Towards an architectural patterns language for Systems-of-Systems

LINA GARCÉS, Department of Computer Systems, University of São Paulo
BRUNO SENA, Department of Computer Systems, University of São Paulo
ELISA Y. NAKAGAWA, Department of Computer Systems, University of São Paulo

Systems-of-Systems (SoS) architectures are inherently dynamic; hence, they must support continuous modification in the behaviour and configuration of these systems at runtime as a result of changes in the environment, new SoS missions, and failures or unavailability of constituents. Modifications should occur without affecting the integrity of constituents, the accomplishment of SoS missions, neither their reliability, security, safety, nor other quality attributes. Architecting SoS requires then important investments in human, time, and economic resources, bringing big challenges. Architectural patterns have been widely used to improve software architecture quality, decreasing costs of design and promoting reuse of good practices and well-known solutions. Nowadays, a great amount of architectural patterns is available; most of them are domain-independent, which harness their selection in specific software projects. The main goal of this paper is to contribute to reduce the work of architects during the choice of better architectural patterns for their SoS. For this, we present a set of patterns that are commonly used to construct such architectures. Additionally, benefits of adopting these patterns are described. To demonstrate their feasibility, we present *HSH-SoS*, a pattern-base architecture for SoS that presents how different patterns can interact to create a solution for SoS. As results, the architectural patterns reported in this work are feasible candidates to compose a language for recurrent problems in SoS architectures. However, formalization of such language is an open issue that we intend to address in future works.

Categories and Subject Descriptors: 500 [**Software and its engineering**]: —*Software architectures*; 300 [**Software and its engineering**]: —*Ultra-large-scale systems*; 300 [**Software and its engineering**]: —*Interoperability*; 300 [**Software and its engineering**]: —*Design patterns*

General Terms: Architectural Patterns

Additional Key Words and Phrases: software architecture, architectural pattern, patterns language, systems-of-systems

ACM Reference Format:

Garcés, L. and Sena, B. and Nakagawa, E.Y. 2019. Towards an architectural patterns language for Systems-of-Systems. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. V (October 2019), 24 pages.

1. INTRODUCTION

Systems-of-Systems (SoS) have been growing in popularity, due to their complexity and adaptive characteristics [Jamshidi 2011; Veríssimo 2006; Bondavalli et al. 2016]. SoS are complex systems composed of a collection of distributed systems (constituents), which gather their resources and capabilities together to provide new and enhanced services aiming at accomplishing missions that are not feasible by any of the constituents working independently [ISO/IEC-24765 2010; Nielsen et al. 2015; Mittal and Rainey 2015; Zoppi et al. 2017]. In general, [Maier 1998], [DeLaurentis 2005] and [Nielsen et al. 2015] defined SoS as a type of complex systems that have five characteristics: (i) operationally independence: each constituent has its individual purpose and continues

Author's address: L. Garcés, ICMC, Lab. 6-204. Avenida Trabalhador São-carlense, 400 - Centro. ZIP: 13566-590 - São Carlos - SP, Brazil; email: linamgr@icmc.usp.br;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 26th Conference on Pattern Languages of Programs (PLoP). PLoP'19, OCTOBER 7-10, Ottawa, Ontario Canada. Copyright 2019 is held by the author(s). HILLSIDE 978-1-941652-14-5

to operate even disassembled from the SoS; (ii) managerial independence: each constituent is developed and maintained by independent organizations; (iii) evolutionary development: SoS development is evolutionary and adaptive, in which structures, functions, and purposes change according to the needs of the SoS; (iv) distribution: constituents are dispersed and must provide a communication protocol to interoperate to the SoS; and (v) emergent behavior: behavior of the SoS emerges as a result of the interdependent collaboration of its constituents.

This scenario leads to the need of building SoS that must be developed and maintained in a way that they can evolve more easily, and in which the impact of changes must be minimized by the operational and managerial independence of the constituents [Nielsen et al. 2015; Zoppi et al. 2017]. The difficulties for building SoS emerge due to the unpredicted collaborations among their constituents [Dersin 2014]. SoS must deal with the loss, inclusion, and replacement of constituents at runtime, besides adjusting their behavior to available resources [Maier 1998]. Therefore, SoS architects must create solutions that benefit quality attributes of interoperability, maintainability, dynamic evolution, flexibility, adaptivity (adaptation in runtime), scalability, and so forth. Moreover, architectural decisions addressing these attributes must be well-defined and documented to facilitate the understanding of SoS architecture by different groups of stakeholders, promoting the quality of this complex systems.

For this purpose, architectural decisions can be captured or documented through architectural patterns (sometimes referred as styles) [Harrison et al. 2007], which are well proven solutions for recurring problems. These patterns help architects to understand the impact of architectural decisions on quality attributes requirements, because they contain information about benefits, consequences, and context of their usage [Harrison et al. 2007].

Architectural patterns constitute a mechanism to capture domain experience and knowledge allowing their reapplication when a similar architectural problem is encountered [Pressman and Bruce R. Maxim 2014]. According to [Bass et al. 2013], an architectural pattern establishes a relationship among: (a) *a context:*, that describes the recurrent situation which origins a problem; (b) *a problem:*, that generally describes the problematic found in the context. Often, this description includes quality attributes that must be met by the pattern; and (c) *a solution:*, that details the successful architectural resolution to the problem. The solution describes the architectural structures that solve the problem, responsibilities of elements and the relationships among them, as well as their runtime behavior during their interactions.

An important aspect of architectural patterns is to make clear what quality attributes are provided by the static and runtime configurations of elements, since software architectures are mostly driven or shaped by quality attributes requirements, which determine and constrain the most important architectural decisions [Clements and Bass 2010]. These patterns represent the first place in the software creation process in which quality requirements should be addressed. Such quality requirements are thus, satisfied by the various structures designed into the architecture, and the behaviors and interactions of the elements that populate those structures [Bass et al. 2013].

In a broad perspective, a pattern language is a structured collection of patterns that rely on each other to transform requirements and constraints into an architecture [Coplien 1998]. A pattern language is a way of subdividing a general problem and its complex solution into a number of related problems and their respective solutions. Each language pattern solves a specific problem in the common context shared by the language. Each pattern can be used separately or with a number of patterns of the language. This means that a pattern alone is considered useful even if the language is not being used.

Due to their benefits, pattern languages have been focus of research for different systems since the beginning of this century. It is possible to find several studies addressing this level of abstraction of architectures [Avgeriou and Zdun 2005; Fernandez and Pan 2001; John et al. 2009; Fehling et al. 2011]. Avgeriou and Zum [2005] presented a language of classical patterns to guide the overall design of architectures independent of the domains. Fernandez and Pan [2001] reported an architectural pattern language for security models. John et al. [2009] proposed a pattern language for usability support, while the work of Fehling et al. [2011] focused on cloud-based applications. Finally, more recent studies investigates a pattern language for scalable systems based on microservices [Márquez et al. 2018], and for big data systems [Sena 2018].

Regarding SoS, investigations on architectural patterns started approximately ten years ago. Ingram et al. [2015] report some relevant characteristics required in an architectural pattern for SoS: i) the extent to which a SoS may need to accommodate legacy constituents systems and unsynchronized evolution; ii) the level of assurance of emergent behaviors, which are impossible to fully predict but need to be avoided; iii) the need for a central decision-making authority; iv) the need for separation of concerns; v) the importance of resilience and adaptability; and vi) the need for a clear chain of command, or accountability for decisions. In this perspective, some studies have investigated taxonomies of architectural patterns and styles to address challenges associated to these systems [Romay et al. 2013; Rothenhaus et al. 2009; Ingram et al. 2015; Garcés et al. 2020], namely, mitigation of the impact caused by changes occurred at constituent systems level, supporting to the understanding of emergent behaviors in runtime, centralization or decentralization of the decision-making authority, separation of concerns of SoS and of their constituents, facilitation of constituent systems collaboration, and adaptability and resilience through SoS dynamic reconfigurations [Ingram et al. 2015].

Despite the existence of relevant works, the research on the development and application of architectural patterns for complex and large-scale systems, as are SoS, is still in its infancy [Kalawsky et al. 2013]. Hence, more efforts must be done to facilitate architects work, decreasing their time spent at searching best alternatives for their SoS, and improving the quality of their architectures.

In this work we report the most common architectural patterns and styles used in SoS, highlighting their contribution to SoS characteristics and quality attributes such as interoperability, flexibility, scalability, dynamism, and adaptations in runtime. These patterns and styles were identified in scientific data libraries and patterns repositories. To demonstrate their applicability in SoS, we present examples of how these patterns and styles could cooperate to create more complete solutions. It is our intention to consider them in a patterns language for SoS architectures.

The remainder of this paper is structured as follows. Section 2 details methods used to identify architectural patterns for SoS. These patterns are described in Section 3, and their application in SoS is reported in Section 4. Some discussions are presented in Section 5. Finally, conclusions and future works are disclosed in Section 6.

2. METHODS

In this section, we present methods used to identify and select the set of architectural pattens / styles more appropriated for architecting SoS. Shortly, the following four steps were conducted:

Step - 1. To understand challenges at architecting SoS. For this, we performed: (i) a study of literature reporting SoS fundamentals [Maier 1998; Jamshidi 2011; Nielsen et al. 2015; Bondavalli et al. 2016; Oquendo 2017]; (ii) an analysis of architectural properties of SoS [Ingram et al. 2015; Maier 1998]; and (iii) an identification of requirements of quality attributes of SoS in different domains as healthcare [Garcés 2018], big data [Sena 2018], crisis and emergency management [Paes et al. 2016], military [de Barros Paes et al. 2018], and smart cities [Cavalcante et al. 2017]. Table II lists SoS requirements found in this step.

Step - 2. To characterize architectural patterns and styles used for constructing SoS architectures. For this, the search string in Table I was executed in the data libraries Scopus and GoogleScholar. Such libraries were selected because they index research works published in other important computer science academic libraries, such as ACM Digital Library, SpringerLink, and IEEE Xplore. The search was constrained to Title-Abstract-Keywords.

Table I. Search String

("architectural pattern?" OR "architectural style?") AND ("System-of-Systems" OR "Systems-of-Systems" OR "System of Systems")

As result of this search, the following studies were identified: [Muller et al. 2009; Rothenhaus et al. 2009; Cuesta and Romay 2010; Nichols and Dove 2011; Cuesta et al. 2013; Kazman et al. 2013; Romay et al. 2013; Weyns and Ahmad 2013; Barnes et al. 2014; Ingram et al. 2014; Ingram et al. 2015; Cuesta et al. 2016]. For the best of our knowledge, they are the only works investigating architectural patterns and styles for SoS architectures. The identified architectural patterns and styles for SoS were: Centralized Architecture, Services-Oriented and Micro-Services Architectures, Publish-Subscribe, Trickle-Up, Reconfiguration Control Architecture, Contract Monitor, Pace-Layer, and Evolution Styles.

Step - 3. To investigate architectural patterns and styles for systems whose architectures present self-* properties [Salehie and Tahvildari 2009], namely, self-configuring, self-healing, self-optimizing, self-protecting, self-management, self-organizing, and reflection. Hence, we identified architectural solutions that allow dynamism and self-organization of software structures in runtime, since they are important properties that SoS architectures must address [Nichols and Dove 2011; Oquendo 2017]. Architectural patterns and styles used in systems with self-* properties are: Layers, Pipes-and-Filters, Publish-Subscribe, SOA, Blackboard, Shared-Data Repository, Master-Slave, Observer-Controller, Enterprise Service Bus, Epistemic Control Loop, Autonomic Manager (MAPE-K), Meta Controller, Deep Model Reflection, Broker, Decorator, and Aggregator [Garcés et al. 2020].

Step - 4. To conduct a search in patterns repositories [van der Aalst and ter Hofstede 2017; Met 2019; AEI 2019; The Hillside Group 2019]. Such repositories were mined to identify possible solutions that could be adopted in SoS architectures. From these repositories, the architectural style AOM (Adaptive Object Model) [Yoder and Johnson 2002] was selected.

Step - 5. To select patterns and styles for SoS. We studied patterns and styles obtained in Steps 2, 3, and 4 and we selected the most common and feasible architectural solutions to support the achievement of SoS requirements investigated in Step 1. Table II lists the architectural patterns / styles that literature have reported as useful to address the specific requirements of SoS architectures.

Patterns/styles selected in this work and listed in the last column of Table II were reported in [Ingram et al. 2015; Avgeriou and Zdun 2005; Kazman et al. 2013; Cuesta et al. 2016; Rothenhaus et al. 2009; Muller et al. 2009; Weyns and Ahmad 2013; Josuttis 2007; Romay et al. 2013; Cuesta et al. 2013; Cuesta and Romay 2010; Yoder and Johnson 2002].

Requirements of SoS architectures were extracted from [Ingram et al. 2015; Maier 1998; Nielsen et al. 2015] and are listed in the second column of Table II.

Step - 6. To apply patterns and styles in a SoS. These patterns and styles were used as basis for constructing SoS architectures in different domains. Some experiences of using these architectural solutions were abstracted and are presented in Section 5.

3. ARCHITECTURAL PATTERNS FOR SOS

In this section, the patterns and styles identified for SoS architectures are described.

Centralized Architecture*.* In a centralized architecture, a central controller, defined as a hub, is responsible for guarantee the correct behavior and allow the accomplishment of missions of the SoS [Ingram et al. 2015]. Control can be characterized as: (i) fully centralized, when just one central controller is responsible for meeting SoS goals; (ii) hierarchical centralized, when constituent systems act as controllers themselves; (iii) hybrid centralized-distributed, if control activities are distributed among different constituent systems and hubs [Ingram et al. 2015].

Service-Oriented Architecture*.* Service Oriented Architecture (SOA) gives support to deal with distributed business processes. Systems participating in a SOA-based system execute business activities and are heterogeneous and under control of different owners [Josuttis 2007]. SOA pattern promotes interoperability among distributed systems to accomplish business functionalities (services) through the easy integration of their capabilities [Josuttis 2007]. SOA-based system behavior and performance can be studied by analyzing service descriptions [Ingram

Table II. Relation between SoS requirements and architectural patterns / styles. Evidences from literature.

ID	SoS Requirements [Ingram et al. 2015; Maier 1998; Nielsen et al. 2015]	Architectural Patterns/Styles
R01	Integration of constituent systems to enhance their collaboration	Centralized Architecture, Publish-Subscribe, SOA (Service-Oriented Architecture), MSA (Micro-Services Architecture), ESB (Enterprise Service Bus), Trickle-up, Pace-Layer, MAPE-K (Autonomic Manager)
R02	Assurance of emergent behaviors	Centralized Architecture, SOA, MSA, Trickle-up, Reconfiguration Control Architecture, Evolution Styles, Reflection, MAPE-K
R03	Central decision making authority	Centralized Architecture, SOA, Publish-Subscribe, Reconfiguration Control Architecture, Trickle-up, MAPE-K
R04	Separation of concerns and low coupling	SOA, MSA, Publish-Subscribe, Pipes-and-Filters, Reconfiguration Control Architecture, ESB, Trickle-up, Pace-Layer, Evolution Styles, Reflection, MAPE-K
R05	Resilience and adaptability of decisions	SOA, MSA, Publish-Subscribe, Pipes-and-Filters, Trickle-up, Reconfiguration Control Architecture, Reflection, MAPE-K, AOM (Adaptive Object-Model)
R06	Dynamic reconfigurations	MSA, Pipes-and-Filters, Reconfiguration Control Architecture, Pace-Layer, Evolution Styles, Reflection, MAPE-K, AOM
R07	Missions accomplishment	Centralized Architecture, Reconfiguration Control Architecture
R08	Clear command chain and accountability of decisions	Centralized Architecture, SOA
R09	SoS evolution	SOA, MSA, MAPE-K, Publish-Subscribe, Pace-Layer, Evolution Styles, Reflection, AMO

et al. 2015]. SOA also improves reliable solutions, since capabilities can be offered by multiple providers (e.g., constituent systems), hence, if a provider is unavailable another can replace its participation. SOA minimizes the impact of modifications and failures of the SOA-based systems on participant systems, and vice-versa, due this pattern promotes loose-coupling. Moreover, SOA improves flexibility and horizontal scalability, since new systems or systems capabilities can be added to a SOA solution due to standardized interfaces and pre defined contracts. Vertical scalability is achieved with the coordination or adaptation of services for creating composed and process services, which are high-level services defined to execute complex business activities work-flows.

Composed and process services are designed using orchestration or choreography approaches. In orchestration, a central controller coordinates all process activities. Choreography approach involves collaboration between participants (services), which are responsible for executing one or more activities. In choreography, no central controller exists, hence, collaboration rules must be defined, since services are unaware of activities performed by other participants [Josuttis 2007]. Choreography allows better scalability than orchestration, since control can be distributed among participants, however, it can impact on the system performance, since the full-decentralized control requires to exchange large amounts of information between participants [Garcés et al. 2020].

Micro-Services Architecture. Micro-Services Architecture (MSA) are an architectural style [Lewis and Fowler 2014] that overcomes development and deployment issues initially presented in SOA [Zimmermann 2017]. MSA can be used as style to structure SoS architectures [Cuesta et al. 2016], since MSA can promote: (i) operational and managerial independence of constituent systems in an SoS, considering each individual system as a micro-service that is geographically distributed and heterogeneous, and maintains its own technologies and data repositories; (ii) evolutionary development of SoS, due the facilities for continuous development and deployment that MSA present, without impacting other micro-services operations; and (iii) decentralized governance that allows scalability and continuous evolution of SoS.

Enterprise Service Bus. Enterprise Service Bus (ESB) is the technical backbone of SOA [Josuttis 2007]. The ESB provides connectivity, data transformation (to allow semantic and syntactic interoperability), protocol transformation (allowing technical interoperability), intelligent routing (e.g., using mediators, point-to-point connectors, or interceptors), means to deal with security and reliability of services, service management (adding, removing,

deactivating, or updating services), and monitoring and logging services operations (to measure Quality of Service - QoS). In the context of SoS, heterogeneous ESB can be integrated to offer large scale mediation among distributed heterogeneous constituent systems.

Publish-Subscribe. Publish-Subscribe pattern allows event consumers (subscribers) to be registered for specific events, and event producers to publish (raise) specific events that reach a specified number of consumers. The Publish-Subscribe mechanism is triggered by the event producers and automatically executes a callback-operation to the event consumers. The mechanism thus takes care of decoupling producers and consumers by transmitting events between them [Avgeriou and Zdun 2005].
In the context of SoS, a constituent system can act as a publisher and/or subscriber. This pattern can be divided in data-centric or content-based publish-subscribe pattern with centralized or decentralized control [Ingram et al. 2015]. Some benefits of this pattern are: (i) performance improvement at communicating data among SoS entities, (ii) promotion of modifiability due to low coupling between entities, (iii) encourage dynamic scalability and evolution, since entities can enter or exit without affecting others entities [Garcés et al. 2020], and (iv) provides resilience, since subscribers can register with multiple publishers for a topic [Ingram et al. 2015].

Pipes-and-Filters. Pipes-and-Filters pattern allows to divide complex task in more easy understood sub-task that are sequentially executed. Each sub-task is implemented as filter that has as responsibility to handle only such task. A filter has a limited amount of inputs and outputs and is connected with other filters though pipes. Filters are unaware of the purpose of their peers. Each filter consumes and delivers data incrementally, which maximizes the throughput of each individual filter, since filters can potentially work in parallel [Avgeriou and Zdun 2005]. This architectural style also supports reusability of filters, since a filter can be added, removed, or composed in different work-flows and connected through multiple pipes, without affecting other filters operations [Hohpe and Woolf 2003]. In the context of SoS, Pipes-and-Filters pattern can grant concurrent execution of adaptations by layers, where each layer is seen as a filter [Garcés et al. 2020].

Trickle-Up Software Pattern. The Trickle-Up pattern is a multi layered pattern that allows separation of concerns for data management. Data object management and fusion logic are independent services that can be binding in runtime to promote control and monitoring of data management [Rothenhaus et al. 2009]. This pattern is principally used when environment data are collected and must be further analyzed and aggregated to create consolidated reports of a situation. In SoS context, the trickle-up pattern can be used for knowledge aggregation and discovering required to support SoS emergent behaviors.

Reconfiguration Control Architecture. The aim of this pattern is to promote dynamic reconfigurations of software architectures. For this, a reconfiguration control entity (e.g., a central controller) is responsible for monitoring constituent systems performance and functionality in the form of metadata obtained from each system. Based on such metadata, the controller, making use of reconfiguration policies, determines when a reconfiguration is necessary and what actions must be executed [Ingram et al. 2015]. In SoS, this pattern can be used to avoid degradation of SoS missions due to modifications or unavailability of its constituent systems.

Contract Monitor. This pattern aims to monitor constituent systems interfaces in order to identify possible deviations from its expected functionalities that can prejudice expected SoS emergent behaviors. For using this pattern, it is assumed that constituent systems interfaces are associated to contracts of behavior, and that composition of such contracts can be correlated to SoS emergent behaviors [Ingram et al. 2015]. Contract monitors can be implemented internally in a constituent system, as an entity, under SoS control, monitoring constituent systems interfaces, or as an external entity monitoring interactions between constituent systems to study emergent behaviors [Ingram et al. 2015].

Pace-Layers. Also named layers of change, is proposed as an initial strategy to design SoS [Romay et al. 2013]. Layers allow the construction of complex behaviors through layers hierarchies, where lower layers implement fast adaptations (i.e., reconfigurations on constituent systems) and higher layers are responsible for time demanding adaptations (i.e., selection of the best policy or plan to achieve SoS missions based on current system

status)[Garcés et al. 2020; Romay et al. 2013]. Lower layers adaptations aim to achieve performance requirements, and adaptations in higher layers seem to address reliability requirements [Garcés et al. 2020]. Moreover, as layers allow separation of concerns, this property supports maintainability and reusability requirements. Interoperability can also be supported by SoS lower layers at establishing well-defined interfaces of constituent systems.

***Evolution Styles**.* Evolution styles have been proposed to modify software architectures of running systems regarding new requirements, technologies, or to achieve self-* properties, e.g., self-healing, self-organization [Cuesta et al. 2013; Garlan et al. 2009]. Evolutions styles are composed of [Cuesta and Romay 2010]: (i) evolution conditions that allow to identify situations that should be handled through system evolution ; (ii) evolution decisions, that are alternatives made as reaction to an evolution condition; (iii) evolutionary steps that realize the evolution decision; (iv) evolution patterns, that are sequence of evolution decisions, being represented as a decision tree; and (v) evolution styles, that are a set of evolution patterns conceptually related. In SoS context, evolution styles could orient reconfigurations of SoS architectures due to emergent behaviors or modifications in missions to be performed by the SoS.

***Reflective Architecture**.* Reflection is defined as "the capability of a system to rationalize and act upon itself" [Cuesta et al. 2001; Maes 1987]. A system with reflection capability is composed of two layers. A bottom layer describing system operations and configuration (e.g., components, interfaces, data, interconnections, etc.), and an upper layer embracing an internal meta-model representing how the system perceives and/or modifies itself [Cuesta et al. 2001]. Such model contain all structural an behavioral aspects of the system and is separated from the application logic components [Avgeriou and Zdun 2005]. Hence, the system will reflect changes performed in the model, and vice versa through the execution of two basic operations named *reification* and *reflection*. Reification is the action to transfer current system status to make alterations in the meta-model. Reflection is the operation of change system configuration or behavior regarding modifications in the meta-model. In this perspective, reflection architecture makes the system more flexible, since it can adapt itself to changing conditions [Cuesta et al. 2001; Cuesta and Romay 2010]. Moreover, reflection allows for coping with unforeseen situations automatically [Avgeriou and Zdun 2005]. In SoS, the reflection architecture can support evolution and changes in runtime, promoting the anticipation of SoS predicted emergent behaviors and the detection of unforeseen ones.

***AOM**.* The Adaptive Object-Model (AOM) is an architectural style used as a strategy to build dynamic and adaptable systems. AOM are also known as a particular kind of reflective architecture [Yoder 2017]. An AOM represents, as metadata, classes, attributes, relationships, and behavior that rule a software system. Hence, the system's architecture is an abstract model that can be instantiated to the system during its operation. The metadata (object model) can be changed at anytime by users to reflect changes in the domain. These changes modify the system's behavior and structure. Hence, the system stores its Object-Model in a database (or repository) and interprets it. Consequently, the model is adaptable; when the descriptive information is modified, the system immediately reflects those changes [Yoder and Johnson 2002].
SoS architectures must be defined in an abstract way [Oquendo 2017], describing possible arrangements or coalitions (among SoS' independent systems and dependent software entities) that could be configured by the central authority to achieve different missions. Hence, the application of the AOM style in SoS can favor the abstract descriptions and modifications of their architectures.

***MAPE-K**.* Autonomic computing has feedback control loops as first-class entities. The most used one is the Autonomic Manager [IBM 2006], or MAPE-K loop, which consists of five components, i.e., Monitor, Analyser, Planner, Executor and Knowledge, and two pairs of sensors and actuators as input and output interfaces, respectively.
Sensors collect information about the monitored system (e.g., constituent system). The *monitor* component filters the accumulated sensor data, and stores relevant events in the *knowledge base* for future reference. The *analyzer* compares event data against patterns in the knowledge base to diagnose and store symptoms. The *planner* interprets symptoms and devises a plan to execute changes in the managed system (e.g., the SoS or parts of it)

through the *actuators* [IBM 2006; Muller et al. 2009]. An autonomic manager maintains its own knowledge (e.g., information about its current state as well as past states) and has access to knowledge that is shared among collaborating autonomic managers. MAPE-K loops have been used as an architectural solution to design software architectures of systems with adaptations at runtime, such as are SoS [Arcaini et al. 2018; Garcés et al. 2020]. Control in autonomic managers can be centralized, decentralized or fully-decentralized [Weyns et al. 2013]. Decentralization level is defined based on how control decisions in the system are coordinated among the MAPE components. In *centralized control*, a single component dedicated to one of the MAPE activity exists. *Decentralized control* is characterized by the existence of multiple components responsible for one of the MAPE activities, i.e., multiple components realizing monitoring activities. In centralized and decentralized control, MAPE components are deployed in a single node. *Fully-decentralized control* is similar to decentralized control, but with multiple MAPE components deployed in multiples nodes [Garcés et al. 2020]. To realize decentralized and fully-decentralized systems, an arrangement of collaborating autonomic managers works towards a common goal [Muller et al. 2009]. Arrangements can be done through sensors and actuators, which communicate autonomic manager status to peers and execute input policies sent from peers.

4. APPLYING ARCHITECTURAL PATTERNS IN SYSTEMS-OF-SYSTEMS

The abovementioned architectural patterns were used by authors during the architectural design of SoS in domains of healthcare [Garcés 2018], crisis and emergency management [Paes et al. 2016], and big data [Sena 2018]. Patterns application was assessed through the conduction of case studies, therefore, evidences presented in this section have been conceptually proven.

Specifically, in this section, we explain the use of architectural patterns and styles through the presentation of the architecture of Healthcare Supportive Home SoS (*HSH-SoS*) [Garcés 2018]. *HSH-SoS* are collaborative SoS, since each constituent system maintains its operational and managerial independence, but constituents collaborate with the HSH system to achieve and evolve the global missions. Examples of constituent systems are smart homes, domotic systems, electronic health records (EHR) systems, Health Information Systems (HIS), telemedicine systems, teleconsultation systems, emergency systems, activity monitoring systems, physiological monitoring systems, smart devices, company robots, mobile apps, smart TV apps, and rehabilitation systems, among others. *HSH-SoS* provide health services to patients in their residence and supply them with autonomy through the interaction and coordination of their constituent sytems. An example of an *HSH-SoS* for patients with diabetes mellitus can be consulted in [Garcés 2019a].

The general representation of *HSH-SoS* architecture is depicted in Figure 1. *HSH-SoS*' structures are organized following the **SOA** pattern, specifically the Service-Oriented Solution Stack (S3) reference architecture [Arsanjani et al. 2007]. *HSH-SoS* are structured in seven layers as presented in Figure 1:

Basic Services layer

It is the lowest level layer, containing two types of services, named constituent systems services and control services.

<u>Constituent Systems Services</u>, consist of capabilities offered by individual systems and that are needed to achieve SoS missions. It is important to highlight that, due to the fact that such services are offered by constituent systems, they are not under the control of the SoS, but they operate and are managed independently by external organizations. Hence, constituent systems services can be developed as **micro-services** and deployed in the SoS architecture following well structured interfaces and contracts [Cuesta et al. 2016].

In the context of *HSH-SoS*, a type of constituent systems' services are those responsible to offer patient's physiological parameters, for instance, body temperature, body measures (e.g., weight, height, waist diameter), heart rate, respiratory rate, blood pressure, oxygen levels (SPO2), among others parameters. Additional services required in *HSH-SoS* are those for defining patient's activities levels, assessing patient's nervous and cardiovascular system, and monitoring patient's environment (e.g., house).

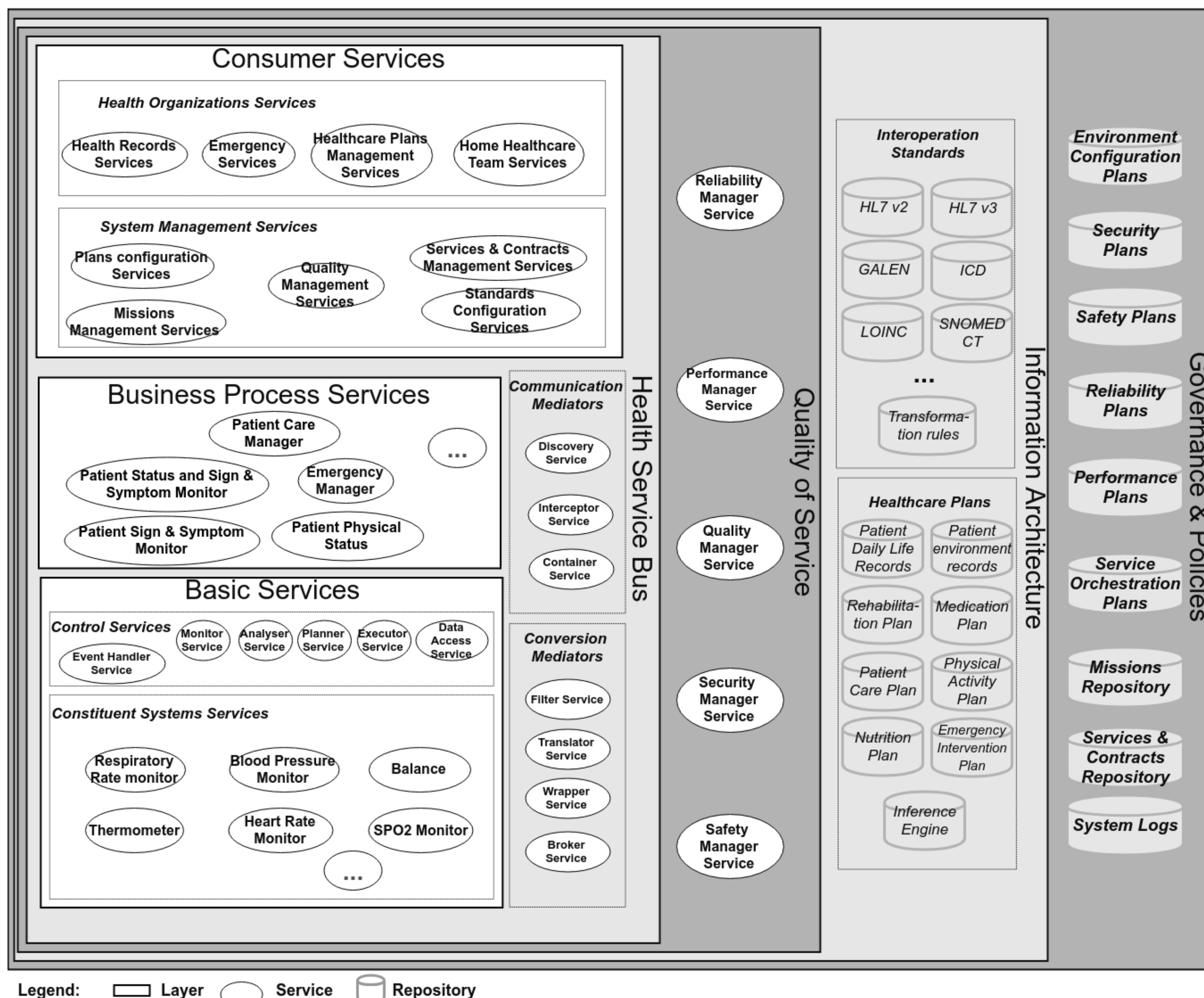

Fig. 1. *HSH-SoS* architecture overview. Application of the SOA pattern. Adapted from [Garcés 2018].

Constituent systems capabilities can be developed as micro-services and deployed in the SoS architecture following well structured interfaces.

Control services, allow orchestration and choreography of basic services and their composition into business process services. In SoS, control services were proposed to the construction of autonomic managers (e.g., **MAPE-K**) as manager services located in the *Business process layer* and *Quality of service layer*. Hence, each control service performs specific functionalities of elements in the MAPE-K pattern, namely monitor (M), analyser (A), planner (P), executor (E), and data access (K). Additionally, a service for handling incoming events is also considered in this layer. The MAPE-K pattern was presented in Section 3.

Each controller service acts as a ***Filter***, that is, executing sequences of tasks (e.g., monitoring, analysing, planning, or executing), and each task is implemented and handled by a specific control service. Moreover, controller services communicate the output of their processing through a directed channel or ***Pipe***, forming thus, a

sequential work-flow. More information about control services behaviors is offered in [Garcés et al. 2018; Garcés 2019b].

In *HSH-SoS*, control services are used to coordinate work-flows of business processes services, such as the *Patient Care Manager* and the *Emergency Manager*, as depicted in Figure 2. The *Patient Care Manager* service is structured following the MAPE-K pattern, and is composed of the *Patient Sign & Symptom* that is an instance of the *Monitor* service, the *Patient Physical Status* as instance of the *Analyser* service, and the *Patient Status and Sign & Symptoms Monitor* as a *Planner* service.

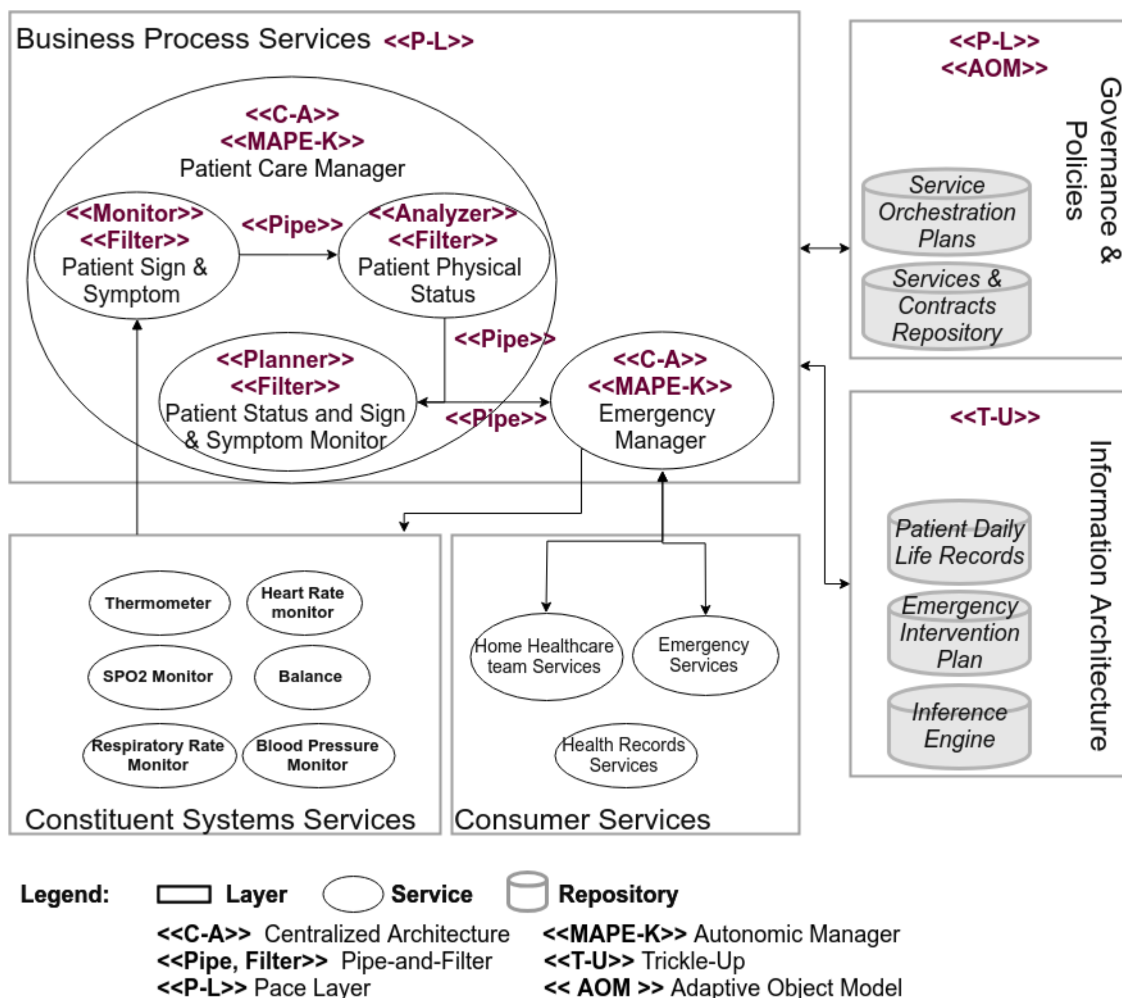

Fig. 2. Patterns involved to achieve a business process in the HSH-SoS.

Business process services layer

Services in this layer, represent macro-flow activities specified in business processes that the SoS must accomplish. These services are composed of basic services, i.e., constituent systems services and control services. Business services realize the logic to orchestrate basic services to execute a business process. In SoS, business process

services are responsible for performing high-level missions, and they are under the control of the SoS (unlike constituent services that are independently controlled).

In *HSH-SoS*, a business process is related to those work-flows required to bring healthcare services at patient's home, e.g., activities performed by the healthcare team to successfully manage patient's condition [Garcés 2018]. Example of business process services in *HSH-SoS* are the ones presented in the top of Figure 2. The *Patient Care Manager* is a composite service that executes the work-flow presented in Figure 3 and for that it coordinates the following services: (i) the *Patient Sign & Symptom* service, a *Monitor*, responsible to: collect patient's health parameters from constituent systems services, aggregate such parameters in coherent clinical information, store such information in the personal health record, and communicate them to the *Patient Physical Status* service; (ii) the *Patient Physical Status* is an *Analyser* responsible to infer the patient's physical status based on historical information stored in the health records and the current patient's parameters. For this, it uses some inference rules stored in the *Inference Engine* repository (which is located in the Information architecture layer); (iii) the *Patient Status and Sign & Symptom Monitor* that receives the inferred patient's status and defines the best intervention plan for the patient, following instructions stored in the *Emergency Intervention Plan* repository. The execution of such plan is the resposibility of the *Emergency Manager* service that communicates and coordinates the involved entities and services, e.g., home healthcare team, emergency (ambulance, firefighters), and other services located in the *Consumer* layer.

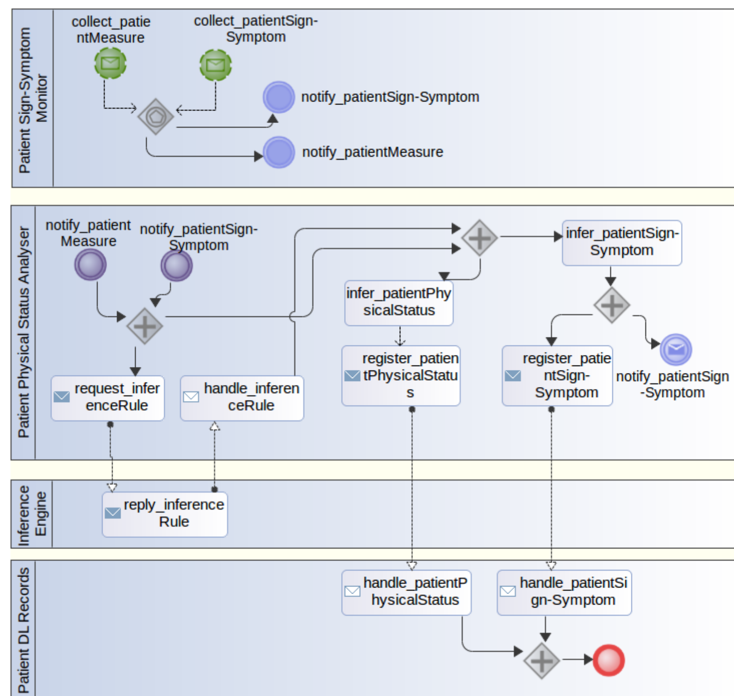

Fig. 3. Excerpt of the orchestration plan for the business process service: "Patient Care Manager". Source: [Garcés 2018].

Besides SOA, MSA, Pipes-and-Filters, and MAPE-K, the following architectural patterns are used to allow the execution of SoS business processes, and thus, the accomplishment of their goals or missions: (i) the *Patient Care Manager* follows a **centralized architecture (C-A)** to control interactions of individual services and ensure the execution of the respective business process. The *Emergency Manager* follows a similar structure;

(ii) The *Information Architecture* layer follows the **Trickle-Up (T-U)** pattern, since repositories in this layer stores aggregated data and information for controlling, monitoring, and decision making purposes, for instance the information stored in the *Patient Records* and *Inference Engine* repositories are used by the *Monitor* (*Patient Sign & Symptom*) and *Analiser* (*Patient Physical Status*) to establish patient's health situation, as described before; (iii) SoS missions are specified through service orchestration following business processes, and are stored in the *Service Orchestration Plans* and *Services & Contracts* repositories. Hence, the *Governance & Policies* layer follows the **AOM** (Adaptive-Object Model) style, since specifications ruling the SoS behavior are represented as metadata stored in repositories allocated in this layer.

Consumer services layer

This layer contains services to handle interaction with users or other stakeholders. An important type of consumer services are the System Management Services, that manage configurations and modifications of the SoS, for instance: (i) *Plans Configuration Services*, offering functionalities to modify the *Environment Configuration Plans* repository in the *Governance & Policies* layer; (ii) *services & contracts management services*, allowing the insertion, modification, or deletion of specifications of services, contracts, and protocols that are allowed in the SoS, and that are stored in the *Services & Contracts Repository*, which is located in the *Governance & Policies* layer; (iii) *Missions Management Services*, providing means to add, modify, or delete missions specifications of SoS from the *Missions Repository*, allocated in the *Governance & Policies* layer; (iv) *Quality Management Services*, allowing the management of quality plans, e.g., security, safety, reliability, and performance plans, which are stored in repositories in the in the *Governance & Policies* layer; and (v) *standards configuration services*, offering functionalities to modify, update, and add *Interoperation Standards* and *Transformation Rules*, which are located in the *Information Architecture* layer. Through services offered in the *Consumer Services* layer, it is possible to follow an evolutionary development during the SoS life cycle, ensuring their sustainability over time.

As aforementioned and depicted in Figure 2, the *Information Architecture* layer implements the **Trickle-Up** pattern, and the *Governance & Policies* layer follows the **AOM** (Adaptive-Object Model) style. Moreover, this last layer together with services in the *System Management* layer are also aligned with the **Pace Layer (P-L)** style, since they are responsible for configuring time demanding adaptations, such as the selection of the best plan for ensuring SoS missions.

System-of-Systems Service Bus (SoSSB) layer

The SoSSB is a specialization of the **ESB** and is analogous to the integration layer specified in the S3 reference architecture [Arsanjani et al. 2007]. In the context of *HSH-SoS*, the SoSSB received the name of *Health Service Bus*.

Capabilities offered by the SoSSB are mediation, routing, and transportation of services messages and events through the Communication mediators services [Garcés 2019b]. The SoSSB also allows message and protocol transformation to grant services interoperability and integration, through the use of Conversion mediators [Garcés 2019b].

For communication, collaboration, and coordination purposes, the SoSSB defines three mediators, as depicted in Figure 1: (i) a *discovery service*, responsible for detecting possible providers of services requested by services in the consumer layer (e.g., stakeholders services), and for verifying if contracts are correctly addressed by participating services; (ii) an *interceptor service*, responsible for routing messages in the SoSSB, verifying permissions of services over operations and data, and recording all operations occurred in the bus into the system logs repository; and (iii) a *container service*, which is in charge of updating services registers, such as capabilities offered, implemented standards, availability level, roles, profiles and quality metrics.

For integration and interoperation ends, SoSSB offers four conversion mediators: (i) a *filter service*, responsible to decode a message and select specific sets of information contained in a message; (ii) a *translator service*, which aims to achieve semantic interoperability, interpreting information contained in a message and translating the

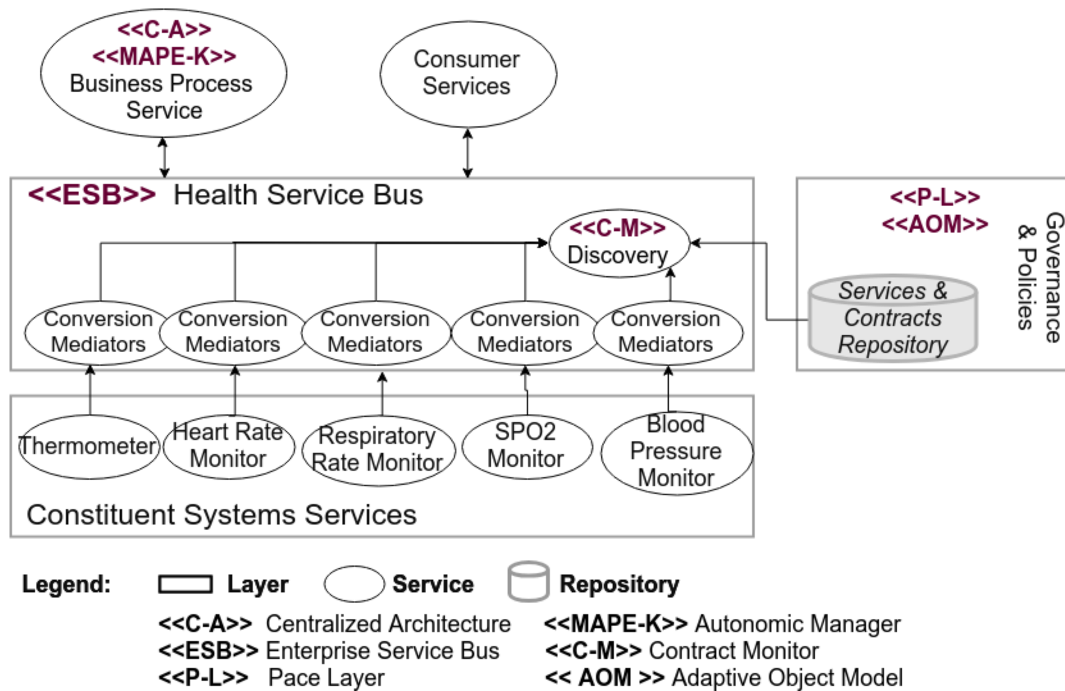

Fig. 4. Integration of constituent systems.

information to be interpreted by other participants; (iii) a *wrapper service*, which intends syntactic interoperability, encoding information in specific message format to be understood by other participants; and (iv) a *broker service* that coordinates services offered by filter, translator, and wrapper services to offer a composed service for collecting messages from a provider and transforming and routing messages to consumers in an interoperable way. More details about software mediators in SoS can be found in [Garcés et al. 2018; Garcés 2019b]. Moreover, to achieve confidentiality and integrity of data, filters, wrappers, and brokers can include operations to implement end-to-end security mechanisms, e.g., encrypting and decrypting of messages.

The SoSSB is designed to improve flexibility and evolution of SoS, since it is possible to add, remove, modify or reconfigure services during SoS operation. For instance in *HSH-SoS*, in Figure 4, the required constituent system services (e.g., thermometer, heart rate monitor, respiratory rate monitor, SPO2 monitor, and blood pressure monitor) are connected to the *Health Service Bus* through conversion mediators to ensure data interoperability. Moreover, they are verified by the *Discovery* service to check if their contracts specifications are adequate for participating in the *HSH-SoS*. Contracts specifications are previously defined and their specification is stored in the *Services & Contracts* repository. Therefore, the *Discovery* service follows the **Contract Monitor (C-M)** pattern. Once contracts are verified, the constituent systems services are authorized to compose a business process service, and hence, to participate in the execution of an SoS mission.

Moreover, ESB can be assembled to allow the integration of other SoS as illustrated in Figure 5. In our example, different *HSH-SoS* can be temporary assembled to allow higher missions, for instance, the monitoring of chronic patients at municipal level. For this, different *Health Service Buses* deployed in patient's residence can **publish** information about patients into a municipal service bus, and communicate such information to health organizations (**subscribers**) for executing healthcare prevention and promotion plans for the community. Moreover, municipal service buses can collaborate and publish relevant information into a national service bus for allowing epidemiological monitoring.

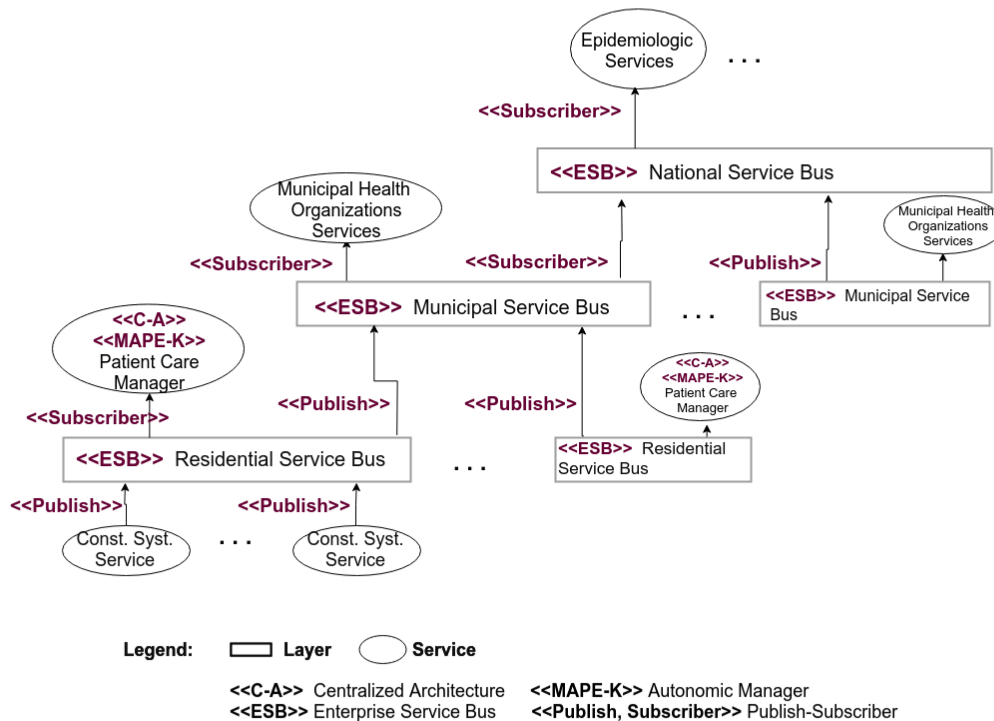

Fig. 5. Scalable architecture of SoS through coalitions of ESB

In this perspective, the coalition of distributed ESB favors scalability of SoS and dynamic evolution of such systems regarding changes of their missions.

Quality managers layer

As defined in the S3 reference architecture [Arsanjani et al. 2007], the Quality of Service (QoS) layer is responsible for capturing, monitoring, logging, and signalling non compliance with non-functional requirements that relate to the service qualities. This layer observes the basic, business process, and consumer services layers, as well as the SoSSB layer, and emits events when it detects or anticipates non compliance with reliability, performance, security, and safety requirements. Henceforth, the aim of QoS layer is to ensure the SoS and their constituents meet non-functional requirements.

In *HSH-SoS*, several *Quality Managers Services* were defined, i.e., reliability, performance, security and safety managers. The internal structure of a *Quality Manager Service* is presented in Figure 6. A quality manager is composed of *Control Services*, *monitor, analyser, planner, executor, data access*, following the **MAPE-K** pattern. These managers are responsible for monitoring quality metrics, analysing non compliance of quality specifications, planning and executing required reconfigurations to maintain the desired quality level of the SoS, as specified in the *Quality Plans* repositories located in the *Governance & Policies* layer.

In *HSH-SoS*, a quality manager was allocated for a specific quality attribute (e.g., safety, security, reliability, performance). Each manager has a **Centralized Architecture (C-A)**. Moreover, each quality manager is responsible to detect possible faults that affect the specific quality and recover the SoS through executing reconfiguration plans. Therefore, quality managers implements the **Reconfiguration Control (R-C)** architectural pattern.

The *Quality of Service* layer adds **Reflection (Refl)** properties to SoS architectures, since such systems can observe their own status at any moment and take decisions for their reconfigurations in order to maintain the

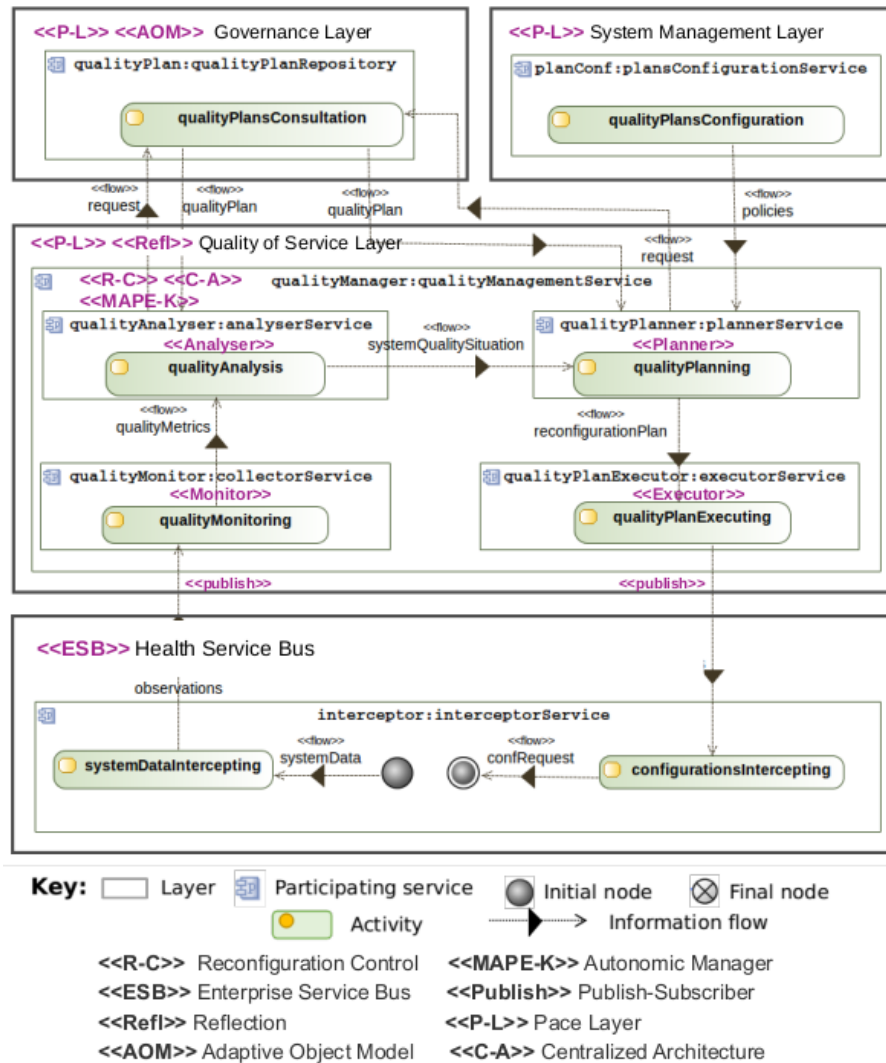

Fig. 6. Internal structure of a Quality Manager - Activities diagram. Adapted from [Garcés 2018].

desired quality levels without degrade their operations. This is also possible because the quality managers based their decisions in quality plans stored as metadata in the *Governance & Policies* layer, following the **AOM** (Adaptive-Object Model) pattern.

Additionally, reconfiguration of *HSH-SoS* is feasible due to the hierarchical structure in layers following the **Pace-Layer (P-L)** pattern: the *Health Service Bus* executes fast reconfigurations at creating, adding, removing services (e.g., constituent systems services, control services, business process services, communication mediators services and conversion mediators services), the *Quality of Service* layer taking decisions about reconfiguration plans for ensuring SoS qualities (e.g., security, safety, reliability, performance), the *System Management* layer modifying SoS plans (e.g., environment configuration plans, missions, service orchestration plans, and quality

plans), and the *Governance & Policies* layer that stores the required metadata to support reconfigurations in all levels.

Information architecture layer

This layer contains knowledge, represented as repositories, offering intelligence to SoS to achieve their missions. This layer is subdivided into the *Interoperation standards layer* and the *Business plans layer*. In the former layer, protocols for information exchange are allocated, and transformation rules between protocols are provided. These rules are used by brokers to allow transformation of messages for interoperable purposes. The latter layer includes all information required to allow the accomplishment of SoS missions, hence, it allocates the business logic that is represented as *Business Plans*. This information can be stored as meta data content, using ontologies or well-defined repositories. Moreover, rules to allow knowledge discovering that enable prediction of SoS behavior (considered as emergent behavior in the SoS context) are defined in this layer. Additionally, data transformation rules are also placed in this layer.

In *HSH-SoS*, the *Information architecture* layer was conceived following the **Trickle-Up** pattern, as depicted in Figure 2. Interoperation standards are represented following this pattern. Examples of such standards are HL7 (High Level Seven) version 2 and 3, GALEN (healthcare codes provided by the Galen Medical Group), ICD (International Classification of Diseases), LOINC (Logical Observation Identifiers Names and Codes), and SNOMED CT (Clinical Terms).

Governance and policies layer

This layer covers all aspects of managing the SoS operations life cycle. It provides guidance and policies for managing service-level agreements, including performance, security, and monitoring [Arsanjani et al. 2007]. Therefore, this layer is applicable to all other layers of an SoS. Specifically, quality plans, reconfiguration plans, repositories of missions, services and contracts specifications, and system logs are contained in this layer. Modifications of information comprised in this layer can be made using the *System management services* located in the consumer layer. Finally, as stated before, the *Governance & Policies* layer is an implementation of **Pace-Layer** and **AOM** (Adaptive-Object Model) styles.

5. DISCUSSIONS

During the application of architectural patterns and styles introduced in this paper, it was possible to identify important collaborations among them to achieve specific SoS requirements. In this section we relate these patterns and styles with the following requirements, which were listed in Table II: (R01) integration of constituent systems to enhance their collaboration; (R02) assurance of emergent behaviors; (R03) necessity of central decision making authority; (R04) separation of concerns and low coupling; (R05) resilience and adaptability of decisions; (R06) dynamic reconfigurations; (R07) missions accomplishment; (R08) clear command chain and accountability of decisions; and (R09) evolution of SoS.

5.1 Integration of constituent systems

SoS architectures must allow the integration of heterogeneous, distributed, and independent systems during the execution of SoS missions [Ingram et al. 2015]. Moreover, when a mission is finished, participating systems must be disassembled without affecting their individual operations and goals. Hence, SoS architectures must allow dynamic plugging and unplugging of constituent systems [Maier 1998].

For this purpose, the architectural styles and patterns showed in Figure 7 can be used. Constituent systems capabilities can be deployed following the **MSA**, a sub-style of **SOA** [Zimmermann 2017]. Services provided by these systems are integrated to the SoS through the SoSSB, a specialization of the **ESB** [Keen et al. 2005]. The SoSSB is responsible for mediating, routing, and conversion of data provided by all services participating in the SoS. The SoSSB can make use of the **Publish-Subscribe** pattern to route messages in an asynchronous way.

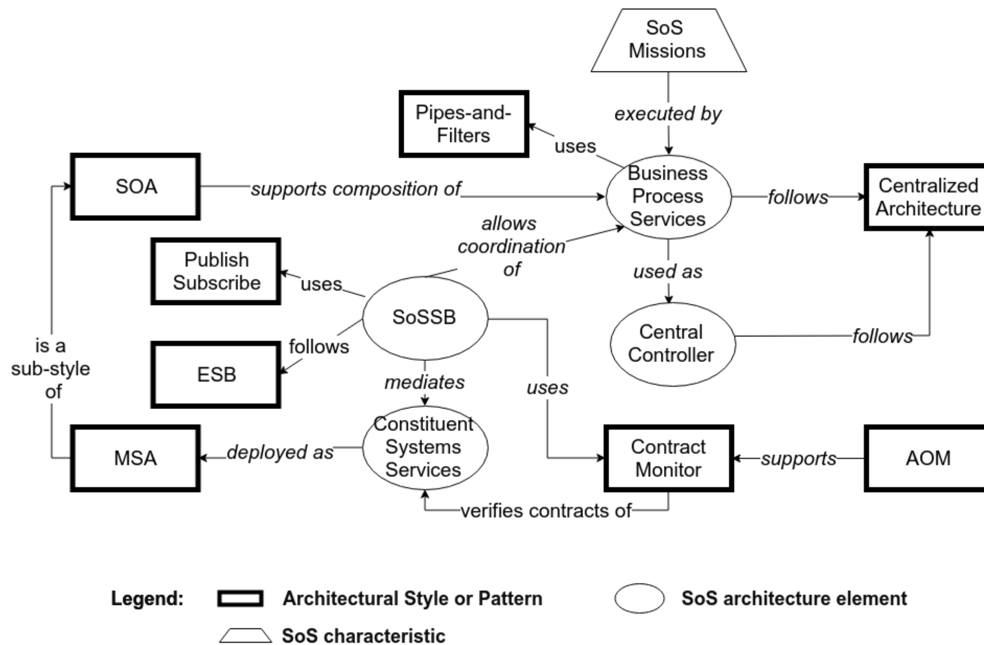

Fig. 7. Architectural styles and patterns used to allow the integration of constituent systems in an SoS.

During the integration process, the SoSSB must verify services contracts before authorizing them to participate in the SoS. For this, the SoSSB applies the **Contract Monitor** pattern. This monitor is supported by the **AOM** style that orients to store contracts specifications (metadata) in repositories to facilitate their change in runtime without affecting SoS operations.

Once the constituent systems services are integrated to the SoSSB, they are enabled to collaborate in the execution of SoS missions. Hence, these services can participate in work-flows coordinated by business process services. Work-flows can be structured following the **Pipes-and-Filters** pattern [Hohpe and Woolf 2003]. In this scenario, business process services are considered central controllers and their responsibilities are the execution of SoS missions and orchestration of constituent systems services capabilities. Therefore, each business process service follows a **Centralized Architecture**.

Following the **SOA** pattern favors the dynamic composition of business process services based on capabilities offered by constituent systems services. Moreover, by deploying capabilities of constituent systems following **MSA** it is possible to plug and unplug these systems with more flexibility without affecting the individual operation of each system [Cuesta et al. 2016].

5.2 Emergent behaviors, missions accomplishment, and evolution of SoS

SoS behaviors emerge as a consequence of constituent systems interactions. These behaviors are required to support the achievement of SoS missions since is through SoS' constituents collaborations that higher goals can be accomplished. In this context, expected emergent behaviors must be granted, as well as undesired behaviors should be prevented. Therefore, SoS architectures need to be structured to permit reasoning about emergent behaviors.

Figure 8 depicts some architectural styles and patterns that, in our experience, support the design of SoS architectures with capabilities for reasoning about their own behavior and structure. By using SOA it is possible to allocate a mission under responsibility of a dedicated business process service, which will accomplish the mission

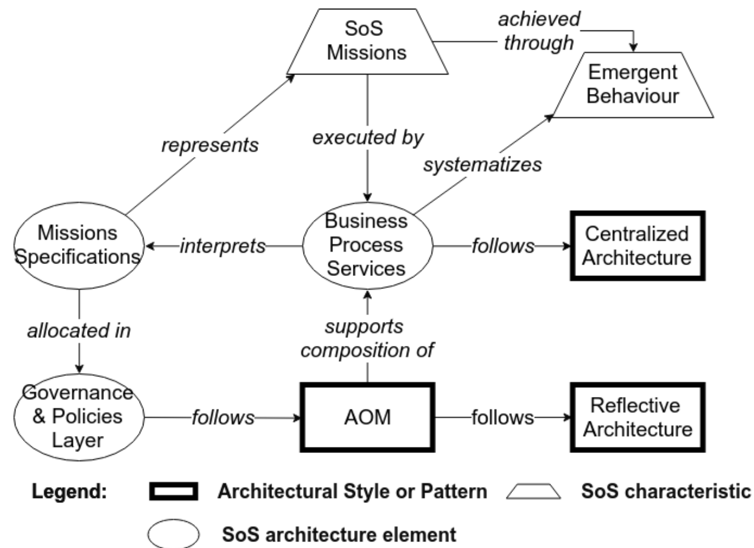

Fig. 8. Architectural styles and patterns supporting the accomplishment of missions and emergent behaviors of SoS.

through the orchestration of constituent systems services. For coordinating a specific mission accomplishment, the responsible business process follows a **Centralized Architecture**. Is through such coordination that SoS behaviors are systematized generating a new behavior as the composition of all capabilities offered by constituent system services.

Moreover, business process services can also be orchestrated by higher-level business process services to compose more complex missions. SOA allow both vertical and horizontal scalability of SoS missions, allowing the manifestation of new behaviors in a (semi) controlled way. Figure 9 illustrates SoS evolution in terms of the increment of their complexity (i.e., an SoS requires to achieve diverse missions in different levels of composition) and size (i.e., the amount of constituent systems increase as new missions are configured in an SoS).

The understanding about how to compose business processes services in runtime and in different levels can be possible by following the **AOM** (Adaptive-Object Model) style, a type of **Reflective Architecture** [Yoder 2017]. This style can be applied to define missions specifications (as metadata), containing information about capabilities of constituent systems, exchanged data, and work-flows required in a mission. Specifications are interpreted and executed by the business process service responsible for the mission.

5.3 Central authority for decision making

SoS can be classified as [Maier 1998; Nielsen et al. 2015]: (i) *Directed*, if the constituent systems are controlled by a central authority to satisfy the SoS missions; (ii) *Acknowledged*, if the constituent systems maintain independent management and missions, but collaborate with the SoS to achieve its missions; (iii) *Collaborative*, when the constituent systems are not forced to follow a central management, but voluntarily collaborate to achieve the SoS missions; and (iv) *Virtual*, if the SoS has not clear missions, hence, the SoS behaviors are highly emergent.

In some types of SoS (e.g., directed and acknowledged) a clear command chain is required [Ingram et al. 2015; Maier 1998]. Hence, it is important that SoS architectures consider a central authority to manage SoS operations and mediate constituent systems interactions.

The establishment of services following a **Centralized Architecture** in higher-layers, i.e., *System Management Services* layer and *Business Process Services* layer (See Figure 1), can be a strategy to create central authorities responsible for specific decision making.

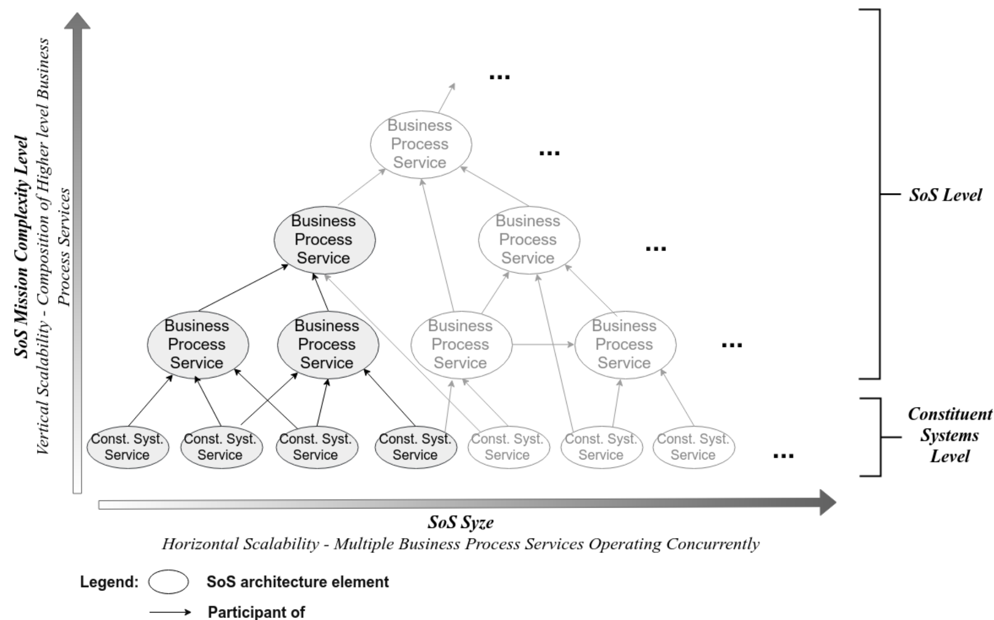

Fig. 9. Evolution of an SoS

5.4 Separation of concerns and low-coupling

SoS architectures must be able to incorporate, sometimes in runtime, independent, distributed, and heterogeneous constituent systems [Maier 1998; Ingram et al. 2015; Oquendo 2017]. Henceforth, these architectures must be designed thinking on separation of concerns and low coupling of their components.

A first strategy to achieve these requirements is to deploy constituent systems capabilities following **MSA**. These systems can be assembled/disassembled on-the-fly from the SoS without affecting neither SoS or other systems operations. As microservices, constituent systems can participate of several missions of an SoS, or even be part of different SoS at the same time. Moreover, structuring the whole SoS following the **SOA** style allow that services under the SoS control can be independent from services offered by constituent system.

Layers also promotes separation of concerns, since different layers can be defined for distinct purposes. For instance, in Figure 1 some layers were specified to organize types of (composed) services, e.g., the business process services responsible for assuring mission execution and quality manager services in charge of guaranteeing the desired quality levels in an SoS.

5.5 Resilience, adaptation, and dynamic reconfiguration

SoS architectures must offer mechanisms to identify faults (e.g., connectivity problems, unavailability of constituent systems, unauthorized operations, or undesired behaviors). When such failures are identified, the SoS must execute actions of prevention or recovering. SoS must execute dynamic reconfigurations of their behaviors or software elements. Therefore, SoS architectures must be designed to allow reconfigurations in runtime [Ingram et al. 2015].

The low-coupling and separation of concerns obtained by applying **MSA** and **SOA** can support restarting and re-integration of services in the SoS. Moreover, by adopting the tuple **ESB** and **Publish-Subscribe** to define the SoSSB (Systems-of-Systems Service Bus) can bring scalability to SoS architectures. This can allow the replication of services and ensure resources availability, dynamic connection of new services, and modifications in SoS structures as described in Section 5.2.

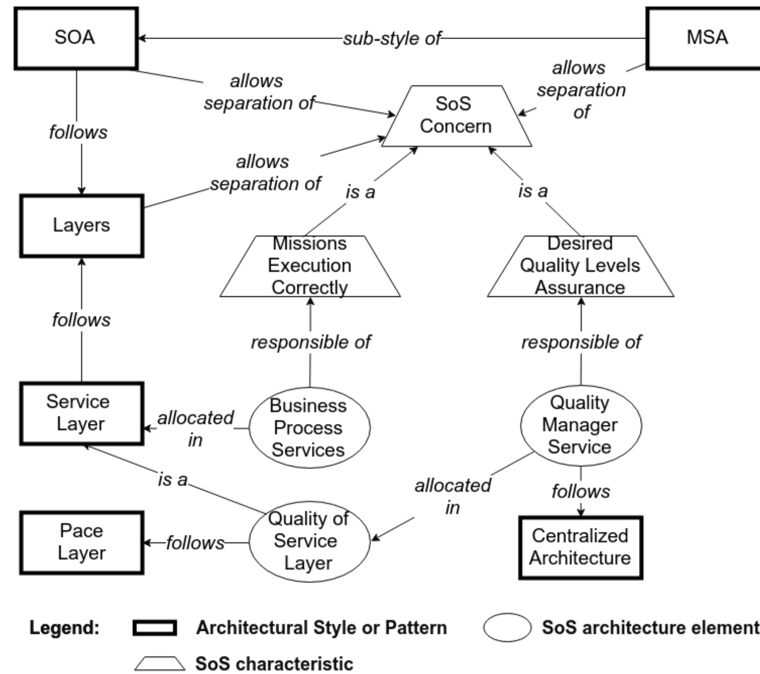

Fig. 10. Architectural styles and patterns supporting separation of concerns and low-coupling in SoS architectures.

Dynamic reconfigurations and adaptations of SoS architectures at runtime also is possible by adopting **Pace-Layer** style to specify the *Governance & Policies* layer, *System Management Services* layer, *Quality of Service* layer, and *SoSSB* layer. **Pace-Layers** can be defined to monitor services allocated in lower layers, and define and execute reconfigurations in different levels. Hence, higher layers modify SoS elements in lower layers. Additionally, hierarchies of **Pace Layers** allows the adoption of **Reflective Architecture** style, enabling SoS architectures to implement reflection capabilities, i.e., these architectures could be able to monitor themselves and define best reconfiguration policies for maintaining SoS operations under desired quality levels.

A dedicated quality manager service can be responsible for assuring quality levels of the SoS architecture (e.g., security, safety, reliability). For this, these managers can follows the tuple **Centralized Architecture** and **Reconfiguration Control Architecture** patterns. Additionally, quality manager services can organize their internal work-flow as **Pipes-and-Filters**, where filters can implement autonomic manager activities as those specified in the **MAPE-K** pattern. Reconfigurations defined by the quality manager services (e.g., reliability manager service, security manager service) can be supported by quality plans allocated in the *Governance & Policies* layer. Decoupling of managers from plans can be guided by the **AOM** (Adaptive-Object Model) style.

5.6 Trade-offs of architectural patterns and styles

It is widely known that combining different patterns and styles for architecting a software system leave to trade-offs between quality attributes. For the patterns and styles exposed in this work, we can identify some liabilities that could occur at using these solutions for SoS architectures.

Security of exchanged data between constituent systems and the SoS could be affected, mainly due to the distribution, low coupling, and separation of concerns of structures and operations in the SoS. Data require to be shared and manipulated to support SoS missions. Hence, data privacy tactics must be included in SoS architectures.

Fig. 11. Architectural styles and patterns supporting dynamic reconfiguration and adaptations of SoS architectures at runtime.

Efficiency of SoS operations could be prejudiced since business operations are distributed over different services layers. Hence, propagation of calls through layers (e.g., using a service bus as mediator) can be inefficient [Harrison et al. 2007]. Moreover, the use of service bus can imply in some communication overhead, due to the multiple messages and events, provided by constituent systems and managed by the SoS, that the bus must route. The adoption of a reflective architecture also could imply low efficiency during reconfiguration procedures, because metadata (i.e., represented as AOM - Adaptive-Object Model) often are inefficient [Harrison et al. 2007].

Reliability of SoS could be also narrowed by adopting central architecture as the style to structure business process services and quality manager services. By doing this, each central controller could be a single point of failure, then, they must be replicated for availability purposes. Moreover, the reliability of SoS operations depend of the robustness of quality protocols that are used by quality managers.

6. CONCLUSIONS AND FUTURE WORKS

In this paper we presented a list of architectural patterns (some of them defined as styles) that have been reported in literature as the most adequate for SoS architectures. The application in conjunction of these patterns was illustrated through the explanation of *HSH-SoS*' architecture. In *HSH-SoS*, patterns and styles were selected to achieve requirements that SoS architectures must address and that were presented in Table II. We believe this work is a first step to consolidate a pattern language and contribute to the quality of further SoS. However, it is required more understanding about how patterns and styles can be used to assure quality attributes requirements in SoS.

As future works we intend to investigate additional solutions (e.g., tactics, techniques, or even other styles and patterns) to overcome the identified liabilities of applying the architectural patterns and styles exposed in this work. Moreover, a deep study about how interoperability can be supported in SoS architectures, is also required. For this, important interoperability strategies [Valle et al. 2019] will be investigated in the context of SoS.

We aim to obtain more evidences for allowing the consolidation of a patterns language for SoS. For this, simulations of SoS architectures constructed based on patterns, styles, and tactics will be performed. This will allow to understand trade-offs of using such solutions in runtime for different types of SoS, i.e., directed, acknowledged, and collaborative. Moreover, we intend to develop a software platform based on these solutions to support the architecture design, and development of SoS.

ACKNOWLEDGMENTS

This work is supported by the Brazilian funding agency FAPESP (Grants: 2018/07437-9, 2017/06195-9, and 2017/22237-3). We would also like to thank Prof. Dr. Olaf Zimmermann, from the Institute für Software at HSR Hochschule für Technik Rapperswil (HSR FHO), who served as a shepherd, for providing us valuable comments that led us to a considerable improvement of the paper.

REFERENCES

2019. A Meta-Architecture, Adaptive Object Model, Reflective Systems Conference. (2019). `https://metaplop.org/papers/`

2019. Service-Oriented Architecture Patterns. (2019). `https://patterns.arcitura.com/soa-patterns`

Paolo Arcaini, Raffaela Mirandola, Elvinia Riccobene, Scandurra, and Patrizia. 2018. A DSL for MAPE Patterns Representation in Self-adapting Systems Paolo. In *European conference on software architectures (ECSA)*, Vol. LNCS 11048. Springer International Publishing, 3–19. DOI:https://doi.org/10.5381/jot.2004.3.5.c7

A. Arsanjani, L. Zhang, M. Ellis, A. Allam, and K. Channabasavaiah. 2007. S3: A Service-Oriented Reference Architecture. *IT Professional* 9, 3 (May 2007), 10–17. DOI:https://doi.org/10.1109/MITP.2007.53

Paris Avgeriou and Uwe Zdun. 2005. Architectural patterns revisited-a pattern language. (2005).

Jeffrey M. Barnes, David Garlan, and Bradley Schmerl. 2014. Evolution styles: Foundations and models for software architecture evolution. *Software and Systems Modeling* 13, 2 (2014), 649–678. DOI:https://doi.org/10.1007/s10270-012-0301-9

L Bass, P Clements, and R Kazman. 2013. *Software Architecture in Practice*. Vol. 3. Addison-Wesley Pearson Education, Boston.

Andrea Bondavalli, Andrea Ceccarelli, Paolo Lollini, Leonardo Montecchi, and Marco Mori. 2016. System-of-Systems to Support Mobile Safety Critical Applications: Open Challenges and Viable Solutions. *IEEE Systems Journal* (2016).

Everton Cavalcante, Nélio Cacho, Frederico Lopes, and Thais Batista. 2017. Challenges to the Development of Smart City Systems: A System-of-Systems View. In *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES'17)*. ACM, New York, NY, USA, 244–249. DOI:https://doi.org/10.1145/3131151.3131189

Paul Clements and Len Bass. 2010. *Relating business goals to architecturally significant requirements for software systems*. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

James O Coplien. 1998. Software design patterns: common questions and answers. *The Patterns Handbook: Techniques, Strategies, and Applications* (1998), 311–320.

C.E. Cuesta and M. P. Romay. 2010. *Elements of Self-adaptive Systems – A Decentralized Architectural Perspective*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20. DOI:https://doi.org/10.1007/978-3-642-14412-7_1

C. E. Cuesta, P. de la Fuente, and M. Barrio-Solárzano. 2001. Dynamic Coordination Architecture Through the Use of Reflection. In *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC '01)*. ACM, New York, NY, USA, 134–140. DOI:https://doi.org/10.1145/372202.372298

C. E. Cuesta, E. Navarro, E.P. Dewayne, and C. Roda. 2013. Evolution styles: using architectural knowledge as an evolution driver. *Journal of Software Evolution: Evolution and Process.* 25 (2013), 957–980.

Carlos E. Cuesta, Elena Navarro, and Uwe Zdun. 2016. Synergies of system-of-systems and microservices architectures. In *SiSoS@ECSA 2016: Inter- national Colloquium on Software-intensive Systems-of-Systems at 10th Eu- ropean Conference on Software Architecture (ECSA Volume 3)*. 1–7. DOI:https://doi.org/10.1145/3175731.3176176

C. E. de Barros Paes, V. V. G. Neto, T. Moreira, and E. Y. Nakagawa. 2018. Conceptualization of a System-of-Systems in the Defense Domain: An Experience Report in the Brazilian Scenario. *IEEE Systems Journal* (2018), 1–10. DOI:https://doi.org/10.1109/JSYST.2018.2876836

Daniel DeLaurentis. 2005. Understanding Transportation as System-of-Systems Design Problem. (01 2005).

Alstom Dersin, Pierre e Transport. 2014. IEEE Reliability Society. Technical Committee on Systems of Systems. http://rs.ieee.org/component/content/article/9/77.html. (2014).

Christoph Fehling, Frank Leymann, Ralph Retter, David Schumm, and Walter Schupeck. 2011. An architectural pattern language of cloud-based applications. In *Proceedings of the 18th Conference on Pattern Languages of Programs*. ACM, 2.

Eduardo B Fernandez and Rouyi Pan. 2001. A pattern language for security models. In *In Proc. of PLoP*, Vol. 1.

Isabella. ; Nakagawa Elisa Garcés, Lina ; Vicente. 2019a. Software Architecture for Health Care Supportive Home System to Assist Patients with Diabetes Mellitus. In *32nd IEEE CBMS International Symposium on Computer-Based Medical Systems* (June). 249–252.

L. Garcés. 2018. *A reference architecture for healthcare supportive home systems from a systems-of-systems perspective*. Ph.D Thesis. University of São Paulo, São Carlos, Brazil.

L. Garcés, S. Martínez-Fernández, Graciano V.V., and Nakagawa E.Y. 2020. Architectural Solutions for Self-Adaptive Systems. (2020).

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 (SBCARS '18)*. ACM, New York, NY, USA, 53–62. DOI:https://doi.org/10.1145/3267183.3267189

Oquendo F. Nakagawa E.Y Garcés, L. 2019b. Software Mediators as First-Class Entities of Systems-of-Systems Software Architectures. *Journal of the Brazilian Computer Society* 25, Article number 8 (2019), 1–23.

D. Garlan, J. M. Barnes, B. Schmerl, and O. Celiku. 2009. Evolution styles: Foundations and tool support for software architecture evolution. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture*. IEEE, Cambridge, UK, 131–140. DOI:https://doi.org/10.1109/WICSA.2009.5290799

N. B. Harrison, P. Avgeriou, and U. Zdun. 2007. Using Patterns to Capture Architectural Decisions. *IEEE Software* 24, 4 (2007), 38–45.

Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

IBM. 2006. *An architectural blueprint for autonomic computing. Third Edition.* Technical Report. Online.

Claire Ingram, Richard Payne, and John Fitzgerald. 2015. Architectural Modelling Patterns for Systems of Systems. In *INCOSE International Symposium*, Vol. 25. Wiley Online Library, 1177–1192.

C. Ingram, R. Payne, S. Perry, J. Holt, F. O. Hansen, and L. D. Couto. 2014. Modelling patterns for systems of systems architectures. In *2014 IEEE International Systems Conference Proceedings*. 146–153. DOI:https://doi.org/10.1109/SysCon.2014.6819249

ISO/IEC-24765. 2010. Systems and software engineering – Vocabulary. *Technical Report, International Organization for Standardization and International Electrotechnical Commission* (2010).

Mohammad Jamshidi. 2011. *System of systems engineering: innovations for the twenty-first century*. John Wiley & Sons.

Bonnie E John, Len Bass, Elspeth Golden, and Pia Stoll. 2009. A responsibility-based pattern language for usability-supporting architectural patterns. In *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 3–12.

N. Josuttis. 2007. *Soa in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., Beijing.

Roy S Kalawsky, Demetrios Joannou, Yingchun Tian, and A Fayoumi. 2013. Using architecture patterns to architect and analyze systems of systems. *Procedia Computer Science* 16 (2013), 283–292.

Rick Kazman, Claus Nielsen, and Klaus Schmid. 2013. *Understanding Patterns for System-of- Systems Integration Understanding Patterns for System-of- Systems Integration*. Technical Report CMU/SEI-2013-TR-017. Software Engineering Institute. 55 pages. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=75750

Martin Keen, Jonathan Bond, Jerry Denman, Stuart Foster, Stepan Husek, Ben Thompson, and Helen Wylie. 2005. *Patterns : Integrating Enterprise Service Buses in a Service-Oriented Architecture* (first edit ed.). IBM International Technical Support Organization. 352 pages.

James Lewis and Martin Fowler. 2014. Microservices. A definition of this new architectural term. (2014). https://martinfowler.com/articles/microservices.html

P. Maes. 1987. Concepts and Experiments in Computational Reflection. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '87)*. ACM, New York, NY, USA, 147–155. DOI:https://doi.org/10.1145/38765.38821

Mark W Maier. 1998. Architecting principles for systems-of-systems. *Systems Engineering* 1, 4 (1998), 267–284.

Gastón Márquez, Mónica M. Villegas, and Hernán Astudillo. 2018. A Pattern Language for Scalable Microservices-based Systems. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings (ECSA '18)*. ACM, New York, NY, USA, Article 24, 7 pages. DOI:https://doi.org/10.1145/3241403.3241429

Saurabh Mittal and Larry Rainey. 2015. Harnessing emergence: The control and design of emergent behavior in system of systems engineering. In *Proceedings of the Conference on Summer Computer Simulation*. Society for Computer Simulation International, 1–10.

H. A. Muller, H. M. Kienle, and U. Stege. 2009. Autonomic Computing Now You See It, Now You Don't. In *Software Engineering*, A. Lucia and F. Ferruci (Eds.). Lecture Notes in Computer Sciences, Vol. 5413. Springer Berlin Heidelberg, Berlin, 32–54.

Craig Nichols and Rick Dove. 2011. 7.1. 3 Architectural Patterns for Self-Organizing Systems-of-Systems. In *INCOSE International Symposium*, Vol. 21. Wiley Online Library, 856–867.

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 Comput. Surv.* 48, 2, Article 18 (Sept. 2015), 41 pages.

F. Oquendo. 2017. Software architecture of self-organizing systems-of-systems for the Internet-of-Things with SosADL. In *2017 12th System of Systems Engineering Conference (SoSE)*. 1–6. DOI:https://doi.org/10.1109/SYSOSE.2017.7994959

Carlos Eduardo de B Paes, Valdemar Vicente Graciano Neto, Flávio Oquendo, Elisa Yumi Nakagawa, and Sao Carlos-SP-Brazil. 2016. Experience Report and Challenges for Systems-of-Systems Engineering: A Real Case in the Brazilian Defense Domain. In *Proceedings of the 10th CBSOFT Workshop on Distributed Development of Software, Software Ecosystems, and Systems-of-Systems (WDES)*.

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

M. P. Romay, C. E. Cuesta, and L. Fernández-Sanz. 2013. O Self-Adaptation in Systems-of-Systems. In *Proceedings of the 1st International Workshop on Software Engineering for Systems-of-Systems*. IEEE, Montpellier, France, 29–34.

K. J. Rothenhaus, J. B. Michael, and Shing. M. T. 2009. Architectural Patterns and Auto-Fusion Process for Automated Multisensor Fusion in SOA System-of-Systems. *IEEE SYSTEMS JOURNAL* 3, 3 (2009), 304–316.

Mazeiar Salehie and Ladan Tahvildari. 2009. Self-adaptive Software: Landscape and Research Challenges. *ACM Trans. Auton. Adapt. Syst.* 4, 2, Article 14 (May 2009), 42 pages. DOI:https://doi.org/10.1145/1516533.1516538

Garcés L. Allian A. Nakagawa E.Y. Sena, B. 2018. Investigating the Applicability of Architectural Patterns in Big Data Systems. In *Proceedings of the 25th Conference on Pattern Languages of Programs (PLoP '18)*. 1–14.

The Hillside Group. 2019. Patterns. (2019). https://hillside.net/patterns/

Pedro Henrique Dias Valle, Lina Garcés, and Elisa Yumi Nakagawa. 2019. A Typology of Architectural Strategies for Interoperability. In *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '19)*. Association for Computing Machinery, New York, NY, USA, 3–12. DOI:https://doi.org/10.1145/3357141.3357144

Wil van der Aalst and Arthur ter Hofstede. 2017. Workflow Patterns. (2017). http://www.workflowpatterns.com

Paulo E. Veríssimo. 2006. Travelling Through Wormholes: A New Look at Distributed Systems Models. *SIGACT News* 37, 1 (March 2006), 66–81. DOI:https://doi.org/10.1145/1122480.1122497

D. Weyns and T. Ahmad. 2013. Claims and Evidence for Architecture-Based Self-adaptation: A Systematic Literature Review. In *7th European conference on Software Architecture*. Springer Berlin Heidelberg, Berlin Heidelberg, 249–265.

D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. Goschka. 2013. On Patterns for Decentralized Control in Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems II*, R. de Lemos, H. Giese, H.A. Muller, and M. Shaw (Eds.). Lecture Notes in Computer Sciences, Vol. 7475. Springer Berlin Heidelberg, Berlin Heidelberg, 76–107.

Joseph Yoder. 2017. Adaptive Object Model. (2017). https://adaptiveobjectmodel.com/

Joseph W. Yoder and Ralph Johnson. 2002. *The Adaptive Object-Model Architectural Style*. Springer US, Boston, MA, 3–27. DOI:https://doi.org/10.1007/978-0-387-35607-5_1

Olaf Zimmermann. 2017. Microservices tenets. *Computer Science - Research and Development* 32, 3 (01 Jul 2017), 301–310. DOI:https://doi.org/10.1007/s00450-016-0337-0

Tommaso Zoppi, Andrea Ceccarelli, and Andrea Bondavalli. 2017. Exploring Anomaly Detection in Systems of Systems. In *The Symposium on Applied Computing (SAC '17)*. ACM, New York, NY, USA, 1139–1146.

Received May 2019; revised July 2019; accepted September 2019

PLoP'19, OCTOBER 7-10, Ottawa, Ontario Canada. Copyright 2019 is held by the author(s). HILLSIDE 978-1-941652-14-5