A Service-Based Architecture for Virtual and Collaborative System of Systems

Rosana T. Vaccare Braga, Iohan Gonçalves Vargas and Thiago Gottardi

[1]Instituto de Ciências Matemáticas e Computação – Universidade de São Paulo
Av. do Trabalhador São-Carlense, 400 – 13560-970 – São Carlos – São Paulo

rtvb@icmc.usp.br,iohan@usp.br,gottardi@icmc.usp.br

Abstract. *System of Systems (SoS) provide means of integrating independent systems aiming at achieving goals that could not be accomplished if these systems were executed isolated. Among the several types of SoS identified in the literature, virtual SoS still pose several challenges to be built. By not depending on a central coordinator, virtual SoS are the most robust type of SoS. On the other hand, collaborative SoS presents the advantage of having a well established emergent behavior design, which we intend to combine with the robustness of virtual SoS. Therefore, in this paper we propose a reference architecture to build SoS of a mixed type between virtual and collaborative. We present a prototype we built as a proof of concept of the proposed architecture that employs services for communication among constituents. We also describe how the architecture has been applied for a SoS in the software engineering tool domain. As conclusions, the architecture was successfully applied to implement a mixed-type SoS since its inception. In addition, the developed prototype can be reused for other SoS instances in different domains.*

1. Introduction

A System of Systems (SoS) is a collection of independent systems and their interrelationships to produce a system that presents additional behavior that is greater than the sum of the parts [Boardman and Sauser, 2006]. They have great potential of obtaining advantages of several existing systems to accomplish emergent behavior. The challenge is to know how to integrate the systems to become constituent systems (CS), considering their relationships and participation in the whole SoS.

Several types of SoS have been identified in the literature. For example, Maier [1998] classifies them as directed (a centrally managed SoS built and managed to fulfill specific purposes), collaborative (a SoS where CS interact to fulfill central purposes guided by central players), and virtual (a SoS with emergent behavior that has not a central management authority).

Maier classified the World Wide Web as a well known Virtual SoS [Maier, 1998]. By not having a central management authority, this category of SoS allows any interested party to create a new CS and join the SoS. At the same time that we want systems to be independent, we need to ensure that the SoS goals are met, i.e., it is important to plan specific emergent behaviors expected to be fulfilled, similarly to a Collaborative SoS design. We cannot discern and distinguish exactly how the system functionality is achieved in a virtual SoS. Nielsen et al. [2015] consider a large-scale complex IT system as a virtual system, justified by the fact that they consist of systems that work together, maybe with mutual interest, but are owned by individual organizations that may be competing. It is important to highlight that we have not found any SoS architectures proposing solutions that satisfy the needs of such systems.

Therefore, we intend to identify how the gap between Virtual and Collaborative SoS designs could be interpolated with a Mixed-Type SoS.

Therefore, this paper has the following goals: (i) to identify a set of requirements for the Mixed-Type SoS, (ii) to present a service-based architecture, which we named MV-SoSA, that serves as a basis when composing new Mixed-Type SoS, (iii) to present an implementation of MV-SoSA that we refer as MV-SoS-Prototype, and (iv) to describe a concrete example of a Mixed-Type SoS employing MV-SoS-Prototype as part of a case study.

The remainder of the paper is organized as follows. In Section 2 we summarize the concepts of SoS and Services. In Section 3 we present the architecture proposal: first we identify the requirements for a Mixed-type SoS (Section 3.1), followed by an overview of MV-SoSA (Section 3.2). In Section 4 we present MV-SoS-Prototype. In Section 5 we describe how MV-SoSA is instantiated to develop a SoS in the software reuse domain. Related work is discussed in Section 6. Finally, the conclusions, as well as future work, are discussed in Section 7.

2. Background

Systems of Systems (SoS) allow the aggregation of independent systems, together with their relationships, to form a whole that is greater than the sum of the parts [Maier, 1998]. Thus, it is possible to achieve complex goals that would not be easily achieved individually by the CS. Emergent behaviors can also arise at any time and should be easy to model.

Maier (1998) classifies SoS based on managerial characteristics, suggesting three types of SoS: Directed, where the integrated SoS is built and managed to fulfill specific purposes and CS maintain an ability to operate independently, but their normal operational mode is subordinated to the central managed purpose; Collaborative, where the CS interact voluntarily to fulfill agreed upon central purposes and central players collectively decide how to provide or deny services; and Virtual, where there is neither a central management authority nor a centrally agreed upon purpose for the SoS, but behavior emerges naturally from the composition.

Maier [1998] has defined five important characteristics of SoS: operational and managerial independence of CS, emergent behavior, geographical distribution, and evolutionary development processes. These characteristics are relevant in the context of this work, as the proposed architecture will allow several independent systems to work together forming a SoS, whilst they can freely enter or leave the composition without having their independence affected at all.

Service-orientation has been presented as a potential mechanism to allow the construction of SoS [Madni and Sievers, 2014]. A service has a public interface that is available and inter-operable, and it can dynamically connect to other services [Mahmoud, 2005].

The use of services is appropriate in distributed software development environments, in which the integration among different tools is required, preferably in a transparent way, in particular for SoS. Indeed, in a systematic review published recently [Vargas et al., 2016], it was verified that a considerable number of publications, approximately 51.72%, have explored the use of services for integration of CS in the SoS context.

3. Architecture Proposal

In this exploratory study, we propose a generic architecture for building a Mixed-Type SoS. We have initially considered SoS in the information systems domain, where each CS can make

available, through services, pieces of information that are potentially of interest to achieve the SoS emergent behavior.

By its own definition, virtual SoS might not be as reliable as required by organizations, because they could enable behaviors not predicted by the CS owners. In the same manner, organizations might also require expected behaviors that cannot be covered by a virtual SoS implementation. On the other hand, the organizations may also require that the SoS must be designed with pre-planned emergent behaviors, unlike virtual SoS, since they cannot be developed to guarantee the emergent behavior. We study a solution to this problem by introducing some characteristics extracted from the collaborative type, more specifically, we suggest that a predictable set of trusted systems is kept by each system willing to participate in the SoS. The implementation of this suggestion maintains the virtual characteristic of the SoS, because: i) CS are not obliged to provide predicted services; ii) CS do not have a coordinator or a previously planned collaboration sequence; and iii) CS have total autonomy to perform their tasks, without depending on a pre-specified choreography.

3.1. Requirements for the Mixed-Type SoS

We have established a set of requirements for the Mixed-Type SoS, listed on Table 1 and explained below. They resulted from the analysis of related works on Collaborative SoS [Mahmood and Montagna, 2012; Black and Fletcher, 2006], and on Virtual SoS [Ramos, 2014], which allowed us to provide a simple design for our Mixed-type SoS. It is important to notice that the classification of SoS is not orthogonal: according to Nielsen et al. [2015], autonomy and ownership have to be maintained, but the SoS type is not expected to be uniform within a SoS, but local subsystems of different types might arise.

#	Requirement
R_1	the proposed System of Systems should be composed by a dynamic set of CS, similarly to a virtual SoS
R_2	the CS must be independent, both from the operational and managerial perspectives
R_3	the CS can enter or exit the SoS at any time, without compromising the working of the independent systems
R_4	each CS should keep a list of trustworthy CS
R_5	each CS can provide services to both users and other trustworthy CS
R_6	components and list of trusted CS must be dynamically changeable
R_7	unique identifiers could be used to identify objects in the SoS
R_8	each CS is responsible to know its managed objects, so the authorization concern should be implemented independently from authentication
R_9	a CS should be capable of performing cascading queries on other trusted CS that also provide the requested functionality and provide a single merged response to the user.

Table 1. Requirements for the Mixed-type SoS

R_1 contains the basic requirement of our work, which is to propose a SoS composed by a dynamic set of CS similarly to a Virtual SoS.

R_2 is important to maintain the fundamental characteristics of SoS as defined in the literature. Basically, it states that each CS should have its own set of capabilities that are kept working even when the system is not part of the SoS. Also, the system management is done independently from the fact that the system is occasionally part of the SoS.

R_3 ensures that CS are free to enter or to exit the SoS according to their own rules, i.e., a CS can manifest its interest in making part of the SoS, but can also decide to leave if this is

the most appropriate action in a certain moment.

R_4 guarantees that only trustworthy systems are allowed to enter the SoS. A mechanism to catalog these systems and check if a system belongs to this catalog when trying to be part of a SoS should be provided.

R_5 depicts the duties of CS in relation to the SoS, i.e., as part of the SoS a CS can provide services that can be useful to other systems or to the SoS as whole, contributing to the emergent behavior. Eventually, a CS could be only a consumer of services provided by other CS.

R_6 is related to the dynamic configuration of CS. It is essential that new systems can be included as members of the SoS at runtime. Analogously, any CS can be removed from the list at anytime.

R_7 is important to ease the retrieval of objects throughout the SoS. There should be a mechanism through which we can identify, based only on an object unique identifier, to what CS it belongs, the identification of the CS it belongs to and its type (class).

R_8 is related to access control. Authentication, i.e. ensuring that the logged users are who they claim to be, could be managed separately by an independent CS or service. However, a CS is responsible for knowing its managed objects and mapping them to groups of users who have read/write access rights. The authorization to perform specific functions or to have access to managed objects could be dealt with by a CS or service that receives the managed objects list as input. In this case, a token could be provided as a way of ensuring that the user has been authenticated/authorized to access the system within a predetermined time period.

Finally, R_9 is related to the ability of CS to perform cascading queries. A single CS should provide the option to the user to perform cascading queries for a single query call. In this manner, the called CS needs to know other trusted CS that also provide the requested functionality of the query, which should be then called recursively, accumulating responses of the requested query into a single response to the user.

3.2. MV-SoSA Overview

Based on the specified requirements, we propose an architecture, named MV-SoSA (Mixed Virtual-System of System Architecture), that serves as a basis when composing new Mixed-type Systems of Systems. Next, we provide more details about MV-SoSA, using two different views (a conceptual model and a layer view), as well as an explanation of how cascading queries work.

3.2.1. Conceptual Model

A conceptual model for MV-SoSA was created by considering the requirements listed in Section 3.1. The diagram for this model is shown in Figure 1. *UID*, defined following R_7 and R_8, represents the unique identifier for objects of the SoS. It is inherited by *ManagedObject*, which was defined according to R_8. *ManagedObject* is used to classify objects that are stored by CS and may be transferred among other CS.

User was defined following R_5 and represents external users of the CS. *SoS* was defined according to R_1. Since the SoS is dynamic, similarly to a virtual SoS, there would be no concrete implementation of this class, because there is no entity that owns or manages the SoS completely. The class was simply created in order to represent that the SoS is composed by

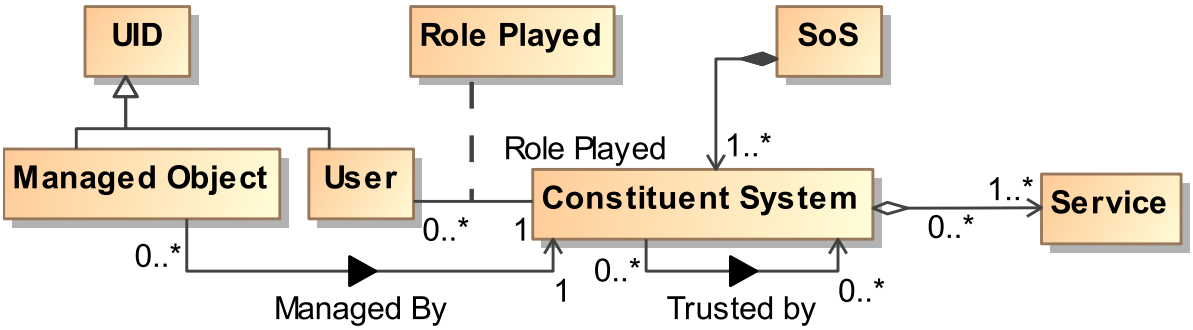

Figure 1. MV-SoSA Conceptual Model.

CS, which, in turn, is represented by *ConstituentSystem*, defined according to R_2, R_3, R_4, R_5, R_6 and R_8. Services are represented by class *Service* (R_5). These services are provided by the CS itself, to its external users or to other trusted CS.

3.2.2. Layer View

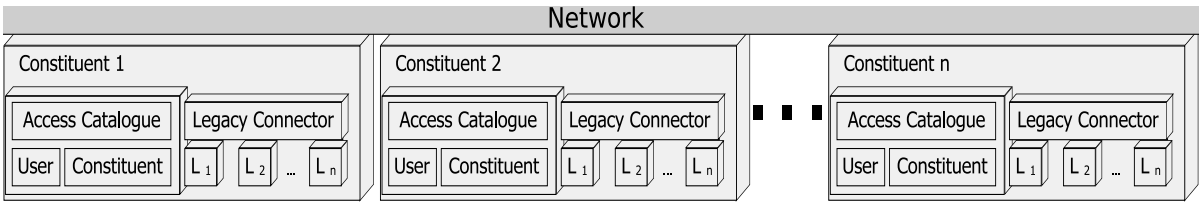

Figure 2. MV-SoSA Layer view.

The layers of the MV-SoSA architecture are shown in Figure 2. On the top of the figure, the network infrastructure is represented as a large single bus. The networking infrastructure itself is not the main objective of this architecture, it just needs to be the same throughout every CS. This way, it could be replaced by any other mechanism that enables communication of the CS.

From the perspective of external users, the CS is a single black box that provides a user profile and permission catalog, as well as other features provided as services. This happens because every call is carried out by a proxy that delegates the call to internal service providers. This proxy is also capable of calling external trusted CS in order to answer cascading queries.

The internal service providers are divided into “Access Catalogue” and “Legacy Connector”. “Access Catalogue” contains the records of trusted external entities, which are either users (including their different roles) or other CS that are entitled to access this CS. The “Legacy Connector” layer aggregates connectors which wrap the actual services targeted at the end-users of the CS. This aggregation strategy allows dynamic connector loading. Then, each CS has a variable set of features implemented by legacy service connectors, which are represented by $L_{1..N}$. Each of these connectors should be implemented to provide a compatible interface to the SoS CS to be able to call the legacy service.

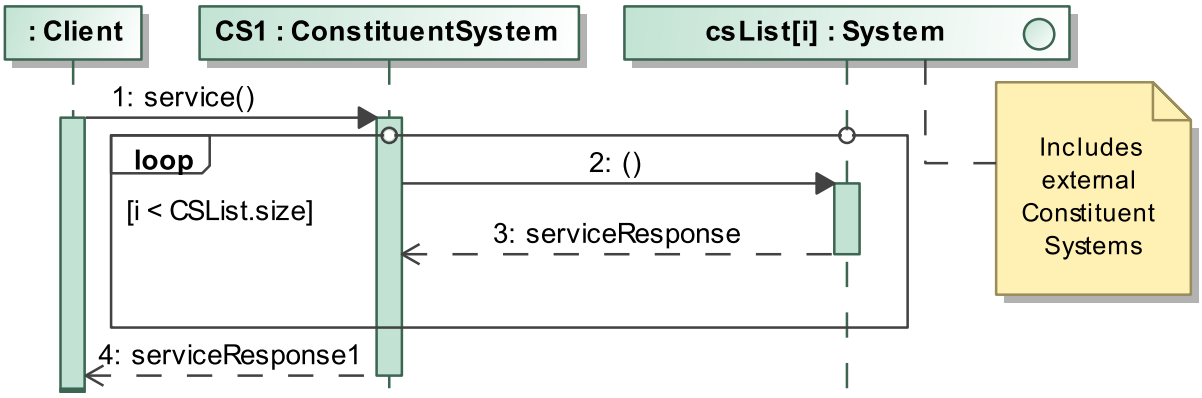

Figure 3. Sequence Diagram of Proxy Call Delegation to System Interface

3.2.3. Proxy Behavior

In order to explain the proxy behavior required to enable cascading queries, as well as dynamic feature emergence, a simplistic perspective is considered, as illustrated in Figure 3.

Consider that the Mixed-Type SoS has been successfully composed after the CS are initialized with a set of trustworthy CS. In this case, the CS ("CS1") may forward any call from external users (referred as "Client") to an internal connector capable of delegating to the requested service provider ("service"), as well as forwarding the call to any other trusted CS. The CS are actually capable of calling more than one internal connector, as well as more than one trusted CS.

The proxy behavior is similar to the proxy pattern by the Gang of Four [Gamma et al., 1995], however, in this case, the systems are dynamic. Therefore, the proxy code does not have the method signature to be called, requiring the dynamic decoding of each client request by interpreting the signature at run-time and then delegate the call to the actual legacy system and/or trusted CS. After the delegated legacy systems and trusted CS reply the request, the proxy must then aggregate the responses into a single message that is returned to the client.

4. Prototype Specification

In this section we present a prototype that follows all the requirements of MV-SoSA, referred in this paper as "MV-SoS-Prototype". This prototype is shown to demonstrate that the mixed-type is possible to be implemented, though this prototype might not cover specific kinds of SoS that are designed to integrate legacy systems. For legacy system integration, it would still require to implement connectors to attach these systems to the new architecture.

In our example, the architecture was designed as an SoS system since its inception, where every CS has basic behavior for integrating services and communicating with other CS that belong to a list of trustworthy systems. In this manner, each CS is modular and is composed by legacy connectors that provide variable sets of services.

In order to encapsulate the common behavior among CS, in our particular implementation, we consider that all CS have a kernel component. This kernel is the basic element of a CS and manages the associated services and CS administrators. The basic responsibility of the kernel is to allow communication among CS. This communication could happen among the service providers that represent actual applications for the end users or other CS. The kernel also acts as a proxy to allow external clients to access both its managed service modules, as

well as with compatible services provided via other CS kernels, which boasts the emergent behavior capability.

4.1. MV-SoS-Prototype Kernel Architecture

The implemented prototype was based on a micro-kernel design with reliability as a primary non-functional feature [Tanenbaum and Bos, 2015]. An overview of the prototype class diagram is shown in Figure 4, where there are two main packages: *common* and *kernel*.

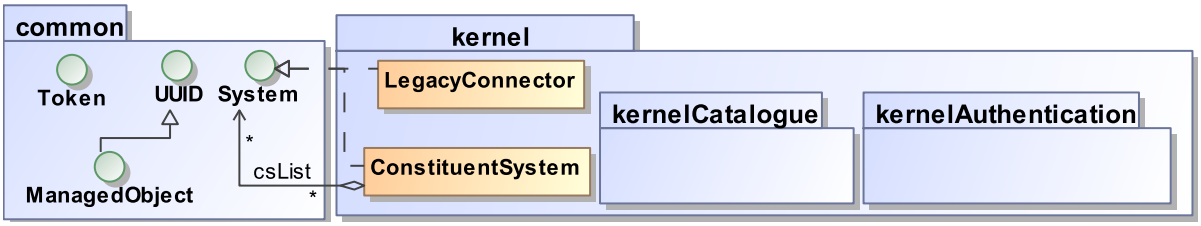

Figure 4. MV-SoS-Prototype Design

The *common* package is composed by interfaces that define data-types used throughout the architecture elements, hence the name *common*. This package is required by every element of MV-SoS-Prototype, including the kernel itself. The *kernel* package represents the actual kernel of the prototype. It was designed to cope with the most minimal and fundamental features of the system.

4.2. Common Types

Since MV-SoS-Prototype allows different CS to manage their objects, it was important to define a Universally Unique Identifier (UUID). This identifier must be hierarchical, allowing each CS to generate valid identifiers without requiring calls to other CS, which can be completely oblivious of object construction.

These identifiers are specified using the *UUID* interface, which defines how the hierarchical parts of the *UUID* are separated, as well as the minimal parts to identify a running instance of kernel and CS.

The CS based on MV-SoS-Prototype also manage objects that are addressed by these UUIDs. Therefore, the interface *ManagedObject* is employed to specify whether an object is managed, transferred and persisted by a specific CS or legacy system.

Since we have identified that the SoS requirements involve users, which could be a human user or another computer system, the *User* interface has been also defined within the *common* package. Our architecture was also planned to allow unified, yet distributed registration for users. Therefore, *User* is also treated as an universal identified object stored into the CS and shared throughout the SoS.

The *System* is an interface to define the signature of required methods needed to implement a CS based on MV-SoSA. Finally, *Token* defines how access tokens that are transferred among CS should be implemented.

4.3. Constituent Kernel

In our architecture, the Kernel of a Constituent System is designed within the *kernel* package shown in Figure 4. The kernel is the basic element of a MV-SoS-Prototype instance, which was inspired by a micro-kernel design. Therefore, it contains the execution entry point that coordinates services required to implement its functionality.

The execution entry point of the CS is defined within *ConstituentSystem* interface, which also extends the *System* of the common package. The actual implementation of the main method is defined in the *ConstituentSystem* class.

The authentication behavior is designed within the *kernelAuthentication* package. Since the kernel has an architecture inspired by micro-kernel design, authentication is defined as an external service. which also increases modularity capability. In this manner, the actual user data is not stored inside the kernel, which can delegate to more than one trusted authorization service to identify users.

Following the design of *authentication* package, the catalog is implemented as a service within *kernelCatalogue* package. Since kernels and legacy connectors are all defined as extensions of *System*, it is possible to store the legacy connectors managed by the current kernel, as well as defining trusted foreign kernels. By this established design, it is also possible to define components that serve more than one CS, increasing flexibility.

5. Evaluation - SoS Application

MV-SoSA has been applied to build a SoS for integrating instances of a reuse tool we have developed at ICMC. In this paper, this tool is simply referenced as RT. It adopts the Reusable Assets Specification (RAS) [OMG, 2005], from OMG, to store the assets in a repository, so any type of reusable asset is allowed, as for example requirements, analysis models, design models, source code, test cases, and processes. It is also based on SOA, i.e., clients can consume the provided services to manage reusable assets, which can be stored in a cloud.

When we consider several independent instances of RT running at different organizations, it is clear that each instance is benefiting from the reuse of its locally stored assets. The idea is to integrate these instances by building a SoS to offer emergent behavior that could not be obtained by independent instances. Notice that the other SoS characteristics described in Section 2, are also present in this SoS, i.e., each instance has its own management and operation; they are distributed geographically; and they could be evolved separately by different organizations, which could adapt the instance to their particular needs.

RT shares the prototype kernel that we have presented in Section 4. Therefore, RT CS can be also CS of our MV-SoS-Prototype. In order to instantiate RT it was necessary to create service connectors based on the "System" interface and to publish the service-oriented interface definitions. Then, the new CS is added to lists of other CS (part of "csList" of Figure 4). It is required to have two or more CS that provide RT Services. According to the current state of our implementation, the CS administrators must set manually the list of trusted systems. Following that possibility, if these CS trust each other, users using any of the CS are then able to access data from every instance, allowing these users to retrieve a single merged result for a query performed on any of these CS, which would not be straightforward if each CS is accessed independently. Other types of emergent behavior can be obtained as well, if necessary. For example, if the code of an asset is stored in a configuration management system, another possible emergent behavior of RT would be to warn other CS about changes and conflicts to asset users.

6. Related Work

Kazman, *et al.* have described a set of design patterns for SoS. They have described patterns such as Facade and Broker, which are similar to the Proxy Pattern, however, they have not shown this pattern itself [Kazman et al., 2013]. In this paper, we have presented an actual implementation of Proxy Design Pattern for a SoS.

Mahmood and Montagna have devised a generic framework for defining a SoS architecture suited for specific applications. They have also presented a Production SoS design by using the proposed framework [Mahmood and Montagna, 2012]. In this paper, our goal was to present a Mixed-type SoS architecture and implementation instead.

Among the definition of complex multi-tier systems that involve the definition of connected subsystems, there is the implementation of a complex robotics collaborative and distributed system [Black and Fletcher, 2006] and a transportation distributed system [Padmadas et al., 2010]. These authors describe several advantages linked to the modularity of the components employed into the design of these systems.

Vierhauser, *et al.* have proposed a software system to monitor the execution of SoS [Vierhauser et al., 2014]. In our SoS implementation, this behavior is also carried by the CS kernel, which monitors the provided subsystem services as well as trusted services provided by other CS.

Ramos devised an approach for SoS development based on statecharts and publish-subscribe mechanisms [Ramos, 2014]. His work was also based on Virtual SoS taxonomy, however, unlike our proposal, it is only suited to local area networks because it depends on event broadcast that is not suitable for a wide area network, e.g. the Internet.

7. Conclusions and Future Work

In this paper, we have established a type of SoS that fills the gap between virtual and collaborative SoS, which we referred simply as Mixed-type SoS. A set of requirements for it was presented, which characterizes a foundation for future SoS research. These requirements were used as basis to design an architecture for Mixed-type SoS, named MV-SoSA. It was designed to be simple and modular to be adapted to several application domains. To the best of our knowledge, Virtual SoS research lack exploratory studies.

Subsequently, MV-SoSA was instantiated to create an actual functional prototype, employed during an exploratory study for an application in the context of software reuse. Our implementation was successfully applied to compose independent systems fulfilling the expected requirements, e.g. emergent behavior, trusting mechanism, authentication, dynamic component loading and modularity.

We expect that our work contributes to efforts on evidencing what could or not be planned for the design of an actual Virtual SoS. Additionally, it could serve as basis to exploratory studies that can try to solve the several challenges presented by Virtual SoS.

As future work, we intend to instantiate MV-SoSA to integrate a broader range of applications pertaining to different domains, as well as conducting experiments on their usability, ease of integration and security. Among these applications, we can cite the evolution of the tool described within Section 5 to be integrated into a project for reuse support [Braga et al., 2016].

8. Acknowledgments

Our thanks to Fundação de Amparo a Pesquisa do Estado de São Paulo (FAPESP, process number 2016/05129-0) and to Coordenaçao de Aperfeiçoamento de Pessoal de Nivel Superior (CAPES) for financial support.

References

Black, R. and Fletcher, M. (2006). Simplified robotics avionics system: A integrated modular architecture applied across a group of robotic elements. In *25th Digital Avionics Systems Conference, 2006 IEEE/AIAA*, pages 1–12.

Boardman, J. and Sauser, B. (2006). System of systems - the meaning of of. In *IEEE/SMC International Conference on System of Systems Engineering*, page 6 pp.

Braga, R. T. V., Feloni, D., Pacini, K., Filho, D. S., and Gottardi, T. (2016). *AIRES: An Architecture to Improve Software Reuse*, pages 231–246. LNCS–9679. Springer International Publishing, Cham.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Kazman, R., Schmid, K., Nielsen, C., and Klein, J. (2013). Understanding patterns for system of systems integration. In *8th International Conference on System of Systems Engineering (SoSE)*, pages 141–146.

Madni, A. M. and Sievers, M. (2014). System of systems integration: Key considerations and challenges. *Systems Engineering*, 17(3):330–347.

Mahmood, A. and Montagna, F. (2012). System of systems architecture framework (SoSAF) for production industries. In *7th International Conference on System of Systems Engineering (SoSE)*, pages 543–548.

Mahmoud, H. (2005). Service-oriented architecture (SOA) and web services: The road to enterprise application integration (EAI).

Maier, M. (1998). *Architecting Principles for "Systems-of-Systems", Systems Engineering*.

Nielsen, C., Larsen, P., Woodcock, J., and Peleska, J. (2015). Systems of systems engineering: Basic concepts, model-based techniques, and research directions. *ACM Computing Surveys*, 48(2):18:1–18:41.

OMG (2005). Reusable asset specification. Available at `http://www.omg.org/spec/RAS/2.2/`.

Padmadas, M., Nallaperumal, K., Mualidharan, V., and Ravikumar, P. (2010). A deployable architecture of intelligent transportation system – a developing country perspective. In *IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, pages 1–6.

Ramos, M. A. (2014). *Bridging software engineering gaps towards system of systems development*. PhD thesis, University of Sao Paulo, Brazil.

Tanenbaum, A. and Bos, H. (2015). *Modern Operating Systems*. GOAL Series. Pearson Prentice Hall.

Vargas, I. G., Gottardi, T., and Braga, R. T. V. (2016). Approaches for integration in system of systems: A systematic review. In *Proceedings of the 4th International Workshop on Software Engineering for Systems-of-Systems*, SESoS '16, pages 32–38, New York, NY, USA. ACM.

Vierhauser, M., Rabiser, R., Grünbacher, P., Danner, C., Wallner, S., and Zeisel, H. (2014). A flexible framework for runtime monitoring of system-of-systems architectures. In *IEEE/IFIP Conference on Software Architecture*, WICSA '14, pages 57–66, Washington, DC, USA. IEEE Computer Society.