A Typology of Architectural Strategies for Interoperability

Pedro Henrique Dias Valle
Department of Computer Systems,
University of Sao Paulo,
São Carlos - SP, Brazil, São Paulo
pedrohenriquevalle@usp.br

Lina Garcés
Department of Computer Systems,
University of Sao Paulo,
São Carlos - SP, Brazil, São Paulo
linamgr@icmc.usp.br

Elisa Yumi Nakagawa
Department of Computer Systems,
University of Sao Paulo,
São Carlos - SP, Brazil, São Paulo
elisa@icmc.usp.br

ABSTRACT

An increasing interest in researching the development, integration, composition, and evolution of large-scale, software-intensive systems (LSSIS) have been observed in the last years. These systems are presented in different domains as connected health, industry 4.0, military, smart cities, smart grids, and smart agriculture. These systems are realized through the cooperation among heterogeneous, independent, highly distributed, individual, and heterogeneous systems to achieve their business goals. For the success of these larger systems, their individual parts must interoperate among them allowing the execution of more complex functionalities. However, there are many concerns that software engineers must overcome during the composition of these systems, that can be related to (i) the misunderstanding of data formats, data semantic, procedures, contracts, standards, quality, and interfaces structures provided by individual systems; and (ii) the absence of high-level architectures to analyze, comprehend, and guide how interoperability can be addressed. The goal of this work is to present a typology of existing and proven strategies for achieving interoperability. Strategies were identified through a systematic search in scientific databases and patterns repositories. The selected strategies were categorized according to the level of interoperability they support, namely, technical, syntactic, semantic, and organizational. In each level, the strategies were divided by the type of solution their propose, i.e., technique, and architectural styles, patterns, and tactics. Moreover, statements explaining how each strategy address interoperability are given. Results of this work can be used by architects to identify and understand solutions for achieving interoperability requirements during the composition of larger systems. The resulting typology in this study is the first step to consolidate a patterns-language for interoperability in software architectures.

CCS CONCEPTS

• **Software and its engineering → Software architectures**; **Interoperability**;

KEYWORDS

Software Architecture, Interoperability, Architectural Pattern, Architectural Style, Tactic, Typology

SBCARS '19, September 23–27, 2019, Salvador, Brazil

ACM ISBN 978-1-4503-7637-2/19/09…$15.00
https://doi.org/10.1145/3357141.3357144

ACM Reference Format:
Pedro Henrique Dias Valle, Lina Garcés, and Elisa Yumi Nakagawa. 2019. A Typology of Architectural Strategies for Interoperability. In *XIII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '19), September 23–27, 2019, Salvador, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357141.3357144

1 INTRODUCTION

During the last years, an increased interest by researching, developing, and evolving large-scale, software-intensive systems (LSSIS) have been observed [5]. Examples of such systems are those based on microservices architectures, IoT (Internet of Things), software ecosystems, big data, and Systems-of-Systems (SoS). These systems have been inquired in different domains as connected health, industry 4.0, military, smart cities, smart grids, and smart agriculture. Their operations depend on interactions among other systems that are heterogeneous, distributed, and sometimes, independent, large-scale, and software-intensive. The building of those systems includes a diversity of stakeholders and multidisciplinary practitioners, that are involved during their whole life cycle, e.g., requirements specification, architectural design, development, testing, deployment, maintaining, and evolution.

Individual systems (also named as constituent systems) participating in larger systems have different life-cycles, the operational environment, programming languages, platforms, goals and business interests. This is due to the fact that they are created and managed by different organizations [43]. Therefore, developers of LSSIS are encouraged to adopt solutions for integrating such heterogeneous, distributed, and independent software systems [1, 40]. To enable the successful integration of these systems and their meaningful data exchange and interactions, it is required an analysis of technical, conceptual, and organizational constraints. This activity is required to ensure the desired interoperation among distributed systems, i.e., their controlled coordination and collaboration to achieve higher business goals [53]. Therefore, interoperability has been claimed as an essential requirement for the successful establishment of LSSIS.

Despite the important role of interoperability in the composition and execution of complex functionalities, software engineers still face interoperability challenges that hamper the building and operation of those systems in an effective and efficient way. Abukwaik and Rombach [1] carried out an online survey, in the industry, with experienced software engineers. Such study identified the main concerns that must be overcome in integration projects, and that are related to [1]: (i) the misunderstanding of data formats, data semantic, procedures, contracts, standards, quality, and interfaces structures provided by individual systems; and (ii) the absence of high-level architectures to analyze, comprehend, and guide how

interoperability can be addressed. Therefore, this work is motivated by the current interoperability challenges that are faced by practitioners during integration projects as reported in [1]. In this context, this work brings an overview of current strategies that could be used during the construction of LSSIS to understand and guide how interoperation requirements could be addressed.

Specifically, this work introduces a typology of strategies organized by interoperability requirements. A typology is restricted to a system of conceptually derived groupings [4, 18]. Requirements are categorized into four classes, namely technical, syntactic, semantic, and organizational interoperability. For each requirement class, four types of strategies were associated, i.e., techniques, technologies, architectural patterns, and architectural styles. To identify the strategies presented in the taxonomy, a systematic search in different scientific databases and pattern repositories was conducted. The use of each strategy for achieving a specific interoperability requirement is justified through theoretical fundamentals and application examples. This typology can support architects and developers participating in integration projects of LSSI, specifically during the selection of proven strategies for addressing interoperability.

The remainder of this paper is structured as follows. Section 2 presents the background in which our paper is placed, showing the main concepts of interoperability and software architecture. Section 3 gives an overview of related works. Section 4 details the methods used to create the typology presented in this paper. Section 5 presents the strategies that were identified for achieving interoperability. Section 6 discusses the main contributions of this paper to the state of the art and highlights threats to the validity of this work. Finally, conclusions and future work are detailed in Section 7.

2 BACKGROUND

In this section, the definition of interoperability, including different classifications that can be found in literature is presented. Moreover, concepts from software architecture are, namely, architectural styles, patterns, and tactics are described as strategies to achieve quality attributes such as interoperability.

2.1 Interoperability

There are several definitions for the interoperability, hence, this term can have many interpretations in different contexts. In this work, interoperability of software systems is considered as the ability of two or more systems or elements to exchange and understand information, and to correctly use the information that has been exchanged [34]. Interoperability has been categorized into different classes [49], however, no consolidated classification of types of interoperability can be found in the literature. In this work, the following four types of interoperability are explored:

- **Technical Interoperability:** It is related to the interoperability of infrastructure and software. Interoperability of infrastructure is the ability of hardware acquired by different organizations to work in a connected way. This type of interoperability is associated with the hardware and software components, networks and equipment that enable machine-to-machine communication to take place, including aspects as open interfaces, connectivity, data integration, middleware, data presentation, data exchange, accessibility and security issues [63, 33].
- **Syntactic Interoperability:** It is referred to the data representation in machine-readable form and, in general, associated with data formats [63]. This type of interoperability aims at identifying elements and rules for structuring the elements, mapping, bridging, and navigating among equivalent elements [65, 49]. In short, the syntactic interoperability assists two or more systems communicating and exchanging data [40].
- **Semantic Interoperability:** It is concerned with the guarantee that the precise meaning of exchanged information is understandable by any other application that was not initially developed for this purpose [49]. In this type, only relevant information can be exchanged or shared. It will support high level, context-sensitive information request over heterogeneous information resources, hiding system, syntax, and structural heterogeneity. To achieve semantic interoperability, both sides must refer to a common information exchange reference model [8].
- **Organizational Interoperability:** It is oriented to the ability of two or more units to provide services to and accept services from other units and to use the services to support them to operate effectively together [41]. Each organization has its own unique culture, capabilities and operating procedures. Organizational Interoperability depends on the successful implementation of technical, syntactical and semantic interoperability [8].

Interoperability has been an important topic both for academia and industry since software systems began to be deployed in a distributed fashion. This characteristic of distribution is presented in almost all systems, being more representative in contemporary systems, for instance, microservices-based systems, IoT, big data, Industry 4.0, mobile systems, SoS, and software ecosystems. Henceforth, systems distribution has put interoperability as one of the highest is considered a great challenge for designing, developing, testing, deploying, and evolving current systems, which are mainly constructed through the integration of components of existing and operating systems. In this perspective, in LSSIS, interoperability should be considered during all software life-cycle, with more focus on the earlies stages of its construction, i.e., software architecture designing, to avoid highest costs at the deployment time.

2.2 Software Architecture

Software architectures have a fundamental role in determining the quality of software systems and guiding further activities in the software development process, mainly because they promote several quality characteristics, e.g., interoperability, performance, portability, adaptability, maintainability, and even interoperability. In this sense, software architectures are essential artifacts to support the success of software systems [9, 50].

The definition of software architecture has evolved over the years. Most of these definitions can be synthesized by the definition of Bass [6, 15, 45] which determines the *"architecture of a software program or computing system is the structure or structures of the*

system, which is composed of software components, the externally visible properties of those components, and the relationships between them".

Software architectures are important because decisions made at the architectural level directly enable, facilitate, or interfere in the achievement of business goals as well as in functional and quality requirements [6]. To support the development of these architectures, Hofmeister *et al.* [30] define the architecting of software systems is composed of three activities: i) *architectural analysis:* this activity articulates architecturally significant requirements (ASRs) accordingly to architectural concerns (e.g., requirements of quality attributes) and the context; ii) *architectural synthesis:* in this activity, architectural decisions (e.g., candidate architectures, architectural patterns, architectural styles, tactics) are proposed to address the architecturally significant requirements; and iii) *architectural evaluation:* this activity ensures that architectural decisions are the right ones to enable the requirements to be satisfied.

Among these activities, the most challenging one is the architectural synthesis, mainly because requirements of quality attributes must be clearly understood and solutions and strategies must be selected for addressing such attributes. In this activity, and depending on the intended requirements, several strategies, namely, architectural styles, patterns, tactics, can be applied to form candidate architectures.

Architectural styles are packages of design decisions that explain a generic design approach for a software system [15]. Another definition used by Bass *et al.* [6] is a specialization of element and relation types, together with a set of constraints on how they can be used [6]. The architectural styles contribute to a good design of architectures, in which they are defined as a coordinated set of architectural constraints that restricts the roles of architectural elements and the allowed relationships among those elements within any architecture [47].

Architectural patterns are ways to capture well-proven design structures, facilitating their reuse [12, 44]. Architectural patterns provide a mechanism to capture domain experience and knowledge to allow it to be reapplied when a new architectural problem is found [58]. In short, an architectural pattern can be considered as a package of design decisions that are found repeatedly in practice [29]. Such patterns have known properties that permit reuse, besides that they describe a class of architectures [6].

Architectural tactics are design decisions that influence the achievement of a quality attribute response tactics directly affect the system's response to some stimulus [6]. Architectural tactics are simple ideas opposed to patterns since they are described in just a few sentences, which they can be refined into a hierarchical structure of tactics. Most patterns are composed or created from several different tactics, and although these tactics might all serve a common purpose, they are often chosen to promote different quality attributes [20].

3 RELATED WORKS

From the software architecture perspective, it is possible to find relevant works investigating relationships between architectural decisions (i.e., patterns, tactics, and styles) and quality attributes [61, 28, 46, 21], as well as taxonomies to systematically organize these decisions in software projects [39]. However, such contributions are generic, lacking specifications for individual quality attributes, such as interoperability.

Regarding strategies to support integration projects, Keshav and Gamble [38] proposed an initial taxonomy between existing integration strategies and the integration elements. The authors classified the integration strategies into three basic integration elements, namely: translator, controller, and extender. Moreover, Rezaei et al. [60] proposed a maturity model for the interoperability of ultra large scale systems. For each maturity level, the model suggests different techniques that can be used for their achievement.

In SoS perspective, Garcés et al. [22] defined a preliminary taxonomy of software mediators. This taxonomy can be used by architects as a library of solutions when designing their SoS, considering essential aspects for such systems as integration/interoperability. The authors defined twelve types of mediators are proposed allowing capabilities of communication and control of interactions, and conversion of messages exchanged through the mediation infrastructure. Additionally, in [24], authors present an initiative of architectural patterns language for SoS, however they are more focused on requirements of reconfiguration, maintainability, and dynamism. In a similar effort, Ingram et al. [35] present modeling patterns for the architectural design of SoS. In the same perspective, Vargas *et al.* [62] carried out a systematic review to investigate the state of the art of SoS Integration and the software engineering methods that assist in the integration between constituent systems of an SoS. As a result, the authors observed that there is a need for additional research to understand why the integration is maturing more slowly than other software engineering area. However, these proposed solutions in SoS context do not consider all interoperability levels (namely, technical, syntactic, semantic, and organizational) required to successfully compose an SoS.

Although the existence of contributions for addressing interoperability from a software architecture perspective, their scope is limited. Besides, there is a lack of studies that bring orientations to practitioners about existing strategies they could use for achieving requirements of interoperability on different levels. The typology presented in Section 4 contributes to the state of the art offering a conceptual classification about existing and proven strategies for achieving interoperability from a software architecture perspective.

4 METHODS

Since a typology is considered a conceptual classification of a problem, it must represent relationships among different concepts types found in the problem domain.

To create the typology of strategies for interoperability, the guidelines presented in [4] were followed.

Step 1: The first step to create the typology was the selection of ***types of interoperability***. For this, a searching for interoperability classifications was made using scientific data libraries as Scopus and Google Scholar. These search engines were selected since they index most of the scientific research made in computer science area. The string used for this purpose was *TITLE(("integration" OR "interoperability") AND ("classification" OR "taxonomy" OR "typology" OR "characterization"))*. As result, eleven studies were

selected for reporting some type of classification of interoperability types, namely [17, 48, 13, 40, 8, 42, 51, 66, 62, 11, 19].

Based on these studies, the following classes of interoperability were identified: programmatic, constructive, operational, conceptual, technological, organizational, semiotic, syntactic, semantic, pragmatic, technology, data, human, institutional, technical, data, infrastructure, application, procedures, service, process, and business.

After a careful analysis, these interoperability concepts were grouped in more representative classes, since some of them are synonyms. Therefore, the classification proposed in [40] was selected to group all interoperability concepts reported in the eleven studies. This classification defines four types of interoperability, i.e., technical, syntactic, semantic, and organizational, and they were described in Section 2.1.

Step 2: To identify approaches for addressing each type of interoperability, a substruction strategy was followed, as suggested in [4]. Such strategy consists on finding most *dimensions* for a single *type of a concept* in the typology. In this work, *dimensions* are referred to strategies for accomplishing interoperability, and *types of concepts* are related to the four interoperability types defined in Step 1.

For this, a new search on scientific databases (i.e., Scopus and Google Scholar) was made, to investigate applied approaches for interoperability in contemporary software projects, e.g., SoS, IoT, big data. The search string used for this search was:

TITLE (("integration" OR "interoperability") AND ("systems of systems" OR "systems-of-systems" OR "system-of-systems" OR "system of systems" OR "big data" OR "internet of things" OR "smart cities"))

As a result, 523 studies were initially obtained. After removing duplicated registers, a set of 338 studies were under analysis. From these, only 65 studies reported at least one strategy for interoperability. All studies can be consulted in the GitHub repository: *<https://github.com/linamgr/SBCARS2019>*.

After reading the 65 studies, different approaches were identified, among them, technologies, techniques, frameworks, platforms, middleware, cloud infrastructures, architectural patterns, architectural styles, and architectural tactics.

In this work, techniques, and architectural styles, patterns, and tactics were the architectural strategies explored. This decision was made based on the fact that technologies, frameworks, platforms, middleware, and cloud infrastructures are broader solutions that adopt the architectural strategies investigated in this work.

Step 3: The relation among interoperability types and strategies was made. This mapping was based on evidence reported by the 65 studies about which kind of interoperability requirement a specific strategy was applied. This classification was carried out by two researchers, which discussed and agreed on strategies categorization for each type of interoperability requirement. As a result, the typology with four types of concepts and four dimensions was constructed, which is explained in Section 5 and depicted in Figure 1.

5 ARCHITECTURAL STRATEGIES

Figure 1 illustrates the resulting typology of architectural strategies for types of interoperability. In each interoperability level, these strategies were divided by the type of solution their propose, as technique, or architectural styles, patterns, and tactics.

In the remainder of this section, each architectural strategy in each category of interoperability is presented. Specifically, for each strategy, it is offered its general description, as well as an example that shows how the strategy contributes to achieving the interoperability level for which it was classified.

5.1 Technical Interoperability

Technical interoperability is related to transmitting data between components or individual systems independently of their distance. This type of interoperability is domain-independent and the meaning of data exchanged is not known by systems exchanging the data [8]. Therefore, technical interoperability only guarantees the correct transmission of bits, but in this interoperability level, the meaning of these bits is unknown [40]. Thus, technical interoperability is focused on communication protocols and infrastructure needed for those protocols to operate [64].

Technical interoperability is the first interoperability requirement that must be addressed in an integration project [60, 40]. For supporting technical interoperability requirements, four different architectural styles were included in the typology, namely file transfer, shared database, remote procedure invocation, and messaging [59]. To exemplify the use of those architectural styles imagine multiple applications, built independently, with different languages and platforms, and it is required those applications to work together exchanging diverse information. A possible solution for this requirement is the use of the file transfer architectural style, in which each application produces files containing the information required by the other application. An implementation of this style is the FTP (File Transfer Protocol) used by operational systems, like Linux. Another solution is the messaging architectural style, that can be used to transfer packets of data frequently, immediately, reliably, and asynchronously, using customizable formats (e.g., as those specified in the TCP - Transmission Control Protocol), enabling communication among the different systems. The shared database is another style for information exchange, and it suggests that applications be integrated store their data in a common shared database or repository. This strategy is one of the most common in big data systems, where information produced by different entities are homogenized and stored in a central repository, and after consulted for visualization or decision making purposed. For the same requirement, remote procedure invocation architectural style suggests that each application be developed as a large-scale object or component with encapsulated data. For implementing this strategy, an interface should be provided to allow other applications to interact with the running application [59]. Moreover, REST (REpresentational State Transfer) is an architectural style widely used in Web based systems, such as web services and microservices. The correct implementation of the REST style is denominated RESTful.

Some of these styles have been implemented in different techniques to achieve technical interoperability requirements. Among them, were identified the OGSA-Data Access and Integration (OGSA-DAI) which orient how data can be stored, accessed and transferred

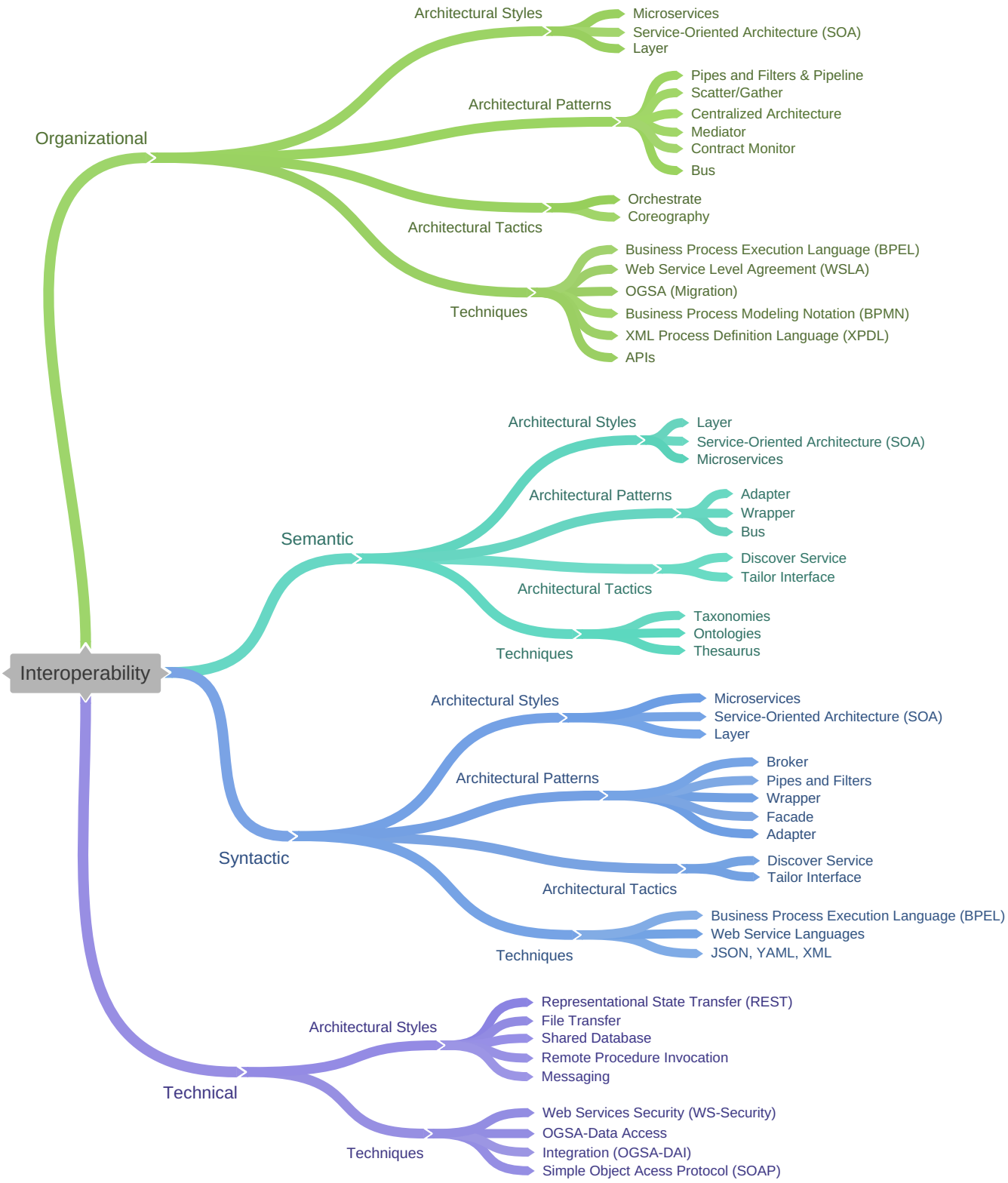

Figure 1: Typology of Architectural Strategies for Interoperability

[60]. Besides, SOAP (Simple Object Access Protocol) is a communication protocol mainly used in service-oriented systems, which allows information exchanging at applying RPC.

5.2 Syntactic Interoperability

Syntactic interoperability is related to providing data formats, well-defined syntax, and encoding (such as message content structure, size of headers, size of the message body, fields contained into a message) [40, 26]. As shown in Figure 1, solutions of all types (e.g., techniques, architectural tactics, styles, and patterns) were identified.

Regarding architectural styles, SOA (Service-Oriented Architecture), microservices, and layers have been used to achieve syntactic interoperability requirements [2, 40, 52, 59]. SOA has been widely used to facilitate the publication of capabilities offered by distributed systems in form of services with well-defined interfaces. Interesting examples about using SOA for interoperability are provided in [25]. Microservices has risen as a sub-style of SOA [67] for overcoming challenges related to interoperability among services. Such challenges were mainly associated with interfaces specifications and continuous integration. Examples of using microservices for interoperability can be consulted in [52]. Layers are one of the most used architectural styles for organizing software systems. For syntactic interoperability, layers are used to separate the type of information embedded in messages packages. An application of layers for semantic interoperability is the TCP/IP packet format.

Five different architectural patterns to achieve requirements of syntactic interoperability were identified. The broker architectural pattern has been used for the coordination of communication among components [59]. For example, servers publish their capabilities (services and characteristics) to a broker, and the clients request a service from the broker and then they redirect the client to a suitable service from its registry [38]. In a complementary way, pipes & filters architectural pattern could be used [38]. An example of using this pattern in interoperability context can be a sequence of messages queues can be used to provide the infrastructure required to implement a pipeline. An initial message queue receives unprocessed messages. A component implemented as a filter task listens for a message on this queue, performs its work, and then posts the transformed message to the next queue in the sequence [57]. The wrapper architectural pattern can also support syntactic interoperability [38]. For example, to achieve integration and interoperation, a wrapper encodes information in a specific message format to be understood by other participants [22]. Wrappers can be used to aggregate new data types in a message, for facilitating the processing of such a message by peers. Another architectural pattern included in the typology is facade. Such pattern hides the complexity of a system and provides a unified interface for a system with multiples components. The unified interface defines an only entry point for clients that would use all functionalities of the system [38]. The adapter architectural pattern also can be used to assist the syntactic interoperability among several systems [38]. Adapters can be used for transforming data formats, vocabularies, and protocols used by different systems in a common standard. In the web systems, adapters are a fundamental strategy to transform messages formats, e.g., from XML to JSON message specification.

Two architectural tactics that can contribute to achieving this type of interoperability requirement were selected in the typology. The tactic of discover service could be used when the different systems that interoperate must be verified and integrated at runtime. For this, this tactic locates a service provider for a required capability, through searching in a known services directory. In this sense, services can be deployed as sets of capabilities that are accessible through some kind of interface [6]. In this same perspective, the tailor interface architectural tactic could be also used for syntactic interoperability, This tactic adds or removes capabilities to an interface. An example of its application is to hide functionalities from untrusted users or to add new permissions to clients over a service [6].

Different techniques were also selected for supporting syntactic interoperability. The Business process execution language (BPEL) could be used for enabling easy and flexible composition of services. Moreover, web service languages for the implementation of services [60], namely, JSON, YAML, XML also are techniques that guide how a message can be formatted to standardize its interpretation by external systems or components.

5.3 Semantic Interoperability

Semantic interoperability is related to data that are conceived as information to be shared, processed, and without ambiguity by systems. In other words, this interoperability level is related to the meaning of the exchanged data as clear definitions of terminology or vocabulary [16]. It is important to highlight that this interoperability level is specific of the domain and context and requires the use of unambiguous information and identifiers [8].

As depicted in Figure 1, the architectural styles SOA, layer, and microservices also can be used to support semantic interoperability requirements, as reported in [2, 40, 52, 59]. The use of these styles has allowed the dissemination of the semantic web 3.0, where services can share well-structured and understood information to homogenize, link, and generate new knowledge, for instance in the case of answer-question services systems [36].

Four architectural patterns were selected in this typology. The adapter architectural pattern allows semantic interoperability between systems exchanging data [38]. This type of interoperability is done through the translation or mapping between different data formats. For example, it is possible to adapt X into Y, where the data type X is different from data type Y, using a transformation rule Z [23]. The wrapper architectural pattern also can be used for this purpose [38], since it adds extra information to data exchanged. For instance, in hospital systems, to construct the whole clinical record of a patient, a wrapper can add information obtained from multiple devices sensing the patient's vital signs. The Enterprise Service Bus (ESB) is another architectural pattern that could be adopted for achieving semantic interoperability. An ESB establishes connectivity protocols, offers capabilities for data and protocol transformation, and message routing [3].

Three techniques have been used to help the building of systems that require semantic interoperability, namely thesaurus, taxonomies, and ontologies. An example of thesaurus is the MetaNet that is a metadata term thesaurus which provides the additional

semantic knowledge that is non-existent within declarative XML-encoded metadata descriptions [32]. An example of taxonomy the one established to support interoperability in the internet of things, which helps the understanding of the meaning of the data that are exchanged among the different systems [54]. The ontologies also can be used to help in information exchange among systems, since ontologies support the specification of the semantics for each terminology used by different systems in an unambiguous fashion [10].

5.4 Organizational Interoperability

Organizational interoperability is related to coordination of distributed workflows and activities that are understood by systems, organizations, or people interacting in business processes. In this case, business processes can be seen as the interaction of multiple organizations to obtain their respective business goals [14]. Organizations that participate in a business process must commit to performing several activities, and the commitments are specified in contracts agreed with other organizations that are taking part in the business process [8].

SOA has been by widely the preferred architectural style for organizational interoperability. SOA (and its sub-style, microservices) allows for the common description of inter-organizational processes when business process definition languages are standardized. Some example of languages is web services defined in WSDL (Web Services Definition Language) or BPML (Business Process Modeling Language) [40]. Additional evidence was identified in the literature about the use of layer and microservices architectural styles to also assist in organizational interoperability [2, 40, 52, 59].

Several architectural patterns also have been used to help in organizational interoperability [38]. In organizational interoperability context, an example of using the pipes-and-filters architectural pattern is when a system just requires to receive messages from its peers, but no request is sent to their constituents, for instance in an IoT based system responsible for monitoring environmental conditions in a smart building. The mediator architectural pattern could be used to intercede interactions among distributed systems. For example, a mediator has a gate used to make available specifications about its interface, as protocols and possible agreements. Middlewares, platforms, and even the cloud serve as an example of capabilities that mediators could implement. The scatter-gather is a conversation pattern that helps the systems to interoperate among them. To exemplify this, image an order processing, where each order item that is not currently in stock could be supplied by one of the multiple external suppliers. The suppliers may or may not have the respective item in stock themselves. For this, it should be requested quotes from all suppliers and decide which one provides with the best term for the requested item. A problem is maintaining the overall message flow when a message needs to be sent to multiple recipients and each of which may send a reply. To solve this, the architectural pattern suggests that use a scatter-gather that broadcasts a message to multiple recipients and re-aggregates the responses back into a single message. This pattern routes a request message to a number of recipients. It then uses an aggregator to collect the responses and distill them into a single response message [55].

The centralized architecture pattern provides a hub which acts as a central point of control and is responsible for ensuring correct communication among systems [35]. An example of this architectural pattern is an anti-guerrilla operation SoS that has a central Theatre Command system, and different constituent systems including UAV scouts; artillery; troops; and communication infrastructures. The central makes operational decisions based upon data sourced from other constituent systems. Therefore, the goals of the SoS are achieved due to the commands of the central which accepts responsibility for delivering SoS functions [27].

The contract monitor architectural pattern should monitoring strategies which indicate precisely how and when a given contract may interact with a component/system [35]. An example of a contract monitor in SoS context is a health monitor observing the interface of a specific constituent system only, to detect contract violations of this constituent system [31]. The service bus pattern can be used for achieving interoperability. It is used for designing and implementing the interaction and communication between mutually interacting software applications in a service-oriented architecture. An example is to use the bus within an organization, or enterprise to manage and govern the use of services within that organization [37].

The orchestration and choreography are two architectural tactics that have been used as strategies to achieve organizational interoperability [6]. Orchestration is a tactic, mainly used in SOA and microservices, that uses a control mechanism to coordinate and manages the invocation of particular services that could be ignorant of each other [6]. An example of the use of this tactic is workflows engines. The mediator pattern can be used for defining a simple orchestration. The orchestration can be specified using the language BPEL [52]. In the same perspective, choreography describes a collection of services that work together to achieve a common and unique goal described by the different web services in interactions with their uses [7]. An example is to use the choreography for proposing an architecture based on the business process transformation centralized technique using graphical representation BPEL [52].

Several techniques have been found to support organizational interoperability. In this interoperability level, Web Service Level Agreement (WSLA) can be used to specify a commitment between a service provider and a client. The OGSA (Open Grid Services Architecture) can be employed for service level attainment and migration. The Business Process Modeling Notation (BPMN) can be applied for understandable and interpretable business process description. The XML Process Definition Language (XPDL) can be applied for process modeling and workflow [49]. Moreover, in SOA, services can expose their capabilities through the use of APIs (Application Programming Interface). Therefore, APIs are a strategy for collaboration and communication of distributed services. An application example of APIs in the fintech domain is offered in [56].

6 DISCUSSION

The strategies presented in the typology in Figure 1 are proven solutions and have been used by software architects and engineers to assist the achieving of interoperability requirements on different levels. The typology provides an overview of existing strategies that

can be used during the building of LSSIS to understand and guide how the interoperability could be addressed. Based on typologies theory [4], it is expected this work supports practitioners works at selecting the strategies for their integration projects.

In this sense, this study contributes to solving some of the interoperability challenges identified by Abukwaik and Rombach [1], namely, the misunderstanding about data structures, procedures, and other interface settings of systems to be integrated, and the lack of a high-level architecture to guide how interoperability requirements could be achieved.

In the first case, architects could use the strategies grouped in the level of syntactic interoperability, as the XML, JSON, and YAML for message formatting. Regarding the misunderstanding of data semantic, the practitioners could use the adapter pattern (grouped in the level of semantic interoperability) to translate the information contained in a message to be interpreted by another system. These transformations could be guided by the use of rules defined in taxonomies, thesaurus, and ontologies. Regarding the interfaces structures provided by individual systems, practitioners can use the tailor interface tactic to add or remove capabilities to an interface, so it can modify published capabilities according to specific requirements. To solve the misunderstanding of procedures, ontologies (strategy grouped in the level of semantic interoperability) could be a good strategy, since they represent a set of domain concepts and the relationships among them, in other words, they assist the specification of the semantics for each terminology used by different systems in an unambiguous fashion.

For the second interoperability challenge, the proposed typology can help practitioners to select among architectural styles and patterns that have been used for different types of interoperability. Specifically, SOA, microservices, and layers can be considered high-level architectures guiding the structure of the system to be integrated, independently of the desired interoperability level. Moreover, architectural patterns related in the typology are proven solutions that can be used for specific types of interoperability requirements. In particular, architectural patterns are well-proven solutions for recurring issues in software architecture. These solutions assist software architectures to understand the impact of their decisions on multiples quality attributes requirements due to they contain information about benefits, consequences, and context of their use [28]. The typology presented in this work is composed of eleven architectural patterns that could contribute to achieving interoperability on different levels. Hence, software architects could use the typology to select an architectural pattern according to the interoperability issue faced, selecting the architectural pattern that best solves their problem. Finally, architectural tactics and techniques can be used as a refinement to construct a well-structured solution by practitioners, since they can be as atomic solutions for very specific interoperability requirements.

The technical interoperability is related, mainly, to messaging exchanges among different systems, in other words, the correct transmission of bits. Few architectural strategies were identified to technical interoperability, due to this kind of requirements can be addressed using well-known techniques and styles highly promoted in web services.

In this perspective, it was possible to show that the proposed typology can be used by software architects to identify and understand different solutions for achieving interoperability in different levels, furthermore such typology contributes to overcome the "*absence of high-level architectures to analyze, comprehend, and guide how interoperability can be addressed in integration projects*" [1]. Moreover, the presented typology is not exclusive to LSSIS and it can be used for any architect to design any class of systems requiring to inter operate and exchange information between them.

6.1 Threats to Validity

In order to minimize biases of this study, the potential threats to validity and some actions carried out to mitigate those threats are discussed below:

- **Missing important strategies:** Aiming to include all architectural strategies for interoperability, a systematic search in scientific data libraries and patterns repositories was carried out. Besides, backward and forward snowballing was conducted using the references and the citations of the initial list of papers returned by the systematic search. Despite this, it is possible we did not identify some architectural solutions for interoperability, henceforth they might have been omitted in the typology proposed in this paper.
- **Appropriateness of strategies:** Architectural strategies presented in the typology were selected based on their applicability to resolve interoperability requirements. Selection was made supported by evidence presented in the analysed studies and through additional consults in pattern repositories. Therefore, the typology only presents relevant strategies to address interoperability requirements at technical, semantic, syntactic, and organizational levels.
- **Incorrect understanding of strategies:** To ensure a correct interpretation of architectural strategies, several examples of their use were consulted, analysed by two researchers, and presented in Section 5, showing how each solution can be applied at each level of interoperability.
- **Scientific databases:** The architectural strategies were identified using scientific databases, which could limit the amount of studies to be analysed, and hence, the number of identified strategies. To mitigate this threat, pattern repositories were used to found all possible solutions that can be used to approach interoperability in software architectures. Therefore, these repositories were used as a complementary mechanisms to increase the searching scope of our solution space.

7 CONCLUSION AND FUTURE WORK

The LSSIS have received considerable attention in the last years, mainly for providing the execution of complex functionalities that individual systems are not able to provide [5]. These systems are composed by heterogeneous, distributed, and independent systems, and the achievement of their business goals are possible through the cooperation between those individual systems. In this scenario, the successful construction of LSSIS highly depends on interoperability capabilities their constituents have.

Architectural synthesis is the activity, within the software architecting process, in which the architect must focus on selecting the most feasible strategies to achieve quality attributes requirements, e.g., security, performance, reliability, safety, and interoperability. In order to contribute to the LSSIS architecting process, a typology was proposed to organize the existing and proven architectural strategies for achieving interoperability requirements. The typology was divided into four groups (namely, techniques, architectural tactics, styles, and patterns) for each type of interoperability requirement. In this sense, the main contribution of this typology is supporting architects to identify and understand solutions for achieving interoperability requirements during the composition of LSSIS.

As future works, it is intended to carry out a case study to validate with practitioners the typology presented in this paper. Moreover, with the case study will be investigated how the typology could contribute to design interoperable environments. It is also intended to systematize and simulate the relationship between the strategies identified in the typology, through techniques of model-driven engineering. Finally, it is aimed to carefully analyze the architectural patterns identified in this study, and consolidate a pattern-language for interoperability in software architectures of LSSIS.

ACKNOWLEDGMENT

This work was supported by the CNPq (National Council for Scientific and Technological Development) and the Brazilian funding agency FAPESP (Grants N.: 2017/06195-9, and 2018/07437-9).

REFERENCES

[1] Hadil Abukwaik and Dieter Rombach. 2017. Software interoperability analysis in practice: a survey. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. ACM, 12–20.

[2] Padroes Arquiteturais. 2019. Repositório online de padrões: architectural pattern. In *http://software-pattern.org/Category/5*. Accessed in 13/07/2019.

[3] Ali Arsanjani, Liang-Jie Zhang, Michael Ellis, Abdul Allam, and Kishore Channabasavaiah. 2007. S3: a service-oriented reference architecture. *IT professional*, 10–17.

[4] Kenneth D Bailey. 1994. *Typologies and taxonomies: an introduction to classification techniques*. Number 102. Sage.

[5] W. C. Baldwin, B. J. Sauser, and J. Boardman. 2017. Revisiting "the meaning of of" as a theory for collaborative system of systems. *IEEE Systems Journal*, 11, 4, 2215–2226.

[6] Len Bass. 2013. *Software architecture in practice*. (3rd ed.). Addison-Wesley.

[7] El Benany and El Beqqali. 2018. Choreography for interoperability in the e-government applications. In *International Conference on Intelligent Systems and Computer Vision (ISCV)*. IEEE, 1–4.

[8] Tim Benson and Grahame Grieve. 2016. *Principles of health interoperability: SNOMED CT, HL7 and FHIR*. Springer.

[9] T. Bianchi, D. S. Santos, and K. R. Felizardo. 2015. Quality attributes of systems-of-systems: a systematic literature review. In *IEEE/ACM International Workshop on Software Engineering for Systems-of-Systems (SESoS)*. Florence, Italy, 23–30.

[10] 2005. *Ontology and semantic interoperability*. (Oct. 2005), 139–160. DOI: 10.1201/9781420036282-7.

[11] Marco Bernardo and Valérie Issarny, editors. 2011. *Interoperability in complex distributed systems. Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–26. ISBN: 978-3-642-21455-4. DOI: 10.1007/978-3-642-21455-4_1.

[12] Frank Buschmann. 1996. *PatternOriented-Software-Architecture-A-System-of-Patterns-Volume-1*. Ashish Raut.

[13] David Chen, Guy Doumeingts, and François Vernadat. 2008. Architectures for enterprise integration and interoperability: Past, present and future. *Computers in Industry*, 59, 7, 647–659. ISSN: 01663615. DOI: 10.1016/j.compind.2007.12.016.

[14] A CHOPRA. 2008. Business process interoperability:extended abstract. In *th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*. P. P. M. Parsons, Ed. Estoril, Portugal: International Foundation for Autonomous Agents and Multiagent Systems Richland, SC.

[15] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. 2002. *Documenting software architectures: views and beyond*. Pearson Education.

[16] DABC. 2004. European interoperability framework for pan-european e-government services (eif 1.0). In *http://ec.europa.eu/idabc/servlets/Docd552.pdf?id=19529*. Accessed in 13/07/2019.

[17] DoD. 1998. *Levels of Information Systems Interoperability (LISI)*. C4ISR. AWG - Architectures Working Group. 139 pp. http://web.cse.msstate.edu/~hamilton/C4ISR/LISI.pdf.

[18] D Harold Doty and William H Glick. 1994. Typologies as a unique form of theory building: toward improved understanding and modeling. *Academy of management review*, 230–251.

[19] European Commission. 2017. European Interoperability Reference Architecture (EIRA) State of play. https://ec.europa.eu/isa2/sites/isa/files/docs/publications/2017-07-18%7B%5C_%7Deira%7B%5C_%7Dstate%7B%5C_%7Dof%7B%5C_%7Dplay%7B%5C_%7Dottawa.pdf.

[20] Eduardo B Fernandez, Hernán Astudillo, and Gilberto Pedraza-Garcıa. 2015. Revisiting architectural tactics for security. In *European Conference on Software Architecture*. Springer, 55–69.

[21] Matthias Galster and Paris Avgeriou. 2012. Qualitative analysis of the impact of soa patterns on quality attributes. In *2012 12th international conference on quality software*. IEEE, 167–170.

[22] Lina Garcés. 2018. *A Reference Architecture for Healthcare Supportive Home (HSH) systems*. PhD thesis. Universiade de São Paulo.

[23] Lina Garcés, Flavio Oquendo, and Elisa Yumi Nakagawa. 2018. Towards a taxonomy of software mediators for systems-of-systems. In *Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse*. ACM, 53–62.

[24] Lina Garcés, Bruno Sena, and Elisa Yumi Nakagawa. 2019. Towards an architectural patterns language for systems-of-systems. In *26TH CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS (PLoP'19)*. HILLSIDE, 1–24.

[25] David Garlan, Felix Bachmann, James Ivers, Judith Stafford, Len Bass, Paul Clements, and Paulo Merson. 2010. *Documenting Software Architectures: Views and Beyond*. (2nd ed.). Addison-Wesley Professional.

[26] Paul Grace, Yérom-David Bromberg, Laurent Réveillère, and Gordon Blair. 2012. Overstar: an open approach to end-to-end middleware services in systems of systems. In *International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 229–248.

[27] Martin Hall-May and Tim Kelly. 2006. Using agent-based modelling approaches to support the development of safety policy for systems of systems. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 330–343.

[28] Neil B Harrison and Paris Avgeriou. 2010. How do architecture patterns and tactics interact? a model and annotation. *Journal of Systems and Software*, 1735–1758.

[29] Neil B Harrison, Paris Avgeriou, and Uwe Zdun. 2007. Using patterns to capture architectural decisions. *IEEE software*, 38–45.

[30] Christine Hofmeister, Philippe Kruchten, Robert L. Nord, Henk Obbink, Alexander Ran, and Pierre America. 2007. A general model of software architecture design derived from five industrial approaches. *Journal of Systems and Software*, 80, 1, 106–126. ISSN: 0164-1212.

[31] W Ling Huang, Jan Peleska, and U Schulze. 2014. Contract support for evolving sos. *Public Document D*.

[32] Jane Hunter. 2001. Metanet: a metadata term thesaurus to enable semantic interoperability between metadata domains. *Journal of Digital information*.

[33] Najhan M Ibrahim and Mohd Fadzil bin Hassan. 2010. A survey on different interoperability frameworks of soa systems towards seamless interoperability. In *International Symposium in Information Technology (ITSim)*. Vol. 3. IEEE. Kuala Lumpur, Malaysia, 1119–1123.

[34] IEEE. 2000. The authoritative dictionary of ieee standards terms. *IEEE Std 100*, 2000, 1–1362.

[35] Claire Ingram, Richard Payne, and John Fitzgerald. 2015. Architectural modelling patterns for systems of systems. In *INCOSE International Symposium* number 1. Vol. 25. Wiley Online Library, 1177–1192.

[36] M. Jang, J. Sohn, and H. K. Cho. 2007. Automated question answering using semantic web services. In *The 2nd IEEE Asia-Pacific Service Computing Conference (APSCC 2007)*. (Dec. 2007), 344–348. DOI: 10.1109/APSCC.2007.69.

[37] Nicolai M Josuttis. 2007. *SOA in practice: the art of distributed system design*. " O'Reilly Media, Inc.".

[38] Reshma Keshav and Rose Gamble. 1998. Towards a taxonomy of architecture integration strategies. In *Foundations of Software Engineering: Proceedings of the third international workshop on Software architecture* number 05. Vol. 1, 89–92.

[39] Philippe Kruchten. 2004. An ontology of architectural design decisions in software intensive systems. In *2nd Groningen workshop on software variability*. Citeseer, 54–61.

[40] Herbert Kubicek, Ralf Cimander, and Hans Jochen Scholl. 2011. Chapter 7 - layers of interoperability. In *Organizational Interoperability in E-Government*. Springer, 85–96.

[41] Christine Legner and Baptiste Lebreton. 2007. Business interoperability research: present achievements and upcoming challenges. *Electronic Markets*, 17, 3, 176–186.

[42] Weizi Li., Kecheng Liu., and Shixiong Liu. 2013. Semiotic interoperability - a critical step towards systems integration. In *Proceedings of the International Conference on Knowledge Discovery and Information Retrieval and the International Conference on Knowledge Management and Information Sharing - Volume 1: KMIS, (IC3K 2013)*. INSTICC. SciTePress, 508–513. ISBN: 978-989-8565-75-4. DOI: 10.5220/0004627005080513.

[43] Azad M Madni and Michael Sievers. 2014. System of systems integration: key considerations and challenges. *Systems Engineering*, 17, 3, 330–347.

[44] Richards Mark. 2015. Software architecture patterns-understanding common architecture patterns and when to use them. (2015).

[45] Shaw Mary and Garlan David. 1996. Software architecture: perspectives on an emerging discipline. *Prentice-Hall*.

[46] Gianantonio Me, Giuseppe Procaccianti, and Patricia Lago. 2017. Challenges on the relationship between architectural patterns and quality attributes. In *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 141–144.

[47] Ali Mesbah and Arie Van Deursen. 2007. An architectural style for ajax. In *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, 1–9.

[48] E. Morris, L. Levine, C. Meyers, P. Place, and D. Plakosh. 2004. Systems of Systems Interoperability (SOSI): Final report. Tech. rep. 2004-TR-004. CMU/SEI-2004, 1–53. https : / / resources . sei . cmu . edu / asset % 7B % 5C _ %7Dfiles / TechnicalReport/2004%7B%5C_%7D005%7B%5C_%7D001%7B%5C_%7D14375.pdf.

[49] Geoffrey Muketha. 2014. A review of agent based interoperability frameworks and interoperability assessment models. *Scholars Journal of Engineering and Technology (SJET)*.

[50] Elisa Yumi Nakagawa, Flavio Oquendo, Paris Avgeriou, Carlos E Cuesta, Khalil Drira, José Carlos Maldonado, and Andrea Zisman. 2015. Foreword: towards reference architectures for systems-of-systems. In *International Workshop on Software Engineering for Systems-of-Systems (SESoS)*. IEEE Press. Florence, Italy, 1–4.

[51] Frâncila Weidt Neiva, José Maria N. David, Regina Braga, and Fernanda Campos. 2016. Towards pragmatic interoperability to support collaboration: a systematic review and mapping of the literature. *Information and Software Technology*, 72, 137–150. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.12.013.

[52] Sam Newman. 2015. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.".

[53] Claus Ballegaard Nielsen, Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, and Jan Peleska. 2015. Systems of systems engineering: basic concepts, model-based techniques, and research directions. *ACM Computing Surveys (CSUR)*, 48, 2, 18.

[54] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. 2019. Interoperability in internet of things: taxonomies and open challenges. *Mobile Networks and Applications*, 796–809.

[55] Repositorio Online. 2019. Architectural patterns (3). In *http://pautasso.info/lectures/s14/sad/8-patterns/patterns3.pdf*. Accessed in 13/07/2019.

[56] Repositorio Online. 2019. Pattern: service level agreement. In *www.microservice-api-patterns.org/patterns/quality/qualityManagementAndGovernance/ServiceLevelAgreement*. Accessed in 13/07/2019.

[57] Pipes and Filters. 2019. Cloud design patterns. In *https://docs.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters*. Accessed in 13/07/2019.

[58] Roger S Pressman and Bruce R. Maxim. 2014. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education.

[59] Repositorio. 2019. Repositório online de padrões: enterprise integration patterns. In *http://www.enterpriseintegrationpatterns.com*. Accessed in 13/07/2019.

[60] Reza Rezaei, Thiam Kian Chiew, and Sai Peck Lee. 2014. An interoperability model for ultra large scale systems. *Advances in Engineering Software*, 67, 22–46.

[61] Anubha Sharma, Manoj Kumar, and Sonali Agarwal. 2015. A complete survey on software architectural styles and patterns. *Procedia Computer Science*, 16–28.

[62] Iohan Gonçalves Vargas, Thiago Gottardi, and Rosana Teresinha Vaccare Braga. 2016. Approaches for integration in system of systems: a systematic review. In *2016 IEEE/ACM 4th International Workshop on Software Engineering for Systems-of-Systems (SESoS)*. IEEE, 32–38.

[63] H Veer and A Wiles. 2008. Achieving technical interoperability-the etsi approach, european telecommunications standards institute. (2008).

[64] Hans van der Veer and Anthony Wiles. 2008. Achieving technical interoperability. *European telecommunications standards institute*.

[65] Kim H Veltman. 2001. Syntactic and semantic interoperability: new approaches to knowledge and the semantic web. *New Review of Information Networking*, 7, 1, 159–183.

[66] Peter Wegner. 1996. Interoperability. *ACM Comput. Surv.*, 28, 1, (Mar. 1996), 285–287. ISSN: 0360-0300. DOI: 10.1145/234313.234424.

[67] Olaf Zimmermann. 2017. Microservices tenets: agile approach to service development and deployment. *Computer Science-Research and Development*, 32, 3-4, 301–310.