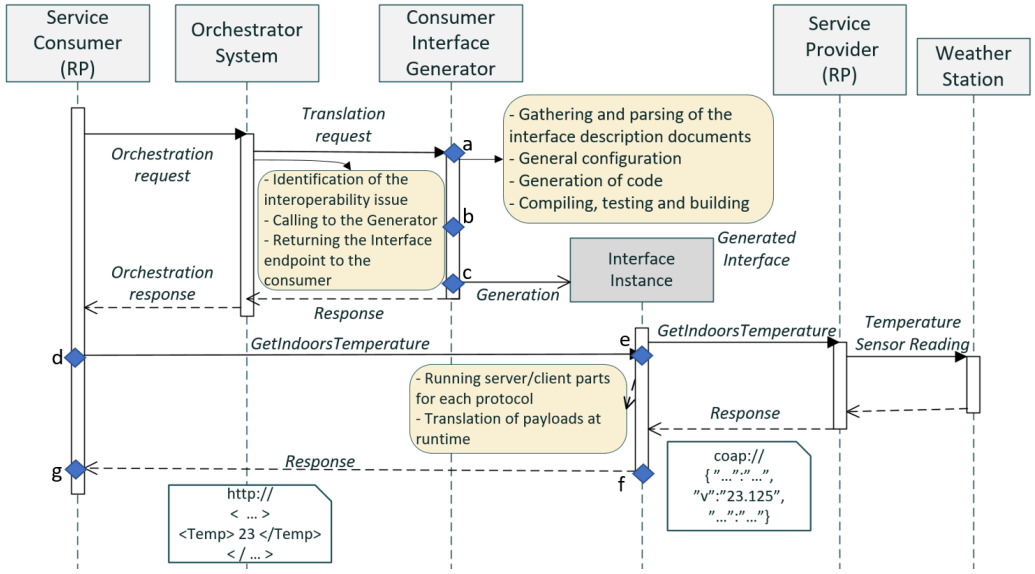

Figure 7. UML Sequence diagram for service consumption using the consumer interface generator.

4.4. Results

In Table 5, it can be observed that the execution time for the code generation and building of the executables lies within a feasible range and is suitable for IoT scenarios. On average, the process of generation, starting when the orchestrator calls the generator system and ending when the JAR files are successfully built, takes less than 50 ms.

Table 5. Experimental Results.

Metric	Averaged Time [ms]	Standard Deviation	Range (Max, Min) [ms]
T_G	25.72	6.33	(64, 28.87)
T_B	17.28	1.13	(24.1, 15.78)
T_I	64.64	15.06	(151, 34)
T_S	101.84	16.46	(193, 53)

The sizes of the generated code are presented in Table 6. The values are dependant on the scenario. Nevertheless, complementary studies have shown that the size does not vary significantly.

Table 6. Sizes of the generated code.

Notation	Meaning	Result
S_G	Size of the generated interface code	24 KB
S_I	Size of the interface JAR (Including dependencies)	11.9 MB

In the case of the use of the generated interface instance, the service consumption time for the presented scenario increases notably. The average time for consumption is approximately 100 ms, and it is possible to achieve 50 ms as a minimum value. The time employed by the interface to retrieve the information from the provider and translate the payloads is approximately 60 ms. Since the average consumption time between the interface and the provider is 33 ms, it is possible to infer that the payload translation takes place within 30 ms on average.

To compare with a reference point, a CoAP Client was implemented to directly consume the data from the provider. The CoAP client interface is a perfect match for the

provider interface. Figure 8 shows a stacked graph of the translation timing in comparison with the direct consumption. Except for a few outliners, there was little variability in the consumption of the service using the interface translation. The CoAP library used to communicate with the provider and the network seems to have been responsible for much of the variability. The direct consumption of the service is also affected by this situation, as shown in the graph.

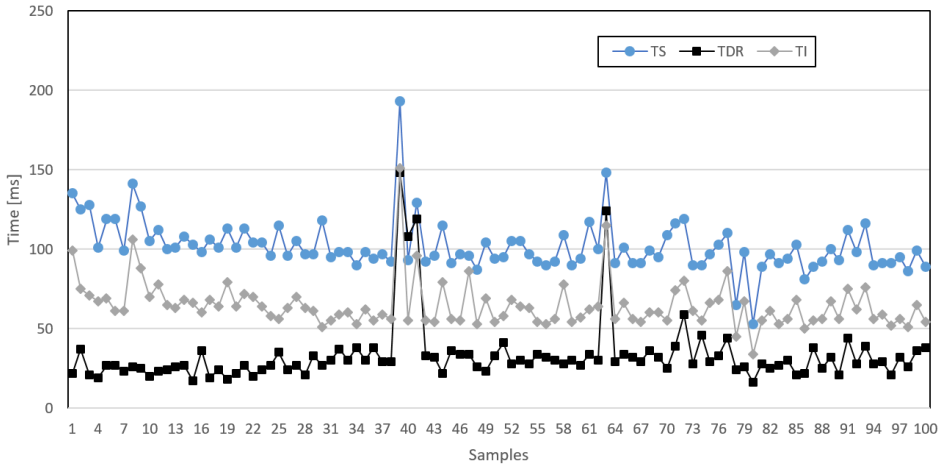

Figure 8. Comparison between the time required to consume the service in the proposed scenario (T_S), which includes the use of interface (T_I), and the time required to consume the service making a direct request ($T_D R$) using a CoAP interface without any interoperability mismatch.

The times presented in this paper are dependent on the network and device characteristics; consequently, they should be considered a proof of concept to demonstrate the validity of the approach and not as strict results. There is room for improvement and optimization, which may more likely result in lower times.

5. Discussion

Experimental results demonstrate that the presented approach is capable of solving interoperability issues at the service interface level. The autonomous generation of new interface instances was performed successfully in the presented scenario.

Despite that the interface translation allows two systems with several interoperability mismatches (protocol, encoding, semantics, payload structure, and datatype) to successfully interchange data, there are also limitations and challenges that need to be overcome. One of the major challenges associated with this approach is the variations in the application interfaces (syntax, data formats, and semantics). The translation of data models and semantics presents an important barrier that needs to be addressed. The approach has been tested for a limited set of payloads. Future experiments need to cover a wider range of data models and consider the introduction of machine learning algorithms and semantic annotations, e.g., reinforcement learning.

Security implications need to be further analyzed. The current implementation of the consumer interface generator system is secured via the HTTPS protocol. The system includes two working modes, *secure* and *insecure*. If the system is started in secure mode, X.509 identity Arrowhead compliant certificates are used for authentication and authorization. The system is considered a trusted peer in the Arrowhead local cloud and follows the security policies stated in the framework [32]. The insecure mode is exclusively used for development.

The translation also implies a time delay. One hundred milliseconds are necessary to consume the service against 40 ms for the direct consumption of the service, a little more than double. We consider that the proposed approach can be used in systems that do not need request-response times of less than 100 ms, regardless of the delay introduced by the

interface instance. Regarding the generation process, it increases the orchestration times by approximately 50 ms. The increase in time is assumable since the orchestration process takes place only once when the consumer system requests the provider endpoint. Once the communication is established, the consumer system uses the same provider endpoint each time that wants to consume the service.

An important clarification is that the service consumption times cannot be compared directly since the consumer does not count with the appropriate interface. Both systems cannot communicate without providing a new interface. The use of interface translation at runtime solves the interoperability misalignment. These problems would require stopping the operation and spending time and resources in the implementation of the code. Usually, the implementation of an interface by hand implies several minutes to hours. The use of the consumer interface generator reduces that time to hundreds of milliseconds and at runtime. Consequently, the generator may potentially decrease the engineering costs associated with interoperability.

Moreover, future work will involve the improvement of the proposed approach, including the design of a wider and robust injection strategy, and the testing of the prototype in industrial scenarios.

This approach can mean a major improvement in comparison with other solutions, such as semantic platforms, WoT, and middleware. These approaches do not try to find a generic solution to the problem, but provide a specific solution that is only useful for a limited period of time and for certain technology.

6. Conclusions

The number of connected devices and systems has increased in an attempt to achieve the Industry 4.0 benefits estimated by analysts. This situation has led to an increase in heterogeneity and a lack of interoperability.

Traditionally, the interoperability problems related to service interfaces are solved manually. Developers spend large amounts of time and effort implementing interfaces that solve these issues. A generic and automated solution is thus needed. This paper proposed a new approach to address service interoperability issues at the interface level based on autonomous code generation. New interface instances are generated to act as a bridge between heterogeneous systems. The instances translate the interfaces based on the service metadata included in the service contracts.

This approach allows solving interoperability mismatches at runtime without any human intervention, consequently reducing the engineering costs and time. The performance of the proposed architecture has been validated through a practical implementation. A prototype of the consumer interface system, the system in charge of the generation, has been tested in a use case scenario. The results showed the potential of the approach. The approach may be used in conjunction with other technologies to provide a generic solution to the interface interoperability problems. However, to obtain the maximum benefits of the approach, limitations such as security considerations, semantic accurate definition, and semantic translation need to be overcome.

Funding: This research was funded by EU ECSEL Joint Undertaking under grant agreement No. 826452.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Acknowledgments: The authors thank the European Commission and the *Arrowhead Tools* project (EU ECSEL Joint Undertaking under grant agreement number 826452) for funding.

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

References

1. Vaidya, S.; Ambad, P.; Bhosle, S. Industry 4.0—A glimpse. *Procedia Manuf.* **2018**, *20*, 233–238. [CrossRef]
2. Boardman, J.; Sauser, B. System of Systems-the meaning of of. In Proceedings of the 2006 IEEE/SMC International Conference on System of Systems Engineering, Los Angeles, CA, USA, 24–26 April 2006; p. 6.
3. Manyika, J.; Chui, M.; Bisson, P.; Woetzel, J.; Dobbs, R.; Bughin, J.; Aharon, D. *The Internet of Things: Mapping the Value Beyond the Hype*; McKinsey&Company, Executive Summary, McKinsey Global Institute: Washington, DC, USA, 2015.
4. Weichhart, G.; Ducq, Y.; Doumeingts, G. An IFIP WG5. 8 State-of-the-Art View on Methods and Approaches for Interoperable Enterprise Systems. In *Advancing Research in Information and Communication Technology*; Springer: Berlin/Heidelberg, Germany, 2021; pp. 222–244.
5. Bhardwaj, S.; Ozcelebi, T.; Lukkien, J.J.; Lee, K.M. Semantic interoperability architecture for smart spaces. *Int. J. Fuzzy Log. Intell. Syst.* **2018**, *18*, 50–57. [CrossRef]
6. 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. [CrossRef]
7. Janowicz, K.; Haller, A.; Cox, S.J.; Le Phuoc, D.; Lefrançois, M. SOSA: A lightweight ontology for sensors, observations, samples, and actuators. *J. Web Semant.* **2019**, *56*, 1–10. [CrossRef]
8. Priesnitz Filho, W.; Ribeiro, C.; Zefferer, T. An Ontology-Based Interoperability Solution for Electronic-Identity Systems. In Proceedings of the 2016 IEEE International Conference on Services Computing (SCC), San Francisco, CA, USA, 27 June–2 July 2016; pp. 17–24.
9. De Biase, L.C.; Calcina-Ccori, P.C.; Silva, F.S.; Zuffo, M.K. The semantic mediation for the swarm: An adaptable and organic solution for the internet of things. In Proceedings of the 2017 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, 12–14 June 2017; pp. 78–79.
10. Ganzha, M.; Paprzycki, M.; Pawłowski, W.; Szmeja, P.; Wasielewska, K. Towards semantic interoperability between Internet of Things platforms. In *Integration, Interconnection, and Interoperability of Iot Systems*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 103–127.
11. Algergawy, A.; Cheatham, M.; Faria, D.; Ferrara, A.; Fundulaki, I.; Harrow, I.; Hertling, S.; Jiménez-Ruiz, E.; Karam, N.; Khiat, A.; et al. Results of the ontology alignment evaluation initiative 2018. In Proceedings of the 13th International Workshop on Ontology Matching Co-Located with the 17th ISWC (OM 2018), Monterey, CA, USA, 8 October 2018; Volume 2288, pp. 76–116.
12. W3C. Web of Things (WoT) Architecture. 2020. Available online: https://www.w3.org/TR/wot-architecture/ (accessed on 2 Auguest 2020).
13. Novo, O.; Francesco, M.D. Semantic Interoperability in the IoT: Extending the Web of Things Architecture. *ACM Trans. Internet Things* **2020**, *1*, 1–25. [CrossRef]
14. Paulheim, H.; Hertling, S.; Ritze, D. Towards evaluating interactive ontology matching tools. In *Extended Semantic Web Conference*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 31–45.
15. Bouloukakis, G.; Georgantas, N.; Ntumba, P.; Issarny, V. Automated synthesis of mediators for middleware-layer protocol interoperability in the IoT. *Future Gener. Comput. Syst.* **2019**, *101*, 1271–1294. [CrossRef]
16. Stoimenov, L.; Davidovic, N.; Stanimirovic, A.; Bogdanovic, M.; Nikolic, D. Enterprise integration solution for power supply company based on GeoNis interoperability framework. *Data Knowl. Eng.* **2016**, *105*, 23–38. [CrossRef]
17. Stoimenov, L.; Djordjević-Kajan, S. An architecture for interoperable GIS use in a local community environment. *Comput. Geosci.* **2005**, *31*, 211–220. [CrossRef]
18. Nawaratne, R.; Alahakoon, D.; De Silva, D.; Chhetri, P.; Chilamkurti, N. Self-evolving intelligent algorithms for facilitating data interoperability in IoT environments. *Future Gener. Comput. Syst.* **2018**, *86*, 421–432. [CrossRef]
19. Nilsson, J.; Sandin, F.; Delsing, J. Interoperability and machine-to-machine translation model with mappings to machine learning tasks. In Proceedings of the 2019 IEEE 17th International Conference on Industrial Informatics (INDIN), Helsinki-Espoo, Finland, 22–25 July 2019; Volume 1, pp. 284–289.
20. Derhamy, H.; Eliasson, J.; Delsing, J. IoT interoperability on demand and low latency transparent multiprotocol translator. *IEEE Internet Things J.* **2017**, *4*, 1754–1763. [CrossRef]
21. De, S.; Niklas, M.; Rooney, B.; Mottok, J.; Brada, P. Towards Translation of Semantics of Automotive Interface Description Models from Franca to AUTOSAR Frameworks: An Approach using Semantic Synergies. In Proceedings of the 2019 International Conference on Applied Electronics (AE), Pilsen, Czech Republic, 9–10 September 2019; pp. 1–6.
22. Rosen, M.; Lublinsky, B.; Smith, K.T.; Balcer, M.J. *Applied SOA: Service-Oriented Architecture and Design Strategies*; John Wiley and Sons: Hoboken, NJ, USA, 2012.
23. Noura, M.; Atiquzzaman, M.; Gaedke, M. Interoperability in internet of things: Taxonomies and open challenges. *Mob. Netw. Appl.* **2019**, *24*, 796–809. [CrossRef]
24. Vinoski, S. Integration with web services. *IEEE Internet Comput.* **2003**, *7*, 75–77. [CrossRef]
25. Bogner, J.; Zimmermann, A.; Wagner, S. Analyzing the relevance of SOA patterns for microservice-based systems. In *ZEUS 2018: Workshop on Services and the Composition, Proceedings of the 10th Central European Workshop on Services and Their Composition: Dresden, Germany, 8–9 February 2018*; CEUR Workshop Proceedings, 2072; RWTH Aachen: Aachen, Germany, 2018; pp. 9–16.
26. Shelby, Z. Embedded web services. *IEEE Wirel. Commun.* **2010**, *17*, 52–57. [CrossRef]

27. Paniagua, C. The Role of Service Contracts in Interoperability Mismatch Identification. In Proceedings of the 2021 IEEE 19th International Conference on Industrial Informatics (INDIN), Palma de Mallorca, Spain, 21–23 July 2021; pp. 1–6.
28. Delsing, J.; Eliasson, J.; van Deventer, J.; Derhamy, H.; Varga, P. Enabling IoT automation using local clouds. In Proceedings of the 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT), Reston, VA, USA, 12–14 December 2016; pp. 502–507.
29. Hu, J.W.; Yeh, L.Y.; Liao, S.W.; Yang, C.S. Autonomous and malware-proof blockchain-based firmware update platform with efficient batch verification for Internet of Things devices. *Comput. Secur.* **2019**, *86*, 238–252. [CrossRef]
30. Varga, P.; Blomstedt, F.; Ferreira, L.L.; Eliasson, J.; Johansson, M.; Delsing, J.; de Soria, I.M. Making system of systems interoperable–The core components of the arrowhead framework. *J. Netw. Comput. Appl.* **2017**, *81*, 85–95. [CrossRef]
31. Rahman, H.; Hussain, M.I. A comprehensive survey on semantic interoperability for Internet of Things: State-of-the-art and research challenges. *Trans. Emerg. Telecommun. Technol.* **2020**, *31*, e3902. [CrossRef]
32. Plósz, S.; Hegedűs, C.; Varga, P. Advanced security considerations in the arrowhead framework. In *International Conference on Computer Safety, Reliability, and Security*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 234–245.