Synergies of System-of-Systems and Microservices Architectures

Carlos E. Cuesta
Vortic3 Research Group
Rey Juan Carlos University
Madrid, Spain
carlos.cuesta@urjc.es

Elena Navarro
Dept. of Computing Systems
Univ. of Castilla-La Mancha
Albacete, Spain
elena.navarro@uclm.es

Uwe Zdun
Software Architecture Group
University of Vienna
Vienna, Austria
uwe.zdun@univie.ac.at

ABSTRACT

Systems-of-Systems (SoS) are being widely embraced by both practitioners and researchers. They share properties such as distribution, evolutionary development (i.e., openness), operational and managerial independence, and emergent behavior. Those properties imply that any element (system) in an SoS is able to operate independently. Similarly, microservices are suggested as a system architecture with a strong emphasis on *independence*, as containers provide the required degree of isolation, and their infrastructure automation frameworks provide the means to deploy them as needed. In a microservices architecture, even data is independently managed; every service maintains its own datastore, and transaction-less interaction is emphasized. Our hypothesis in this work is that while the two approaches have been treated separately in the literature so far, they share many common characteristics, and it would be fruitful to investigate their synergies. In this paper, we analyze to what extent microservices architectures can be understood as a kind of system-of-systems, explaining some of the success of the microservices approach a consequence of their SoS properties. In addition, the best practices proposed for microservices can enable a conscious, controlled, and manageable introduction of SoS concepts into system architectures, if they are needed.

CCS Concepts

•**Software and its engineering** → **Software architectures; Ultra-large-scale systems;** *Abstraction, modeling and modularity; System description languages;*

Keywords

System-of-Systems; Microservices Architecture; Scalability; Self-Adaptation; Emergent Behavior; Container; IoT

1. INTRODUCTION

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSoS 2016 Copenhagen, Denmark

© 2017 ACM. ISBN XXX-XXXX-XX-XXX/XX/XX.

DOI: XX.XXX/XXX_X

The complexity of software systems is growing constantly, due to both the evolution in the users' demand and the advances in technologies, processes, and their applications. This leads to an increasing number of *System-of-Systems* (SoS): systems that are defined by assembling other systems which are themselves operated and managed independently. This independence facilitates the appearance of emergent behavior in the SoS, which is not supported by any of its assembled systems on its own.

These features have caused a wide interest in such SoSs, leading to the application of their concepts in different domains, namely smart grids [5], national transportation systems [2], healthcare systems [7], national defense systems [6], and many more. For instance, in these domains the SoS approach can be followed with the goal to achieve a higher overall resilience and reliability, among other properties. Every constituent system would operate independently, no matter whether the other systems, e.g., crash (or are down) periodically, or are attacked, or even destroyed.

Systems-of-systems are no silver bullet. They have many unwanted properties, as already noted in the literature [3]. For instance, there can be dependent and cascading failures that propagate from a constituent system to different systems of the SoS. They can manifest non-linear correlations, a.k.a. *copulas* [17], which could make dealing with violations, errors, or unwanted system properties more difficult. Also, the sheer number, complexity, and heterogeneity of events in the SoS makes (centralized) event processing not only more challenging, but even unattainable in some scenarios [15]. Moreover, chaotic behavior might emerge, induced not only because of their nonlinear dynamics, but also because of the potentially chaotic human behavior [10]. And these are just a few of the possible unwanted properties.

However, our systems are evolving, willingly or unwillingly, into SoSs, because monoliths are broken down into SoS structures in order to deal better with their growth and complexity. Mergers and acquisitions force companies to interconnect their formerly independent systems. Independent systems need to be integrated, in order to cope with the continuously growing inter-connectivity of systems. And these are only a few of the potential reasons for SoS emergence.

The fact that many software systems have been developed as monoliths is also of much interest to software development and architecture related communities. In this context, a *monolith* is a system that is developed and tested as a unit, implemented by a single team, and run as a single logical executable. This often leads to problems in the long

run. For instance, when the system must be scaled up, it is not considered which functionality is actually demanded by users: the whole system is just replicated and deployed. If the system needs to be updated, even for small changes, the whole system must be deployed again. As a result, system parts cannot easily be independently developed, tested, or evolved. In order to face such problems, *microservices* [13] have been proposed as an approach to develop a single application as a suite of services, each one running in its own process and communicating with others by means of lightweight mechanisms [11].

While the two approaches have been treated quite separately in the literature so far, it is remarkable that they share many common characteristics. Moreover, many SoSs and microservice-based systems share the common origin of a monolithic architecture being broken down into smaller systems (i.e., parts). Therefore, studying the emergence of SoSs (e.g., by studying small-scale SoS or emerging SoS), as well as "good" migration paths to SoS structures is important, in order to be able to better deal with their aforementioned unwanted properties.

As microservices are largely seen as best practices for certain kinds of distributed systems, on the one hand, this paper explores common characteristics of the two approaches, with the goal to study microservice-based systems as a "controlled" or "well-designed" migration path towards an SoS, as opposed to an unwilling or uncontrolled migration into an SoS structure. On the other hand, established results on SoS properties can be used as a theoretical framework to explain many successful aspects of microservice architectures.

2. FEATURING SYSTEMS-OF-SYSTEMS

Although there is no standard definition for System-of-Systems (SoS), perhaps the most commonly used one is that by Maier [12], which can be summarized in the sentence as: *"A system-of-systems is an assemblage of components which individually may be regarded as systems"*. It is often assumed that a System-of-Systems has the implicit property of (a very big) *size*. This is probably due to its nature beyond composition, and also to its origins in Systems Engineering. Many of the better known examples describe systems which are actually large. Consider the well-known cases of airports, smart cities, or smart power grids.

Certainly there is a relationship to the notion of Ultra-Large-Scale Systems [14], where "size" is explicitly and clearly emphasized, and which can be considered similar to SoS to some extent. However, where the qualifier *ultra-large* must focus on *scale*, the post-compositional structure of a system-of-systems rather focuses on *complexity*. Its features derive from the complexity of its inner flow of information, defining the feedback loops that cause its adaptive behavior.

In fact, the assumption about size often makes it difficult to experiment with, and even to reason about, systems-of-systems. The details about many systems-of-systems, such as today's a smart grid are not easily accessible; and even when they are, the volume of data makes it complex to perform any detailed analysis. This is true even in *software-intensive* systems-of-systems, as they are often physically linked to those large infrastructures.

However, neither size nor complexity are the defining criteria for systems-of-systems. Maier [12] established that a system made up of subsystems is a System-of-Systems if and only if it fulfills the following two criteria:

1. *Operational independence.* Considering an SoS as an assemblage of Systems, this feature establishes that every one of its constituent systems must be able to operate independently, supporting specific customers' needs on its own.

2. *Managerial independence.* Every system assembled as part of an SoS is able to operate, no matter whether the whole SoS exists or not. In fact, usually the systems in an SoS are even acquired and integrated independently of other systems of that SoS.

These two criteria have been accepted by both researchers and practitioners as mandatory to distinguish an SoS from other systems. However, there are four additional criteria, also present to some extent in Maier's work, but which were stated in their original form by Sage and Cuppan [16], and which have attracted much attention by the community. These criteria are:

3. *Geographical distribution.* A System-of-Systems is made up of geographically distributed subsystems. Maier does not claim this is a defining criterion, and he even provides examples of systems-of-systems that do not fulfill this specific requirement. However, such geographically distributed subsystems will usually be operated and managed independently even when assembled, facilitating the fulfillment of the two defining criteria by Maier.

4. *Emergent behavior.* A System-of-Systems has behaviors that emerge from the collaboration of its assembled subsystems, and which are not supported by any of them in an isolated way. Sage and Cuppan even claim this is the most desirable criterion, as it provides new and unexpected uses for an SoS.

5. *Evolutionary development.* A System-of-Systems is continuously modifying its structure, function, and purposes as it is used; that is, it continuously evolves over time. A system-of-systems may even seem not to be fully integrated and functional at first. This evolving nature of systems-of-systems also facilitates complying with the previous criterion, that of emergent behaviour.

Finally, there is another criterion identified by DeLaurentis [2]. Some authors have also recognized this requirement, but it is not always claimed as necessary:

6. *Heterogeneity of constituent systems.* The assembled subsystems in an SoS should have a significantly different nature that leads them to operate on different time scales and dynamics. Sage and Cuppan also appreciate this and define it as an interesting criterion to be expected in a *Federation of Systems* (FoS), but not in a System-of-Systems. These authors assume that a system-of-systems can have centralized control and authority, to some extent. However, this is almost nonexistent in a FoS, as the goals of the federation are only achieved by the collaboration and cooperation among the constituent partners. However, DeLaurentis rules out this distinction, as he does emphasize the importance of this criterion for the development of an SoS.

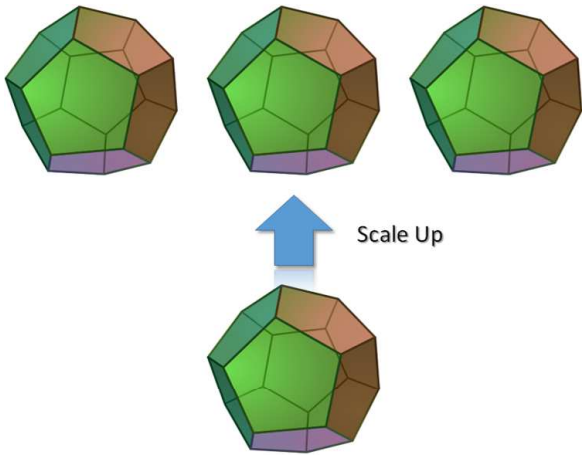

Figure 1: Scaling up a Monolith Architecture

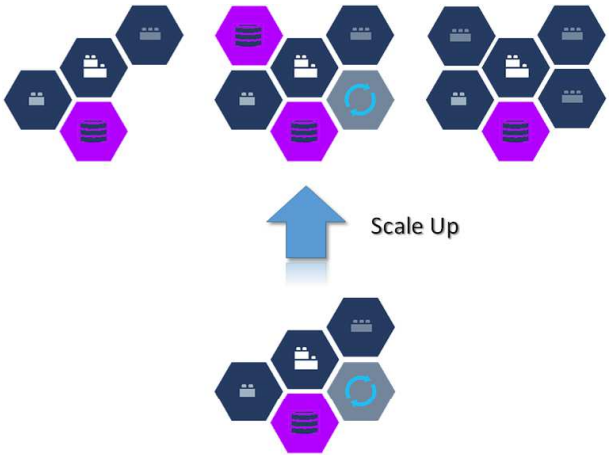

Figure 2: Scaling up a Microservices Architecture

3. SYNERGIES BETWEEN SYSTEMS-OF-SYSTEMS AND MICROSERVICES ARCHITECTURES

Traditionally, many of the applications deployed during the last decades have been developed as *monoliths*; that is, pieces of code that are managed as a single unit. Just consider, e.g., the layered style, one of the most widely used styles for the development of information systems. In many layered architectures, by applying this style, layers for the User Interface, the Business Logic and Persistence are designed, developed, and deployed as a single process. Even when these layers are distributed in an n-tier architecture, they are often tightly coupled, i.e., not fully independent systems or services.

In a monolith, the whole system is run as a single logical executable that is deployed as a whole. Therefore, when the system has to be *changed*, the changes affect the *whole* monolith, making it necessary to re-deploy a new version of the whole system. Moreover, whenever *scalability* becomes a requirement of the system, this is satisfied by replicating the *whole* monolith, as depicted in Fig. 1, instead of just instantiating those resources that are really needed.

Both, support for change and scalability, have been two of the main reasons which also led to the definition of *Microservice Architectures* (MSA) as an architectural style. Lewis' and Fowler's [11] describe this style as *"an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery."* As illustrated in Fig 2, when applying this style, the monolith is broken down into different independent and highly cohesive pieces, that can be developed using different programming languages, database managers, middleware technologies, and so on, each satisfying different functionality parts. This greatly simplifies the way in which *change* can be managed, as every piece can be modified, tested, and deployed independently. Moreover, *scalability* can also be more easily achieved, as different instances of these pieces can be created and deployed on different virtual machines and/or servers.

Therefore, the MSA style describes architectures in terms of independently, automatically deployable, and highly cohesive *microservices*, which use decentralized control of language and data. Indeed, both microservices architectures and systems-of-systems share similar characteristics, including the separation of a monolithic system [12] into a distributed network of independent elements. This claim sets the foundations of our thesis in this paper, namely that many *Microservices Architectures can be understood as a kind of System-of-Systems*; and one of the factors leading to the success of the microservices approach is that their properties are close to the properties of an SoS.

That is, there are synergies between Microservices Architectures and systems-of-systems that *can* and *should* be exploited. In particular, as discussed in Section 1, systems-of-systems have also many undesirable properties, and the emergence of an SoS might unwillingly happen as a consequence of external influences, such as mergers and acquisitions or the intent to cope with the continuously growing inter-connectivity of systems. Hence, the aforementioned synergies can also be used to exploit the best practices proposed for microservices to enable a conscious, controlled, and manageable introduction of SoS concepts into system and software architectures.

In the following, the conceptual correspondences among Microservices Architectures and Systems-of-Systems are described in more detail by explaining how the defining features of Microservices Architectures [11] can be understood as the foundations to satisfy the features of an SoS:

1. *Componentization via Services.* This feature of Microservices Architectures defines that the components that compose such a system are units of software developed as *services* implementing well-defined interfaces. The utilization of services facilitates that each component is managed as an independent process, which can also be independently deployed. *Microservices* are components with a strong emphasis on independence, as *containers* provide the required degree of isolation, and their *automatic deployment* frameworks provide the means to deploy them as needed, even in terms of *scaling up* the microservices. This feature of Microservices Architectures has consequences on achieving a number of SoS properties:

	Operational and managerial independence	Geographical distribution	Emergent behavior	Evolutionary development	Heterogeneity of constituent systems
Componentization via Services	Helps to	Helps to	Eases	Eases	Enables
Organized around Business Capabilities	Eases	Helps to			
Products not Projects	Enables			Eases	
Smart Endpoints and Dumb Pipes	Enables	Eases	Eases	Eases	Eases
Decentralized Governance and Data Management	Eases	Eases	Eases	Enables	Enables
Infrastructure Automation	Enables			Enables	
Design for Failure	Enables	Helps to	Helps to	Enables	Enables
Evolutionary Design			Helps to	Enables	Enables

Table 1: Synergies between System-of-Systems and Microservices Architectures

(a) It helps to achieve *managerial independence* as every service can be started or stopped independently, using the provided infrastructure. Moreover, it also helps to achieve *operational independence* as services have well defined interfaces and are independently deployed.

(b) The conception of units of software as independent services facilitates that not only different technologies, but also different communication protocols, can be used for the development of the SoS. Therefore, the location of every constituent system is not constrained, helping to satisfy the *geographical distribution* property of SoSs.

(c) When services are used as a componentization mechanism, they are expected to be highly cohesive units. This helps to manage composite macro-architectures consistently, and thus to control emergence at the macro-level; hence, componentization via services eases dealing with the *emergent behavior* property of SoSs.

(d) This feature also eases the *evolutionary development* property of SoSs, because single changes are usually confined to services that can be independently evolved and deployed as required. Moreover, Microservices Architectures use evolution-supporting mechanisms such as *service contracts* or *service-level agreements* (SLA) to facilitate easier evolution. Finally, this feature also eases different versions of the same service to be independently deployed, just like different versions of the SoS can be managed.

(e) Finally, it enables the *heterogeneity of constituent systems* because services, in contrast to other componentization concepts and mechanisms, ease the use of heterogeneous system implementations and technologies *per se*. For instance, services are used to deal with heterogeneous middleware technologies and protocols; and, in addition, microservices suggest heterogeneity of databases and database technologies.

2. *Organized around Business Capabilities.* Many applications have been designed by focusing on the technology instead of the business capabilities. Just consider layered-style monoliths, as described above, which prescribe UI, logic, and persistence layers. The development of these layers is usually carried out by teams specialized on the technology supporting such layers. In consequence often, as stated by Conway's Law [1], the system's design eventually mirrors the structure of the organization that develops it. However, Microservices Architecture encourages the development of systems around *business capabilities*. This means that now full-stack developers are required, as they must not only have the competences to use the different technologies involved, but they have to focus on a specific business line and perform different kinds of software engineering activities in this context including development, testing, evolution, deployment, and operations. Functional teams are disbanded in favour of cross-functional teams, composed of experts on, e.g., database, UI, deployment, testing, etc., who work cooperatively towards some business capability. When considering this feature from the SoS point of view, it supports the following SoS properties:

   (a) It enables the fully *operational and managerial independence* of an SoS, as the Microservices Architecture can be organized around SoS capabilities that can be developed by independent teams and, consequently, should be able to operate independently of the SoS as a whole.

   (b) It helps to deal with *geographical distribution*, as the systems of the SoS can be distributed according to business locations, or as demanded by the scalability of the services. For example, services for EU customers can be hosted closer to their data hosted on a database in the EU.

3. *Products not Projects.* The Microservices Architecture approach suggests a clear disruption in the software lifecycle, as developers do not initiate a transition phase to maintenance whenever the software is built,

but the team keeps handling the product during all its lifetime. This is also obviously related to a *continuous deployment* approach and should facilitate a closer relationship between developers and users throughout the product lifetime. This Microservices Architecture feature supports some SoS properties, namely:

(a) It enables *operational independence*, as the responsibilities are clearly defined and maintained across the lifecycle of the systems of the SoS.

(b) It eases *evolutionary development*, as the development teams do not focus on building a complete functionality for the SoS, but on establishing an ongoing relationship with the business to facilitate that evolution.

4. *Smart endpoints and dumb pipes.* Microservices Architecture aims to move the "intelligence" from complex communication mechanisms to the endpoints, that is, to the microservices themselves. These endpoints use simple communication protocols, such as HTTP, keeping the microservices architecture as close to Web's own mechanisms as possible. This feature strengthens the synergies between Microservices Architecture and SoS because:

(a) Microservices are developed as highly decoupled and cohesive components that rely on simple protocols for their orchestration, and therefore a microservices architecture enables *operational independence* as described for SoS.

(b) It also eases *geographical distribution* as intelligence resides in the services, and thus they, as the systems of the SoS, can be distributed more easily and independently.

(c) As the intelligence is located in the endpoints, the systems of the SoS become more independent. Independence of constituent components raises the chances that *emergent behavior* can appear at all.

(d) The high decoupling and cohesion of the microservices eases the *evolutionary development* of the SoS; that is, a major change should not affect the whole SoS but it is likely to affect only one of the constituent systems.

(e) The use of dumb pipes eases the *heterogeneity of the constituent systems* of the SoS because systems of the SoS just produce and consume messages, not requiring any knowledge about the technologies and functionality used by the other microservices.

5. *Decentralized Governance and Data Management.* This feature of Microservices Architectures is related to the aforementioned organization around business capabilities, as it leads naturally to *decentralized governance*. Nowadays, teams are self-directed, able to make decisions about the design, instead of standardizing on single technology platforms, chosen and imposed by a centralized team. The use of services as components also facilitates this decentralization, as different technologies can be deployed as required, according to the needs of the specific business capability being developed. In a microservices architecture, even *data* are decentralized and independently managed; every service maintains its own datastore, emphasizing *non-transactional interaction* to overcome temporal coupling. From a SoS perspective, this feature relates to the following properties:

(a) Decentralization eases *operational and managerial independence*, as every cross-functional team can make decisions about the functionality and operation of the constituent systems of the SoS which it is responsible for.

(b) Decentralized governance implies that the systems of the SoS can be *geographically distributed* more easily. Teams responsible for their development, operations, and management do not need to be collocated because they act independently, facilitating a faster reaction to problems and urgent needs. Moreover, as every system of the SoS maintain its own datastore, the location of that datastore does not impose any geographical constraint on the location of the services which are independent of those data.

(c) Therefore, the more decentralized the systems of an SoS are, the more independent they become. As already noted, the independence of constituent components raises the chances that *emergent behavior* eventually appears.

(d) As data and governance aspects can evolve independently, this feature also enables the independent evolution of the systems of the SoS, facilitating its *evolutionary development.*

(e) Decentralized governance and data management also enable the *heterogeneity of the technologies and concepts* used for the constituent systems, as there is not a "standard" technology used all throughout the SoS. On the contrary, each cross-functional team will make decisions according to the specific needs of the system.

6. *Infrastructure Automation.* Development teams make extensive use of tools and techniques that facilitate the automation of Continuous Integration (CI) and Continuous Delivery (CD) while building, deploying, and operating microservices. In fact, Microservices Architecture is recognized as the *de facto* standard architecture style for DevOps [4, 9]. DevOps encourages a higher collaboration between developers and IT teams to achieve a faster software delivery. This feature facilitates the support of the following SoS properties:

(a) Infrastructure automation, as used for DevOps, CD or CI, enables the *operational independence* of the systems of the SoS, because it enables that these systems are built, tested, and deployed independently.

(b) Infrastructure automation also enables a new level of *evolutionary development*, as it facilitates that each system of the SoS can be incrementally and independently developed, possibly with significantly shorter evolution (and release) intervals.

7. *Design for Failure.* Microservices Architectures rely on the introduction of different mechanisms for monitoring and logging, which are able to detect and to *react* to failures as soon as possible. These systems are built upon the assumption that failures eventually emerge; their design intends to keep the system operational in those situations, not affecting the user experience. This feature is important for supporting the following SoS properties:

   (a) It enables *operational independence*, because the systems of the SoS are able to perform their operations even when some of them are failing.

   (b) It helps to achieve a reasonable degree of availability even in the context of *geographical distribution*, as the systems of the SoS are able to exploit *partial failure* features of distributed systems.

   (c) As Microservices Architectures often rely on *event-based* collaboration, or other loosely coupled interconnections which foster system independence, the chances that *emergent behavior* appears are again raised. More importantly, the design for failure of microservices architectures, in the context of SoS, helps to manage the already described *unwanted emergent behavior*, thanks to the introduction of their monitoring and logging mechanisms.

   (d) Humble and Farley [8] point out that there are some problems that may arise when Continuous Deployment techniques are not properly understood and applied by the team, such as infrequent or buggy deployments, or poor application quality. As Microservices Architecture prescribes the introduction of monitoring mechanisms to check architectural elements, failures can be detected early in the process, enabling the *evolutionary development* property of SoSs.

   (e) Apart from being mandatory for microservices architectures, design for failure also enables coping with problems like temporal unavailability, which might occur due to the *heterogeneity of the constituent systems*. Also, finding bugs across these heterogeneous constituent systems is simpler thanks to presence of the monitoring and logging mechanisms.

8. *Evolutionary Design.* In order to break down the monolith into components, the design of the Microservices Architecture usually considers two important properties, *replaceability* and *upgradeability*, which must be satisfied by the final system. That is, components can evolve without hindering the evolution of any other components, or even their testability. When considering these properties in the context of SoS features, they provide the following synergies:

   (a) Considering that systems of the SoS are designed to be evolved independently, following the Microservices Architecture approach, evolutionary design (together with *infrastructure automation*) leads to continuous upgrades and releases that are independently made by independent teams. Again, this kind of independence raises the chances that *emergent behavior* appears. Potentially faster and more independent evolution cycles also help to cope with *unwanted emergent behavior* more easily, as different teams and microservices can independently adapt to such emerging situations.

   (b) The design for evolution facilitates that the systems of the SoS can be independently replaced and upgraded, thus enabling an *evolutionary development*. In summary, these two features are obviously related.

   (c) Evolutionary design of the systems of an SoS facilitates their independent evolution and replaceability. This does not only enables, but in the long run might even multiply, the *heterogeneity* of the constituent systems, as their nature –without a centralized governance structure– becomes more divergent as they independently evolve.

Table 1 summarizes our discussion on the synergies between Microservices Architectures and SoS.

4. CONCLUSIONS

After the discussion presented in the previous section, it can be argued that every microservices-related feature, either leads to multiple system-of-systems features, or directly supports them, as summarized in Table 1. However, it must also be clarified that there is not a bidirectional relationship among them; that is, *not* every SoS is a microservices architecture. For example, many SoSs might have certain centralized structures for governance or communication, such as an Enterprise Service Bus. Others do not support infrastructure automation in any of their systems, or do not provide service-based componentization, etc.

Similarly, *not* every system starting as a microservices architecture would eventually lead to an SoS. For instance, if the microservices of a microservices architecture are neither managed nor operated independently, a defining property of SoSs would not be fulfilled.

However, our analysis of the potential synergies shows that Microservices Architectures can enable a conscious, controlled, and manageable introduction of SoS concepts into system architectures. Their features, such as design for evolution, componentization via services and infrastructure automation, can lay the foundation to define a *reengineering* process that replaces a monolith applying a SoS approach, which is better distributed and more independently operated. It can also lead to an alignment of development processes and team structures as suggested by the Microservices Architecture approach.

Finally, SoS properties can help to explain and better understand some of the success factors of the microservices approach. The overall consequence of this conceptual correspondence between the properties of both architectural styles is the *emergent behavior*, which also might imply *decentralized governance*: usually there is no central control in either of the approaches.

5. ACKNOWLEDGEMENTS

This work has partially been funded by the Spanish Ministry of Economy and Competitiveness and by FEDER funds from the EU under the Grants CoMobility (TIN2012-31104),

insPIre (TIN2012-34003), Multiply@City (TIN2016-78103-C2-1-R), and Vi-SMARt (TIN2016-79100-R).

6. REFERENCES

[1] M. E. Conway. How Do Committees Invent? *Datamation*, 14(4):28–31, 1968.

[2] D. DeLaurentis. Understanding Transportation as a System-of-Systems Design Problem. In *43rd AIAA Aerospace Sciences Meeting and Exhibit*, Reston, Virigina, Jan. 2005. American Institute of Aeronautics and Astronautics.

[3] P. Dersin. Systems of Systems. IEEE-RS-TC-SoS White Paper, IEEE Reliability Society. Technical Committee on Systems of Systems, Oct. 2014. http://rs.ieee.org/component/content/article/9/77-system-of-systems.html.

[4] V. Farcic. *The DevOps 2.0 Toolkit. Automating the Continuous Deployment Pipeline with Containerized Microservices*. Leanpub, Feb. 2016.

[5] J. Gao, Y. Xiao, J. Liu, W. Liang, and C. P. Chen. A survey of communication/networking in smart grids. *Future Generation Computer Systems*, 28(2):391–404, 2012.

[6] R. K. Garrett, S. Anderson, N. T. Baron, and J. D. Moreland. Managing the interstitials. A system of systems framework suited for the ballistic missile defense system. *Systems Engineering*, 14(1):87–109, 2011.

[7] Y. Hata, S. Kobashi, and H. Nakajima. Human health care system of systems. *IEEE Systems Journal*, 3(2):231–238, 2009.

[8] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2011.

[9] M. Hütterman. *DevOps for developers*. Apress, 2012.

[10] W. Karwowski. A review of human factors challenges of complex adaptive systems: Discovering and understanding chaos in human performance. *Human factors*, 54(6):983–995, 2012.

[11] J. Lewis and M. Fowler. Microservices: a definition of this new architectural term. http://martinfowler.com/articles/microservices.html, Mar. 2004.

[12] M. W. Maier. Architecting Principles for System of Systems. *Systems Engineering*, 1(4):267–284, 1998.

[13] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen. *Microservice Architecture. Aligning Principles, Practices and Culture*. O'Reilly, June 2016.

[14] L. Northrop, editor. *Ultra-Large-Scale Systems. The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon, June 2006.

[15] F. Paraiso, G. Hermosillo, R. Rouvoy, P. Merle, and L. Seinturier. A middleware platform to federate complex event processing. In *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*, pages 113–122. IEEE, 2012.

[16] A. P. Sage and C. D. Cuppan. On the Systems Engineering and Management of Systems of Systems and Federations of Systems. *Information Knowledge Systems Management Journal*, 4(2):325–345, Dec. 2001.

[17] P. Shah, N. Davendralingam, and D. A. DeLaurentis. A conditional value-at-risk approach to risk management in system-of-systems architectures. In *System of Systems Engineering Conference (SoSE), 2015 10th*, pages 457–462. IEEE, 2015.