2016 21st International Conference on Engineering of Complex Computer Systems

Formally Describing the Architectural Behavior of Software-intensive Systems-of-Systems with SosADL

Flavio Oquendo
IRISA – UMR CNRS
Univ. Bretagne Sud, France
e-mail: flavio.oquendo@irisa.fr

***Abstract*—Software-intensive systems are often independently developed, operated, managed, and evolved. Progressively, communication networks have enabled these independent systems to interact, yielding a new kind of complex system, i.e. a system that is itself composed of systems, the so-called System-of-Systems (SoS). By its complex nature, SoS exhibits emergent behaviors.**

Nowadays, none of the Architecture Description Languages (ADLs), which have been developed for modeling the architectural behavior of single software-intensive systems, has the expressive power to formally describe the architectural behavior of Software-intensive SoSs.

For addressing this research challenge, we propose a novel ADL, called SosADL, specially conceived for formally describing the architecture of Software-intensive SoSs. It embodies SoS architectural concepts and constructs encompassing the formal description of software architectures from both the structural and behavioral viewpoints.

This paper presents SosADL focusing on the description of the architectural behavior of Software-intensive SoSs. It describes SosADL from its behavioral viewpoint enabling to specify independent systems, mediators among these systems, coalitions of mediated systems, and the architectural conditions that enforce the production of emergent SoS behaviors. It illustrates SosADL through an excerpt of a real application for architecting a Flood Monitoring and Emergency Response SoS.

Keywords—Software Architecture Description Language; Software-intensive System-of-Systems; Formal Behavioral Modeling.

I. INTRODUCTION

The complexity of software and the complexity of systems reliant on software have grown at a staggering rate. In particular, software-intensive systems have been rapidly evolved from being stand-alone systems in the past, to be part of networked systems in the present, to increasingly become systems-of-systems in the coming future.

De facto, the pervasiveness of the communication networks increasingly has made possible to interconnect systems that were independently developed, operated, managed, and evolved, yielding a new kind of complex system, i.e. a system that is itself composed of systems, the so-called System-of-Systems (SoS) [14][25]. SoSs are evolutionary developed from independent systems to achieve missions not possible to be accomplished by a single system alone. They are architected to exhibit emergent behavior.

This is the case of SoSs found in different areas as diverse as aeronautics, automotive, energy, healthcare, manufacturing, and transportation; and applications that address societal needs such as flood monitoring, environmental monitoring, emergency coordination, traffic control, smart grids, and smart cities.

It is worth noting that complexity is intrinsically associated to SoSs by its very nature that implies emergent behaviors [12], i.e. new behaviors that stem from the interactions among constituent systems, but cannot be deduced from the behaviors of the constituent systems themselves. It means that the behavior of the whole SoS cannot be predicted through analysis only of its constituent systems, or stated simply: "the behavior of the whole SoS is more than the sum of the behaviors of its constituent systems". Note also that in SoSs, missions are achieved through emergent behaviors drawn from the local interactions among constituent systems.

Hence, complexity poses the need for separation of concerns between architecture and engineering: (i) architecture focuses on designing and reasoning about interactions of parts and their emergent properties; (ii) engineering focuses on designing and constructing such parts and integrating them as architected.

Definitely, a key facet of the design of any software-intensive system, being a single system or a system-of-systems, is its software architecture (i.e. the fundamental organization of the system embodied in its constituents, their relationships to each other, and to the environment, and the principles guiding its design and evolution, as defined by the ISO/IEC/IEEE Standard 42010 [11]).

Conceiving Software Architecture Description Languages (ADLs) has been the subject of intensive research in the last 20 years resulting in the definition of several ADLs for modeling initially static architectures and afterwards dynamic architectures of (often large) single systems [15][16][27][17]. However, none of these ADLs has the expressive power to describe the architecture of a Software-intensive SoS [9][13], in particular regarding the formal modeling of architectural behavior.

To fill this gap, we have defined SosADL [22][23], a novel ADL specially conceived for formally modeling the architecture of Software-intensive SoSs from both structural and behavioral viewpoints. SosADL is formally grounded in the π-Calculus for SoS [24].

This paper focuses on the presentation of SosADL from the behavioral viewpoint, enabling to formally describe SoS architectural behaviors. From this viewpoint, the behavior of each constituent system of an SoS is described as well as how these constituent systems can be mediated to form coalitions producing SoS emergent behaviors. The formal basis of SosADL is provided by the π-Calculus for SoS.

978-1-5090-5527-2/16 $31.00 © 2016 IEEE
DOI 10.1109/ICECCS.2016.35

This novel ADL brings the following contributions to the state-of-the-art:

- its novel formal foundation: we conceived a novel process calculus in the family of the π-Calculus [18], named π-Calculus for SoS [24] (its formal definition was presented in the 38th International Conference on Communicating Process Architectures – CPA 2016; see [24] for details);
- its novel architectural concepts and the language constructs concretely embodying these SoS architectural concepts: grounded on the π-Calculus for SoS, we conceived a novel architectural language based on the separation of concerns between architectural abstractions at design-time and architectural concretions at run-time (its architectural structure viewpoint and its underlying semantics were presented in the 11th IEEE SoS Engineering Conference – SoSE 2016; see [22] and [23] for details; and its architectural behavior viewpoint is the subject of this paper);
- its validation based on a real pilot project and related case study on how to formally describe Software-intensive SoS architectures was presented in the 15th IEEE International Conference on Systems, Man, and Cybernetics – SMC 2016 (see [26] for details).

The remainder of this paper is organized as follows. Section II introduces the notion of Software-intensive SoS. Section III gives the motivation for conceiving a novel ADL for Software-intensive SoS. Section IV briefly presents the concepts of SosADL from the structural viewpoint. Section V presents SosADL constructs for describing SoS architectures from the behavioral viewpoint. Section VI illustrates SosADL in the description of SoS architectural behaviors through an excerpt of a field study of a real SoS. In section VII, we outline the implemented toolset and address the validation of SosADL. In section VIII, we compare SosADL with related work. To conclude we summarize, in section IX, the main contributions of this paper and outline future work.

II. Notion of System-of-Systems (SoS)

The notions of system (in general) and software-intensive system (in particular) are defined in the ISO/IEC/IEEE Standard 42010 [11]. A system is a combination of constituents organized to accomplish a specific behavior for achieving a mission. Hence, a system exists to fulfill a mission in its environment. A software-intensive system is a system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole.

The notion of Software-intensive SoS is however relatively new, being the result of the ubiquity of computation and pervasiveness of communication networks.

An SoS is a combination of constituents, which are themselves systems, that forms a more complex system to fulfill a mission, i.e. this composition forms a complex system that performs a mission not performable by one of the constituent single systems alone, creating emergent behavior.

For intuitively distinguishing an SoS from a single system, it is worth to recall that every constituent system of an SoS fulfills its own mission in its own right, and continues to operate to fulfill its mission during its participation in the SoS as well as, once disassembled from the encompassing SoS, afterwards. For instance, an airport, e.g. Paris-Charles-de-Gaulle, is an SoS, but an airplane alone, e.g. an Airbus A380, is not. Indeed, if an airplane is disassembled in components, none of them is a system in itself. In the case of an airport, the constituent systems are independent systems that will continue to operate, e.g. the air traffic control and the airlines, even if the airport is disassembled in its constituents.

Precisely, a Software-intensive SoS is defined as a software-intensive system, which is itself composed of other systems, exposing the five characteristics cited below [14].

In an SoS, constituent systems must be:

- operationally independent,
- managerially independent,
- physically decoupled, geographically distributed.

The SoS as a whole must have:

- evolutionary development,
- emergent behaviors.

Moreover, the actual constituent systems are generally not known at design-time and are only identified at run-time.

The combination of these defining characteristics turns the architecture of SoSs to be naturally highly evolvable, frequently changing at run-time. SoSs are said to have evolutionary architectures (with the meaning that they dynamically adapt or evolve at run-time as a consequence of evolutionary development and emergent behaviors).

III. Motivation for a Novel ADL for SoS

The distinctive feature between an SoS and a single system is the nature of their constituents, specifically their level of independence and the exhibition of emergent behavior [7]. The independence of the constituents of an SoS implies its evolutionary development. Emergent behavior is drawn from local interactions of independent constituent systems: local interactions must enable the emergence of global behaviors, i.e. behaviors that cannot be performed by any of the constituent systems alone to achieve SoS missions [29].

In the last two decades, much work has addressed the issue of describing software architecture (in the sense of architecture of software-only systems as well as software-intensive systems) from the structural and behavioral viewpoints. Most of the work carried out addressed static architectures (i.e. architectures which do not change at run-time) and some tackled dynamic architectures (i.e. architectures which may change at run-time according to planed reconfigurations decided at design-time) [17]. Therefore, we must pose the question whether these ADLs provide enough expressive power for describing SoS architectures.

To address this question, i.e. "*do current ADLs provide enough expressive power for describing SoS architectures?"*, we analyze in *Table I* what are the implications of the SoS characteristics when compared with the expressive of power of ADLs designed for single systems. For each SoS defining characteristic, also in *Table I*, we answer that question.

TABLE I. SOS DEFINING CHARACTERISTICS VS. SINGLE SYSTEM ADLS

SoS Defining Characteristics	Do current ADLs provide enough expressive power for describing SoS architectures?
Operational independence of constituent systems	• *No*: Single system ADLs have been defined based on the notion of constituent components whose operation is controlled by the system (where that control is encoded by the system architect at design-time), which is not the case of SoSs. Moreover, the concrete components (instances) are known at design-time, which is not the case of SoSs either.
Managerial independence of constituent systems	• *No*: Single system ADLs have been defined based on the notion of constituent components whose management is controlled by the system (decisions are taken by the system architect at design-time), which is not the case of SoSs.
Physical decoupling and geographical distribution of constituent systems	• *No*: Single system ADLs have been defined based on the notion of logically distributed components. None supports the notion of physical mobility, in particular regarding unexpected local interactions among constituents that physically move near to each other, as it is the case of SoSs.
Evolutionary development of SoS	• *No*: Single system ADLs have been defined based on the principle that development is carried out initially based on concrete constituents which are known at design-time, which is not the case of SoSs.
Emergent behavior drawn from SoS	• *No*: Single system ADLs have been defined based on the principle that all behaviors are explicitly defined. None supports the notion of emergent behavior, as it is the case of SoSs.

Based on the conclusion of the analysis carried out (see [9] for the details), summarized in the answers shown in *Table I*, it is clear that no existing ADL for single systems was conceived to formally describe SoS architectures. Even advanced ADLs such as Darwin [8], Wright [1] and π-ADL [2][21] fall short in the description of such evolutionary architectures because of their limited expressive power in terms of run-time evolution and lack of support for emergent behaviors. In addition, using these single system ADLs, it is not possible to describe an architecture for which the architect does not know the concrete components at design-time.

Indeed, an SoS architecture must describe how constituents will enable the emergence of SoS-wide behaviors derived from local ones (by acting only on local interactions, without being able to act on the constituent systems themselves).

Therefore, defining a formal ADL able to describe the software architecture of SoSs, in particular enabling the specification of architectural behavior, is a major research challenge as no existing ADL has the expressive power to describe such evolutionary architectures [25] (see also [9] for more details on the limitations and inadequacies of single system ADLs for describing SoS architectures).

In addition, an ADL for describing SoS needs to cope with the fact that concrete constituent systems are in general not known at design-time, being only identified at run-time.

IV. SOSADL FROM THE STRUCTURAL VIEWPOINT

ADLs conceived for describing the software architecture of single systems have serious drawbacks when confronted with the requirements for describing the software architecture of SoSs (summarized in *Table I*). To overcome these limitations, a novel ADL is needed to enable the description of Software-intensive SoS architectures in formal terms.

The SosADL was conceived to overcome these limitations from both structural and behavioral viewpoints.

Recall that software architecture is defined to be the fundamental organization of a system embodied in its constituents, their relationships to each other and to the environment, and the principles guiding its design and evolution [11].

In ADLs for single systems, the core concepts are the one of *component* to represent the constituents, the one of *connector* to represent the relationships among constituents, and the one of *configuration* to represent their composition.

As the semantics of these concepts do not cope with the nature of SoS architectures, we defined novel concepts in SosADL, named in terms aligned with the SoS terminology to fit the semantics needed for describing SoS architectures.

First, the architectural concepts of component of a single system, connector among components of a single system, and configuration of connected components of a single system were generalized for coping with the more general semantics of constituent *system* of an SoS, *mediator* among constituent systems of an SoS, and *coalition* of mediated constituent systems of an SoS.

Second, using SosADL, SoS architectures are represented in abstract terms (recall that concrete systems that will become constituents of the SoS are generally not known at design-time). The defined abstract architecture will then be evolutionarily concretized at run-time, by identifying and incorporating concrete systems (see [10] for details on the automated resolution for creating concrete SoS architectures with SosADL).

Note that an abstract architecture is the expression of all possible concrete architectures in declarative terms. A concrete architecture is the actual architecture that operates at run-time.

Based on these ADL design decisions, we have conceived SosADL. Thereby, with SosADL, in an SoS architecture description:

- *Systems* are SoS architectural elements defined by intention (declaratively in terms of abstract systems) and selected at run-time (concretized);
- *Mediators* are SoS architectural elements defined by intention (declaratively in terms of abstract mediators) and created at run-time (concretized) to achieve a goal, part of a mission [29] (note that its architectural role is to mediate the interaction of constituent systems for contributing to emergent behavior);
- *Coalitions* are SoS architectural compositions of mediated constituent systems, defined by intention (declaratively in terms of possible systems and mediators and policies for their on-the-fly compositions) and evolutionarily created at run-time (concretized) to achieve an SoS mission in a specific environment.

TABLE II. CONCEPTS IN SINGLE SYSTEM ADLS VS. SOSADL

Sys. ADLs	SosADL
Software architectural concepts	
Notion of component	***System***. Systems are the constituents of an SoS: a system has its own mission, is operationally independent, managerially independent, and may independently evolve; its focus is on capabilities to deliver system functionalities. • Note that the concept of constituent system subsumes the concept of component in single system ADLs: a component can be perceived as a "constituent system that is completely subordinated", oppositely to constituent systems in general that are, by definition, independent. • Constituent systems exist independently of the SoS and may enter or leave the SoS at run-time by their own decision.
Notion of connector	***Mediator***. Mediators mediate the interaction of constituent systems in an SoS: a mediator has the purpose to achieve a specific emergent behavior by mediating the local interaction among different constituent systems. • Note that the concept of mediator in SosADL subsumes the concept of connector in single system ADLs: mediators are expressed using constraints and have a coordination role, while connectors are basically communication channels for passing values or control among constituents. • Note that mediators are both operationally and managerially dependent of the SoS, and evolve under its control for achieving emergent behaviors (an SoS has total control on mediators: it can create, evolve and destroy mediators at run-time).
Notion of configuration	***Coalition***. A coalition constitutes a temporary alliance for combined action among constituent systems connected via mediators (it is dynamically formed to fulfill the SoS mission through emergent behaviors). • Coalitions can be decomposed and recomposed in different ways or with different systems in order to fulfill a specified SoS mission. • Coalitions are declared by expressing the SoS policies to select and bind existing constituent systems using mediators created by the SoS itself. • Note that the SoS totally controls its mediators, but not at all its constituent systems, which are independently operated, managed, and evolved.
Design-time use of the architectural concepts	
Abstract architecture defined by intension	***Abstract architecture***. Only constraints are specified in an SoS abstract architecture description (it includes constraints to select possible constituent systems and contracts to be fulfilled by constituent systems in mediators). • By the nature of SoSs, concrete constituent systems are generally not known at design-time.
Run-time use of the architectural concepts	
Concrete architecture defined by extension	***Concrete architectures***. Concrete systems coping with the specified system abstractions may enter or leave an SoS at run-time (the SoS has no control on concrete systems); mediators oppositely are dynamically created and evolve under the SoS control. • A concrete architecture evolves to cope with evolution of constituents for continuously achieving its mission under abstract architectural constraints.

In *Table II* we characterize how the architectural concepts embodied in SosADL are different and generalizes the ones of ADLs for single systems.

For a detailed description of SosADL from the structural viewpoint, see [22].

V. SOSADL FROM THE BEHAVIORAL VIEWPOINT (LANGUAGE DEFINITION)

Now, we will focus on the definition of the behavioral constructs embodied in SosADL. The abstract syntax is defined by abstract productions rules[1].

The abstract syntax for expressing behaviors is defined in *Fig. 1* (focusing on a subset, with some eluded definitions for sake of brevity).

Abstract syntax of behaviors

constrainedBehavior ::= behavior$_1$
| valuing$_1$. constrainedBehavior$_1$
| **behavior** name$_1$ **is {** behavior$_1$ **}**
| **constraint** name$_1$ **is {** constraint$_1$ **}**
| **compose {** constrainedBehavior$_0$ … **and** constrainedBehavior$_n$ **}**

behavior ::= baseBehavior$_1$
| valuing$_1$. behavior$_1$
| **repeat {** behavior$_1$ **}**
| **apply** name$_1$ **(** value$_0$ …**,** value$_n$ **)**
| **compose {** behavior$_0$ … **and** behavior$_n$ **}**

baseBehavior ::= action$_1$. behavior$_1$
| **choose {** action$_0$. baseBehavior$_0$
or action$_1$. baseBehavior$_1$ … **or** action$_n$. baseBehavior$_n$ **}**
| **if** constraint$_1$ **then {** baseBehavior$_1$ **} else {** baseBehavior$_2$ **}**
| **done**

action ::= baseAction$_1$
| **tell** constraint$_1$
| **untell** constraint$_1$
| **check** constraint$_1$
| **ask** constraint$_1$

baseAction ::= **via** connection$_1$ **send** value$_0$
| **via** connection$_1$ **receive** name$_0$ **:** type$_0$
| **do** internalAction$_1$
| **unobservable**

connection ::= name$_1$ | name$_1$…::name$_n$

valuing ::= **value** name$_1$ **is** type$_1$ **=** value$_0$ | typing

typing ::= **datatype** name$_1$ **is** type$_0$ **{** function$_0$ … **and** function$_n$ **}**

Fig. 1. Abstract syntax of typed behaviors and values

[1] The SosADL abstract grammar definition uses the following notation for the abstract production rules: (i) keywords are written with bold; (ii) non-terminals are written without bold; (iii) a sequence of zero, one or more elements is written: $Element_{min}$, …, $Element_{max}$, where the value of *min* specifies the minimum number of elements (*0* specifies possibly no elements, *1* specifies at least one element) and the value of *max* specifies the maximum number of elements ($Element_n$ specifies any number of elements); (iv) alternative choices are written separated by |.

Note that as all statements have behavior as scope, the scope *behavior* will be explicitly shown in the continuations. Let us now present the constructs used in behavior declaration, starting from simpler ones (from bottom up in *Fig. 1*).

Typing: ***datatype*** *name* ***is*** *type* ***{…} .*** *behavior* declares a value type named *name* according to the declaration of type *type* in the scope of a behavior *behavior*. Functions for manipulating the values of the declared type may be defined.

Valuing: ***value*** *name* ***is*** *type = value* ***.*** *behavior* declares a variable named *name* of type *type* possibly initialized with *value* in the scope of a behavior *behavior*. Values declare variables that are local, restricted to the scope of *behavior*, which however may be extruded.

Action: *action* ***.*** *behavior* expresses the capability of a behavior to execute an *action* and then to continue as *behavior*.

Behaviors execute by performing actions. The capabilities for action are expressed through output and input prefixes when dealing with communication, constraint-handling prefixes when dealing with constraints, and a silent prefix when dealing with internal actions. An output action ***via*** *connection* ***send*** *value* expresses the capability to send a value *value* via the connection *connection*. An input action ***via*** *connection* ***receive*** *value* ***:*** *type* expresses the capability to receive a value *value* of type *type* via the connection *connection*. The internal action ***unobservable*** expresses the capability to enact an action, internally, invisibly.

Besides expressing unobservable actions, internal actions can be expressed using ***do*** *internalAction*. In this case, explicit internal actions can be executed by applying operators and functions defined in data types.

An assert action ***tell*** *constraint* expresses the capability to assert a constraint in the local environment of a behavior. A retract action ***untell*** *constraint* expresses the capability to retract a constraint from the local environment of a behavior.

A check action ***check*** *constraint* expresses the capability to check if a constraint is consistent with the told constraints in the local environment of a behavior.

An entail action ***ask*** *constraint* expresses the capability to check if a constraint is entailed from the told constraints in the local environment of a behavior.

Thereby, actions express the capability of sending or receiving values via connections, handling constraints, or enacting unobservable actions.

In addition, a base library of primitive types equipped with functions is provided. It includes mono-valued types as well as multi-valued types, e.g. the following functions are provided for manipulating sequences (ordered collection of values in which repetitions are allowed):

- *sequence****::select{*** *name* ***suchthat*** *constraint* ***}*** selects elements of a sequence *sequence*: it returns a sequence with the selected elements of *sequence* that satisfy the specified *constraint* where *name* is an iterator traversing the *sequence*;
- *sequence****::collect{*** *name* ***suchthat*** *value* ***}*** collects elements of a sequence *sequence*: it returns a sequence with the results of applying an expression *value* that returns a value to each element of the sequence.

Condition: ***if*** *constraint* ***then*** **{** $behavior_1$ **}** ***else*** **{** $behavior_2$ **}** expresses the capability of a behavior to proceed as $behavior_1$ if the *constraint* is true, otherwise as $behavior_2$.

Choice: ***choose*** **{** $behavior_1$ ***or*** $behavior_2$ **}** expresses the capability of a behavior to choose either the capability of $behavior_1$ or the capability of $behavior_2$. When one of the capabilities is exercised, the other is no longer available. Thereby, the choice will proceed either as $behavior_1$ or (exclusive or) as $behavior_2$. The concrete syntax generalizes choice for any number of behaviors.

Repeat: ***repeat {*** *behavior* ***}*** expresses the capability of a behavior to be iteratively repeated.

Inaction: ***done*** is the behavior that can do nothing (inaction is not a basic construct: it can be defined as an empty choice). It represents the end of an execution.

Composition: ***compose*** **{** $behavior_0$ … ***and*** $behavior_n$ **}** expresses the capability of a behavior to parallel compose the capabilities of $behavior_0$ … and $behavior_n$. After composition, they proceed independently and can interact via bound connections. Thereby, the composition will proceed by exercising the capabilities of independent actions in $behavior_0$ … and $behavior_n$, or jointly exercising a capability of interaction, i.e. when, via a bound connection, one behavior sends a value and the other receives the value yielding an unobservable communication action.

Constraints: assertions can be defined in terms of constraints using ***constraint*** name ***is*** **{** *constraint* **}** for being reused in constructs handling constraints.

Constrained composition: composition of behaviors can be constrained using ***compose*** **{** $behavior_0$ … ***and*** $behavior_n$ ***and*** $constraint_0$ … ***and*** $constraint_n$ **}**. It expresses the capability of a behavior to parallel compose the capabilities of $behavior_0$ … and $behavior_n$ subject to $constraint_0$ … ***and*** $constraint_n$. Note that the case of composition without constraints is subsumed by the constrained composition.

Application: to apply abstractions and create concretions, application is used. An application has the form ***apply*** *abstraction* **(** $value_0,\ldots,value_n$ **)**. Applications instantiate abstractions yielding running behaviors.

Abstraction: An abstraction may define architectures, constituent systems or mediators. Connections are used to express gates in the case of systems and duties in the case of mediators.

VI. SosADL from the Behavioral Viewpoint (excepts from a real SoS application)

As seen, SosADL provides architectural concepts and notation for describing SoS architectures from both structural and behavioral viewpoints. The SosADL notation for describing SoS architectures from the structural viewpoint is detailed in [22]. Hereafter we will focus on its expressiveness as an ADL for describing SoS architectures from the behavioral viewpoint by focusing in an excerpt of a field study carried out for architecting and engineering an SoS for Flood Monitoring and Emergency Response.

Flood Monitoring and Emergency Response SoSs address the problem of flash floods, which constitute a significant threat in different countries during rainy seasons.

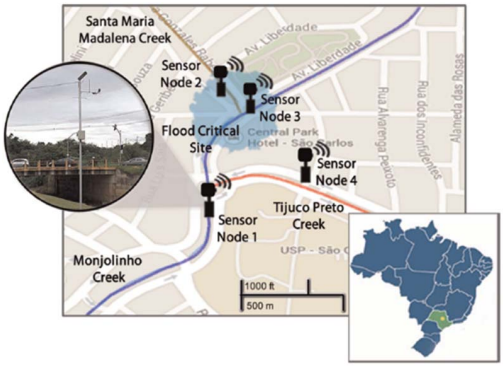


Fig. 2. Monjolinho river crossing the city of Sao Carlos with deployed wireless river sensors

Flash flood becomes particularly critical in cities that are crossed by rivers such as the city of Sao Carlos, SP, Brazil, crossed by the Monjolinho river as shown in *Fig. 2*.

To address this critical problem, we have architected with SosADL a Flood Monitoring and Emergency Response SoS.

To highlight the main architectural concepts and constructs of SosADL, we will use a subset of this Flood Monitoring and Emergency Response SoS, which is itself an SoS. The Urban River Monitoring SoS [6] is based on two kinds of constituent systems: wireless river sensors (for measuring river level depth via pressure physical sensing) and a gateway base station (for analyzing variations of river level depths and warning on the risk of flash flood).

For achieving the stated mission by creating emergent behavior, the architecture of this SoS needs to be rigorously conceived. In particular, resilience (even in case of low charge in the battery of sensors or failure) needs to be managed as well as its operation in an energy-efficient way.

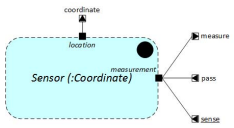
A. Describing abstract SoS architectures with SosADL

Let us describe now an SoS abstract architecture focusing on the Urban River Monitoring SoS and then its concretization in the case of the Monjolinho river.

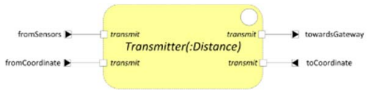
In SosADL, the different kinds of constituent systems are declared as system abstractions as shown in *Fig. 3*.

The *Sensor* abstract constituent system declares three gates: *measurement* that comprises connections for handling measures, *location* that includes a connection for handling GPS coordinates, and *energy* (hidden in *Fig. 3*) that has connections for managing energy consumption. For the sake of brevity, we will not present the full description of the *Sensor* abstract system, focusing only on its behavior. The behavior exposed by the *Sensor* system is shown in the top of *Fig. 3*.

This behavior specification first declares a variable to store the value of the sensor's GPS coordinate, then tell publicly this value sending it via its gate *location* to its neighbors. Subsequently, it receives the power threshold enabling operation and repeatedly (if the remaining power is greater than the power threshold) tells that it is able to operate and will choose between sensing raw data from the sensor device and transmitting the corresponding measure (sending a tuple with both its GPS coordinate and the measure converted from the raw data received) or just forwarding a measure received from a neighbor sensor via connection *pass*.




```
system Sensor(lps: Coordinate) is { …
  behavior sensing is {
     // 'behavior' executes when the 'system' is created
     value sensorcoordinate is Coordinate = lps
     tell sensorlocation is {sensorcoordinate = lps}
        // assert the specified constraint with name
        // and type extrusions
     via location::coordinate send sensorcoordinate
     via energy::threshold receive powerthreshold
     repeat {
        via energy::power receive powerlevel
        if (powerlevel > powerthreshold) then {
           // 'if' tests the condition without blocking
           tell powering is {powerlevel > powerthreshold}
           choose{
              via measurement::sense receive data
              via measurement::measure send
                 tuple{coordinate=lps,depth=data::convert()}
           } or {
              via measurement::pass receive data
              via measurement::measure send data
           }
        }
     }
  }
}
```




```
mediator Transmitter(distancebetweengates:Distance) is { …
  duty transmit is {…
  } assume {
     property inrange is {
        via location::fromCoordinate receive sendercoordinate
        via location::toCoordinate receive receivercoordinate
        ask distanceofsensors is {
           sendercoordinate::distance(receivercoordinate)
             < distancebetweengates
           // the distance between the sender and
           // the receiver must be smaller than
           // the parameter of the mediator
        }
     }
   } guarantee {
     protocol transmitpact is {
        repeat {
           via transmit::fromSensors receive measure
           via transmit::towardsGateway send measure
        }
     }
  }
```

```
architecture WnsMonitoringSosArchitecture() is {…
  behavior coalition is compose {
     sensors is sequence{Sensor}
     gateway is Gateway
     transmitters is sequence{Transmitter}
  } binding {…
     forall{isensor1 in sensors, isensor2 in sensors
        suchthat
           exits{itransmitter in transmitters
              suchthat
                 (isensor1 <> isensor2) implies
                 unify one{itransmitter::fromSensors}
                      to one{isensor1::measurement::measure}
                 and unify one{itransmitter::towardsGateway}
                      to (one{isensor2::measurement::pass}
                 xor unify one{itransmitter::towardsGateway}
                      to one{gateway::notification::measure}
           }
     // multiplicities are 'one', 'none',
     // 'lone' (none or one),
     // 'any' (none or more),
     // 'some' (one or more), 'all'
     }
} guarantee {…}
```

Fig. 3. Excerpt of an SoS Architecture Description in SosADL

If not, it does nothing, but continue to scan the power level of the battery to check if the remaining power became greater than the threshold. Whenever it is the case, it will be able to operate again.

Let us now formulate the guarantees of sensor constituent systems that have been formalized in SosADL. Every sensor node guarantees, by its behavior, that, if operating, it will always be able to transmit measures (own measures or ones from its neighbors) via the connection *measure* of *measurement* gate. It is the case whenever the power available in the sensor node is greater than the power threshold, and if not, it will wait for the battery to sufficiently recharge again (in the case of the Monjolinho river, some sensor nodes include batteries charged by solar panels). Once recharged, it will continue to operate as guaranteed.

For the sake of brevity, we will not present the description of the *Gateway* abstract constituent system.

Let us define now, in the middle of *Fig. 3*, a *Transmitter* mediator, whose purpose is to transmit measures from each sensor towards the gateway using multihop transmissions. It has duties to be fulfilled by gates of constituent systems. In the *Transmitter* mediator, the purpose is to bind on the one hand with a constituent system having a gate that will commit to fulfill the duty providing measures to the mediator through connection *fromSensors* and on the other hand with another constituent system having a gate for consuming measures from the mediator through connection *towardsGateway*. The commitment of the mediator itself (see middle of *Fig. 3*) is to forward from the input connection to the output one without any processing in-between, if and only if the distance calculated from the GPS coordinates of these two bound sensors is in the range of transmission of the mediator.

Let us now describe the SoS abstract architecture focusing in the declaration of the coalition, expressing the policies defining possible bindings between mediated constituents.

As shown in the bottom of *Fig. 3*, a coalition may involve possibly many sensors, exactly one gateway and possibly many transmitters. This coalition does not specify which constituent systems will exist at run-time, but simply what are the specifications of possible systems that may exist and which are the required conditions for forming a coalition among the systems identified at run-time to participate in the SoS. It does not specify either which concrete system will be connected to which concrete system through a mediator. It simply specifies what gates can fulfill what duties.

The SoS architecture is thereby intentionally described. This suitably copes with the five intrinsic SoS characteristics.

B. Creating concrete SoS architectures with SosADL

For constructing the initial concrete architecture, mediators are synthesized taking into account the range of data transmission and the geographical location (based on GPS coordinates) of constituent systems. First the initial coalition is created from selected sensors and gateways available at the Monjolinho river. Then, it evolves to continuously achieve its mission, automatically supporting the re-architecture of the SoS for satisfying the set of told constraints (see [10] for details on the constraint solving mechanism and how concrete architectures are created and evolve).

VII. The SosADL Implementation and Validation

A major impetus behind developing formal languages for describing SoS architectures is that their formality renders them suitable to be manipulated by software tools. The usefulness of an ADL is thereby directly related to the kinds of tools it provides to support architectural description, but also analysis and evolution, in particular in the case of SoSs. In addition, tools pave the way to apply formal ADLs in real-scale projects.

A. The SoS Architecture Development Environment

We have developed an SoS Architecture Development Environment (ADE) for supporting architecture-centric formal development of SoSs using SosADL. This toolset, called SosADE, is constructed as plugins in Eclipse Mars. It provides a model-driven architecture development environment where the SosADL meta-model is defined in EMF (with the textual concrete syntax expressed in Xtext, the graphical concrete syntax developed in Sirius, and the type checker implemented in Xtend, after having being proved with Coq) is transformed to different meta-models and converted to input languages of analysis tools, e.g. UPPAAL for model checking, DEVS for simulation, and PLASMA for statistical model checking. In particular we have conceived a novel logic, named DynBLTL [28], for expressing correctness properties of dynamic software architectures and verifying these properties with a statistical model checking method [3].

B. Validating SosADL in a Field Study (in vivo)

SosADL, supported by its toolset, has been applied in different case studies and pilot projects for architecting SoSs. Especially, it has been applied for architecting the Flood Monitoring and Emergency Response SoS partially presented in this paper. Deployed in the Monjolinho river crossing the City of Sao Carlos, Brazil, it provided a real set for validating SosADL and assessing its toolset.

Let us now briefly present this field (in vivo) study in *Table III*. As stated in this table, the aim of this field study was to assess the fitness for purpose and the usefulness of SosADL to support the architectural design of real SoSs.The designed SoS abstract architecture was described in SosADL, edited using the SoS architecture *editor* (note that the excepts of the SoS architecture description in this paper are screen captures of the editor), validated using the SoS architecture *validator* (by simulation in DEVS), had its assume-guarantee properties verified using the SoS architecture *verifier* (by model checking in UPPAAL), and analysis of extreme cases due to the uncertainties of the river environment and actual availability of the constituent systems using the SoS architecture *analyzer* [3] (by statistical model checking in PLASMA).

The concretization of the SoS abstract architecture for the concrete case of the Monjolinho river was based on the actual installed sensors and gateway station. It was performed using the SoS architecture *constructor* (based on the Kodkod SAT-solver [10]). Using the architecture-based *synthesizer*, we generated a concrete implementation of the SoS. Finally, the SoS architecture-based *evolver* has been applied to assess the support for the evolutionary development of this SoS.

TABLE III. FIELD STUDY OF SOSADL

Architecting a Flood Monitoring and Emergency Response SoS	
Purpose	The aim of this field study related to the development of a Flood Monitoring and Emergency Response SoS was to assess the fitness for purpose and the usefulness of SosADL to support the architectural design of real-scale SoSs.
Stakeholders	The SoS stakeholder is DAEE (Sao Paulo's Water and Electricity Department), a government organization of the State of Sao Paulo, responsible for managing water resources, including flood monitoring of urban rivers. This SoS also involves as stakeholders the different city councils crossed by the Monjolinho river, the policy and fire departments of the city of Sao Carlos that own Unmanned Aerial Vehicles (UAVs) and have cars equipped with Vehicular Ad-hoc Networks (VANETs). Also are involved the hospitals of the city of Sao Carlos (with ambulances equipped with VANETs). The population, by downloading an App from DAEE, is also involved as target of the alert actions. They may also register for getting alert messages by SMS.
Mission	The mission of this SoS is to monitor potential floods and to handle related emergencies: detection of imminent floods and warning of citizens in risky areas.
Problem statement	In this field study, the wireless sensor network for urban river monitoring was enhanced with aerial and terrestrial vehicles (for the sake of brevity, not shown in the subset presented in this paper): UAVs and VANETs. UAVs (microcopters with eight propellers and a camera) have as mission to enforce the resilience of the monitoring SoS, i.e. to maintain an acceptable level of service in the face of energy shortages and failures to normal operation, by serving as router or by serving as sensor data mule. They may also transmit images in real time for reducing the risks of false positives. In addition, the UAVs may provide data dissemination to VANETs embedded in vehicles crossing the risky areas, thereby ensuring that vehicles driving towards a possible flood area can be warned to avoid certain roads.
Constituents	Constituent systems are sensor nodes and a gateway in a Wireless Sensor Network (WSN); Unmanned Aerial Vehicles (UAVs); Vehicular Ad-hoc Networks (VANETs); SMS multicasting; DAEE Apps in Smartphones. Note that transmitter mediators are dynamically created to maintain the connectivity of WSN.
Emergent behaviors	In order to fulfill its mission, this SoS needs to create and maintain an emergent behavior where sensor nodes (each including a sensor mote and an analog depth sensor) and microcopters (each including communication devices) will coordinate to enable an effective monitoring of the critical areas of the river and whenever a risk of flood is detected, to prepare the emergency response for vehicles approaching the flood area and inhabitants that live in potential flooding zones.
SoS architecture	The SoS architecture was described in SosADL as a collaborative SoS: the architecture self-organize itself based on mediators for connecting sensors and forming multihop ad-hoc networks, using UAVs when needed, as well as always being ready to send flood alert messages towards VANETs or inhabitants.

The result of this field study (of which some excepts have been presented in this paper) has showed that SosADL met the requirements for describing real-scale SoS architectures. As expected, using a formal ADL compels the SoS architects to study different architectural alternatives and take key architectural decisions based on SoS architecture analyses.

Learning SosADL in its basic form was quite straightforward; however, using the advanced features of the language needed interactions with the SosADL expert group. The SoS architecture *editor* and the *validator* were in practice the key tools in the learning and use of SosADL and the *verifier* and *analyzer* were the key tools to show the added value of formally describing SoS architectures.

In fact, a key identified benefit of using SosADL was the ability, by its formal foundation, to validate and verify the studied SoS architectures very early in the application lifecycle with respect to the SoS correctness properties, in particular for studying the extreme conditions in which emergent behaviors were not able to satisfy the SoS mission.

VIII. RELATED WORK

Software-intensive SoS is yet in its infancy. According to [9], ca. 75% of the publications addressing Software-intensive SoSs appeared in the last 5 years and ca. 90% in the last 10 years.

We carried out a Systematic Literature Review (SLR)[2] to establish the state-of-the-art on architecture description of SoSs [9], which permitted to collect, evaluate, and summarize related work addressing the following research question: *which modeling languages (including ADLs) have been used to describe SoS architectures from the behavioral viewpoint?*

As a result of the SLR [9], the following modeling languages have been identified as the main ones used for SoS architecture description: UML [31] (semi-formal), SysML [30] (semi-formal), and CML [17] (formal). These findings are compatible with the findings of another SLR conducted independently, reported in [13].

SysML is an extension of UML for systems modeling. It was the baseline of the European FP7 projects COMPASS [4] and DANSE [5] for which they developed extensions of SysML for SoS.

DANSE did not develop an ADL, but used SysML to semi-formally describe executable SoS architectures that can then be tested against interface contracts. The tests are applied to traces (obtained by executing SoS architectures) against interface contracts expressed on GCSL (Goal Contract Specification Language) [5].

COMPASS developed a formal approach, in contrast to DANSE that extended a semi-formal one. In COMPASS, CML [20] was specifically designed for SoS contract modeling and analysis.

CML is not an ADL. It is a contract-based formal specification language to complement SysML: SysML is used to model the constituent systems and interfaces among them in an SoS and CML is used to enrich these specifications with

[2] We conducted automatic searches on the major publication databases related to the SoS domain (IEEE Xplore, ISI Web of Science, Science Direct, Scopus, SpringerLink, and ACM Digital Library), after having defined the SLR protocol (see [9] for details on the SLR).

interface contracts. A CML model is defined as a collection of process definitions (based on CSP/Circus [20]), which encapsulate state and operations written in VDM (Vienna Development Method) [20] as well as interactions via synchronous communications.

CML is a low-level formal language, of which a drawback (stated by their authors) is that SysML models when mapped to CML produce huge unintelligible descriptions (it was one of the lessons learned from COMPASS [20]).

Compared with CML, SosADL enables the formal description of the whole SoS architecture, while CML does not, focusing only on contracts of interactions. Moreover, SosADL subsumes CML in terms of expressive power by its mathematical foundation based on the π-Calculus for SoS [23], subsuming CSP/Circus and providing an assertion language with an expressive power equivalent to VDM.

Besides formalisms specifically conceived for SoS, other formalisms have been proposed in the literature for modeling Service-Oriented Architectures (SOA), that share some concerns with SoS, including different variants of process calculi of services [32], e.g. SCC [32], SRML [32], COWS [32], and SOCK [32]. These languages have been designed to address different levels of abstractions; some of them (such as SRML) have focused on the representation of composite services from the business modeling perspective, and others (such as COWS, SCC, and SOCK) focused on the mathematical foundation perspective. Regarding architectural characteristics, the former ones provide constructs with a level of abstraction that is too high (i.e. hiding most of the architectural features) and the latter ones too low (i.e. where architectural features are only partially exposed inside many non-relevant details). That is, they do not provide the right level of abstraction for both describing an architecture in its own right and verifying the architectural properties.

Complementary to architecture, a new generation of programming languages and component frameworks have been designed to develop a specific class of SoS, the so-called "ensembles" (an SoS that is only composed of homogeneous systems) [33], in particular DEECo (Dependable Ensembles of Emerging Components) and SCEL (Service Component Ensemble Language) [33]. Both can be used to develop the implementation of SoS architectures designed with SosADL, i.e. SoS architectures described and analyzed with SosADL can be transformed into implementation models using DEECo and SCEL.

In summary, based on the study of the state-of-the-art carried out through this SLR [9], SosADL is positioned as the first ADL having the expressive power to formally describe SoS architectures from both structural and behavioral viewpoints, no existing ADL being able to express the evolutionary behavioral of SoS architectures [9][13]. Formally, by its mathematical foundation for behavior modeling, SosADL subsumes CML and complements different works on mission specifications, e.g. GCSL, while filling the gap between high-level semi-formal SoS architecture descriptions and low-level SoS modeling formalisms used as input for automated analysis. Regarding detailed design and implementation in specific styles, it is complementary to formalisms developed for SOA design as well as for "ensemble" programming.

IX. Conclusion and Future Work

This paper presented SosADL, an ADL specifically conceived for describing SoS architectures, focusing in the concepts and constructs supporting the description of SoS architectures from the behavioral viewpoint. This novel ADL enables the description of evolutionary architectures, which maintain emergent behavior supporting on-the-fly coalitions for sustaining SoS missions [10][29].

SosADL brings major contributions beyond the state-of-the-art: (i) it is the first full formal ADL having the expressive power to address the challenge of describing evolutionary architectures of Software-intensive SoSs from both structural and behavioral viewpoints; (ii) it enables to describe SoS software architectures abstractly at design-time without knowing which will be the actual concrete systems in the SoS at run-time; (iii) it enables to select and dynamically create concrete architectures grounded on its dynamic constraint solving mechanism.

More precisely, from the behavioral viewpoint, SosADL is the first ADL having the expressive power to describe the evolutionary behavior of SoS architectures. It enables: (i) to specify the assumptions and guarantees of behaviors of constituent systems; (ii) to specify the conditions in which mediators need to be created/synthesized to connect constituent systems to meet specific goals as well as the assumptions and guarantees of mediating behaviors; (iii) to specify the constraints for composing coalitions for combined action among constituent systems connected via inferred mediators. In particular, coalitions are dynamically formed to fulfill the SoS mission through emergent behaviors drawn by mediating local interactions among constituent systems.

By its formal underpinnings, SosADL supports automated verification of behavioral correctness properties of SoS architectures and supports behavioral validation through simulation of executable specifications. It is particularly important in the architectural design of trustworthy SoSs.

SosADL has been applied in several case studies and pilot projects. Particularly, it was the subject of a field study in the development of a real Flood Monitoring and Emergency Response SoS (partially presented in this paper), where the suitability of the language itself and the supporting toolset has been validated.

On-going and future work is mainly related with the application of SosADL in industrial-scale projects. They include joint work with DCNS for applying SosADL to architect naval SoSs, with IBM for applying SosADL to architect smart-farms in cooperative settings, and with SEGULA for applying SosADL to architect SoSs in the transport domain.

References

[1] Allen, R.; Garlan, D.: "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 6 (3), July 1997, pp. 213-249.

[2] Cavalcante, E.; Batista, T.V.; Oquendo, F.: "Supporting Dynamic Software Architectures: From Architectural Description to Implementation", *Proc. of the 12th Working*

IEEE/IFIP Conference on Software Architecture (WICSA), Montreal, Canada, May 2015 pp. 31-40.

[3] Cavalcante, E.; Quilbeuf, J.; Traonouez, L.M.; Oquendo, F.; Batista, T.V.; Legay, A.: "Statistical Model Checking of Dynamic Software Architectures", *Proc. of the 10th European Conference on Software Architecture (ECSA)*, LNCS, Springer, Copenhagen, Denmark, September 2016.

[4] COMPASS: Comprehensive Modelling for Advanced Systems of Systems, http://www.compass-research.eu

[5] DANSE: Designing for Adaptability and Evolution in System-of-Systems Engineering, http://www.danse-ip.eu

[6] Degrossi, L.C. et al.: "Using Wireless Sensor Networks in the Sensor Web for Flood Monitoring in Brazil: Lessons Learned", *Proc. of the 10th International Conference on Information Systems for Crisis Response and Management (ISCRAM)*, Baden-Baden, Germany, May 2013, pp. 1-5.

[7] Firesmith, D.: *Profiling Systems Using the Defining Characteristics of Systems of Systems (SoS)*, Software Engineering Institute, SEI Technical Report: CMU/SEI-2010-TN-001, February 2010, 87 p.

[8] Foster, H. et al.: "Specification and Analysis of Dynamically-Reconfigurable Service Architectures", *Rigorous Software Engineering for Service-Oriented Systems*, Springer LNCS 6582, 2011, pp. 428-446.

[9] Guessi, M.; Nakagawa, E.Y.; Oquendo, F.: "A Systematic Literature Review on the Description of Software Architectures for Systems-of-Systems", *Proc. of the 30th ACM Symposium on Applied Computing (SAC)*, Salamanca, Spain, April 2015, pp. 1-8.

[10] Guessi, M.; Oquendo, F.; Nakagawa, E.Y.: "Checking the Architectural Feasibility of Systems-of-Systems using Formal Descriptions", *Proc. of the 11th IEEE System-of-Systems Engineering Conference (SoSE)*, Kongsberg, Norway, June 2016.

[11] ISO/IEC/IEEE 42010:2011: *Systems and Software Engineering – Architecture Description*, December 2011, 46 p.

[12] Johnson, C.W. (Ed.): "Complexity in Design and Engineering", *Reliability Engineering & System Safety*, 91 (12), December 2006, pp. 1475-1588.

[13] Klein, J.; van Vliet, H.: "A Systematic Review of System-of-Systems Architecture Research", *Proc. of the 9th International Conference on Quality of Software architectures (QoSA)*, Vancouver, Canada, June 2013, pp. 13-22.

[14] Maier, M.W.: "Architecting Principles for Systems-of-Systems". *Systems Engineering*, Vol. 1 (4), 1998, pp. 267-284.

[15] Malavolta, I. et al.: "What Industry Needs from Architectural Languages: A Survey", *IEEE Transactions on Software Engineering*, Vol. 39, No. 6, June 2013, pp. 869-891.

[16] Medvidovic, N.; Taylor, R.: "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, Vol. 26 (1), January 2000, pp. 70-93.

[17] Mesli-Kesraoui, S.; Kesraoui, D.; Oquendo, F.; Bignon, A.; Toguyeni, A.; Berruet, P.: "Formal Verification of Software-intensive Systems Architectures described with Piping and Instrumentation Diagrams", *Proc. of the 10th European Conference on Software Architecture (ECSA)*, LNCS, Springer, Copenhagen, Denmark, September 2016.

[18] Milner, R.: *Communicating and Mobile Systems: The π-Calculus*, Cambridge University Press, 1999, 174 p.

[19] Nakagawa, E.Y.; Gonçalves, M.B.; Oquendo, F. et al.: "The State of the Art and Future Perspectives in Systems-of-Systems Software Architectures", *Proc. of the 1st ACM International Workshop on Software Engineering for Systems-of-Systems (SESoS)*, Eds. F. Oquendo et al., Montpellier, France, July 2013, pp. 13-20.

[20] Nielsen, C.B. et al.: "Systems-of-Systems Engineering: Basic Concepts, Model-based Techniques, and Research Directions", *ACM Computing Survey*, Vol. 48 (2), September 2015, pp. 1-41.

[21] Oquendo, F.: "π-ADL: Architecture Description Language based on the Higher-order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures", *ACM SIGSOFT Software Engineering Notes*, Vol. 29 (3), May 2004, pp. 1-14.

[22] Oquendo, F.: "Formally Describing the Software Architecture of Systems-of-Systems with SosADL", *Proc. of the 11th IEEE System-of-Systems Engineering Conference (SoSE)*, Kongsberg, Norway, June 2016.

[23] Oquendo, F.: "π-Calculus for SoS: A Foundation for Formally Describing Software-intensive Systems-of-Systems", *Proc. of the 11th IEEE System-of-Systems Engineering Conference (SoSE)*, Kongsberg, Norway, June 2016.

[24] Oquendo, F.: "The π-Calculus for SoS: Novel π-Calculus for the Formal Modeling of Software-intensive Systems-of-Systems", *Proc. of 38th International Conference on Communicating Process Architectures 2016 (CPA)*, Copenhagen, Denmark, August 2016.

[25] Oquendo, F.: "Software Architecture Challenges and Emerging Research in Software-intensive Systems-of-Systems", *Proc. of the 10th European Conference on Software Architecture (ECSA)*, LNCS, Springer, Copenhagen, Denmark, September 2016.

[26] Oquendo, F.: "Case Study on Formally Describing the Architecture of a Software-intensive System-of-Systems with SosADL", *Proc. of 15th IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Budapest, Hungary, October 2016.

[27] Ozkaya M.; Kloukinas C.: "Are We There Yet? Analyzing Architecture Description Languages for Formal Analysis, Usability, and Realizability", *Proc. of the 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Santander, Spain, September 2013, pp. 177–184.

[28] Quilbeuf, J.; Cavalcante, E.; Traonouez, L.M.; Oquendo, F.; Batista, T.V.; Legay, A.: "A Logic for Statistical Model Checking of Dynamic Software Architectures", *Proc. of the 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA)*, Springer, Corfu, Greece, October 2016.

[29] Silva E.; Batista, T.; Oquendo, F.: "A Mission-Oriented Approach for Designing System-of-Systems", *Proc. of the 10th System-of-Systems Engineering Conference (SoSE)*, San Antonio, TX, USA, May 2015, pp. 346-351.

[30] SysML – OMG Standard: *Systems Modeling Language*, http://www.omg.org/spec/SysML

[31] UML – OMG Standard: *Unified Modeling Language*, http://www.omg.org/spec/UML

[32] Wirsing, M.; Hölzl, M. (Eds.): *Rigorous Software Engineering for Service-Oriented Systems*, Springer, 2015, 748 p.

[33] Wirsing, M. et al. (Eds.): *Software Engineering for Collective Autonomic Systems*, Springer, 2015, 537 p.