Supporting Simulation of Systems-of-Systems Software Architectures by a Model-Driven Derivation of a Stimulus Generator

Valdemar Vicente Graciano Neto[1,2,3], **Carlos Eduardo Barros Paes**[2,3], **Flavio Oquendo**[1], **Elisa Yumi Nakagawa**[3]

[1]Instituto de Informática
Universidade Federal de Goiás (UFG)
Goiânia – GO – Brazil

[2]Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP) – São Carlos – SP – Brazil

[3]Departamento de Computação – Pontifícia Universidade Católica de São Paulo (PUC-SP) – São Paulo – SP – Brazil

[4]IRISA-UMR CNRS/Université de Bretagne-Sud – Vannes – France

`valdemarneto@inf.ufg.br, carlosp@pucsp.br`

`flavio.oquendo@irisa.fr, elisa@icmc.usp.br`

Abstract. *Systems-of-Systems (SoS) often support critical domains such as air traffic control, and emergency and crisis response. Hence, their software architectures must be validated, guaranteeing that they conform to their specification. However, SoS exhibit dynamic properties that bring difficulties to a static validation. In this direction, simulations can aid by offering a dynamic view for software architecture specifications of SoS. However, to be reliable, simulation models must reproduce the real conditions of the surrounding environment in which it will be deployed. Conversely, manually stimulating simulations is tiring, repetitive, and error-prone. Then, we explored the possibility of automatically deriving a 'stimulus generator', that continuously emits stimulus for the entities being simulated, supporting an early identification of failures, and enabling correction before the deployment. In this direction, we present our method to automatically derive a stimulus generator from a software architecture description of SoS. The main contribution of this paper is providing such method, suppressing the necessity of manually coding it. We evaluated our proposal with an example of a real Flood Monitoring SoS. Preliminary results point out that the stimulus generator automatically produced is reliable, emitting the expected outputs, and suitably triggering the simulation.*

1. Introduction

Software systems have become increasingly complex, forming alliances known as Systems-of-Systems (SoS)[1], and supporting critical domains. Due to that, SoS has a

[1]For sake of simplicity, SoS will be used interchangeably to express singular and plural.

potential to cause damages, losses, and hazards. They must be constructed to be trustworthy, i.e., their operation must reliable in a way that people can safely rely and trust on their success to accomplish their missions correctly, without failures, not causing accidents, working as expected, and rearranging themselves to keep their execution [Oquendo and Legay 2015, Graciano Neto et al. 2016]. Then, it is imperative supporting the SoS software development life cycle, in particular Verification and Validation (V&V) activities, guaranteeing that the SoS yield the results expected by the missions assigned to them, with an early identification of problems in SoS operation, assuring that the SoS performs exactly what it is intended to [Michael et al. 2009].

Simulations are a recurrent approach in SoS Engineering (SoSE) to support such early anticipation of failures [Himmelspach and Uhrmacher 2007, Michael et al. 2009, Sauser et al. 2010, Zeigler et al. 2012, Mittal and Rainey 2015, Wachholder and Stary 2015]. In particular, simulations are useful for SoS, since they can externalize the inherent dynamics associated to SoS operation. However, to be reliable, SoS simulations must reproduce the real conditions in which it will be deployed. A manual approach to reproduce such stimulus, which consists of stimulating the simulation with inputs during its run-time, is costly, error-prone, tedious, and not correspondent to the real rhythm in which the stimulus could be received. Additionally, manually coding a stimulus generator is repetitive and domain-dependent, since each new SoS requires a distinct stimulus generator. In order to reduce the costs associated to its engineering, a possibility is automating the creation of such stimulus generator by a model-driven derivation based on the SoS architectural descriptions that inherently stores information about the inputs and outputs expected by the SoS.

In this paper we present specifically how to automatically derive a stimulus generator for simulation purposes of SoS software architectures by means of a model transformation. We document our SoS architecture in a high-level of abstraction using SosADL, a new architectural description language (ADL) specially tailored for SoS [Oquendo 2016]. Furthermore, we adopt DEVS (a standard formalism for simulation approaches in SoSE community) as the dynamic viewpoint to simulate SoS. Indeed, we establish a transformation from SosADL to DEVS (SosADL2DEVS) to deal with simulations of software architectures of SoS while preserving their specification. The main contribution of this paper is presenting such method, externalizing how to proceed with this transformation, extracting such information about the stimulus from an architectural description of SoS to support the creation of stimulus generators. We evaluated our proposal with a small-scale instance of a Flood Monitoring SoS. Preliminary results are promising, with an effective generation of a functional simulation. Our method brings straightforward derivation of simulation code and traceability between software architecture and simulation models. This paper is structured as follows: Section 2 outlines the concepts involved in our proposal. Section 3 distills our method. Section 4 presents a brief evaluation we carried out. Section 5 brings final considerations and points out for forthcoming work.

2. Simulation of SoS Software Architectures

Simulations are a recognized approach to deal with SoS dynamicity. They correspond to an imitation of the operation of a real-world process or system over time, and involve the generation of artificial stimulus and the observation of the effects to draw inferences concerning the operational characteristics of the real system that is represented [Banks 1999].

They (i) provide a visual and dynamic viewpoint for SoS software architectures, reproducing stimulus the system can receive from a real environment, (ii) enable the prediction of errors, diagnosing them and permitting corrections, and (iii) support the observation of expected and unexpected emergent behaviors of an SoS. In fact, simulations are one of the main groups of evaluation approaches for software architectures, typically relying on a high level implementation of the software architecture for evaluating their performance and accuracy [Bosch 2000, Santos et al. 2014]. SoS exhibit a further degree of complexity due to their dynamic nature, i.e., their operation, in particular emergent behaviors, are not visible in static specifications such as conventional software architecture documentations that adopts diagrams in UML or SySML. Thus, SoS software architectures demand an additional view that captures dynamic aspects of SoS operation.

Software architectures are often described through modern Architectural Description Languages (ADL) that offer canonical constructs to properly specify those architectures. Their formalism levels include informal (usually based on lines, rectangles, and figures denoting structures), semi-formal, and formal [Garlan et al. 2010]. Remarkable examples include Darwin (semi-formal) [Foster et al. 2011], Wright (formal) [Allen and Garlan 1997], π-ADL (formal) [Oquendo 2004], UML[2] (semi-formal) and SySML[3] (semi-formal). However, those ADL have not been developed for properly capturing SoS' dynamics [Guessi et al. 2015]. Recently, a novel ADL called SosADL was proposed for describing the architecture of SoS. It provides architectural concepts and notation formally defined in terms of the π-calculus, and also supports the specification of emergent behaviors [Oquendo 2016]. However, SosADL is not executable yet, and a dynamic approach is still required to support a plain visualization of SoS operation. Hence, an approach to support dynamic aspects of SoS software architectures descriptions is required since a long time and it is still an issue [Zave 1993, Sauser et al. 2010, Graciano Neto and Nakagawa 2015].

In this direction, simulations can offer such dynamic approach for visualization of SoS operation. Nevertheless, simulations often depend on some internal structure that imitates the surrounding environment of an SoS, delivering the stimulus that are supposed to be received by the SoS to trigger its operation [INCOSE 2016]. A stimulus generator is a virtual simulation entity responsible for playing the role of environment, such as temperature, wind, water level, and noise, or an entity which produces internal events in the systems, i.e., it imitates the reception of inputs that the constituent system can produce to itself, such as the collecting of an external data, the level of battery, of its geographic position. Developing such structure usually relies on (i) an specification of ports, inputs, outputs, and state diagram formalism in a low level of abstraction, (ii) a distinct stimulus generator for every different SoS that we produce, and (iii) a careful investigation on SoS requirements and architecture specification to elicit which stimulus should be provided, which can be costly, and error-prone. In fact, the development of stimulus generators for simulation purposes is not a new trend [Al-Hashimi 1995, Kitchen and Kuehlmann 2007]. Meanwhile, such approaches for automatically creating stimulus generator for simulation of SoS software architectures have not emerged.

Recent studies have investigated the adoption of simulation in software engineer-

[2]http://www.omg.org/spec/UML/2.5/
[3]http://www.omg.org/spec/SysML/1.4/

ing [de França and Travassos 2015, de França and Travassos 2016], and simulation has certainly been applied for SoS development [Xia et al. 2013, Graciano Neto et al. 2014, Bogado et al. 2014]. Regarding simulation of software architectures, Palladio[4] offers a solution [Becker et al. 2009]. However, there is no support for simulation of SoS software architectures. Discrete Event Systems Specification (DEVS) is a well-established formalism for simulating SoS in virtual environments [Zeigler et al. 2012]. However, it does not preserve the architectural details of SoS software architecture specification and rely in a low-level of abstraction formalism. Bogado et al. also rely on discrete event formalism for representing software architectures of monolithic software [Bogado et al. 2014], whilst Alexander et al. also deal with simulation of software architecture and dynamic aspects, but they do not address software architectures of SoS in a strict way, relying on the broader discussion of systems architecture perspective [Alexander et al. 2015].

2.1. SosADL and DEVS

In short, SosADL describes SoS, which can be expressed as a combination of architecture declarations, systems declarations, and mediators declarations[5]. An architecture declaration has an intrinsic behavior declaration, data types, and gates declarations. Gates are abstractions that enable the establishment of connections. A connection can be established to receive stimulus from or act on the environment, or simply for a communication between constituents. Furthermore, the connection can be for input, output, or for both. Data types can have inherent functions, and functions can have expressions associated. Mediators and systems, as well as the SoS architecture itself, have gates, data types, and behaviors. Systems play the role of constituents in an Architecture Behavior Declaration, and Systems are mediated by mediators. SosADL supports emergent behavior representation by the idea of coalition, a temporary alliance for combined action among constituents connected by mediators. Those behaviors are specified as part of the coalition behavior, documenting how constituents should interact to accomplish a given set of missions[6].

In turn, DEVS is structured based on atomic and coupled models. Atomic models represent constituents, and coupled models represent the communications among constituents, materializing the SoS as a whole and constituents' interoperability. In DEVS, Atomic models have the following elements: (i) a labeled state diagram, that performs transitions due to input or output events; (ii) abstract data types definition, (iii) global variables definition, (iv) variables initialization, (v) ports definition, and (vi) events definition. An atomic model with only a state diagram specification and ports definition is already executable. Coupled models are expressed as a System Entity Structure (SES), i.e., a formal structure governed by a small number of axioms that expresses how atomic models communicate. A straightforward generation of DEVS code does not guarantee the simulation be executable. This happens because the SoS operation is deeply related to the stimulus received from the environment that triggers the accomplishment of a mission. Thus, it is necessary to elaborate a specific entity in the simulation model that is responsible for delivering the expected stimulus that drive the operation of the SoS: the stimulus generator. Next section discusses how to automatically produce such stimulus

[4]http://www.palladio-simulator.com/

[5]Mediators are architectural elements that establish communication between two or more constituents

[6]More details about the syntax of architecture descriptions in SosADL and its elements can be found in [Oquendo 2016].

generator from SosADL models, properly transforming SosADL in DEVS.

3. A Model-Driven Derivation of a Stimulus Generator for Simulation of SoS Software Architectures

We present our approach relying on a real example: a Flood Monitoring SoS (FMSoS), i.e., a SoS composed of smart sensors (i.e., complex sensors that embed software) to monitor the occurrences of floods in an urban area in the city of São Carlos in Brazil. Rivers cross the city and, when the rains are intense, floods recurrently occur, causing losses, damage, and iminent danger for population. The FMS we describe is concerned to validate a single emergent behavior: *flood alert*. Such SoS consists of a collaborative SoS, in which there is not a central authority that orchestrate the constituents' functionalities to accomplish missions. Figure 1 gives an illustration of the FMSoS[7]. Sensors are spread on the river's edges with a regular distance among them, and mediators exist between every pair of sensors in a pre-established distance between them. The data collected by sensors are collected and transmitted until reaching the gateway. In case of flood, the gateway emit an alarm for the public authorities. All of the codes presented henceforth are based on this example.

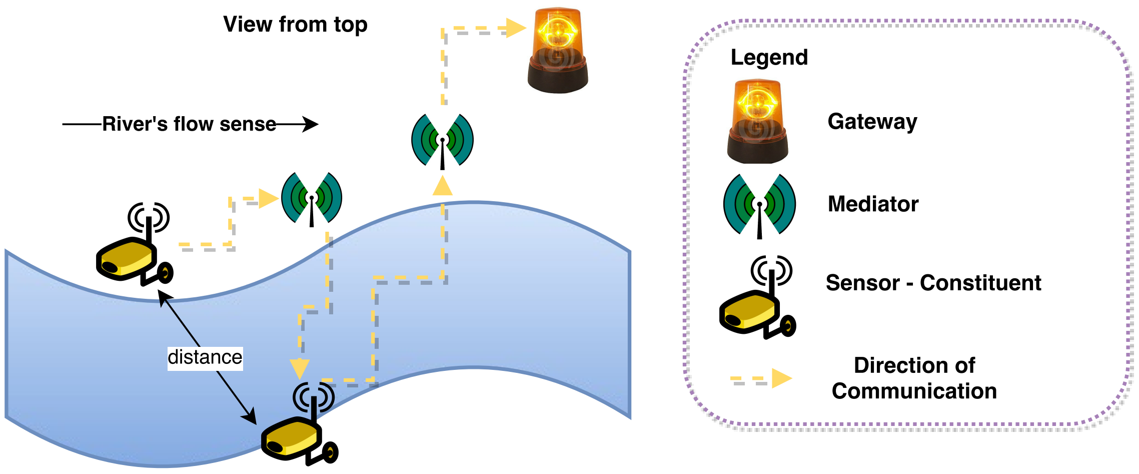

Figure 1. A Flood Monitoring System-of-System (FMSoS).

FMSoS is an SoS, since it exhibits [Maier 1998]:
Operational independence of the constituents, i.e., smart sensors perform their own missions despite being out of the scope of the SoS;
Managerial independence of constituents, since a diversity of stakeholders and enterprises might independently own, deliver, and manage the smart sensors;
Distribution, since they interoperate through a communication network;
Evolution, since the SoS evolves as a consequence of changes in the configuration or functionality of smart sensors; and
Emergent behavior, since one unique constituent could not deliver a flood alert by itself. For instance, if only one sensor performs its activities in an urban area, it could not notify a flood on time, being not effective. Conversely, if it is used outside of the urban area,

[7]Credits for the images used to compose the figure: http://goo.gl/TTOlAa, http://goo.gl/QCUAKY, http://goo.gl/a9Y0Dw.

trying to previously diagnostic the flood, it would be so far that could not reach a gateway to communicate to communicate the danger. Also, it might emit a false alert, since the flood could be limited to another place. Hence, the flood alert is a result of the interoperability among a diversity of constituents working in cooperation, spread along the river edge. Furthermore, we deal with a weak emergent behavior, since they can not be predicted by static models (simple emergent behaviors), and they are reproducible in simulations (but not in static models), emerging predicted behaviors but still with potential to non-predicted behaviors that must be handled.

We selected SosADL as the SoS software architecture specification notation, and DEVS as the simulation formalism. We adopted a DEVS dialect called DEVS Natural Language (DEVSNL) that enables programming atomic and coupled models in a human-like format in tools such as MS4ME[8]. A SosADL code is submitted as input for a XTend script that materializes the Code Generator. A functional code written in DEVSNL is generated as output, establishing a SosADL2DEVS transformation. Briefly, considering a broad view of the transformation, the concepts of System and Mediator in SosADL are transformed into Atomic Models in DEVS. Behaviors are materialized into labeled state diagrams in DEVS that configures the constituents operation. Architecture, Coalition, and SoS become Coupled Models. Connections and gates become Ports, and Data Types and Functions are mapped in data types and functions in DEVS.

Each statement (line of code) within a SosADL behavior is converted in one or more state transitions in DEVS. In DEVS, transitions can occur due to (i) a data received, expressed as `?data`, (ii) a data sent, expressed as `!data`, and (iii) a spontaneous transition, without any input or output. This is the approach we used to generate constituents, mediators, and gateway[9]. However, the derivation of the stimulus generator is quite different. In SosADL, there is a special type of connection called **environment**, that abstracts interaction of an SoS with the surrounding environment, emitting outputs to the environment, or receiving stimulus from it, e.g., when the system is a sensor. Moreover, SosADL offers a library called Localization, that offers invoking Global Positioning System (GPS) functionalities.

All of the SosADL aspects must be traduced to DEVSNL to create a functional simulation. However, there are no straightforward elements in DEVSNL to automatically produce environment stimulus. Thus, it is necessary to create a stimulus generator that delivers the expected inputs the constituents wait to perform transitions and to start their behavior execution. SosADL models are analyzed by the transformation algorithm, searching for environment connections and callings for localization libraries. For convention, constituents are analyzed first, since they are the frontier of a SoS, and state transitions for the stimulus generator are created to serve their necessities. After that, Mediators are investigated. Lastly, Gateways are analyzed. Since usually constituents starts the SoS operation due to the stimulus they receive from environment, we prioritize the analyzes of constituents. If there are congruent statements between constituents and mediators, i.e., a same type of stimulus that must be received for both, a unique transition is created in the stimulus generator. Each connection specified as an `environment` con-

[8]http://goo.gl/NmBBuu

[9]We do not discuss this mechanism with details in this paper, since the focus is the representation and derivation of a stimulus generator. Other details are discussed in a forthcoming paper.

nection produces one transition in the specification of the state diagram in the resulting stimulus generator. Hence, the stimulus generator consists of a special type of system (in the context of the simulation) that has a continuous behavior (a behavior materialized as a loop) to emit stimulus by output state transitions, starting and keeping the SoS in operation.

In DEVS, such stimulus generator is also represented as an Atomic Model. Listing 1 shows an excerpt of a code in SosADL that specifies one of the sensors that compose the FMSoS. Some parts are hidden since they do not influence in the discussion of stimulus generation derivation. It is possible to see that the gate energy offers two environment connections (Lines 12 and 13): one to receive a threshold (a limit of energy that is considered enough to keep the sensor in operation), and `power`, that is used to receive the level of battery available. Within the behavior `sensing`, it is possible to see the sensor receiving its coordinate and receiving the energy threshold and power level. In turn, Listing 2 shows the code in DEVSNL that specifies the stimulus generator produced using our approach. We changed the names of the states to become them more readable. It illustrates not only the transitions generated from the code of the sensor showed in Listing 1, but also from the codes of mediators and gateway (Lines 5 to 15). In Listing 2, the stimulus generator has three output ports (Lines 1 to 3) that simulates the collection of the geographic positions (lps), power level, and the reception of the water level by the mediators, sensors, and gateway. Figure 3 depicts a state diagram equivalent to the DEVS code presented in Listing 2. It delivers the aforementioned data, and comes back to the state `LPSsent`, forming a loop that keeps the stimulus continuously running and offering the stimulus for the operation of the SoS.

4. A Brief Evaluation

We adopted a specification in SosADL of a real Flood Monitoring SoS already in operation to evaluate our proposal [Horita et al. 2015]. Our aim was evaluating if the simulation (automatically produced) would run as expected and deliver, as a result, a single emergent behavior. A brief video demonstrating the generated simulation is available externally[10]. It shows the simulation running and the stimulus generator successfully delivering the outputs necessary to the simulation execution. Indeed, the approach was submitted to domain experts. They classify our approach as suitable to generate a stimulus generator that is reliable and correspondent to the stimulus specified in high-level in the SoS software architecture specification.

Regarding threats to validity, we can mention the possibility of failures if the SoS architect forget to qualify the environment connections in SosADL with the keyword `environment`. If it occurs, the simulation can fail, since the expected input can be never received. Indeed, any error regarding the declaration of environment connections at design-time can affect the final simulation. Moreover, more accurate evaluation with larger contexts and applications are still required.

5. Final Remarks

This paper presented a model-driven solution to automatically derive a stimulus generator to be used in a simulation of SoS software architectures. Our proposal contributes

[10]`https://goo.gl/pdGCIC`

Listing 1. A specification of a Sensor in SosADL.

```
//'with' imports declarations suppressed
// Description of Sensor as a System Abstraction
library WsnSensor is {
  system Sensor( lps:Coordinate ) is {
  // Declaration of local types hidden
  gate measurement is {
      connection pass is in { MeasureData }
      connection measure is out { MeasureData }
  }

    gate energy is {
      environment connection threshold is in { Energy }
      environment connection power is in { Energy }
    }
    gate location is {
      connection coordinate is out { Coordinate }
    }

    behavior sensing is {
          via location::coordinate send sensorcoordinate
      via energy::threshold receive powerthreshold
      repeat {
        via energy::power receive powerlevel
        if( powerlevel > powerthreshold ) then {
                  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
 } } }}}}
```

Listing 2. Code DEVSNL for a stimulus generator.

```
generates output on lps!
generates output on powerLevel!
generates output on sense!

to start hold in s0 for time 1!
after s0 output lps!
from s0 go to LPSsent!
hold in LPSsent for time 1!
after LPSsent output powerLevel!
from LPSsent go to powerLevelSent!
hold in powerLevelSent for time 1!
after powerLevelSent output sense!
from powerLevelSent go to
    waterLevelSent!
hold in waterLevelSent for time 10!
from waterLevelSent go to LPSsent!
```

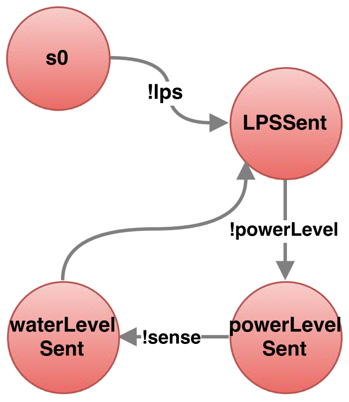

Figure 2. A State Diagram equivalent to the DEVS code generated for the stimulus generator of a FMSoS.

by automating the process of generation of that stimulus generator, bringing productivity, traceability between the models. This solution is a proposal of a joint effort of two research groups: SofTware ARchitecture Team (START/ICMC-USP) and ArchWare (IRISA/UBS), and is part of a broader approach to support validation of emergent behaviors in software architecture of SoS by the adoption of model driven derivation of a simulation of SoS. Future works include scaling the solution, testing the confidence of the model-driven transformation, and conducting experimental studies to evaluate our proposal.

Acknowledgments: We thank CNPq (grant:201230/2015-1/SWE) and FAPESP (grant: 2014/02244-7).

References

Al-Hashimi, B. (1995). *The Art of Simulation Using PSpice: Analog and Digital*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition.

Alexander, P., Nicolaescu, A., and Lichter, H. (2015). Model-based evaluation and simulation of software architecture evolution. ICSEA, pages 153 – 156.

Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249.

Banks, J. (1999). Introduction to simulation. In *Proceedings of the 31st Conference on Winter Simulation: Simulation—a Bridge to the Future - Volume 1*, WSC '99, pages 7–13, New York, NY, USA. ACM.

Becker, S., Koziolek, H., and Reussner, R. (2009). The palladio component model for model-driven performance prediction. *J. Syst. Softw.*, 82(1):3–22.

Bogado, V., Gonnet, S., and Leone, H. (2014). Modeling and simulation of software architecture in discrete event system specification for quality evaluation. *Simulation*, 90(3):290–319.

Bosch, J. (2000). *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. Addison-Wesley, New York, USA.

de França, B. B. N. and Travassos, G. H. (2015). Simulation based studies in software engineering: A matter of validity. *CLEI Electron. J.*, 18(1).

de França, B. B. N. and Travassos, G. H. (2016). Experimentation with dynamic simulation models in software engineering: planning and reporting guidelines. *Empirical Software Engineering*, 21(3):1302–1345.

Foster, H., Mukhija, A., Rosenblum, D. S., and Uchitel, S. (2011). *Rigorous Software Engineering for Service-Oriented Systems: Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*, chapter Specification and Analysis of Dynamically-Reconfigurable Service Architectures, pages 428–446. Springer, Berlin, Heidelberg.

Garlan, D., Bachmann, F., Ivers, J., Stafford, J., Bass, L., Clements, P., and Merson, P. (2010). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition.

Graciano Neto, V. V., Guessi, M., Oliveira, L. B. R., Oquendo, F., and Nakagawa, E. Y. (2014). Investigating the model-driven development for systems-of-systems. ECSAW '14, pages 22:1–22:8, New York, NY, USA. ACM.

Graciano Neto, V. V. and Nakagawa, E. Y. (2015). A biological inspiration to support emergent behavior in systems-of-systems development. WDES' 15, pages 33–40, Belo Horizonte, Brazil. SBC.

Graciano Neto, V. V., Oquendo, F., and Nakagawa, E. Y. (2016). Systems-of-systems: Challenges for information systems research in the next 10 years. GRANDSI-BR/SBSI, pages 1–3, Florianópolis, Brazil. SBC.

Guessi, M., Graciano Neto, V. V., Bianchi, T., Felizardo, K. R., Oquendo, F., and Nakagawa, E. Y. (2015). A systematic literature review on the description of software architectures for systems of systems. In *SAC*, pages 1433–1440, Salamanca, Spain.

Himmelspach, J. and Uhrmacher, A. M. (2007). Plug'n simulate. ANSS '07, pages 137–143, Washington, DC, USA. IEEE Computer Society.

Horita, F. E., de Albuquerque, J. P., Degrossi, L. C., Mendiondo, E. M., and Ueyama, J. (2015). Development of a spatial decision support system for flood risk management in Brazil that combines volunteered geographic information with wireless sensor networks. *Computers & Geosciences*, 80:84 – 94.

INCOSE (2016). The Guide to the Systems Engineering Body of Knowledge (SEBoK). SEBoK v. 1.6 released 25 March 2016.

Kitchen, N. and Kuehlmann, A. (2007). Stimulus generation for constrained random simulation. ICCAD '07, pages 258–265, San Jose, California. IEEE Press.

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

Michael, J. B., Riehle, R., and Shing, M. T. (2009). The verification and validation of software architecture for systems of systems. In *SoSE*, pages 1–6, Albuquerque, USA.

Mittal, S. and Rainey, L. (2015). Harnessing Emergence: The Control and Design of Emergent Behavior in System of Systems Engineering. In *SCS*, pages 1–10, San Diego, CA, USA. SCSI.

Oquendo, F. (2004). π-ADL: An Architecture Description Language Based on the Higher-order Typed π-calculus for Specifying Dynamic and Mobile Software Architectures. *SIGSOFT Softw. Eng. Notes*, 29(3):1–14.

Oquendo, F. (2016). Formally Describing the Software Architecture of Systems-of-Systems with SosADL. In *SOSE*, pages 1–6, Kongsberg, Norway. IEEE.

Oquendo, F. and Legay, A. (2015). Formal Architecture Description of Trustworthy Systems-of-Systems with SosADL. *ERCIM News*, (102).

Santos, D. S., Oliveira, B., , 1, M. G., Oquendo, F., Delamaro, M., and Nakagawa, E. Y. (2014). Towards the evaluation of system-of-systems software architectures. WDES, pages 53 – 57. SBC.

Sauser, B., Boardman, J., and Verma, D. (2010). Systomics: Toward a Biology of System of Systems. *IEEE Trans. on Systems, Man, and Cybernetics*, 40(4):803–814.

Wachholder, D. and Stary, C. (2015). Enabling emergent behavior in systems-of-systems through bigraph-based modeling. In *SOSE*, pages 334–339, San Antonio, USA. IEEE.

Xia, X., Wu, J., Liu, C., and Xu, L. (2013). A model-driven approach for evaluating system of systems. In *ICECCS*, pages 56–64.

Zave, P. (1993). Feature interactions and formal specifications in telecommunications. *Computer*, 26(8):20–29.

Zeigler, B. P., Sarjoughian, H. S., Duboz, R., and Souli, J.-C. (2012). *Guide to Modeling and Simulation of Systems of Systems*. Springer.