A simulation-driven model-based approach for designing softwareintensive systems-of-systems architectures

Valdemar Vicente Graciano Neto

▶ To cite this version:

Valdemar Vicente Graciano Neto. A simulation-driven model-based approach for designing softwareintensive systems-of-systems architectures. Multiagent Systems [cs.MA]. Université de Bretagne Sud; Universidade de São Paulo (Brésil), 2018. English. NNT : 2018LORIS489 . tel-02146340

HAL Id: tel-02146340
https://theses.hal.science/tel-02146340v1

Submitted on 3 Jun 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE / UNIVERSITE DE BRETAGNE SUD
sous le sceau de l'Université Bretagne Loire

pour obtenir le titre de
DOCTEUR DE L'UNIVERSITE BRETAGNE SUD

Mention : Informatique

Ecole doctorale: MathSTIC

Présentée par

Valdemar Vicente GRACIANO NETO

Préparée à l'unité mixte de recherche 6074
Institut de Recherche en Informatique et Systèmes Aléatoires
Université Bretagne Sud

Une approche dirigée par les simulations et basée sur les modèles pour la conception des architectures logicielles des systèmes-des-systèmes à logiciels prépondérants

Thèse soutenue le 27 mars 2018, devant le jury composé de :

M. Amar RAMDANE-CHERIF
Professeur des Universités, Université de Versailles Saint-Quentin, France
/ Rapporteur

Mme. Cecilia Mary Fischer RUBIRA
Professeur des Universités, Université de Campinas, Brésil
/ Rapporteur

M. Jair CAVALCANTI LEITE
Professeur des Universités, Université Fédérale du Rio Grande do Norte, Brésil
/ Examinateur

Mme. Jennifer PEREZ-BENEDI
Maître de Conférences HDR, Université Polytechnique de Madrid, Espagne
/ Examinateur

Mme. Elisa Yumi NAKAGAWA
Maître de Conférences HDR, ICMC, Université de São Paulo (USP), São Carlos, Brésil
/ Codirecteur de thèse

M. Flavio OQUENDO
Professeur des Universités, IRISA – Université Bretagne Sud, Vannes, France
/ Directeur de thèse

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: ______________________

Valdemar Vicente Graciano Neto

A simulation-driven model-based approach for designing software-intensive systems-of-systems architectures

Doctoral dissertation submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP and to the Université Bretagne Sud - UBS, Institut de recherche en informatique et systèmes aléatoires - IRISA, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics (ICMC-USP) and Doctorat en Informatique (UBS), in accordance with the international academic agreement for PhD double degree signed between ICMC-USP and UBS. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics / Computer Science

Advisor: Prof. Dr. Elisa Yumi Nakagawa (ICMC-USP, Brazil)
Advisor: Prof. Dr. Flavio Oquendo (IRISA-UBS, France)

USP – São Carlos
April 2018

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

G736s

Graciano Neto, Valdemar Vicente
A simulation-driven model-based approach for designing software-intensive systems-of-systems architectures / Valdemar Vicente Graciano Neto; orientador Elisa Yumi Nakagawa; coorientador Flavio Oquendo. -- São Carlos, 2018.
217 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2018.

1. Systems-of-Systems. 2. SoS. 3. Model-Based Engineering. 4. Software Architecture. 5. Simulation. I. Nakagawa, Elisa Yumi , orient. II. Oquendo, Flavio, coorient. III. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Valdemar Vicente Graciano Neto

Uma abordagem digirida por simulação e baseada em modelos para projeto de arquiteturas de sistemas de sistemas intensivos em software

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC–USP e à Université Bretagne Sud – UBS, Institut de recherche en informatique et systèmes aléatoires – IRISA, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional e Doutor em Informática (UBS), de acordo com o convênio acadêmico internacional para dupla titulação de doutorado assinado entre o ICMC-USP e a UBS. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional / Ciência da Computação

Orientadora: Prof. Dr. Elisa Yumi Nakagawa (ICMC-USP, Brazil)
Orientador: Prof. Dr. Flavio Oquendo (IRISA-UBS, France)

USP – São Carlos
Abril de 2018

To my lovely mother (in memoriam) and my grandparents, who founded my path until here.

ACKNOWLEDGEMENTS

It’s closing time again. And as it is broadly known, the best part of a journey is the path, not the final destination. I have been surrounded by phenomenal people during these almost five years of walking. I met many people who changed my life. By the way, *change* was one of the most constant words in the trajectory. I moved out five times during the doctorate course. And in all the changes, I found people with open hearts willing to welcome me.

I am thankful for many things and I thank many people. Maybe a chronological order is fair enough.

First of them, my entire family, who supported me during all these years; in special, my grandmother and grandfather, Glória and Amâncio, who are my foundations; and my mother (in memoriam), who would be so proud of what I achieved. After them, I must say thank you to prof. Dr. Gelson Cruz (EMC/UFG), who motivate me as a good friend to being a PhD. I also say thank you to prof. Dr. Juliano Lopes de Oliveira (INF/UFG) and prof. Dr. Auri Marcelo Rizzo Vincenzi (DC/UFSCar), who formally recommended me through official letters to the Postgraduate Program in Computer Science and Computational Mathematics of Institute of Computer Science and Computational Mathematics, University of São Paulo, which recently achieved the maximum grade in the quality scale of the Brazilian Commission for the Improvement of Higher Education Personnel (CAPES). I am so glad to finish my PhD in such a prestigious university and program.

I thank prof. Dr. Elisa Yumi Nakagawa, the kind and lovely woman who welcomed me with open arms and continuously taught me during all these years; who answered all my emails (with no exception), who answered emails after midnight, on holidays, and on weekends; who read my texts, made great contributions to improve them, and worked as a true partner in all endeavors. Thank you for your valuable trust and supervision.

To prof. Dr. Flavio Oquendo, a man of great simplicity, huge proficiency, wisdom, expertise, and knowledge. Thank you for the precious teachings all the time I have been in France. It was an incredible experience that has enriched me, and that I will hold for my entire life.

After them, I must say thank you to prof. Dr. Omar Khalil and prof. Me. João

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

Ricardo Braga. They prepared my entire documentation to apply for my current work at the Federal University of Goiás. Without them, I would not be there, and I will never forget what they did for me. I also thank all my friends in Formosa, Goiás, where I lived for almost two years (in special, Ítalo Dutra and Ariane).

Thank you, Alessandra Bueno, my analyst and friend, who supported me before and during my entire PhD. Without her, I would not have finished it.

I must also say thank to Foundation for Research Support of the State of Goiás (FAPEG), for the financial grant under number 2013.009.97100854, and National Council for Scientific and Technological Development (CNPq), for the financial grant and for supporting me for the international doctoral internship under grant number 201230/2015-1/SWE.

I say thank you to my colleagues in Federal University of Goiás, in particular prof. Dr. Vinícius Sebba Patto, and prof. Dr. Sérgio Teixeira de Carvalho, for the friendship, and for supporting me on the organization of the Brazilian Symposium on Information Systems (SBSI) in Goiânia in 2015. I also recall and thank prof. Dr. Les Foulds, who assisted me during my PhD, carefully reviewing my papers.

As I moved a lot, I have friends to say thank you in many places. My lovely Goiânia friends: José Camilo Cintra, Priscila Cassimiro, Wesley Crisóstomo, Luiz Loja, Danilo Oliveira, Sofia Larissa Costa, Arthur Gleyton (in memoriam), Fabiana Freitas, Patra Gomes.

My dear friends I have met in São Carlos: Leo Santorsula, Dona Zeza, Liara Rodrigues, Tamar Rafael, Caio Vinicius Reis, Joseph McCarthy, Tiago Medeiros, Tiago Volpato, Paulo Santos, Ronaldo Junior, Patricia Nunes, Gisa, Fabio, Leandro, Laura, Leonardo Oichenaz, Michel Fernando, Danilo Kutsmi, Grazi, Carol, Marieli, Victor Padilha, Jhonatan Lima, Patricia Righete. I also thank prof. Angela Giampedro, for the awesome English review in my texts and the thesis, itself.

Friends that I have met abroad: Pamela Carreño, Arthur Brenaut, Jared Barnett, Carlos Eduardo Paes, Daniela Tuffani, Mariano Zocine, Angelique Mangenot, Madis, Ségal, Manoj, Mathiew, William Andeole, Jamal El Hachem, Istvan David, Delphine.

Software Architecture Team (START), good friends for the entire life. You have a special place inside my heart: Milena Guessi, Lina Garcés, Lucas Bueno R. De Oliveira, Brauner Oliveira, Daniel Soares, Marcelo Benites, Ana Allian, Bruno Sena, Wallace Manzano, Gabriel Abdalla.

Friends from the Software Engineering Laboratory (Labes): Kamilla Takayama (thank you for being so kind and taking care of me when I was sick), Carlos Diego Damasceno

(thank you for the final sprint with press of my thesis copies), Danilo Reis, Livia Degrossi, Elias Nogueira, Ricardo Ramos, Rafael Durelli, Patricia Scandurra, Faimison Porto, Silvana Morita, Rachel Reis, Clausius. A unique word traduces you in my life: *smiles*.

My good friends spread in Brazil: Marcelo Cristian, Daniel Cancelier, Vitor Perciccote, Ju, Ivan, Paulo, Davi Viana, Rodrigo Pereira dos Santos, Everton Cavalcante, Fabio Basso, Sofia Costa.

Friends from Special Commitee on Information Systems of the Brazilian Computer Society (CE-SI/SBC - 2015-2017): Sean Siqueira, Renata Araujo, and Clodis Boscarioli.

Maybe my memory was unfair with some friend that I did not mention him/her here. But be sure that you are equally important to this achievement.

"What we know is a drop, what we don't know is an ocean."
(Sir Isaac Newton, 1643 – 1727)

ABSTRACT

GRACIANO NETO, V. V. **A simulation-driven model-based approach for designing software-intensive systems-of-systems architectures** . 2018. 217 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2018.

Context: Software-intensive systems have been increasingly interoperated forming alliances termed as Systems-of-Systems (SoS). SoS comprises a collection of systems joined to achieve a set of missions that none of the systems can accomplish on its own. Each constituent system keeps its own management, goals, and resources while coordinating within the SoS and adapting to meet SoS goals. Applications of SoS range from traffic control to emergency response and crisis management. As SoS often support critical domains, such systems must be correct by dealing with malfunction or defects and avoiding failures that could cause extensive damage and losses to the users.

Problem: Correct SoS operations depend on a precise specification and a rigorous attestation of its operational consistency. However, besides limitations on languages to jointly capture SoS structure and behavior, predictions on the SoS operational consistency rely on constituent systems not totally known at design-time. Therefore, SoS have been developed and deployed without evaluating their operations, since current languages do not support such precision in evaluation.

Objectives: This thesis provides solutions founded on a formal architectural description language to support an early evaluation of SoS operation regarding SoS structure and behavior by means of simulations.

Contribution: The main contributions of this project comprise (i) a model transformation approach for automatically producing simulation models from SoS software architecture descriptions, combining SoS structure and behavior description in a same solution, (ii) a SoS software architecture evaluation method for SoS operation prediction considering the inherent changes that can occur, (iii) environment modelling and automatic generation of stimuli generators to sustain the SoS simulation, delivering data to feed such simulation, and (iv) a method for the automatic synchronization between the runtime descriptive architecture (changed at runtime due to dynamic architecture) and its original prescriptive architecture based on model discovery and recovery mechanisms and a backward model transformation.

Evaluation: We conducted case studies to assess our solutions using Flood Monitoring

SoS and Space SoS.

Results: Our solutions support a high accuracy to (i) produce fault-free and fully operational simulations for SoS software architectures, (ii) support evaluation and prediction of SoS operation at design-time, (iii) automatically generate stimuli generators to sustain and feed the simulation execution, and (iv) maintain the synchronization between the runtime architecture and the intended version of the SoS architecture.

Conclusions: We concluded that the proposed solutions advance the state of the art in SoS software architecture evaluation by offering solutions to predict the SoS operations effectiveness to maintain a continuous operation despite architectural changes, providing more trust for users that futurely shall rely on SoS services.

Keywords: Systems-of-Systems, SoS, Model-Based Engineering, Software Architecture, Simulation.

RESUMO

GRACIANO NETO, V. V. **Uma abordagem digirida por simulação e baseada em modelos para projeto de arquiteturas de sistemas de sistemas intensivos em software** . 2018. 217 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2018.

Contexto: Sistemas intensivos em software tem sido interoperados para formar alianças conhecidas como Sistemas-de-Sistemas (SoS). Domínios de aplicação de SoS variam do controle de tráfego ao gerenciamento de situações de crises e emergência. Devido à criticidade destes domínios, tais sistemas precisam ser confiáveis e robustos, lidando com potenciais defeitos e mal funcionamento, e evitando falhas que poderiam causar ameaças à integridade dos usuários.

Problema: O funcionamento correto de um SoS depende da especificação precisa e da garantia rigorosa da consistência de suas operações. Entretanto, além das limitações nas linguagens quanto à especificação de ambos estrutura e comportamento do SoS, prever seu comportamento depende da especificação de constituintes que não são totalmente conhecidos em tempo de projeto e de seu comportamento emergente. Neste sentido, SoS têm sido desenvolvidos e implantados sem a devida avaliação de seus comportamentos, uma vez que as linguagens disponíveis atualmente não dão suporte a uma especificação precisa destes comportamentos.

Objetivos: Este projeto de doutorado relata avanços teóricos e práticos fundamentados em uma linguagem de descrição arquitetural formal para permitir a predição e avaliação do comportamento e estrutura dos SoS com base em simulações.

Contribuições: As principais contribuições deste projeto envolvem (i) uma transformação de modelos para produzir automaticamente modelos de simulação para descrições de arquitetura de software de SoS, combinando estrutura e comportamento em uma mesma solução, (ii) um método de avaliação de arquitetura de software de SoS para prever o comportamento do SoS considerando sua dinâmica inerente, (iii) modelagem do ambiente e derivação automática de geradores de estímulos entregando dados continuamente e sustentando a execução de simulações de SoS, e (iv) um método para promover a sincronização automática entre modelos descritivos e prescritivos de arquitetura de software de SoS baseados em mecanismos de descoberta e recuperçaão de modelos, e transformação de modelos reversa.

Avaliação: Estudos de caso foram conduzidos para avaliar as soluções nos domínios de

Monitoramento de Enchentes e Espacial.

Resultados: As abordagens propostas exibem alta acurácia no que tange (i) a produzir simulações operacionais e sem falhas para arquiteturas de software de SoS, (ii) ao suporte à avaliação, ainda em tempo de projeto, do comportamento que emerge da operação do SoS, (iii) à derivação automática de geradores de estímulos para entrega contínua de dados e manutenção da execução das simulações geradas, e (iv) à manutenção do alinhamento entre os modelos descritivos e prescritivos da arquitetura do SoS avaliado.

Conclusões: Conclui-se que as abordagens propostas avançam o estado da arte no projeto de arquiteturas de Software de SoS ao permitir prever, em tempo de projeto, como o SoS vai operar em tempo de execução, permitindo estabelecer estratégias para manter a simulação rodando, e sua operação contínua, mesmo com as mudanças arquiteturais inerentes ao seu funcionamento, provendo mais confiabilidade para os usuários futuramente dependerão de seus serviços.

Palavras-chave: Sistemas-de-sistemas, SoS, Arquitetura de Software, Engenharia Baseada em Modelos, Simulação.

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

RÉSUMÉ

GRACIANO NETO, V. V. **Une approche dirigée par les simulations et basée sur les modèles pour la conception des architectures logicielles des systèmes-des-systèmes à logiciels prépondérants**. 2018. 217 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2018.

Contexte: Les systèmes à logiciels prépondérants sont de plus en plus intéropérables formant des alliances nommées: Systèmes-des-Systèmes (SdS). Les applications des SdS peuvent aller des systèmes de gestion du trafic jusqu'aux systèmes de gestion de crises. Étant donné que les SdS supportent souvent les domaines critiques, ils doivent être fiables en traitant les disfonctionnements ou les défauts et en évitant les défaillances qui pourraient causer des dégâts et pertes importantes aux utilisateurs.

Problème: Ajuster les opérations d'un SdS dépend d'une spécification précise et une attestation rigoureuse de sa consistance opérationnelle. Cependant, en plus des limitations des langages pour capturer conjointement la structure et le comportement des SdS, les prédictions de la consistance opérationnelle des SdS reposent sur leurs systèmes constitutifs qui ne sont pas totalement connus au moment de la conception. Par conséquent, les SdS ont été développés et déployés sans évaluation de leurs opérations, puisque les langages actuels ne supportent pas ce type de précision lors de l'évaluation.

Objectif: Ce projet de thèse fournit des solutions théoriques et pratiques basées sur un langage formel de description d'architectures pour supporter une évaluation précoce des opérations du SdS par rapport à la structure et le comportement du SdS à travers les simulations.

Contributions: Les contributions essentielles de ce projet comprennent (i) une approche de transformation des modèles pour produire automatiquement des modèles de simulation à partir des descriptions des architectures logicielles du SdS, combinant la description structurelle et comportementale du SdS dans la même solution, (ii) une méthode d'évaluation de l'architecture logicielle du SdS pour la prédiction des opérations du SdS tout en considérant les changements inhérents qui peuvent se produire, (iii) modélisation de l'environnement et génération automatique des générateurs de stimulus pour soutenir la simulation des SdS, livrant des données pour nourrir tel simulation, et (iv) une méthode pour la synchronisation automatique entre l'architecture descriptive d'exécution (qui change à l'exécution par suite de l'architecture dynamique) et son architecture prescriptive d'origine basée sur des

mécanismes de découverte et de récupération de modèles et une transformation de modèle rétrograde.

Évaluation: Nous avons conduit des cas d'études pour évaluer nos approches en utilisant le SdS de surveillance des inondations et le SdS d'espace.

Résultats: Notre approche montre une précision importante pour (i) produire des simulations des architectures logicielles des SdS sans failles et complètement opérationnelles, (ii) supporte une évaluation et une prédiction fiable des opérations du SdS à la phase de conception, (iii) génère de manière automatique des générateurs de stimuli pour soutenir et nourrir l'exécution de la simulation et (iv) maintien la synchronisation entre les versions descriptives et prescriptives de l'architecture du SdS.

Conclusion: Nous avons conclu que les approches proposées font évoluer l'état de l'art de l'évaluation des architectures logicielles des SdS en offrant des solutions pour prédire l'efficacité des opérations du SdS pour maintenir une opération continue malgré les changements architecturaux, fournissant plus de confidence aux utilisateurs qui reposent dans l'avenir sur les services du SdS. .

Mots-clés: Systèmes de systèmes, SoS, architecture logicielle, simulation, évaluation.

LIST OF FIGURES

1. Scientific methodology. 7
2. Proposed solutions. 11
3. Formalisms and models used in studies of MBSE for SoS, as shown in Table 1. 20
4. An excerpt of SosADL abstract syntax (GRACIANO NETO, 2016). . . 24
5. An approach for the transformation of SosADL models into DEVS simulation models (GRACIANO NETO, 2017). 49
6. Patterns expressed as diagram classes in UML. 53
7. Illustration of the addition of a constituent in a simulation of SoS software architecture. 60
8. Process of remotion of constituent in a simulation of a SoS software architecture. 61
9. Substitution of constituent in the simulation of a SoS. 63
10. Process of reorganization of architecture in the simulation of a SoS software architecture. 64
11. An illustration of the relation between DRC and constituents simulated. 65
12. A flood monitoring system-of-systems (FMSoS) (GRACIANO NETO *et al.*, 2016). 67
13. Relation between percentage of data received in gateways and alerts triggered. 69
14. Relation between data received in one or more gateways and SoS scale. 71
15. Box plot for data received for 50 different architectural configurations. . 72
16. Illustration of a Brazilian SoS for data collection via satellites (INPE, 2017). 74
17. Activity diagram of the business process followed by a space SoS for the monitoring of the Amazon. 75
18. Brazilian territory map adapted from (DNIT, 2017). 77
19. Three orbital trajectories containing two satellites each (CARVALHO *et al.*, 2013). 81
20. Comparison between data obtained in case study: simulation time. . . 86
21. Maximum number of constituents in each simulation. 86
22. Number of architectural changes at runtime. 87

23. Diversity of constituents. 87
24. Relation between accuracy and data loss. 88
25. An illustration of communication between DCP and a satellite. 91
26. Model discovery mechanism for SoS software architectures. 106
27. Model discovery mechanism for SoS software architectures. 109
28. An illustration of the initial state of the FMSoS architectural configuration. 118
29. Excerpt of a FMSoS architecture restored through reverse transformation (part of the code is hidden for the reader convenience). 120
30. Relation between number of constituents and mutation coefficient. . . . 121
31. Average of mutation coefficient samples. 122
32. SoSADL2DEVS transformation (GRACIANO NETO, 2017). 133
33. Stimuli-SoS workflow. 135
34. A flood monitoring system-of-systems (FMSoS) Architecture. 137
35. An illustration of part of a FMSoS. 139
36. A real picture of a human dummy used to classify floods risk. 142
37. Water level with a human dummy. 142
38. Illustration of how an automaton is derived from SoSADL system specification to create a functional stimuli generator. 146
39. Monitoring depth of water in simulation during data processing. 149
40. A labeled state diagram corresponding to a *sensing* behavior extracted from SosADL code. 204
41. An illustration of the relation between Dynamic Reconfiguration Controller (DRC) and constituents being simulated. 212

LIST OF SOURCE CODES

Source code 1. Code in SosADL for a mediator. 49
Source code 2. An atomic model for a mediator generated in DEVSNL. 50
Source code 3. Description of an architecture of an FMSoS in SosADL. 57
Source code 4. Coupled model for FMSoS generated in DEVS. 59
Source code 5. Model discovery mechanism. 110
Source code 6. Coupled model for FMSoS generated in DEVS. 112
Source code 7. Coalition specified in SosADL. 113
Source code 8. A SoS abstract architecture specified in SosADL. 114
Source code 9. A specification of a sensor in SoSADL. 136
Source code 10. DEVS code for a stimuli generator. 147
Source code 11. Code in SosADL for a mediator. 199
Source code 12. Transformation code specified in Xtend. 200
Source code 13. An atomic model for a Mediator generated in DEVSNL. 202
Source code 14. Code in SosADL for a behavior called *sensing* of a smart sensor. . 203
Source code 15. State diagram code corresponding to a Smart sensor behavior generated in DEVSNL. 204
Source code 16. Description of an architecture of an FMSoS in SosADL. 207
Source code 17. Transformation rules specified in Xtend for the transformation of a SosADL model into DEVS model. 209
Source code 18. Coupled Model for FMSoS generated in DEVS. 210
Source code 19. Dynamic reconfigurator controller structure. 211
Source code 20. A transformation excerpt that supports generation of DEVS simulation of SoS software architecture with support to dynamic reconfiguration. 213
Source code 21. Excerpt of a satellite modelled in SosADL. 215

LIST OF TABLES

1. Transformations for SoS domain (Entries marked with asterisk (*) represent that transformations are actually mentioned or considered as a possibility of future work, but not performed). 35
2. Comparison between formalisms for SoS simulation and software architecture specification considering aforementioned language requirements. 41
3. Mapping between SosADL and DEVS. 47
4. Conflicts and compatible instructions in DEVS. 54
5. Patterns for input in DEVS simulation models. 55
6. Output pattern for DEVS simulation models. 56
7. Mapping of SosADL into DEVS. 56
8. A sample of data collected by a sensor and sent to a gateway. 67
9. Number of DCP constituents and data for each space SoS architectural configuration. 77
10. Percentage of data transmitted to the satellite and simulation time. . . . 79
11. Data loss for the space SoS simulation. 79
12. Telecommands in space SoS simulation. 79
13. Space SoS architectural configurations for Constellation of Satellites. . . 82
14. Percentage of data transmitted by each architectural configuration and received in ground. 84
15. Percentage of data loss in satellite constellation simulation. 84
16. Results of telecommands and photographs requests, taken, and returned to ground. 84
17. Percentage of missions accomplished in Scenario 3. 84
18. Number of lines of code produced by our patterns for FMSoS. 89
19. Number of lines automatically generated for simulations 1 and 2 of space SoS - NoM means Number of Models. 90
20. Comparison between related approaches. 94
21. Mapping between DEVS and SosADL. 111
22. Part of the results collected during the case study. 119
23. A sample of data sent by sensors. 149
24. Comparison between co-related approaches. 155
25. Mapping of SosADL into SES/DEVS 206

CONTENTS

1 Introduction . . . 1
1.1. Problem Statement and Justification for the Research . . . 4
1.2. Scientific Methodology . . . 7
1.3. Research Questions, and Objectives . . . 9
1.4. Summary of Contributions . . . 11
1.5. Thesis Outline . . . 13
2 State of the Art on Model-Based Software Engineering for Systems-of-Systems . . . 15
2.1. Foundations on MBSE and SoS . . . 15
2.2. MBSE for SoS . . . 17
2.2.1. Domain-Specific Modeling Languages for Systems-of-Systems . . . 19
2.2.2. Missions in SoS . . . 21
2.2.3. Software Architecture for SoS . . . 22
2.2.4. Dynamic Software Architectures . . . 25
2.2.5. Prescriptive and Descriptive Architectural Models . . . 26
2.2.6. Simulation Models for SoS . . . 28
2.2.7. Stimuli Generators . . . 29
2.2.8. Evaluation, Testing, Verification, and Validation for SoS . . . 31
2.2.9. Deployment and Maintainability . . . 32
2.2.10. Model Transformations . . . 33
2.2.11. Transformation Tools, Models, and Languages . . . 38
2.3. Final Remarks . . . 40
3 ASAS: A Model-Based Approach for the Simulation and Evaluation of Software Architectures of Systems-of-Systems . . . 45
3.1. Presentation of ASAS Approach . . . 46
3.1.1. Correspondences between SosADL and DEVS models . . . 47
3.1.2. Generation of Constituent Models . . . 49
3.1.3. Patterns for SoS Simulation . . . 51
3.1.4. Generation of Coupled Models . . . 55
3.1.5. A Dynamic Reconfiguration Mechanism for Supporting SoS Dynamic Architectures . . . 59
3.2. Evaluation . . . 63

3.2.1. Scenario 1: Flood Monitoring SoS 66
3.2.2. Scenario 2: Space SoS with One Satellite 72
3.2.3. Scenario 3: Space SoS with Satellites Constellation 80
3.2.4. Synthesis 85
3.3. Discussion 89
3.4. Final Remarks 100
4 Back-SoS: a Model-Based Approach for Reconciliation between Descriptive and Prescriptive Models of Systems-of-Systems Software Architectures . . 105
4.1. Presentation of Back-SoS Approach 106
4.1.1. Architectural Drift in SoS architectures 107
4.1.2. Model Recovery and Discovery Mechanism at SoS Concrete Architectural Level 108
4.1.3. Architectural Evaluation 110
4.1.4. Reconciling Descriptive and Prescriptive SoS Software Architectures . . 111
4.1.5. Mechanisms to Check Conformance between Abstract and Concrete Software Architectures 114
4.2. Evaluation 115
4.3. Discussion 122
4.4. Final Remarks 129
5 Stimuli-SoS: A Model-Based Approach for Automatic Creation of Stimuli Generators in Simulations of Software Architectures of Systems-of-Systems 131
5.1. Presentation of Stimuli-SoS 133
5.1.1. A Systematic Approach to Derive Stimuli Generators 133
5.1.2. Model Transformation 134
5.2. Evaluation 137
5.2.1. Scenario Description 137
5.2.2. Case Study Protocol 141
5.3. Discussion 150
5.4. Final Remarks and Forthcoming Steps 156
6 Conclusions 159
6.1. Solutions 159
6.2. Limitations 160
6.3. Possible Extensions and Future Work 161

References 165
APPENDIX A List of Publications 193

APPENDIX B Specification and Details on Transformation of SoSADL models into DEVS models . . . 199
B.0.1. Generation of atomic models . . . 199
B.0.2. Generation of coupled models . . . 205
B.0.2.1. Dynamic reconfiguration controller structure . . . 210
APPENDIX C A Satellite specified in SosADL . . . 215

CHAPTER

1

INTRODUCTION

Software has been increasingly embedded into mechanical, electrical, hydraulic, and pneumatic systems. Software supports them to offer a higher precision of their functionalities with automation of operation, which makes them smarter. Such systems have become *software-intensive*, i.e., software intensively contributes, influences, and impacts on their design, construction, deployment, and evolution (ISO, 2011; GONCALVES *et al.*, 2014). Our society has become highly dependent on services provided by software-intensive systems, and due to that, increasingly more complex solutions have been required. However, systems operating alone have not achieved them successfully, which has pressured them to interoperate, i.e., communicate, exchange data, and use the information exchanged to deliver results (HIMSS, 2013). In this perspective, a distinct class of systems, known as Software-Intensive Systems-of-Systems (SoS)[1] has emerged. A SoS comprises a number of operationally and managerially independent software-intensive constituent systems that work together to offer complex functionalities that could not be delivered by any one of them in isolation (MAIER, 1998; JAMSHIDI, 2009; GUESSI *et al.*, 2015; INCOSE, 2016). Moreover, SoS are often designed to accomplish missions, i.e., high-level goals assigned to the entire SoS to be achieved through exploiting the set of functionalities delivered by the constituent systems (SILVA *et al.*, 2014). SoS are likely to form the next generation of software-intensive systems (JAMSHIDI, 2008; BOEHM, 2006) and often support missions in critical domains, such as smart traffic control and emergency and crisis response (ROAD2SOS, 2013). Important investments in SoS Engineering have been made; for instance, Saudi Arabia has invested 70 billion dollars in smarter cities and South Africa has conducted a 7.4 billion dollars smart city project (CERRUDO, 2015). Therefore, their correct operation is of paramount interest and the construction of reliable SoS must be

[1] For sake of simplicity, the acronym SoS will be used herein to express both singular and plural.

investigated.

SoS share important dimensions (MAIER, 1998), such as: (i) managerial independence, i.e., constituents are owned and managed by distinct organizations and stakeholders; (ii) operational independence, since constituents also perform their own activities, even when they are not accomplishing one of the SoS' missions; (iii) distribution, i.e., their constituents are dispersed requiring connectivity to communicate; (iv) evolutionary development, since SoS evolve due to the evolution of their constituents parts; and (v) emergent behavior, which corresponds to complex functionalities that emerge from the interoperability among constituents. Constituents cooperate with their individual capabilities to deliver complex functionalities, some of them deliberately planned to be accomplished as emergent behaviors, which comprise a realization of the pre-established missions (DAHMANN; JR.; LANE, 2008; INCOSE, 2016). Moreover, SoS can exhibit an opportunist nature, i.e., a system can become spontaneously available for joining other systems to form a SoS, and leave the SoS when the mission finishes. Remarkable examples include smart cities, smart grids, smart buildings, and all a plethora of smart-* systems (OQUENDO, 2016c; ICS-CERT, 2015; FITZGERALD *et al.*, 2013).

Software-intensive SoS holds software architectures. Software architectures correspond to the fundamental structure of a software system, which comprises software elements, relations among them, and the rationale, properties, and principles governing their design and evolution (BASS; CLEMENTS; KAZMAN, 2012; ISO, 2011). A software-intensive architecture of a SoS is its fundamental structure, which includes its constituents and connections between them, their properties as well as those of the environment (NIELSEN *et al.*, 2015). SoS software architectures are particularly highly dynamic, i.e., they continuously change in response to addition, substitution, and deletion of constituents. In SoS software architectures, an *architectural configuration* is the current state and organization of an arrangement of interoperable software-intensive systems at a given point of time, also known as *coalition* in SoS domain. A *dynamic architecture* can change its own structure at runtime, exhibiting several architectural configurations during its execution, whereas a *dynamic reconfiguration* is the ability of an architecture or simulation has to reconfigure its own structure at runtime. Therefore, a dynamic architecture incorporates dynamic reconfiguration support.

Single systems can be validated and verified in a satisfactory way using analytical methods and techniques. However, complex systems, a class of systems of which SoSs are part, require methods that support their proper validation and verification. Emergent behavior is a particular SoS characteristic triggered by the reception of stimuli and data exchanged between the constituents, and between them and their environment (GRAHAM,

2013). Such behaviors are an holistic phenomenon manifested through a certain number of interactions among the constituents that produce a global result that could not be delivered from any one of them in isolation (MAIER, 1998). Emergent behaviors comprise a scenario in which the whole possesses properties not possessed by their parts, so that if a whole is reduced to its parts in analysis the emergent properties are not discoverable by the analysis (OQUENDO, 2017). Examples of emergent behaviors include *home security* behavior, which could emerge from a set of individual systems installed in a smart home, and a *traffic jam*, which is the resultant behavior that several cars may raise together depending on the traffic network and conditions, which is not predictable by knowing only the behavior of cars. Another example is the so-called *phantom traffic jams*, where fast-moving traffic suddenly congeals into a slow-moving jam for no apparent reason (OQUENDO, 2017).

Emergence is deliberately and intentionally planned and designed for SoS (BOARDMAN; SAUSER, 2006), i.e., the SoS engineer is the major player for creatively exploring the functionalities delivered by the constituents, assembling them for innovative purposes. Hence, guaranteeing that SoS are going to exhibit an expected set of behaviors highly depend on predicting how constituents interoperate at runtime. As the level of uncertainties and variables increases due to the number of constituents involved, analytic solutions for SoS evaluation demand a dynamic view, i.e., a model that captures the SoS behavior and enables its evaluation at design-time. SoS must be analyzed under a multitude of perspectives, and the views can be distinguished into two families (CARLE *et al.*, 2012): static views, focusing on systems properties, and dynamic views, focusing on the representation of the software architecture behavior. As SoS exhibit emergent behaviors, dynamic views are especially interesting. Simulations allow to observe, at design-time, the behaviors (intended or not) that emerge at runtime, allowing SoS engineers to verify and validate them.

Emergent behaviors can be classified under two perspectives (MITTAL; RAINEY, 2015): *Intention* and *Type*. An emergent behavior can be *Predicted* or *Unpredicted* (CHALMERS, 2006). Predicted emergent behavior consists of behaviors intentionally designed to emerge at runtime, whilst unpredicted emergent behavior corresponds to that one that emerges as a co-lateral effect of specific conditions or runtime configurations, with the potential to cause losses to the SoS operation. Considering the type, four categories exist (MITTAL; RAINEY, 2015): *Simple*, *predicted*, *strong*, and *spooky*. *Simple* emergent behaviors are emergent properties readily predicted by simplified models of the SoS. They are produced in lower complexity through models that abstract the SoS (only intentional predicted behaviors emerge since the model is overly simple). *Predicted* emergent behaviors are those readily and consistently reproducible in simulations of the system, but not in static models. They are partially predicted in advance (desired behaviors are predicted,

but undesired can also appear). *Strong* emergent behaviors are consistent with SoS known properties, but are not reproducible in any model of the system. Direct simulations may reproduce the behavior, but inconsistently, and simulations do not predict where the property will occur (desired behaviors exist, but unpredicted behaviors are the majority). Finally, *spooky* emergent behaviors are inconsistent with known properties of the SoS, not reproducible or subject to simulation, such as life itself, not predicted. For the scope of this thesis, we deal with the predicted ones.

1.1. Problem Statement and Justification for the Research

Single systems can cause serious damage while operating alone. When they are combined to work together, the possibilities of failures dramatically increase. Hence, due to dynamic properties as emergent behaviors and dynamic architectures, and to the high complexity that can be faced in regards to the amount and variety of different systems involved into the SoS operation, SoS also exhibits a high degree of *uncertainty.* As SoS aim at supporting critical domains, risks of damages, financial losses, and threats to human lives may arise. Therefore, SoS must be constructed to be trustworthy, i.e., their operation should be reliable for users that trust on their services to correctly operate, work as expected, and keep operations in progress, with no failure or accidents (NAMI; SURYN, 2013; STEINHOGL, 2015; OQUENDO; LEGAY, 2015; GRACIANO NETO; OQUENDO; NAKAGAWA, 2016). Hence, trust influences the degree at which users will rely on the services provided by a SoS (MOHAMMADI *et al.*, 2014).

Problem I. Lack of notations that encompass representation of static and dynamic aspects of SoS software architectures.

For the avoidance of rework and delays in the SoS development project, evaluation activities must start early in the development process. Requirements models, such as UML sequence models, textual requirements, use case diagrams, SysML block diagrams (KASSAB; NEILL; LAPLANTE, 2014), or architecture models (one of the 4+1 views - development view, logical view, physical view, process view, and scenarios (KRUCHTEN, 1995) could be used for evaluation purposes. However, requirement models are often not rich enough in details to provide support for SoS evaluation. In turn, architectural models (i) inherently hold the SoS structure, including constituent elements, their relation, dynamics, and non-functional requirements description, (ii) enable SoS specification for verification and validation (V&V), and evaluation purposes, and (iii) are richer in details than requirement models, which can be incomplete, imprecise, and rudimentary to support a reliable evaluation process.

Problem II. SoS are critical and their software architectures should be specified and designed using an approach that supports the prediction of their behaviors.

Architectural evaluation enhances quality attributes, including those related to trustworthiness, and minimizes faults and failures that affect software quality (BALCI, 1997; LEMOS; GACEK; ROMANOVSKY, 2002; NAMI; SURYN, 2013). Therefore, the support of evaluation activities in SoS guarantees the software governing the SoS operation yields the expected results (MICHAEL; RIEHLE; SHING, 2009). In particular, SoS evaluation demands an association between static and dynamic views and automation for detecting as many potential failures as possible. Manual approaches for evaluation are often unfruitful, as they deal only with static aspects and small-scale samples. Architectural models can be adopted due to the level of details they exhibit and dynamic models, as simulations, can be employed to anticipate failure, and exhibit the SoS behavior prior to its deployment.

However, SoS impose barriers on evaluation activities, as they exhibit dynamic properties, as emergent behaviors and dynamic architectures. Emergent behaviors comprise a holistic phenomenon that occurs at runtime as a consequence of interoperability of constituents and is explicitly planned (or not) by SoS engineers as a form of exploiting constituents' capabilities (MAIER, 1998; NIELSEN *et al.*, 2015; GRAHAM, 2013; FITZGERALD; LARSEN; WOODCOCK, 2014; WACHHOLDER; STARY, 2015; MITTAL; RAINEY, 2015). Dynamic architectures regard the ability of SoS to self-adapt their own architecture at runtime, i.e., join new constituents, rearrange their own structure, and substitute or eliminate constituents, which are recurrent activities during their normal operation. Such characteristics cannot be totally validated through the adoption of static specification, demanding non-static specifications that externalize the SoS dynamics and enables its visualization (MICHAEL; RIEHLE; SHING, 2009; DOBRICA; NIEMELE, 2002; MICHAEL *et al.*, 2011).

Problem III. An evaluation method for SoS software architectures should adopt descriptions that combine static and dynamic view to precisely capture their structure and behavior, enabling the observation of the impact of dynamic architecture on the functionalities being provided.

Architectural evaluation activities demand some type of specification and SoS engineering requires strategies and means for dealing with the inherent dynamics, complexity, and often large dimensions of SoS. Model-Based Software Engineering (MBSE) has been recommended as a response to those demands, as it comprises a software development approach that prescribes models to capture knowledge acquired from the domain, supporting the specification required for evaluation purposes (RAMOS; FERREIRA; BARCELO, 2012;

RAMOS; FERREIRA; BARCELO, 2013; INCOSE, 2016). MBSE also prescribes model transformations for automating the generation of software code from abstract models, taming complexity and enabling the processing of SoS properties for specification, evaluation, verification, and validation purposes (GRACIANO NETO *et al.*, 2014; FALKNER *et al.*, 2016). Moreover, MBSE can also make models executable, i.e., they support the management of dynamic aspects visualized at runtime.

Currently, we could mention four major categories of techniques for evaluating software architectures (DOBRICA; NIEMELE, 2002; MICHAEL; RIEHLE; SHING, 2009; ABRAHAO; INSFRAN, 2017): scenario-based, simulation-based, mathematical/logical-based, and experience-/metric-based. Since SoS architectures demand a dynamic approach, simulation-based approach matches such requirements. Nonetheless, despite the architectural description languages (ADL) support for evaluating software architectures, known ADL have not supported evaluation of both static and dynamic properties in a same approach, in particular including the entire set of requirements imposed by SoS, as emergent behaviors, dynamic architectures, complexity, and a large variety and amount of systems (GUESSI *et al.*, 2015).

Problem IV. Inconsistent operational states in a SoS can lead to malfunction, disasters, and losses. Representation of the surrounding environment of a SoS should then involve representation of potentially unpredicted conditions and continuous variables related to the environment where it will be deployed.

As we selected simulations as the dynamic view for SoS architectural specifications, other problems arise. Another source of uncertainty of SoS is its surrounding environment. Even using a simulation to predict behavior of SoS and its constituents, and how they achieve missions, the behavior of a SoS in real world is dramatically influenced by its surrounding environment. Architectural specifications often hold details about the intended environment of a systems, documenting how this influences the system results. However, many languages used to describe software architectures have not tackled environment representation. Moreover, when dealing with simulations, it is important to support it with data that represent the environment, and continuously delivering it during its execution.

Problem V. New architectural arrangements that arise at runtime due to dynamic architecture can potentially originate harmful inconsistencies between SoS architectural description and SoS architecture at runtime, which should be prevented.

Finally, as SoS progresses with its operation, constituents can join it, leave it, or the entire architectural arrangement can be organized to better fit a set of requirements. As such, an increasing inconsistency between the initial architectural specification and the runtime

version of such SoS can take place. These inconsistencies, which characterize *architectural drift*, can cause a phenomenon called *architectural erosion*. For software architectures of single software systems, architectural erosion is known to have a negative impact on the quality of software systems, such as for maintainability, evolvability, performance, and reliability (GURGEL *et al.*, 2014; TERRA *et al.*, 2012), potentially interrupting the operation of the system. Tailored mechanisms have been established for the identification and prevention of architectural erosion as well as for attaining architectural consistency. However, architectural erosion of SoS software architectures has not been properly addressed by any existing mechanism (SILVA; BALASUBRAMANIAM, 2012).

1.2. Scientific Methodology

This thesis was conducted as scientific research project. As such, a methodology was followed according to well-defined steps, as depicted in Figure 1, which involved:

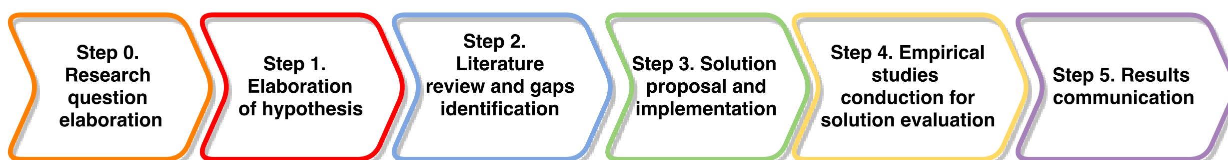

Figure 1 – Scientific methodology.

- **Step 0. Research question elaboration.** At the first moment, a research question motivated the conduction of this PhD research: *How can we evaluate SoS software architectures at design-time?* Once such question was raised, the scientific research itself was started. SoS are often huge and complex, involving several constituents with many restrictions, policies, and business rules. Moreover, SoS are often built for critical domains, and they must be reliable. At this direction, evaluation activities could be considered essential, as they can reduce potential errors, failures, and misunderstandings at specification and design levels. Evaluation demand some sort of specification. Hence, models could solve this need, which led us to consider the research hypothesis;

- **Step 1. Elaboration of hypothesis.** In light of all the identified and aforementioned requirements, the hypothesis established was *a model-based (MB) approach can support architectural evaluation activities for software-intensive SoS at design-time.* Models (i) capture the domain expertise acquired during the software development, and (ii) support automated evaluation. Moreover, models leverage abstraction, i.e., unnecessary details are excluded, and the attention can be focused on specific facets

of the problem. Besides, MB approaches bring automation about code generation, potential for simulation (and a consequent anticipation of problems before coding and deploying the product), and potential of reuse;

- **Step 2. Literature review and gaps identification.** Literature was reviewed to identify the gaps about specification and evaluation of SoS software architecture models, and to elicit existent mechanisms, methods, models, and approaches in the state of the art about MB approaches for SoS. Results were published in a workshop paper (GRACIANO NETO *et al.*, 2014). During the review, we identified that the application of MB methods in SoS engineering were still embryonic and could be further exploited (STEINHOGL, 2015). Later, we identified that MB approaches are a common systems engineering practice (NIELSEN *et al.*, 2015), and were adopted for several studies (around 59.38% of included studies in a systematic mapping) for managing systems complexity by enabling engineers to better understand requirements, develop candidate architectures, and verify design decisions early in the development process (LANA *et al.*, 2016);

- **Step 3. Solution proposal and implementation.** Considering the results of our literature review, we decided to adopt a model transformation to harmonize formalisms that could support both the precise specification of SoS software architectures and the execution of such models. SoSADL and DEVS were selected from the state of the art. We associated them by means of a model transformation (SoSADL2DEVS). We relied on such model transformation to build other advances, namely (i) an automatic generation of a mechanism for dynamic reconfiguration control, (ii) the externalization of patterns to automatically generate constituents behaviors with no conflicting rules (for simulation purposes), (iii) automatic generation of stimuli generators to feed the simulation and sustain its execution, preventing the need of a human interaction during the study conduction, and (iv) the elaboration of an evaluation method on such transformation. Moreover, we also identified the emergence of architectural erosion due to dynamic architecture, also proposing a solution for reconciliation between the runtime architecture and the intended one, as further explained later.

- **Step 4. Empirical studies conduction for solution evaluation.** Once solutions were proposed and implemented, we evaluated them according to rigorous scientific protocols. Experiments were not feasible, as such studies require another technique to compare the results with. We did not find directly related solutions. Hence, we opted for an exploratory but relevant empirical method, namely case studies (RUNESON; HöST, 2009), as our empirical source of evidence;

- **Step 5. Results communication, confirmation of refutation of hypothesis.** Once all the proposed approaches were evaluated, we confirmed our hypothesis that claim a model-based (MB) approach can support architectural evaluation activities for software-intensive SoS at design-time. Results were communicated through reporting and scientific publications.

Next section detail the research questions, and solution drawn as a result of the conduction of the aforementioned research methodology.

1.3. Research Questions, and Objectives

We claim an MB approach can aid this endeavor as it: (i) fosters the creation and adoption of software specifications expressed as models, (ii) relies on those models to automatize architectural evaluation activities through simulations, (iii) predicts the SoS behavior and dynamics under environmental conditions, and (iv) reduces uncertainty by predicting runtime conditions.

The following research questions were derived from the main research question:

RQ1: How can the evaluation of SoS architectures be supported?
Rationale: Considering the aforementioned difficulties, a novel evaluation approach must be established to support evaluation of SoS architectures. Such approach must combine dynamic and static characteristics of SoS, covering representation of structure and behavior.

RQ2: How can SoS dynamic behaviors be anticipated and predicted at design-time?
Rationale: SoS development suffers from challenges imposed by the inherent uncertainty related to its operation. Hence, a novel approach must predict, at design-time, how changes in the SoS architecture impact on the behaviors designed to be accomplished, and anticipate how changes must be performed to reactivate SoS operation in cases that an architectural change cause malfunction.

RQ3: How can SoS architectural description be continually consistent with its runtime configuration, despite its inherent dynamic architecture?
Rationale: As SoS progresses its operation, dynamic architecture results in different architectural configurations. New configurations can cause a lack of conformance with the original SoS architectural specification, what can cause mismatches between the implemented SoS in execution and the planned one. Hence, it is prominent to establish some mechanism to keep its documentation continually synchronized with its current

configuration.

RQ4: How can the surrounding environment be modelled for a SoS simulation purpose?
Rationale: SoS are developed to be deployed in a highly dynamic environment that must be modeled for the prediction of situations to which they will be subjected.

The following solutions were defined in this thesis:

1. *Automatic generation of simulation models for SoS*: we aimed at establishing a model-based transformation approach to combine dynamic and static descriptions of a SoS software architecture in a unique approach. We decided to describe SoS software architectures with the use of SosADL models and automatically transforming them into simulation models documented in Discrete Event System Specification (DEVS), a formalism for systems simulation;

2. *Modeling of a SoS surrounding environment and automatic generation of stimuli generators that support SoS simulation*: the surrounding environment is an important concern for SoS, as it directly impacts on the way they perform their activities. We aimed at modeling the environment at a certain abstraction level and automatically creating a structure that continuously produces stimuli for SoS simulation, anticipating possible failures, imitating the environment, and reducing costs of an early and inadvisable SoS deployment;

3. *Evaluation of SoS software architectures through simulations*: we aimed at considering a established and well-defined method to support evaluation of SoS architectural descriptions in regards to the functionalities it should offer, despite the dynamics of its architecture. SoS architectures could then be analyzed at runtime based on the simulations produced by our transformation approach; and

4. *A platform to support SoS architectural evaluation by means of simulation*: one of the contributions is the development of a platform that advances the state of the practice by supporting SoS architectural evaluation activities through SoS architectural simulation and evaluation;

5. *Automatic restoration of software architectural models*: an approach should be established to support analyzing the emerging architectural configurations. After the selection the best architectural configuration that matched the quality attributes expected for a trustworthy SoS, the architectural model can be updated according to that configuration, to proceed with SoS development.

The next section addresses the contributions of the thesis in accordance with the proposed objectives.

1.4. Summary of Contributions

This PhD thesis reports results of the establishment of approaches to deal with the highlighted gaps.

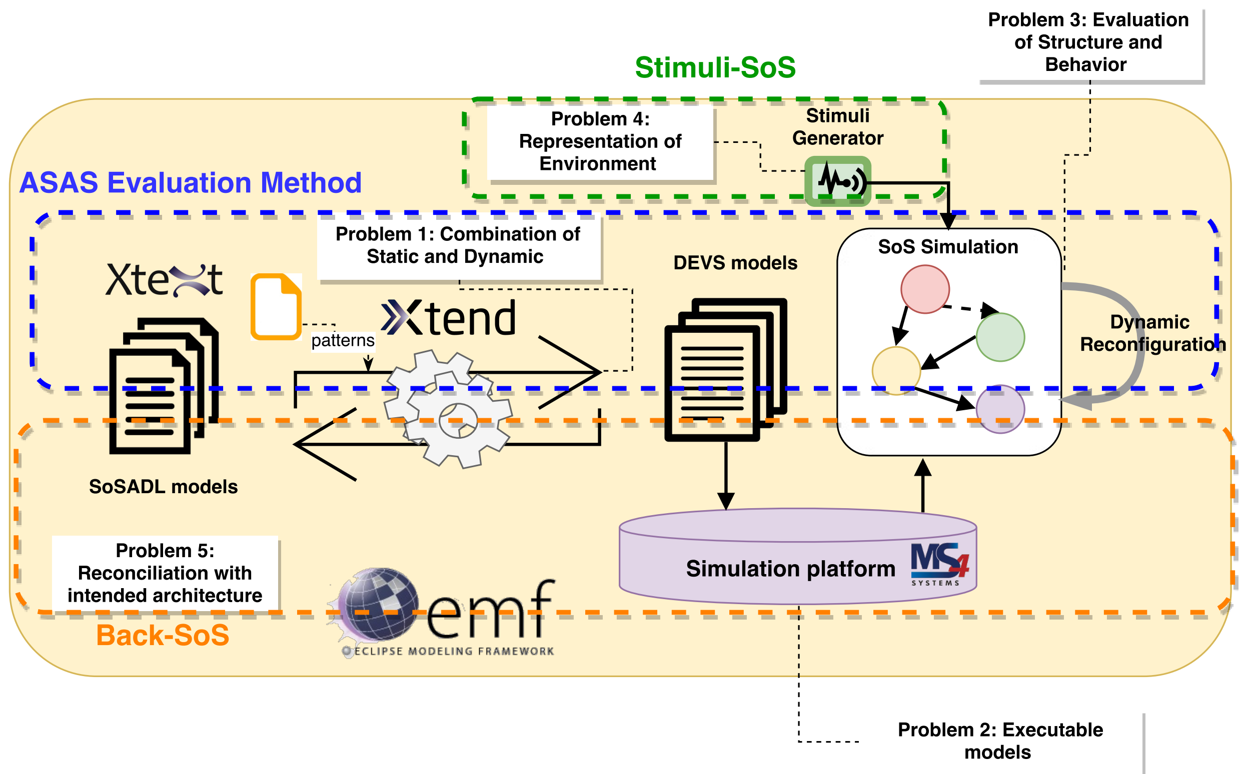

Figure 2 – Proposed solutions.

Figure 2 summarizes the contributions of this thesis, which are a compilation of three separate solutions. The solutions were designed to match the highlighted problems. Those solutions comprise:

1. **ASAS, an evaluation approach for SoS software architectures based on simulations** - we consider a pre-defined set of metrics associated with quality attributes related to trustworthiness ((i) *correctness*, (ii) *configuration quality*, (iii) *dependability*, and (iv) *performance*[2] (MOHAMMADI *et al.*, 2014; KUMAR; KHAN; KHAN, 2015; HASSELBRING; REUSSNER, 2006; NAMI; SURYN, 2013; MICHAEL

[2] In our research, we do not focus on other aspects such as security, privacy, data, or usability

et al., 2011)), evaluating SoS architectural descriptions based on the simulations produced by our transformation approach, and through a an evaluation method so-named ASAS (A Model-Based Approach to Simulate and Evaluate Software Architectures of Systems-of-systems). The main aim of ASAS approach is the support for the evaluation of SoS software architectures for increasing the level of trustworthiness for SoS. The core of ASAS is a model transformation that receives SoSADL models as input and produces simulation models documented in DEVS (GRACIANO NETO *et al.*, 2018b);

2. **Specification of patterns for conception of functional DEVS simulations with no conflicting rules** - engineering simulations can be unfruitful, as many instructions can be conflicting and simulation guides do not use to teach how to engineer such type of software. We discovered and reported patterns to guide how to engineer SoS simulations, and encapsulated them in the model transformation used in ASAS approach to automatically generate consistent simulations. Moreover, such knowledge can be reused to specify other simulations that use DEVS formalism. Results were reported in a conference paper (GRACIANO NETO *et al.*, 2018c);

3. **A mechanism for supporting SoS dynamic architectures** - we proposed SoS dynamic architecture canonical operations, and supported SoS simulation comprising all such operators, reproducing at runtime how a SoS could behave when deployed in the real world (MANZANO; GRACIANO NETO; NAKAGAWA, 2018) (such mechanism is included in ASAS);

4. **Stimuli-SoS, a method for representation of the SoS surrounding environment and automatic creation of stimuli generators to sustain SoS simulations** - the surrounding environment is an important concern for SoS, as it directly impacts on the way they perform their activities. As a software architecture description often documents environmental issues at some level, we automatically create structures that continuously produce stimuli for SoS simulation, anticipating possible failures, imitating the environment, and reducing costs of an early and inadvisable SoS deployment (GRACIANO NETO *et al.*, 2016; GRACIANO NETO *et al.*, 2017);

5. **Back-SoS, a mechanism for reconciliation between current SoS and its respective architectural documentation through a backward model transformation** - as a SoS progresses its operation, new architectural configurations take place, arising inconsistencies between original architectural documentation and current one. A backward model transformation was established, together with mechanisms for model recovery, discovery, and reconciliation, for realign both models, the runtime, and the original (GRACIANO NETO *et al.*, 2018b).

1.5. Thesis Outline

This chapter presented an overview of the context in which this research is settled, the problems that exist and that must be overcome, and the contributions that this thesis offer considering the highlighted gaps. The remaining chapters of this thesis are organized as follows.

Chapter 2 addresses the state of the art of MBSE for SoS, an overview of the foundations on which this thesis has been built, terminology and key concepts and results of the systematic literature review conducted. The results published in (GRACIANO NETO *et al.*, 2014) were updated to be included in the thesis.

Chapter 3 describes ASAS approach, which is the core of our proposal. ASAS is based on a model transformation from SoS architectural models to simulation models. SoS models were specified using SoSADL, whilst simulation models used DEVS. The chapter also presents an approach for the automatic generation of SoS simulation with support to dynamic reconfigurations at runtime. Results were reported in (GRACIANO NETO, 2016; GRACIANO NETO *et al.*, 2018b; GRACIANO NETO *et al.*, 2018c) and the content was extended to the thesis body and submitted to a journal (GRACIANO NETO *et al.*, 2018a).

Chapter 4 presents Back-SoS, an approach to prevent architectural drift in SoS architectures, and reconcile the runtime architectural arrangement with the SoS software architectural description. Results were published in (GRACIANO NETO *et al.*, 2018a) and submitted to a journal (GRACIANO NETO *et al.*, 2018b).

Chapter 5 presents Stimuli-SoS, an approach that automatically produces structures known as *stimuli generators*, which are virtual entities deployed in a simulation for imitating the surrounding environment. Results were published in (GRACIANO NETO *et al.*, 2016; GRACIANO NETO *et al.*, 2017).

Finally, Chapter 6 concludes the thesis, revisiting contributions, summarizing limitations, and presenting perspectives for future research.

CHAPTER 2

STATE OF THE ART ON MODEL-BASED SOFTWARE ENGINEERING FOR SYSTEMS-OF-SYSTEMS

Model-Based Software Engineering (MBSE) has been applied to SoS development. Models comprise a resource to tame complexity, leveraging the abstraction of software production, and documenting the design decisions in a computer-based format that allows to process it and even automatically generate software code. This chapter covers the state of the art about how model-based approaches have been exploited in SoS domain. Section 2.1 briefly presents foundations about model-based approaches. Section 2.2 presents a review on the state of the art on MBSE approaches for SoS. Section 2.3 concludes the chapter.

2.1. Foundations on MBSE and SoS

A *model* is a selective, reduced, and accurate representation of a system that concisely captures all the essential properties for a given set of concerns (SELIC, 2012). Such model is, essentially, a machine-readable abstraction of the reality represented using a given language (textual or visual). Examples of languages used to represent models in MBSE approaches include UML (OMG, 2015), SySML (OMG, 2017), and π-ADL (OQUENDO, 2004). These models are often driven by a more abstract concept that restricts how a model can be built. These abstract models are known as metamodels. Metamodels encapsulate the lexicon (the canonical constructs of a language) and the syntactic rules (the relations between the elements) that are allowed in the construction of a certain type of model. Models necessarily need to conform to their respective metamodels

(CANOVAS; MOLINA, 2010; CICCHETTI *et al.*, 2008), and this is analyzed during a validation of a model against its metamodel. After the model is validated, a transformation can be performed.

Models are part of the state of the practice in software development. They capture the expertise acquired to realize complex problems and potential solutions through the use of abstraction (SELIC, 2003), being successively transformed along the software development process until reaching software code. In short, requirements are input for architectural design; architectural design is refined to a detailed design; and the detailed design culminates in software code properly. Then, the software development process can be considered a difficult, expensive, and error-prone (OPHEL; OPHEL, 1993) manual succession of model transformations.

Along decades of software engineering, some portions of the aforementioned model transformations were automated through the use of model transformers, as a result of emergence of Model-Driven Architecture (MDA) specification by OMG (MILLER; KINGDOMERJI, 2003). Model transformers are model compilers that receive models and their respective metamodels as input and, through the use of model transformations, transform source models in target models. Target models can be graphical or textual (code). As models are described by metamodels, metamodels are described by metametamodels, and metametamodels describes themselves. The OMG standard metametamodel is the Meta Object Facility Language (MOF) (OMG, 2006). Another metametamodel is Ecore[1], an Eclipse standard used for implementations in Eclipse Modelling Framework (EMF) (KLEPPE; WARMER; BAST, 2003).

Transformations are mappings between a source metamodel and a target metamodel, according to a transformation definition (LEVENDOVSZKY *et al.*, 2002). A transformation definition is a set of transformation rules that describe how a model in a language can be transformed into another model (MENS; GORP, 2006). Such rules establish a traceability relation between source and target models, linking their language constructs through a mapping between those elements.

Model transformations are classified in three categories (MENS; GORP, 2006): Model-to-Model (M2M), Model-to-Text (M2T), and Text-to-Text (T2T). The first type encompasses the situation where the input of a model transformation is a model (as a diagram) and the output is a model as well (for instance, textual requirements to architectural models). Target and source models in a M2M transformation can also conform to the same metamodel, which is known as a *endogenous* transformation. Second type consists of a model transformation where the input is a model and output is a textual

[1] http://eclipse.org/ecoretools/

artifact (as HTML pages, software code, or scripts). And the last consists of transformation from a textual model to another textual model (as XML for PostgreSQL). Transformations may be unidirectional or bidirectional. Unidirectional transformations can be executed in one direction only, for example when a target model is computed (or updated) based on a source model. A transformation from a source model to a target model is termed as *forward transformation*, whilst a transformation from a given target model back to a source model is known as *backward transformation*. Bidirectional transformations can be executed in both directions, which is useful in the context of synchronization between models and code (CZARNECKI; HELSEN, 2003; STEVENS, 2008; CZARNECKI *et al.*, 2009; STEVENS, 2010; ZAN; PACHECO; HU, 2014). Bidirectional transformations can be achieved using bidirectional rules or by defining two separate complementary unidirectional rules, one for each direction. Most of the approaches do not provide bidirectionality (CZARNECKI; HELSEN, 2003).

M2M transformations usually use graph patterns (LARA; GUERRA, 2005; MENS *et al.*, 2005; SCHURR; NAGL; ZUNDORF, 2008; GREENYER; RIEKE, 2012). M2M transformations translate between source and target models, which can be instances of the same or different metamodels (CZARNECKI; HELSEN, 2003). In this approach, a model is considered as a graph, with nodes (as classes) and edges (as the relations among classes), and a transformation is essentially performed as a graph transformation, when manipulations are performed to transform nodes in other nodes, merge nodes, separate nodes, creating and removing edges. Other approaches are direct-manipulation, relational, structure-driven, and hybrid approaches (CZARNECKI; HELSEN, 2003). M2T (or Model-to-code (M2C)) category is distinguished between visitor-based and template-based approaches (CZARNECKI; HELSEN, 2003). In former, a very basic code generation approach consists of providing some visitor mechanism to traverse the internal representation of a model and write code to a text stream to generate code. In latter, a template of a program in the target technology is filled with code transformed from a source model.

2.2. MBSE for SoS

SoS is challenging, as they are highly dynamic, assembled by constituents often developed with COTS (Commercial Off-The-Shelf), engineered in all sort of heterogeneities comprising distinct operational systems, communication mechanisms, programming languages, and middleware technologies (GOKHALE *et al.*, 2008). Efforts have been performed to establish strategies, techniques, and methods to deal with the classic concerns of software engineering regarding software-intensive SoS (FRANCE; RUMPE, 2007; CALINESCU; KWIATKOWSKA, 2010), as SoS impose complexity, namely:

- Combining constituents, allowing them to interoperate, requires working on COTS with which they are usually engineered. Indeed, SoS require configuring middleware support to enable constituents communication, abstracting inherent heterogeneities regarding external data representation, operating systems, programming languages, network patterns, and communication mechanisms (BAY, 2002; GOKHALE *et al.*, 2008; BALASUBRAMANIAN *et al.*, 2009; FARCAS *et al.*, 2010; KAZMAN *et al.*, 2013);

- Managing constituent systems requires controlling their presence in the SoS, taking into account their intrinsic dynamics and volatility in a running SoS (BRYANS *et al.*, 2013; BATISTA, 2013);

- Configuring and deploying SoS usually relies on manually creating and handling large text files. This is necessary to ensure a suitable and correct configuration and deployment. However, this is a laborious and error-prone task, since those files have huge dimensions and high complexity (BARBI *et al.*, 2012);

- Ensuring that the software generated for SoS faithfully corresponds to models used to specify them is currently a manual process. There is a lack of specialized tools to address SoS complexities, and a low consensus due to diversity of models and languages for SoS modeling (FARCAS *et al.*, 2010);

- Assuring, in the opposite side, that software of SoS keeps synchronized with models used to generate it (FARCAS *et al.*, 2010); and

- Constituents must also perform their functions independently, not only exclusively to accomplish a mission of the whole SoS. This requires that code of independent work and mission need to co-exist in a same software entity, claiming for an adequate and modularized architecture (MAIER, 1998).

At this direction, MBSE has received attention to solve some of the aforementioned problems, specially in SoS domain (FRANCE; RUMPE, 2007; FARCAS *et al.*, 2010; FISCHER; SALZWEDEL, 2011; BARBI *et al.*, 2012; MITTAL; MARTIN, 2013). MBSE considers models as the primary artifacts of software development and has achieved prestige by gains in time-to-market, reducing effort and cost, and increasing productivity, traceability, and software quality (SENDALL; KOZACZYNSKI, 2003; SELIC, 2003). Besides, automation in the generation of software code adds value to the production of software, promoting traceability between models and code. As knowledge is registered in abstract models and model transformations, MBSE also potentially fosters reuse.

2.2.1. Domain-Specific Modeling Languages for Systems-of-Systems

Adoption of MBSE requires the use of models and domain-specific languages (DSL). Such DSL comprise a computer language specialized to a particular application domain. In regards to SoS, an specific type of DSL is often used termed as ADL. As SoS must be trustworthy, they must be carefully evaluated about their functional and non-functional properties. Hence, a enough detailed model is required for such evaluation activities. For this context, architectural models are the best option, since they inherently hold the SoS structure (constituent elements, their relation, and dynamics), which are important elements for validation and verification. Besides, architectural elements are potentially executable, which can support validation of dynamic properties. Moreover, architectural models are richer in details than requirement models, which can be too incomplete, imprecise, rudimentary to support a reliable V&V process. Hence, in this section we broadly discuss ADL and other languages for SoS architecture specification.

MBE or MBSE has been adopted in several studies situations (GRACIANO NETO *et al.*, 2014; NIELSEN *et al.*, 2015; LANA *et al.*, 2016). Literature reviews have reported a systematic use of MBSE in SoS development. Lana et al. report that around 59.38% (19 out of 32 included studies from 1994 too 2015) of included studies in a systematic mapping use MBSE principles for managing systems complexity by enabling engineers to better understand requirements, develop candidate architectures, and verify design decisions early in the development process (LANA *et al.*, 2016). Nielsen et. al also discuss an overview of MBSE techniques to SoS. Nevertheless, MBSE has also been recognized as a challenge for SoS engineering. Methods and supporting tools need to be adapted and evolved to support SoS. Diverse approaches involving models and model transformations have been reported in literature (BRYANS *et al.*, 2013; MITTAL; MARTIN, 2013; RAMOS; FERREIRA; BARCELO, 2012; LEWIS; SMITH; BEAULIEU, 2011; BARBI *et al.*, 2012; TU; ZACHAREWICZ; CHEN, 2011; NEEMA *et al.*, 2009; GOKHALE *et al.*, 2008; LANG; SCHREINER, 2009). Figure 3 visually summarizes the most common formalisms that have been adopted in the last years in literature. Arrows indicate model transformations directions, illustrating sources and targets used in each considered study.

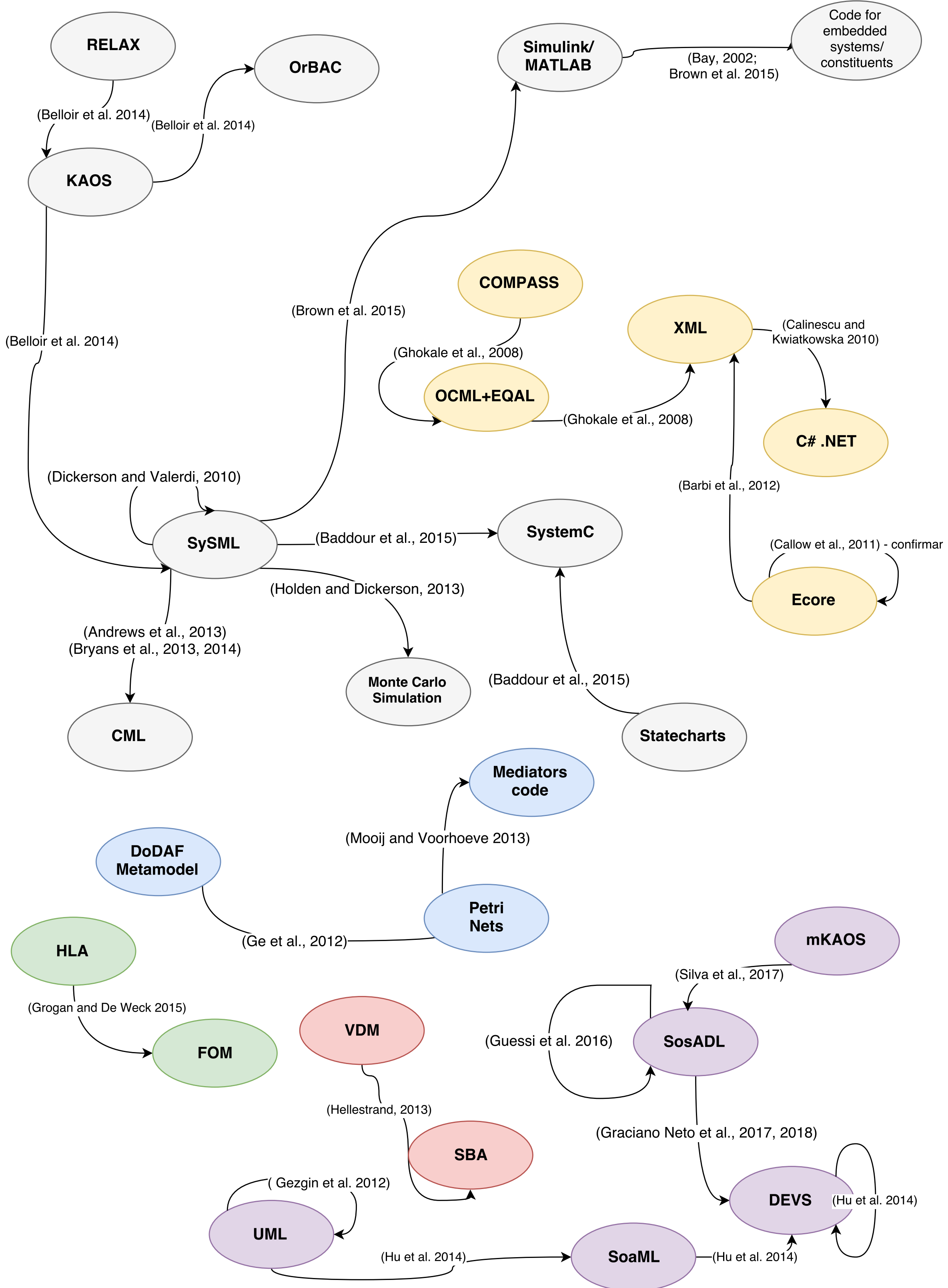

Figure 3 – Formalisms and models used in studies of MBSE for SoS, as shown in Table 1.

The most relevant aspects to be analyzed in studies that adopt MBSE for SoS development are (i) languages, models, metamodels, and purposes/concerns represented in those models during model-based software development for SoS (requirements, architecture, testing); and (ii) types of transformation executed (M2M, M2T, T2T), and whether they are unidirectional (forward or backward) or bidirectional; and (iv) the tools adopted, if they are customized and/or commercial, and which models play the role of source and target models in those transformations. Hence, our discussion is made on top of these parameters.

Many models and languages have been used or suggested to represent SoS. We discuss, as follows, which models have been used to describe SoS and what they have documented.

2.2.2. Missions in SoS

SoS are concerned with the fulfillment of *missions*, i.e., a set of goals to be accomplished by (i) performing tasks based on capabilities (functionalities) of constituent systems, and (ii) interactions among constituent systems leading to emergent behaviors. Individual missions are realized by constituent systems themselves whereas global missions of an SoS are accomplished through emergent behaviors (SILVA *et al.*, 2014; SILVA; BATISTA; CAVALCANTE, 2015; SILVA *et al.*, 2016; SILVA; CAVALCANTE; BATISTA, 2017). Silva et al. (SILVA *et al.*, 2014) have reviewed the literature to collect studies investigating how to deal with mission specification for SoS. After analyzing 12 studies with initiatives in this direction, they have elaborated a conceptual model of missions for SoS. They have proposed mKAOS, an extension of KAOS methodology (LAMSWEERDE, 2001) to model missions (SILVA; BATISTA; CAVALCANTE, 2015) in a goal-oriented notation. mKAOS encompasses six different models that allow specifying missions of SoS and defining relationships between these missions and other aspects of the SoS (such as emergent behavior and capabilities of the constituent systems), regardless of implementation details.

Lorenzo Alvarez et al. also progress on missions specification for SoS using a MBSE approach (ALVAREZ *et al.*, 2016). They adopt SySML to model scientific space missions involving many constituents of a SoS in European Space Agency. They use ESAAF (European Space Agency Architectural Framework), which consists of a prescriptive modeling methodology based on a set of architectural views to guide System-of-systems (SoS) design and integration. ESAAF is based on the well-known architectural frameworks, such as DoDAF (USA Dept of Defense AF), MoDAF (UK Ministry of Defence AF), TOGAF (Open Group AF) and Zachman Framework. Their model is organized in the following views: (i) Requirements view: Modeling of Euclid requirements, including traceability,

budgeting, justification and change control; (ii) Architecture view: Modeling of architecture and structure of the Euclid system, including interaction and interconnection between elements, characteristics, models; (iii) Verification view: Modeling of system verification logic, including activities, levels and flows; and (iv) Lifecycle view: Modeling of operational and data flows for the Euclid system, including operational timelines, data transmission and communication flows. They adopt Enterprise Architect to support mission modeling through SysML notation.

2.2.3. Software Architecture for SoS

Software-intensive SoS exhibit a software architecture, which comprise the fundamental structure of a software system, its software elements, relations among them, and the rationale, properties, and principles governing their design and evolution (BASS; CLEMENTS; KAZMAN, 2012; ISO, 2011). In SoS context, a software architecture involves the SoS in its fundamental structure, including its constituents and connections between them, properties of the constituents and of the environment (NIELSEN *et al.*, 2015). These concepts are important, as architectural models can be adopted for evaluation activities, contributing to improve the SoS quality (NAKAGAWA *et al.*, 2013).

Constituents and SoS itself are often specified using some ADL. An ADL is a domain specific language adopted to specify the structure of a system (ISO, 2011; NIELSEN *et al.*, 2015; GUESSI *et al.*, 2015). While current ADL support both representation and evaluation of SoS (GUESSI *et al.*, 2015), they still lack mechanisms to capture uncertainty, dynamism, and potentially undesired behaviors that can emerge from SoS architecture configurations (GRACIANO NETO *et al.*, 2014). Languages often adopted to describe software architectures, such as UML[2], SysML[3], and CML[4], lack expressiveness for describing SoS, specially regarding to: (i) a partial description of constituents, which are not totally known at design time; (ii) environmental modeling; and (iii) dynamic architecture. Other initiatives have proposed approaches that use model transformations from an architectural model (π-ADL, SysML, HLA, DoDAF[5]) to some simulation formalism (Go language, Simulink[6]). However, these approaches do not support: (i) SoS software architecture specification (CAVALCANTE; OQUENDO; BATISTA, 2014; CAVALCANTE *et al.*, 2016), (ii) dynamic architecture and constituents not known at design time (FALKNER *et al.*, 2016), and (iii) the concept of SoS software architecture with all the necessary details to guarantee precision in representation (XIA *et al.*, 2013; ZEIGLER *et al.*, 2012).

[2] UML, http://www.uml.org/
[3] SysML, http://sysml.org/
[4] CML, http://www.compass-research.eu/approach.html
[5] DoDAF, US Department of Defense Architecture Framework, 2010.
[6] Simulink, www.mathworks.com/products/simulink/

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

Many languages have been adopted for specifying SoS architectures (GUESSI *et al.*, 2015; KLEIN; VLIET, 2013).

Composable Adaptive Software Systems (COMPASS) defines a modeling paradigm that allows a SoS to be deployed via mediators. Besides, it allows to model constituents and aspects of the application, validate syntactic, semantic, and binary compatibility of the assembled constituents, and generate the systemic meta-data as descriptors to be used for middleware purposes (GOKHALE *et al.*, 2008). They assume a SoS as a *net-centric* and distributed set of intercommunicating systems.

Pavon present an approach to engineer SoS based on agents simulation. Models are used to specify/document SoS, analyze it, validate it, tackle target platform heterogeneity, using of transformations and model-driven technologies to generate graphical editors and model transformations/code generation and deployment (PAVON; GOMEZ-SANZ; PAREDES, 2011), with evaluation using a Water Management Policies System. Self-Management Modeling Language (SelfMML) is adopted to model the self-adaptation ability of a SoS architecture.

Systems Modeling Language (SysML), a semi-formal systems modeling notation, is adopted in many studies to model SoS architectures (MITTAL; MARTIN, 2013; BRYANS *et al.*, 2013; ANDREWS *et al.*, 2013). Despite that fact, SysML (OMG, 2017) lacks of specific structures to model some aspects of SoS software architecture SoS. It can represent multiple systems through the use of block diagrams, but its models are static, preventing representation of dynamic properties, as dynamic architecture and emergent behaviors (GUESSI *et al.*, 2015). Dahmann recently advocated the adoption of SySML as a suitable language to represent SoS architectures (DAHMANN *et al.*, 2017). They provide a working example of a Cross-Domain Maritime Surveillance and Targeting (CDMaST) SoS developed in Defense Advanced Research Projects Agency (DARPA), and report the use of IBM Rational Rhapsody to animate SySML models, making them executable. They indeed report the adoption of Unified Profile for DoDAF and MODAF (UPDM) as the architectural framework to document SoS using SySML. However, no clue on dynamic architectures and reconfiguration is given.

Unified Modeling Language (UML) is also proposed as a possible language to represent SoS software architectures (GUESSI *et al.*, 2015). Dagli and Kilicay-Ergin suggest to create a language for SoS stakeholders using DoDAF and use UML for capturing different SoS static views via models (GUESSI *et al.*, 2015). Cook et al. also propose MSC Assertions, which is a formal-language extension for UML message sequence diagram superimposed with UML statecharts to validate SoS runtime behaviors (COOK; DRUSINKSY; SHING, 2007; GUESSI *et al.*, 2015). Griendling and Mavris propose to use

UML in a methodology termed Architecture-based Technology Evaluation and discuss UML limitations to represent executable models, suggesting the use of Discrete Event notations (GRIENDLING; MAVRIS, 2011). Capability Tradeoff (ARCHITECT) UPDM is also another formalism identified to model SoS. It consists of an ADL that provides UPDM (Unified Profile for DoDAF and MODAF) provides a consistent, standardized means to describe DoDAF 1.5 and MODAF 1.2 architectures in UML-based tools as well as a standard for interoperability (HAUSE, 2010b; HAUSE, 2010a). Mordecai et al. also propose the use of UML and/or a UML profile (such as SysML) to implement MoBIE ontology, an ontology of a conceptual modeling framework for modelbased interoperability engineering (MoBIE) for SoS (MORDECAI; ORHOF; DORI, 2017).

Even eXtensible Markup Language (XML) has been recommended at some level to represent some aspect of a SoS (BARBI *et al.*, 2012; MITTAL; MARTIN, 2013; GOKHALE *et al.*, 2008). In general, XML is automatically generated to specify some configuration files for middleware or deployment.

Bigraph-based modeling offers a solution for mathematically representing SoS using graphs (WACHHOLDER; STARY, 2015; GASSARA; BOUASSIDA; JMAIEL, 2017; GASSARA *et al.*, 2017). Nodes represent constituents, and edges represent their interoperability links. Authors claim that bigraphs allow the modeling of SoS including their constituents (e.g., sub-systems, features, in- formation resources, etc.) by means of their structural as well as behavioral characteristics. Structural characteristics are materialized by links (i.e. system connectivity). They further specify reaction rules to model SoS constituents behaviors and reaction to external stimuli.

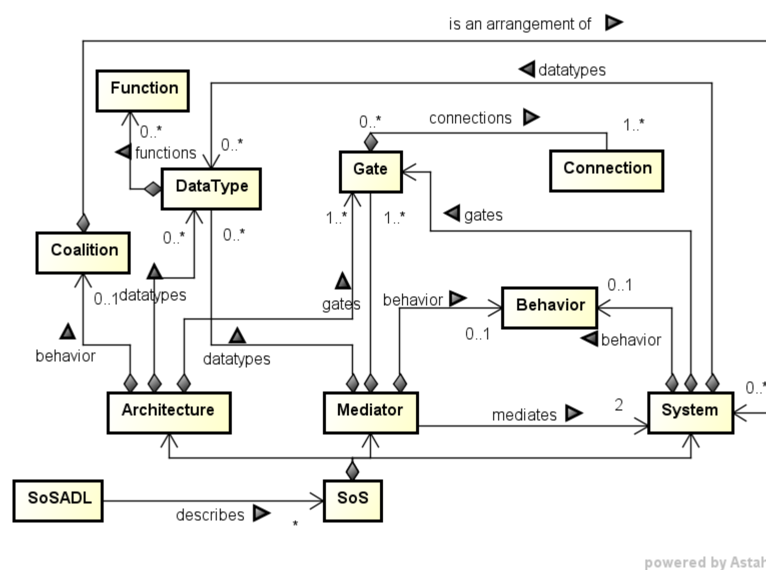

Figure 4 – An excerpt of SosADL abstract syntax (GRACIANO NETO, 2016).

SosADL is a novel ADL for specifying SoS (OQUENDO, 2016a; OQUENDO, 2016b; OQUENDO, 2016c). SosADL supports the specification of abstract architectures in which constituents are known at design-time and abstract connectors (also referred to as mediators) that can be dynamically realized for composing concrete architectures of this SoS. SoSADL is formally founded on π-calculus, while still offer a syntax for specification in high-level of abstraction (OQUENDO, 2016c). In SoSADL, architectures are abstractly defined by coalitions, which represent temporary alliances among constituents that collaborate via mediators (OQUENDO, 2016c). Such coalitions can be dynamically formed and changed at runtime. Coalition behaviors document how constituents interoperate to accomplish a given set of missions. Figure 4 shows an excerpt of SoSADL metamodel. Mediators are first-class elements representing communication links between two or more constituents (WIEDERHOLD, 1992). The architecture of a SoS defines policies for assembling abstract types of systems and mediators as coalitions, which are further characterized by behavior, data type, and gate declarations. Gates are abstractions that enable the establishment of connections. A connection can receive stimulus from or act on the environment, hence enabling the communication among independent elements. Data types can have inherent functions, and functions can be associated to expressions. Mediators and systems can also specified in terms of gates, data types, and behaviors.

SoSADL is an executable language, as it exhibits an operational semantics, i.e., a recipe of how SosADL statements should be interpreted to be converted in executable models (PLOTKIN, 2004; OQUENDO, 2016b). If an execution mechanism is not available, strategies (as model transformations) can be established to provide a dynamic view to enable the visualization of SoS dynamic architectures, a topic discussed in the next section.

2.2.4. Dynamic Software Architectures

SoS software architectures are inherently dynamic since constituents can freely join or leave the SoS structure at any moment. SoS dynamic architectures, also known as evolutionary architecture, are considered a consequence of the inherent operational and managerial independence of SoS constituents (OQUENDO, 2016a). Dynamic software architectures (DSA) are not a novel trend (OREIZY *et al.*, 1998; ALLEN; DOUENCE; GARLAN, 1998; LEMOS; GACEK; ROMANOVSKY, 2002; COSTA; PÉREZ; CARSÍ, 2007; MEDVIDOVIC; TAYLOR, 2010). They comprise of software architectures that exhibit dynamic reconfiguration ability, i.e., the ability to self-adapt its own structure at runtime due to a diversity of reasons (ZUNIGA-PRIETO *et al.*, 2018), including fault-tolerance (LEMOS; GUERRA; RUBIRA, 2006; ANDERSSON *et al.*, 2009). Such ability is essential to minimize system disruptions while new or modified constituents are being

joined into a SoS. Therefore, such characteristic is inherent to SoS and required to provide trust for SoS operation.

Several dynamic architecture description languages (DADLs) were proposed, including Rapide (LUCKHAM; VERA, 1995), Darwin (MAGEE; KRAMER, 1996), Dynamic ACME (GARLAN; MONROE; WILE, 1997), Chemical Abstract Machine-based formalism (WERMELINGER, 1998), Dynamic Wright (ALLEN; DOUENCE; GARLAN, 1998). However, they worked at component-level of single systems, dealing only with predicted components, with no support for multiple systems interoperating, constituents not known at design-time, and constituents joining or leaving a whole SoS at runtime. Such languages relied purely on structural models and captured a very limited perspective of the application being represented, consequently failing to support prediction of runtime behavior and sustainment of software operation, even for isolated systems (OREIZY *et al.*, 1998).

Advances have recently been achieved regarding DSA management. Cavalcante et al. addressed the existing gap between architecture descriptions and their respective implementations in the context of large-scale, dynamic software systems. They introduced the dynamic reconfiguration support provided by π-ADL (OQUENDO, 2004), a formal ADL for describing dynamic software architectures under structural and behavioral viewpoints based on an initial running architecture decomposed into its constituent architectural elements, modified and (re)composed to form a new, evolved architecture (CAVALCANTE; BATISTA; OQUENDO, 2015). They also advanced the state of the art by proposing two approaches for managing dynamic reconfiguration, namely: (i) an exogenous approach, in which an architectural element has the control over the other architectural elements and centralizes the reconfiguration actions to be applied on the architecture; and (ii) an endogenous, decentralized approach, in which the architectural elements themselves are able to perform the reconfiguration actions.

Research groups have investigated and proposed self-adaptation abilities for SoS architectures (FIRESMITH, 2010; ROMAY; CUESTA; FERNáNDEZ-SANZ, 2013; BATISTA, 2013; WEYNS; ANDERSSON, 2013). However, there is still a lack of studies on the specification and implementation of DSA, besides studies on their simulation and dynamic reconfiguration support.

2.2.5. Prescriptive and Descriptive Architectural Models

To attain to best practices when developing SoS, software architectures must be specified under two complementary perspectives (VALERDI; ROSS; RHODES, 2007; TAYLOR; MEDVIDOVIC; DASHOFY, 2010): (i) prescriptive models, which captures the design decisions made prior to the system's construction (it is the as-conceived or

as-intended architecture); and (ii) descriptive models, which describe how the system has been built (it is the as-implemented or as-realized architecture). Abstract architectures are the prescriptive architectures for SoS, as they define at design time the baseline from which a family of SoS architectural configurations (or concrete architectures) can be dynamically established. In turn, concrete architectures can be regarded as descriptive models, describing how individual actions of constituents are combined to produce a desired behavior. The concrete architecture is subject to simulation. As such architecture dynamically evolves, the new concrete architecture that results from such changes may conform (or not) to the original abstract architecture. When considering SoS context, such changes are frequent, and can be beneficial or harmful for SoS behaviors. A new architectural configuration may even offer better functionalities than the original architecture because it has new functionalities or a new structure that was not envisaged, and that emerged from the architectural changes. Regardless, abstract and concrete architectural models of a SoS can grow further apart, becoming increasingly inconsistent with each other.

Three different models are used for description of SoS architectures (PERRY; WOLF, 1992; HELDAL *et al.*, 2016; VALERDI; ROSS; RHODES, 2007; TAYLOR; MEDVIDOVIC; DASHOFY, 2010):

- SoS abstract architecture, which captures the SoS intended general structure, specifying the potential types of constituents that can join the SoS and the feasible links that they can establish to interoperate;
- SoS architectural instance, which consists of one possible SoS software architecture models. This model is one realization (between many possible instances) of the abstract architecture. It specifies the exact number and types of each constituent that can be part of such SoS, and the connections to be established between them; and
- SoS concrete architecture, which shows the runtime version of a SoS. Such models are often materialized as simulation models or the runtime real architecture and its corresponding model.

To prevent inconsistencies between these models, anti-erosion techniques can provide mechanisms for (GURP; BOSCH, 2002; GURGEL *et al.*, 2014): (i) explicitly defining the intended architecture of a system, which includes the description of dependency rules among components, and (ii) checking whether the system implementation conforms to the intended design. Mechanisms for controlling architecture erosion have been traditionally centered on architecture repair. This process typically involves (SILVA; BALASUBRAMANIAM, 2012): (i) using some reverse engineering mechanism to extract the implemented

architecture from source artifacts (recovery); (ii) mechanisms to infer its intended architecture (model discovery); and (iii) applying fixes to the eroded parts of the implementation (reconciliation). Regarding SoS, mechanisms must be established to bridge prescriptive and descriptive architectural models in order to avoid the aforementioned problems derived from architectural degradation. However, for SoS context, these techniques are still scarce. Besides, the concepts of architectural drift and erosion for SoS domain have not even been consensual.

2.2.6. Simulation Models for SoS

The ability to animate models can help one better understand modeled behavior. Novices and experienced developers will both benefit from the visualization of modeled behavior provided by model animators. Model animation can give quick visual feedback to novice modelers and can thus help them identify improper use of modeling constructs. Experienced modelers can use model animation to understand designs created by other developers better and faster (FRANCE; RUMPE, 2007). Simulations provide animations for models.

SoS architectures should offer a dynamic viewpoint to support a suitable evaluation of their dynamic behavior and architecture, besides predicting and preventing unexpected behaviors. Simulation-based approach is a well-known and well-accepted formalism for SoS development and for software architecture evaluation (DOBRICA; NIEMELE, 2002; MICHAEL; RIEHLE; SHING, 2009; BOSCH, 2000). Nonetheless, despite the ADL support for validation of software architectures and the crosscutting nature of ADL to the validation strategies, the majority of the known ADL have not supported both the specification and dynamic properties (emergent behaviors and dynamic architecture) of SoS architectures (GUESSI *et al.*, 2015). Simulations have supported dynamic properties for SoS (NIELSEN *et al.*, 2015; MITTAL; RAINEY, 2015; MICHAEL; RIEHLE; SHING, 2009; SAUSER; BOARDMAN; VERMA, 2010; ZEIGLER *et al.*, 2012; WACHHOLDER; STARY, 2015; FRANÇA; TRAVASSOS, 2016). Such approaches (MICHAEL *et al.*, 2011; FRANÇA; TRAVASSOS, 2016; WACHHOLDER; STARY, 2015; XIA *et al.*, 2013): (i) support the validation of expected emergent behaviors, (ii) empower the observation of unexpected emergent behaviors; (iii) enable the prediction of errors, diagnosing them and permitting corrections; and (iv) provide a visual and dynamic viewpoint, reproducing stimuli that the system can receive from an operational environment. DEVS is a well-known SoS simulation language to achieve SoS simulation (ZEIGLER *et al.*, 2012).

DEVS is a modeling formalism for SoS based on the idea of atomic and coupled models (TENDELOO; VANGHELUWE, 2017; VANGHELUWE, 2008; ZEIGLER *et al.*,

2012; MITTAL *et al.*, 2008). Atomic models represents individual entities in the SoS (for instance, systems), while coupled models represent a combination of atomic models. 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.

Bagdatli and Dimitri developed a notation and a simulation execution engine to model and execute a high level architecture of a system of systems, it was named as a High-level Architecture (HLA) Discrete Event Simulation: HADES (BAGDATLI; MAVRIS, 2012). They discuss a plethora of underlying simulation paradigms, including Probability Calculations, Markov Chains, Queueing Models, Petri Nets, Discrete Event Systems Specification (DEVS), and Agent Based. They conclude that DEVS is the best approach, and implemented their engine on that. Recently, Falcone et al. also proposed a simulation based on HLA (FALCONE *et al.*, 2017).

Tomson and Preden propose MACE simulation framework, consisting of a simulator application and a binding mechanism for agent and mediator code for simulating complex SoS that make use of a proactive middleware (TOMSON; PREDEN, 2013).

Vierhauser et al. describe the REMINDS tool suite for runtime monitoring of SoS developed in response to industrial monitoring scenarios. REMINDS provides tool support for interoperating and monitoring runtime systems, extracting events and data, defining constraints to check expected behavior and properties, and visualizing constraint violations to facilitate diagnosis.

Moisescu et al. propose the use of colored petri nets (CPN) to simulate SoS composed by cyberphysical constituent systems for precision agriculture domain (MOISESCU *et al.*, 2017).

2.2.7. Stimuli Generators

Modelling and simulation (M&S) are vital elements within processes for analysis and design of SoS. M&S enable visualization of SoS dynamics (VANGHELUWE, 2008; FRANCE; RUMPE, 2007; CARLE *et al.*, 2012; BALDWIN; SAUSER; CLOUTIER, 2015; FALKNER *et al.*, 2016). Several application domains adopt M&S (VANGHELUWE, 2008). Simulations correspond to an imitation of the operation of a real-world process or system

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

over time, and involve generation of artificial stimuli and the observation of its outcome to draw inferences about the operation of real systems that they represent (ZEIGLER *et al.*, 2012; VANGHELUWE, 2008; BANKS, 1999). As such, M&S promote: (i) a visual and dynamic viewpoint for SoS software architectures, reproducing stimuli the system can receive from a real environment; (ii) prediction of errors, diagnosing them and enabling corrections, and (iii) observation of expected and unexpected emergent behaviors of an SoS (BOSCH, 2000; SANTOS *et al.*,).

Baldwin et al. summarize current techniques found in the literature to simulate SoS (BALDWIN; SAUSER; CLOUTIER, 2015). Event-based modeling is the most prominent approach, as researchers can program different states a system undergoes to comprehend the behavior of the SoS as a whole (BALDWIN; SAUSER; CLOUTIER, 2015). In particular, DEVS is the most popular event-based simulation formalism (CHOI; KANG, 2013). It represents SoS, providing the required dynamic view of SoS. However, a straightforward generation of DEVS code does not guarantee the simulation is executable. This happens because the SoS operation is deeply related to the stimuli received from the environment that triggers the simulation execution. Hence, it is necessary to elaborate a specific entity in the simulation model that is responsible for delivering expected stimuli that drive the operation of the SoS: the stimuli generator.

Regardless of the approach adopted to simulate SoS, simulations often depend on some internal structure that imitates the surrounding environment of an SoS, delivering stimuli that are assumed to be received by the SoS to trigger its operation (INCOSE, 2016). The environment comprises the SoS surroundings, such as temperature, wind, water level, and noise; and and/or conditions in which a system operates, such as battery level and geographic position (GRACIANO NETO *et al.*, 2016). Environment is local to each system. By the nature of SoS, environments are only partially known at design-time (OQUENDO, 2016a).

There are two alternatives to deliver stimuli to a simulation (SANCHEZ-MONTANES; KONIG; VERSCHURE, 2002; BRUNEAU; CONSEL, 2013; RAHMAN *et al.*, 2014; PICCOLBONI; PRAVADELLI, 2014; YANG *et al.*, 2012). The first one is adding a portion of code to the body of each constituent in the simulation, randomly producing data (BOGADO; GONNET; LEONE, 2014). However, this approach brakes the separation of concerns principle, decreasing maintainability, as this code will be tangled to the constituent operational code. The second alternative is to materialize all stimuli into a single artificial entity known as *stimuli generator*. This structure becomes part of the simulated SoS, continuously delivering stimuli to SoS. Hence, stimuli generators imitates the SoS surrounding environment, automating the stimuli input (RAHMAN *et al.*, 2014;

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

PICCOLBONI; PRAVADELLI, 2014; YANG *et al.*, 2012; AL-HASHIMI, 1995; KITCHEN; KUEHLMANN, 2007; PLAZA; MARKOV; BERTACCO, 2007).

Developing stimuli generators require a careful investigation of SoS requirements and architecture specification to elicit which stimuli should be provided. Such tasks can bring additional cost to the SoS development and might be error-prone when a manual approach is used to transform architectural elements in software code. Moreover, stimuli generator can be used as an interface between the simulator actually employed and other industrial simulators used to imitate real environments, such as flight simulators, or a river simulation for flood monitoring SoS. This association between two types of simulator is known as *co-simulation* (BARTON; PANTELIDES, 1994; GOMES, 2016). This approach is broadly adopted by industry to large-scale test. Meanwhile, despite the potential of stimuli generators to support co-simulation approaches in SoS development, such approaches for automatically creating this stimuli generator for simulation of SoS have not been widely investigated.

Model-Based Engineering (MBE) techniques have been investigated in the context of SoS (FRANCE; RUMPE, 2007; GRACIANO NETO *et al.*, 2014; FALKNER *et al.*, 2016; GRACIANO NETO *et al.*, 2016; LANA *et al.*, 2016). They represent a software engineering approach in which models are the main basis, spanning all activities that make up the software development process (SENDALL; KOZACZYNSKI, 2003; NIELSEN *et al.*, 2015). MBE has been supported by a broad set of tools that are available to achieve a proper level of automation using transformation tools, such as Xtend (BETTINI, 2013) and Acceleo (ECLIPSE, 2012). Model transformations are the heart of MBE (SENDALL; KOZACZYNSKI, 2003). Model transformations are a well-accepted approach that aids software engineers in establishing correspondences between models (SUN *et al.*, 2008). It consists of a program, often written in a declarative manner, that transforms an input model in an output model (SENDALL; KOZACZYNSKI, 2003). MBE can be exploited to generate stimuli generators.

2.2.8. Evaluation, Testing, Verification, and Validation for SoS

Michael et al. discuss verification and validation in SoS domain (MICHAEL; RIEHLE; SHING, 2009; MICHAEL *et al.*, 2011). They propose alternatives for validation and verification in that domain, and introduce a mathematical model to link nonfunctional requirements of software systems to their SoS architecture, and present an approach to evaluate the quality of software architecture considering requirements.

Zapata et al., in turn, investigate testing for SoS (ZAPATA *et al.*, 2013). Authors claim that an important challenge in SoS Testing methodologies is to establish a test suite

that check that the complete SoS mission and objectives are achieved. This is a hard problem, as testing for every interface in the SoS leads to an exponential complexity. They propose a software testing technique named Basis Path Testing, which is a white-box technique that creates a control flow graph from each of the constituents behaviors to design an optimal test suite. This test suite is a set of paths that traverse through the functions, which are assumed linearly independent and that can be used to create a test strategy that will exercise all of the program's functions at least once to verify and validate their functionality. By applying Basis Path Testing analysis to the constituent systems in a SoS, the tester can develop an optimal test suite that will guarantee that all possible independent paths, all possible logical decisions, and all their interfaces are executed at least once.

Falkner et al. propose an approach for measuring performance of SoS architectures (FALKNER *et al.*, 2016). They developed an environment called MEDEA, which is a MBSE-based *system execution environment* that supports evaluation and performance prediction. They model SoS using many views through GraphML, a platform-independent language that supports modelling of interfaces, behavior, and workload of SoS and its constituents. They also perform simulations in their environment.

Meinke reports preliminary results of the creation of a Learning-based testing (LBT) for cyber-physical systems-of-systems (CO-CPS). Author proposes a paradigm for fully automated requirements testing that combines machine learning with model-checking techniques (MEINKE, 2017)

Yun et al. propose a mutation analysis approach for SoS policy testing (YUN; SHIN; BAE, 2017). Mutation analysis is a systematic way of evaluating test cases using artificial faults called mutants. As a general mutation framework for SoS policy testing, we present an overview of mutation analysis in SoS policy testing as well as the key aspects that must be defined in practice. Authors provide a case study using a traffic management SoS with the Simulation of Urban Mobility (SUMO) simulator. The results show that the mutation analysis is effective at evaluating fault detection effectiveness of test cases for SoS policies at a reasonable cost.

2.2.9. Deployment and Maintainability

Barbi et al. present a model-driven approach to configure and deploy SoS based on a framework for distributed applications (BARBI *et al.*, 2012). Configuration of SoS involves the production of many configuration files (at least, at this framework-based approach) which describes the structure of the SoS in general, configuration parameters, and interoperability issues. Such configuration files uses to reach a considerable size and

complexity due to the hundreds of lines of code. Additionally, handle text files manually is error-prone. Possibly, repetitive code must also exist in the resultant configuration file (not explicitly mentioned); the lack of a supporting tool. Three different architectural views are used: Structure View: it describes the structure of the components of a SoS with the relations and interactions between them; Deployment View: it describes the components distribution on the nodes (e.g. the A Process runs on the F Server); Activity View: it describes the boot and shutdown order of the system components. They create a tool called ACTUAL (Automation of the Configuration and deploymenT of distribUted AppLications) based on GMF. OCL constraints were added to to the metamodel to support the error control mechanism. XSLT and XSL were used to implement the transformations from models to configuration files.

Andren et al. also work on SoS deployment by proposing a DSL for Smart Grid Architecture Model (SGAM), using MBSE principle to automate and shorten the design process of use cases, also automating code generation and deployment of power utility applications (ANDRÉN; STRASSER; KASTNER, 2017).

2.2.10. Model Transformations

Model transformations are a valuable resource. Transformations in MBSE can be seen as a specialization of reduction procedures in computability theory, in which an algorithm transforms one problem into another problem. A transformation establishes traceability between two models, besides enabling a conversion of one formalism into another one. Many transformations have been carried out in MBSE domain for a diversity of purposes, including mapping between static and dynamic models, and automatic refinements in a same model (usually, a M2M transformation). Table 1 summarizes some instances of these transformations found in literature, the type of transformation, source models, and target models found.

Neema et al. present a framework for SoS simulation with Information Fusion capabilities (NEEMA *et al.*, 2009). DEVS is used to model the simulation environment; GME for infrastructure model interpretation; and Simulink to model the Unmanned Aerial Vehicles (UAV). A Case study on Command and Control (C2) domain is presented with a simulation of a urban atack with bombs. The combat team is based on autonomous vehicles (constituents) and those together acts as a SoS. Model-driven practices (MIC, specifically) are used to support a simulation environment for that scenario, and to perform models fusion. Transformations and metamodels are used, but there are no details about technologies they use to transform models, except for GME. Solution reported works only on one proprietary middleware platform. They use a set of views to describe the system

(analogous to 4+1 Kruchten views set), and a mission scenario is mentioned as a kind of model to represent missions, and they use Colored Petri Nets to model such missions.

Tu et al. present an ongoing research which uses MBSE to perform reverse engineering over legacy Enterprise Information Systems, and using the originated models to compose a Federation of Information Systems, with broad coverage of interoperability issues (TU; ZACHAREWICZ; CHEN, 2011). As legacy information systems (IS) often are required to interoperate and costs to implement new ones are high, they propose a process based on MDA to address what they call Model Reverse Engineering, culminating in a bidirectional transformation approach, to extract models from legacy IS, and to generate a Federation. Thus, bidirectionality does not cover the Federation/SoS reverse engineering as a whole, but only from their federates (constituents). They use High Level Architecture (HLA) as a basis to create the environment of distribution simulations (it is a framework for simulation too). They use MoDisco, a integrant part of Eclipse Modeling Project used to perform Model Discovery, an approach to extract models from legacy code.

Barbi et al. perform bidirectional transformations[7] by conducting a M2T transformation via XSLT to map SoS functionalities to configuration files for deployment purposes, from ACTUAL-model to XML, and vice-versa (BARBI *et al.*, 2012). OCL constraints were added to to the metamodel to support the error control mechanism. XSLT and XSL were used to implement the transformations from models to configuration files.

Gezgin et al. perform a M2M graph transformation to deal with dynamic architecture of SoS (GEZGIN *et al.*, 2012). They propose a visual approach and tool that run a graph transformation to automatically update a SoS architecture, converting one architectural configuration into another one. They mention a variation of UML called MECHATRONIC UML, and show an example of a SoS changing its architecture.

Hellestrand address cars as constituents of SoS (HELLESTRAND, 2013). He models a car using a DSL called Vehicle Dynamics Model (VDM), and transforms it to another formalism termed Specification-Based Architecture (SBA). He claims that many SBA models can be simulated in association, forming an executable model of SoS.

Belloir et al. transformed requirements expressed in a requirements engineering

[7] Currently, a bidirectional transformation is understood by MBSE community as a single mapping that works for both sides (source and target), i.e., it is capable of linking two models and identically transforming in forward and backward directions using the same transformation. This type of transformation is often specified in a bidirectional transformation language (such as BiGUL (KO; ZAN; HU, 2016)). However, as this is not a reality for SoS domain yet, for the scope of this thesis, a bidirectional transformation is that one in which a pair of transformations (one forward and another one backward) are used to link two SoS models. For the scope of this thesis, such type of transformation can also be referred as a round-trip engineering approach.

Table 1 – Transformations for SoS domain (Entries marked with asterisk (*) represent that transformations are actually mentioned or considered as a possibility of future work, but not performed).

Study ID	Types of transformations	Source languages and models:	TARGET Languages and Models
(BAY, 2002)	M2T	MATLAB	Code for Embedded systems
(TOLK; DIALLO; TURNITSA, 2007)*	M2M	Ontology	Ontology
(GOKHALE *et al.*, 2008)	M2M, M2T	COMPASS to OCML+EQAL, and later to XML (target)	XML code for Middleware for deployment purposes
(DICKERSON; VALERDI, 2010)	M2M	SysML	SysML
(CALINESCU; KWIATKOWSKA, 2010)	M2T	Metamodels in XML	C# .NET
(GE *et al.*, 2012)	M2M	DoDAF Metamodel (DM2) - Core Data Elements	Colored Petri Nets (CPN)
(GEZGIN *et al.*, 2012)	M2M	Mechatronic UML	Mecnatronic UML
(BARBI *et al.*, 2012)	M2T	Ecore, GMF, + OCL	XML
(HOLDEN; DICKERSON, 2013)*	M2M	SysML	Tactical Situation/Mission Scenario (TacSit/MS) Models
(BRYANS *et al.*, 2013; BRYANS *et al.*, 2014a; BRYANS *et al.*, 2014b)	M2M	SysML	CML
(ANDREWS *et al.*, 2013)	M2M	SysML	CML
(HELLESTRAND, 2013)	M2M, M2T	Vehicle Dynamics Model (VDM)	Specification-Based Architectures, Finite state machines, and software code for Electronic Control Unit (ECU)
(MOOIJ; VOORHOEVE, 2013)	M2T	Petri-Net Models	Executable Adapters
(HU *et al.*, 2014a; HU *et al.*, 2014d; HU *et al.*, 2014c)	M2M, M2T	UML/SysML Activity Diagram (AD) -> SoaML ->	DEVS -> DEVS
(BELLOIR *et al.*, 2014)*	M2M, M2T	Textual to RELAX, RELAX to KAOS	KAOS to OrBAC or KAOS to SySML (transformation chain, but as a possible work)
(BROWN *et al.*, 2015)*	M2M	SySML, UML	Simulink
(GROGAN; WECK, 2015)	M2M	HLA (High-Level Architecture) (IEEE, 2010)	FOM
(BADDOUR; PASPALIARIS; HERRERA, 2015)	M2M, M2T	SySML and Statecharts	SystemC
(GRACIANO NETO, 2016)	M2M	SosADL	DEVS
(GUESSI; OQUENDO; NAKAGAWA, 2016)	M2M	SosADL+Alloy	SosADL
(GRACIANO NETO *et al.*, 2017)	M2M	SosADL	DEVS
(SILVA; CAVALCANTE; BATISTA, 2017)	M2M	mKAOS	SosADL
(GRACIANO NETO *et al.*, 2018b)	M2M	SosADL	DEVS

language for Dynamic Adaptive Systems (DAS) called RELAX to KAOS, and subsequently to SySML models associated with security verification policies specified using OrBAC (BELLOIR *et al.*, 2014).

Grogan and De Weck adopt High-Level Architecture (HLA) to model SoS (GROGAN; WECK, 2015). HLA is a general purpose architecture for distributed computer simulation systems (IEEE, 2010). They convert HLA models in Federate Object Models (FOM) to enable a set of interoperable simulations that represent the operational SoS.

Bryans et al. present a set of views to represent interfaces in SoS. Four views are used and one more for structural representation (BRYANS *et al.*, 2013). Each view is represented by a SySML diagram, and after modeling a SoS, these models are used as input to manually transform them into CML (COMPASS Modelling Language), the first language developed specifically to model and design SoS. Views involve Interface Connectivity (SySML Block diagram, expliciting funcionalities offered and required for it constituent), Interface Definition (shows each funcionality signature, with datatypes, types used as input and returned types SySML diagram close to Class Diagram in UML), Interface Behaviour (SySML Sequence Diagram), and Protocol Definition (represent states and transitions of the SoS, using State Machine). A case study is performed using a flight booking example.

Gokhale et al. claim that large-scale Distributed Real-time Embedded (DRE) Systems are critical to many areas and uses to be interconnected via networks to form systems of systems (GOKHALE *et al.*, 2008). They claim additionally that delivering DRE systems supporting Quality of Service (QoS) properties is complex since DRE are mostly engineered using Component-Based Software Engineering and based on COTS of hardware and software, and since they have to be deployed with a middleware-based communication, and this part is subject to changes along the DRE development lifecycle because changes and evolution in COTS technologies (specially hardware) demands evolution in middleware configuration, what spend substantial time and effort. They propose an approach based on Model-Driven Middleware (MDM), which combines MBSE and QoS-enabled middleware to address composition and integration of such constituents, deploying them in a environment with a correct middleware configuration. They describe CoSMIC (Component Synthesis using Model-Integrated Computing), a technology that relies on COMPASS (COMPosable Adaptive Software Systems), a well-established language to solve problem of packaging component functionalities. They represent SoS using COMPASS, transforming such model for a combination of Option Configuration Modeling Language (OCML) and Event QoS Aspect Language (EQAL) (OCML+EQAL). Such models are subsequently transformed by CoSMIC tool in metadata documented in XML that can be used in the underlying

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

middleware.

Guessi et al. propose a methodology to address refinenement of SoS abstract architectures, automatically deriving feasible concrete architectures, i.e., SoS software architectures that are able to achieve pre-defined goals and properties (GUESSI; OQUENDO; NAKAGAWA, 2016). They adopt SosADL to specify abstract and concrete SoS software architectures. Besides, they adopt Alloy to specify constraints and properties on the abstract architectural model. This association of Alloy and SosADL is input for a model transformation, that generates a feasible SoS concrete architecture as output.

Silva et al. (SILVA; CAVALCANTE; BATISTA, 2017) proposed a M2M model-transformation to automatically refine m-KAOS missions, transforming them into SosADL models, in which constituents match the goals, if they are at the same semantic granularity.

Andren et al. (ANDRÉN; STRASSER; KASTNER, 2017) present a model-based approach that proposes a Domain-Specific Language (DSL) termed Power System Automation Language (PSAL) based on Smart Grid Architecture Model (SGAM)[8] to support a formal description a smart grid architecture (in particular its use cases). They implemented PSAL using the Xtext environment for the Eclipse IDE, and implemented a a validation test case for a laboratory environment. A transformation between PSAL and SCL (System Configuration Language)[9] was implemented.

Graciano Neto et al. propose a model transformation to map SosADL models in DEVS models, as explained during this thesis (GRACIANO NETO, 2016; GRACIANO NETO, 2017; GRACIANO NETO *et al.*, 2018b). In (GRACIANO NETO, 2016), simulation models created from SosADL models are used to validate emergent behaviors. In (GRACIANO NETO, 2017), a SosADL construct is used to automatically extract from the SoS software architectural description details about the surrounding environment modeling. As a result, a model transformation creates stimuli generators to immitate the surrounding environment of a SoS during its simulation execution. After, (GRACIANO NETO *et al.*, 2018b) report the use of SosADL2DEVS transformation as a means to obtain dynamic models that support architectural evaluation of SoS software architecture according to pre-defined functional properties, i.e., missions.

Holden and Dickerson (HOLDEN; DICKERSON, 2013) designed a model-based conceptual framework for termed Relational Oriented Systems Engineering and Technology Tradeoff Analysis (ROSETTA) framework for performing technology tradeoff and design studies with respect to flight training system of systems (FTSoS). They mention the

[8] A layered three-dimensional architectural framework to design smart grid architectures (ANDRÉN *et al.*, 2013).
[9] an XML based description language used to write configuration files for smart grid domain.

potential of their approach to support capability assignment to constituents, and an automated assessment of the resulting architecture via simulations. They propose to model missions in SysML via behavioral diagrams, and assess the SoS architecture via Monte Carlo Simulation.

2.2.11. Transformation Tools, Models, and Languages

Several MBSE tools and models have emerged and been used for SoS development. We can mention transformation languages as Acceleo and Xtend for M2T transformations (GRACIANO NETO *et al.*, 2018b); and ATL and QVT for M2M transformations. Xtext and Ecore are the most common platforms to specify models, metamodels, and grammars (ANDRÉN; STRASSER; KASTNER, 2017; SILVA; CAVALCANTE; BATISTA, 2017).

With respect to tools, Eclipse Modeling Framework (EMF), constructed under the Eclipse Modeling Projecta (EMP) [10], is a well-known instance that provides a large set of tools that deliver, besides model transformations support, several useful functionalities, as model management, model edition, visual model designs, and animation. Many of such tools offer both M2M and M2T transformations (CZARNECKI; HELSEN, 2003). EMF, in particular, is a complete modeling framework that offers a large range of tools and functionalities to support MBSE. It supports code generation facility, capabilities for building tools, and other applications based on structured models (STEINBERG *et al.*, 2009). It offers a complete set of technologies to perform all sort of operations over models.

EMF offers tools for (STEINBERG *et al.*, 2009):

- Abstract Syntax Development: Net4J[11], Teneo[12];
- Concrete Syntax Development: Graphical Modeling Framework[13], Graphiti[14], Xtext[15] (BETTINI *et al.*, 2013);
- Model Development: BPMN2[16], eTrice[17], MoDisco[18], OCL[19], Papyrus[20], Sphinx[21];

[10] http://www.eclipse.org/modeling/
[11] https://wiki.eclipse.org/Net4j
[12] https://wiki.eclipse.org/Teneo
[13] http://eclipse.org/modeling/gmp/
[14] https://eclipse.org/graphiti/
[15] https://eclipse.org/Xtext/
[16] http://eclipse.org/bpmn2-modeler/
[17] https://eclipse.org/etrice/
[18] https://eclipse.org/MoDisco/
[19] http://www.omg.org/spec/OCL/
[20] http://eclipse.org/papyrus/
[21] http://sphinx-doc.org/develop.html

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

- Model Transformation: ATL (M2M)[22], Acceleo[23], JET[24], Xpand (M2T)[25];
- Technology and Research: Atlas MegaModel Management[26], Atlas Model Weaver[27], MOFScript[28], VIATRA2[29], Epsilon[30];

Barbi et al. created a tool called ACTUAL (Automation of the Configuration and deploymenT of distribUted AppLications) based on GMF to implement the transformations from models to configuration files (BARBI *et al.*, 2012).

Mooij and Voorhoeve model a SoS using Petri net models in a tool called Marlene (MOOIJ; VOORHOEVE, 2013). They propose a methodology to solve incompatibilities issues between systems intended to be interoperated through the use of adapters, also known by them as mediators or glue logic. They define an adapter as a (small) additional system that is compatible with each of the original systems. They represent constituents using petri-nets, and automatically produce adapters (mediators) to support their communication.

Pavon presents INGENME (supports metamodel management, graphical editor generation for a metamodel, and code generation), INGENIAS Development Kit (IDK) as an agent-oriented language to model such systems; SelfMML to support self-management modeling (PAVON; GOMEZ-SANZ; PAREDES, 2011).

Baddour et al. propose SCV^2 tool, which allows SoS simulation. They adopted XSLT with the SAXON transformation engine to automate the simulator code-generation procedure, by automatically converting statechart diagrams (documented in SysML) into executable SystemC code.

Gassara et al. propose BiGMTE (BiGraph Matching & Transformation Engine), a tool for bigraph rewriting (GASSARA; BOUASSIDA; JMAIEL, 2017). BiGMTE tool provides an implementation of bigraph matching and transformation. It allows to execute the application of a reaction rule on a given bigraph to be rewritten. Execution is based on graph rewriting. Actually, they represent a SoS and its constituents by means of agents having a certain structure and behavior.

[22] https://eclipse.org/atl/
[23] https://eclipse.org/acceleo/
[24] http://eclipse.org/modeling/m2t/?project=jet
[25] https://www.eclipse.org/modeling/m2t/?project=xpand
[26] http://raweb.inria.fr/rapportsactivite/RA2006/atlas/uid26.html
[27] https://eclipse.org/gmt/amw/
[28] http://eclipse.org/gmt/mofscript/
[29] https://eclipse.org/viatra2/
[30] http://eclipse.org/epsilon/doc/etl/

2.3. Final Remarks

MBSE is not a panacea. There is an associated cost associated with its usage. However, it is precisely the attempt to bring benefits that avoids the damages already seen in other engineering by the use or neglect of models (SELIC, 2012). Literature reviews have been conducted in the last years about the adoption of models and their applications for SoS (RAMOS; FERREIRA; BARCELO, 2012; GRACIANO NETO *et al.*, 2014; GUESSI *et al.*, 2015; LANA *et al.*, 2016; WORTMANN; COMBEMALE; BARAIS, 2017).

Recently, Wortmann et al. proposed MBSE as a key enabler for complex SoS, and an element of what is known as Industry 4.0 (WORTMANN; COMBEMALE; BARAIS, 2017). After analyzing 222 studies, they noticed that 47 employ UML and variants, 36 use DSLs specific to Industry 4.0 challenges, 26 employ knowledge representation techniques, and 19 papers use AutomationML. 40% of the contributions address Industry 4.0 challenges with new DSLs, language profiles of UML or SysML, or metamodeling techniques, which can also be considered challenges for SoS.

We surveyed the literature to find a formalism that could properly support SoS software architecture modeling and simulation (GUESSI *et al.*, 2015). For this purpose, such formalism should meet the following language requirements, supporting:

1. SoS simulation,
2. Specification of dynamic architectures,
3. Multiple constituents modeling,
4. Constituents interoperability modeling, and
5. SoS software architecture specificities and precision, including environment modeling.

Table 2 summarizes the comparison between potential formalisms to support SoS software architecture representation, simulation, and stimuli generator derivation.

We decided to search for a software architecture notation, an ADL or modeling notation, that could support all the concepts necessary to represent SoS and that could be simulated. The following modeling languages have been identified as the key ones used for SoS architecture description: Darwin (semi-formal) (FOSTER *et al.*, 2011), Wright (formal)

Table 2 – Comparison between formalisms for SoS simulation and software architecture specification considering aforementioned language requirements.

Approach	1	2	3	4	5
Simulink/MATLAB	Yes	No	Yes	Yes	No
SysML	Yes	No	Yes	Yes	No
UML/Executable UML	Yes	No	Yes	No	No
DEVS (ZEIGLER *et al.*, 2012)	Yes	Yes	Yes	Yes	No
CML	Yes	No	Yes	Yes	No
Darwin	No	Yes	No	No	No
Wright	No	Yes	No	No	No
π-ADL	No	Yes	Yes	Yes	No
SoSADL	No	Yes	Yes	Yes	Yes

(ALLEN; GARLAN, 1997), π-ADL (formal) (OQUENDO, 2004), UML[31] (semi-formal), SysML[32] (semi-formal), and SoSADL (OQUENDO, 2016a) (GUESSI *et al.*, 2015).

Wright and Darwin were not designed to model SoS architectures. The aforementioned requirements are not covered as these ADL were created to model monolithic systems. SysML was the backbone of two European projects (COMPASS[33] and DANSE[34]) for which they developed extensions for SoSs. DANSE did not develop an ADL, but used SysML for semi-formally describe executable architectures that are then tested against contracts. However, SysML is a UML Profile, and not necessarily an ADL. Moreover, the adoption of SysML to model SoS would require multiple models, each one being simulated individually, and the simulations being interoperated, what is costly. Then, SysML does match our approach requirements, and, despite being adopted for software architecture modeling, is not strictly an ADL. UML shares the same drawbacks.

COMPASS developed a formal approach, in contrast to DANSE that extended a semi-formal one. In COMPASS, CML was specifically designed for SoS modeling and analysis. However, 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 a SoS and CML is used to enrich these specifications with contracts. A CML model is defined as a collection of process definitions (based on CSP/Circus), which encapsulate state and operations written in VDM as well as interactions via synchronous communications. CML is a low-level formal language, of which a key drawback is that SysML models when mapped to CML results in huge unintelligible descriptions (it was

[31] http://www.omg.org/spec/UML/2.5/
[32] http://www.omg.org/spec/SysML/1.4/
[33] http://www.compass-project.eu/
[34] http://danse-ip.eu/home/

one of the lessons learned from COMPASS) (OQUENDO, 2016a).

Finally, π-ADL is a formal language to model distributed architectures. However, despite π-ADL provides architectural description models for concurrent and communication processes, it does not provide straightforward abstractions for some SoS' particular concepts, such as mediator, coalition, environment modeling, and it does not support modeling of abstract architectures and simulations. SoSADL is more expressive than π-ADL for the description of SoS software architectures, with additional elements, such as gates, duties, guarantees, properties and mediators. In SoSADL, architectural descriptions are intentional and abstract, whereas in π-ADL, such descriptions are declarative and concrete. In addition, from a formal point of view, SoSADL includes other formalisms besides the π-calculus, which is the only one that π-ADL possesses.

Hence, only SoSADL matched the majority of requirements we raised. SoSADL is a language formally founded on π-Calculus for SoS, a novel process calculus extended from original π-calculus, conceived for enabling the formal architecture description of software-intensive SoS. It can be considered correct by construction, as the formal semantics of such calculus is defined by means of a formal transition system, expressed as labeled transition rules, which are formulated as proof rules (OQUENDO, 2016b). In short, SoSADL describes SoS, which can be expressed as a combination of architecture, systems, and mediators declarations[35]. Each architecture declaration is expressed in terms of its intrinsic behavior, data types, and gates, i.e., abstractions that enable the establishment of connections. A connection is established to receive stimulus from or act on the environment, or to simply communicate with other constituents. Furthermore, a connection can be used to receive, send, or do both actions. Data types can have inherent functions, and functions can have associated expressions. Mediators and systems as well as the SoS architecture itself also 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 representation of emergent behavior by means of a coalition, i.e., a temporary alliance that allows constituents to perform a combined action. These emergent behaviors are specified as part of the coalition behavior, documenting how constituents should interact to accomplish a given set of missions[36].

SoSADL an executable language founded on an operational semantics defined in a formalism termed as π-calculus (OQUENDO, 2016b). SoSADL demands an execution mechanism that runs their models, converting the specification of the SoSADL constructs

[35] Mediators are architectural elements that establish communication between two or more constituents (WIEDERHOLD, 1992)

[36] Additional details about the syntax of architecture descriptions in SoSADL can be found in (OQUENDO, 2016a).

in operational portions. Such mechanism, materialized as a model transformation that generates simulation models, matches the majority of our requirements and offers a denotational executable semantics with DEVS that conforms to the operational semantics in π-Calculus for SoS. SosADL holds the environment description, whilst a simulation formalism run the stimuli generators created.

Simulation formalisms (or notations subject to simulation) such as Simulink/MATLAB[37], Executable UML (HU *et al.*, 2014a), or SySML[38] do not support any one of the aforementioned requirements. Even if such formalisms could represent SoS constituents by means of multiple models, each one representing one individual constituent, hence, simulating them would require interoperability between individual simulations (known as distributed simulations), which demands further adaptations and costly implementations. DEVS, in turns, was developed specially to represent SoS architectures. Hence, we chose DEVS as the formalism to simulate our SoS software architectures.

However, even DEVS lacks important characteristics for expressing SoS software architectures, such as: (i) the language only deals with the notion of ports; there is no notion of connections and gates separately, that is a remarkable paradigm of software architectures (Components, Connections, and Values/Constraints define a software architecture (BASS; CLEMENTS; KAZMAN, 2012; ISO, 2011)) used by SoS modeling (CAVALCANTE; BATISTA; OQUENDO, 2015); (ii) In DEVS, every major entity of an architecture is represented as the same type of model (called atomic model). As the single abstraction available, an atomic model prevents a complete characterization of the software architecture with the diversity and typical heterogeneity of constituents that form a SoS; (iii) even though it supports environment modeling, its inherent syntax does not have any specific mechanism for representing the surrounding environment, which is an important aspect of any software architecture, including software architectures of SoS (ISO, 2011); (iv) it does not offer a mechanism to automatically create stimuli generators; and, finally, (v) since SoS architectures are dynamic, i.e., their constituents are not necessarily known at design-time and they can join or leave the SoS at runtime, it lacks the notion of abstract architectures, i.e., a description of constituents and their *potential connections* with other constituents, and how they could be adapted at runtime. Even though DEVS supports dynamic reconfiguration, i.e., the modeling and simulation of architectures of SoS that adapt themselves at runtime, the language still lacks support for abstract architectures, thus requiring all constituents that take part in the simulation to be known. Therefore, it is not allowed for a new constituent, i.e., a constituent that has not been predicted at design-time, to join the coalition at runtime.

[37] https://www.mathworks.com/products/simulink.html
[38] http://www.omgsysml.org/

As observed in this chapter, model transformations have been conducted between different formalisms, and around 20 different formalisms have been proposed and used to describe and simulate SoS architectures and other aspects. However, many of them still lack support for dynamic architectures, emergent behaviors, and environmental modeling, which are essential for SoS software architectures. Next chapter discusses how we associated SosADL models and DEVS models in an approach termed ASAS, which deals with SoS software architecture evaluation considering its inherent dynamic architecture.

CHAPTER

3

ASAS: A MODEL-BASED APPROACH FOR THE SIMULATION AND EVALUATION OF SOFTWARE ARCHITECTURES OF SYSTEMS-OF-SYSTEMS

Systems-of-Systems (SoS) often support critical application domains and exhibit dynamic architectures that, despite their several different configurations at runtime, they should assure a correct operation. Architectural description languages (ADL) have not captured the SoS inherent dynamism or enabled architectural visualizations, which hampers the making of decisions on possible, valid, or best architectural configurations assumed by a SoS during runtime. This chapter presents ASAS, an approach for the automatic generation of simulation models by a model transformation and support to SoS architectural evaluation. ASAS also predicts architectural arrangements that leverage SoS operation and supports the evaluation of pre-determined quality attributes. The approach was evaluated by means of a case study with two different scenarios, namely Space SoS and Flood Monitoring, in which the model transformation and support for the evaluation of SoS behaviors were assessed. ASAS successfully supported the evaluation through automatically generated simulations, enabling the observation of (i) multiple missions in a same SoS, (ii) SoS dynamic architecture, and (iii) the percentage of achievement of each mission in each SoS, with a large set of constituents in both cases.

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

3.1. Presentation of ASAS Approach

ASAS prescribes the use of SosADL for the documentation of SoS software architectures, DEVS as a formalism for simulation (ZEIGLER; KIM; PRAEHOFER, 2000), and a model transformation that maps both formalisms. SosADL is semantically founded on π-calculus for SoS (OQUENDO, 2016b) and provides all abstractions required for the specification and validation of SoS behaviors. As a matter of fact, the transformation establishes an approach that maps all those abstractions related to SosADL to DEVS. To be executable, a language must have an operational semantics, i.e., a recipe for the interpretation of SosADL statements, so that they become executable models (PLOTKIN, 2004). Our approach provides such an operational semantics for SosADL through an equivalent denotational semantics, i.e., a formalism that specifies the meanings of statements and expressions in programming languages (STOY, 1977). Model transformations provide the operational semantics for SosADL models through DEVS executable models, i.e., DEVS models have equivalent definitions of the way SosADL models should be executed.

ASAS is structured according to the following steps:

Step 1. Design of a SoS architecture in SosADL: the SoS software architecture must be specified, and, in our approach, this step must be performed using SosADL. A SoS software architecture can be specified in abstract or concrete level. Abstract architectures represent potential connections that can be established between the potential types of constituents to be part of such SoS, as constituents are not necessarily known at design-time. The derivation of concrete instances of an abstract SoS software architecture was investigated in (GUESSI; OQUENDO; NAKAGAWA, 2016). In our approach, we adopted an initial concrete instance that can be changed at runtime for a better fitting of stakeholders' requirements and adaptation to the stimuli received from the environment;
Step 2. Evaluation planning: this step comprises the selection of metrics and aspects to be measured during the SoS architectural evaluation that must be deliberately planned, and include a set of missions to be evaluated through SoS simulations;
Step 3. Execution of the model transformation: after the evaluation plan has been structured and the SoS software architecture has been specified in SosADL, the model transformation can be executed for the production of the DEVS simulation code used for evaluation purposes;
Step 4. Deployment: this step involves the management of the files obtained as the outcome of the transformation, and their deployment in a MS4ME[1] environment;
Step 5. Simulation Execution and Architectural evaluation: this step consists in the launching of the simulation in an MS4ME environment, monitoring through observation

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

and possible interaction with the simulation through the exercising of multiple SoS architectural configurations. Data and execution traces are logged in text files during this process to be further examined; and

Step 6. Analysis: it consists in the inspection of the execution traces in log files, and drawing of conclusions according to a pre-established set of missions and corresponding metrics.

3.1.1. Correspondences between SosADL and DEVS models

ASAS is based on a DEVS dialect called DEVS Natural Language (DEVSNL) (ZEIGLER *et al.*, 2012), which enables the programming of atomic and coupled models expressed as DEVS in a human-like format by tools, as MS4ME. Since DEVS and SosADL are founded upon rigorous formalizations, a transformation should preserve the correspondences between the fundamental concepts. Expressions and values are suppressed in this representation[2].

Table 3 – Mapping between SosADL and DEVS.

SoS concept	SosADL	DEVS
Capability	Behavior Declaration	Atomic Model Behavior
Connection	Connection Declaration	DEVS Port
Constituent System	System Declaration	Atomic Model
Data Types	Data Type Declaration	Data Type
Function	Function Declaration	DEVS Function
Gate	Gate Declaration	DEVS Port
Mediator	Mediator Declaration	Atomic Model
SoS Architecture	Coalition	Coupled Model

Correspondences were established between both models, as shown in Table 3. SoS concepts are mapped onto SosADL and DEVS, as follows:

Connections. A connection can be established for receiving stimuli from an environment, acting on the environment or simply enabling communication between constituents. A communication is established when data are sent by a constituent and received by another. Connections are mapped, in DEVS, into ports (input and output ports).

Constituent System. Systems play the role of constituents in SoS. When transformed

[2] More details on the syntax of architecture descriptions in SosADL and its elements can be found in (OQUENDO, 2016a).

into DEVS models, constituents become atomic models. They are represented as single entities in the simulation model, and their behaviors are expressed as a state machine.

Data Type Declaration. Data types can be specified for the scope of a SoS or constituent or a mediator. Data types and values in SosADL become data types in DEVS. They can be abstract data types (ADT) or simple data, as state variables, and become state variables, global values, or ADT, which can be exchanged in messages among systems in DEVS, since the state transitions in DEVS are triggered by message content.

Functions. Functions are blocks of code that can be used within a behavior specification. Function declarations are converted to functions in DEVS.

Gates. In SoS and SosADL, gates are abstractions that enable the establishment of connections, and connections are abstractions of links between gates of distinct communicating entities. The gate concept is suppressed, and the connections linked to that gate become a DEVS Port.

Mediator. Systems are mediated by mediators, which are architectural elements concerned with the establishment of communication links between two or more constituents (WIEDERHOLD, 1992; INVERARDI; TIVOLI, 2013). When transformed into DEVS models, they follow the same rationale of constituent systems, i.e., mediators also become atomic models.

SoS Architecture. An architecture declaration has an intrinsic behavior declaration (coalition), data types, and gates declarations. SoS can be expressed as a combination of architecture, systems, and mediators. Coalitions constitute temporary alliances for combined actions among systems connected via mediators and are dynamically formed to fulfill the SoS mission through emergent behaviors (OQUENDO; LEGAY, 2015). In fact, they guide the way interactions between the constituents will be performed and how their functionalities are explored to accomplish a mission. Such structures are summarized in the coupled model and specify the way they compose the SoS architecture and promote the SoS operation.

Figure 5 shows the approach for the implementation of the solution proposed. An SoS architectural description written in SosADL is verified against the EBNF abstract syntax of SosADL described in Xtext[3]. If the SosADL code conforms to the Xtext grammar, the code is submitted as input to an Xtend[4] script that represents the Code Generator. A functional code written in DEVSNL is generated as output.

[3] <https://eclipse.org/Xtext/>
[4] <http://www.eclipse.org/Xtend/>

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

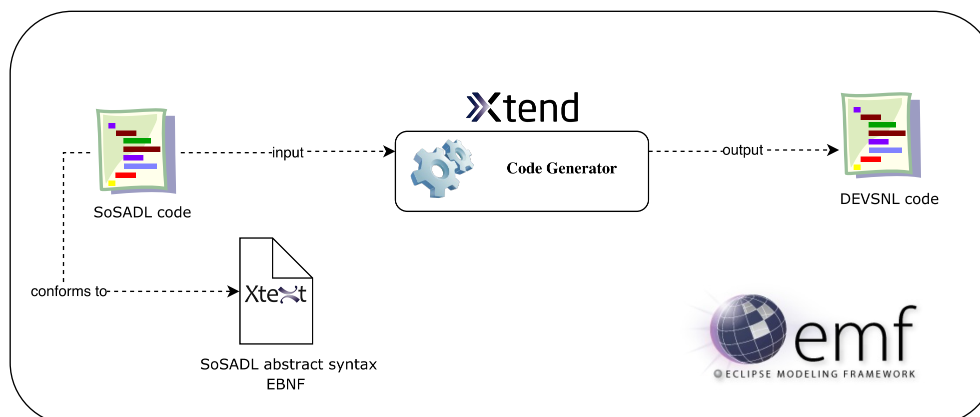

Figure 5 – An approach for the transformation of SosADL models into DEVS simulation models (GRACIANO NETO, 2017).

3.1.2. Generation of Constituent Models

Listing 11 shows a simplified code of a mediator specified in SosADL. This code is mapped into an atomic model written in DEVS depicted in Listing 13. The transformation is performed by the code specified in Xtend available in Listing 12. In Listing 11, data types are defined on Lines 2-6. Duties (in this context, only a name for the designation of gates and mediators) with their respective connections are defined on Lines 8-16. The behavior of the mediator is specified between Lines 18 and 23, and shows that the mediator (i) receives constituents coordinates (Lines 19 and 20), (ii) receives data from the sensors (Line 22) and (iii) forwards such data to a gateway (Line 23). This sequence of actions is performed in a loop. Details on the model transformation that maps SoSADL models into DEVS models are available in Anex B.

```
1  mediator Transmitter( distancebetweengates:Distance ) is {
2    datatype Abscissa
3    datatype Ordinate
4    datatype Coordinate is tuple { x:Abscissa, y:Ordinate }
5    datatype Depth
6    datatype Measure is tuple { coordinate:Coordinate, depth:Depth }
7
8    duty transmit is {
9      connection fromSensors is in { Measure }
10     connection towardsGateway is out { Measure }
11   }
12
13   duty location is {
14     connection fromCoordinate is in { Coordinate }
15     connection toCoordinate is in { Coordinate }
16   }
17
18   behavior transmitting is {
19     via location::fromCoordinate receive coordinate
20     via location::toCoordinate receive coordinate
```

```
21        repeat {
22          via transmit::fromSensors receive measure
23          via transmit::towardsGateway send measure
24        }
25      }
26    }
```

Source code 1 – Code in SosADL for a mediator.

Lines 2-6 in Listing 11 represent the definition of data types in SosADL. Data types are transformed by Lines 1-16 in Listing 12 to produce Lines 1-22 in Listing 13. Lines 8-16, which specify the connections and duties (gates) of a mediator in SosADL in Listing 11, are transformed into Lines 24-27 in Listing 13, whereas Lines 18-23, which represent the behavior of a mediator specified in SosADL in Listing 11, are transformed into Lines 24-42 in Listing 13.

```
 1 A Distance has a value!
 2 the range of Distance's value is Integer!
 3 use distance with type Distance!
 4
 5 A Abscissa has a value!
 6 the range of Abscissa's value is Integer!
 7 use abscissa with type Abscissa!
 8 A Ordinate has a value!
 9 the range of Ordinate's value is Integer!
10 use ordinate with type Ordinate!
11 Coordinate has x and y!
12 the range of Coordinate's x is Abscissa!
13 the range of Coordinate's y is Ordinate!
14 use coordinate with type Coordinate!
15
16 A Depth has a value!
17 the range of Depth's value is Integer!
18 use depth with type Depth!
19 Measure has coordinate and depth!
20 the range of Measure's coordinate is Coordinate!
21 the range of Measure's depth is Depth!
22 use measure with type Measure!
23
24 accepts input on FromCoordinate with type Coordinate!
25 accepts input on ToCoordinate with type Coordinate!
26 accepts input on FromSensors with type Measure!
27 generates output on Measure with type Measure!
28
29 to start hold in s0 for time 1!
30 hold in s0 for time 1!
31 from s0 go to s1! //Unobservable
32 passivate in s1!
33 when in s1 and receive Coordinate go to s2!
34 passivate in s2!
35 when in s2 and receive Coordinate go to s3!
36 passivate in s3!
```

```
37 when in s3 and receive Measure go to s4!
38 hold in s4 for time 1!
39 after s4 output Measure!
40 from s4 go to s5!
41 hold in s5 for time 1!
42 from s5 go to s3! //Unobservable
```

Source code 2 – An atomic model for a mediator generated in DEVSNL.

3.1.3. Patterns for SoS Simulation

Simulations are software that rely on dozens of lines of code and demands techniques for its engineering. Simulation codes are often driven by labeled state machines based on discrete input and output events, as Discrete Event Systems (DEVS) (ZEIGLER *et al.*, 2012). However, some of the instructions are conflicting, and the number of lines of code can be huge, which leads to difficulties and high costs for production and maintainability. Such state machines can reach large dimensions, which makes the production of codes repetitive and error-prone. Under this perspective, the identification of patterns can aid the conception of simulations for SoS, supporting the automatic generation of codes from specifications in a high level of abstraction, such as architectural specifications of software of SoS.

Patterns are standard solutions for recurrent problems that emerge in a domain (VLISSIDES *et al.*, 1995). As they enable the reuse of solutions, they can (i) contribute to the trustworthiness expected from SoS, once the reuse of a well-succeeded solution can foster the construction of the correct product, and (ii) reduce costs and time. Since simulation has become increasingly dominant and used by various industries, techniques and methods must be designed for the effectiveness of developers (GRAY; RUMPE, 2016). However, the literature lacks techniques and software engineering methods for simulations in SoS context. SoSE is a novel discipline, and general principles and patterns still must be discovered[5]. DEVS is well-recognized formalism for specification of SoS simulations. In DEVS, constituents operations are specified via a labeled state machine, i.e., a state machine in which transitions occur due to data input or output, or time elapsed. DEVS variants include probabilistic, non-deterministic, and finite deterministic ones. As non-determinism is unfeasible, deterministic DEVS versions are more common on platforms, such as FD-DEVS (Finite Deterministic DEVS), implemented in platforms as MS4ME[6].

In DEVS, a constituent system is driven by a state machine specified according to

[5] J. Fitzgerald, S. Foster, C. Ingram, P. G. Larsen, and J. Woodcock. Model-based engineering for systems of systems: the COMPASS manifesto. COMPASS, October 2013.

[6] http://www.ms4systems.com/pages/main.php

a well-defined set of primitives. A state in DEVS can be either a 'hold state', or a 'passive state' (exclusively). The execution flow is maintained at a hold state for a certain amount of time until it is automatically changed to another state (via an internal transition). On the other hand, in a passivate state, simulation remains indefinitely (or until it receives a message that triggers an external transition). Below are the basic constructs for a state machine in DEVS:

Passivate State (PS). State in which the execution flow remains until some event (input or output) has caused a transition to another state.

```
passivate in STATENAME!
```

Hold State (HS). State in which the execution flow remains for a well-defined time, such as 5 seconds.

```
hold in STATENAME for time 5!
```

Initial State (IS). State marked as initial for the simulation execution. A simulation holds only one initial state specified by the following syntax:

```
to start passivate in STATENAME!
or
to start hold in STATENAME for time 5!
```

Internal Transition (IT). A simple transition that specifies the current state and a next state. Every HS must exhibit one internal transition.

```
from FROMSTATE go to TOSTATE!
```

Output Transition (OT). A transition that produces an output of a message prior to the internal transition.

```
after STATENAME output OUTPUTMESSAGE!
```

External Transition (ET). Transition that defines an input message the model might receive. Such a message causes a transition from a state to another. Both states and the expected input message must be specified. Any state can have one or more external transitions defined. The syntax is:

```
when in FROMSTATE and receive INPUTMESSAGE go to TOSTATE!
```

The code of a state diagram in DEVS is based on inputs and outputs, and consists of an arrangement of statements that guide the operation of a constituent. However, System Engineering guides usually do not teach how to group those statements conveniently for the

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

avoidance of conflicts (for example, a hold state also marked as passivate causes conflicts), which may cause fail, error, or interruption of the simulation.

Since DEVS is based on inputs and outputs, we have established one pattern for input and another one for output. The context is the same for both (DEVS simulation models), and recurrent problem changes only in function of the purpose, i.e., input or output. The former prescribes an input causes a transition when, at some state, it receives a datum, whereas, according to the latter, once an output occurs spontaneously (without any triggering event), it should remain in that state for one second (the time can be specified according to convenience), perform the output, and transit to the next state.

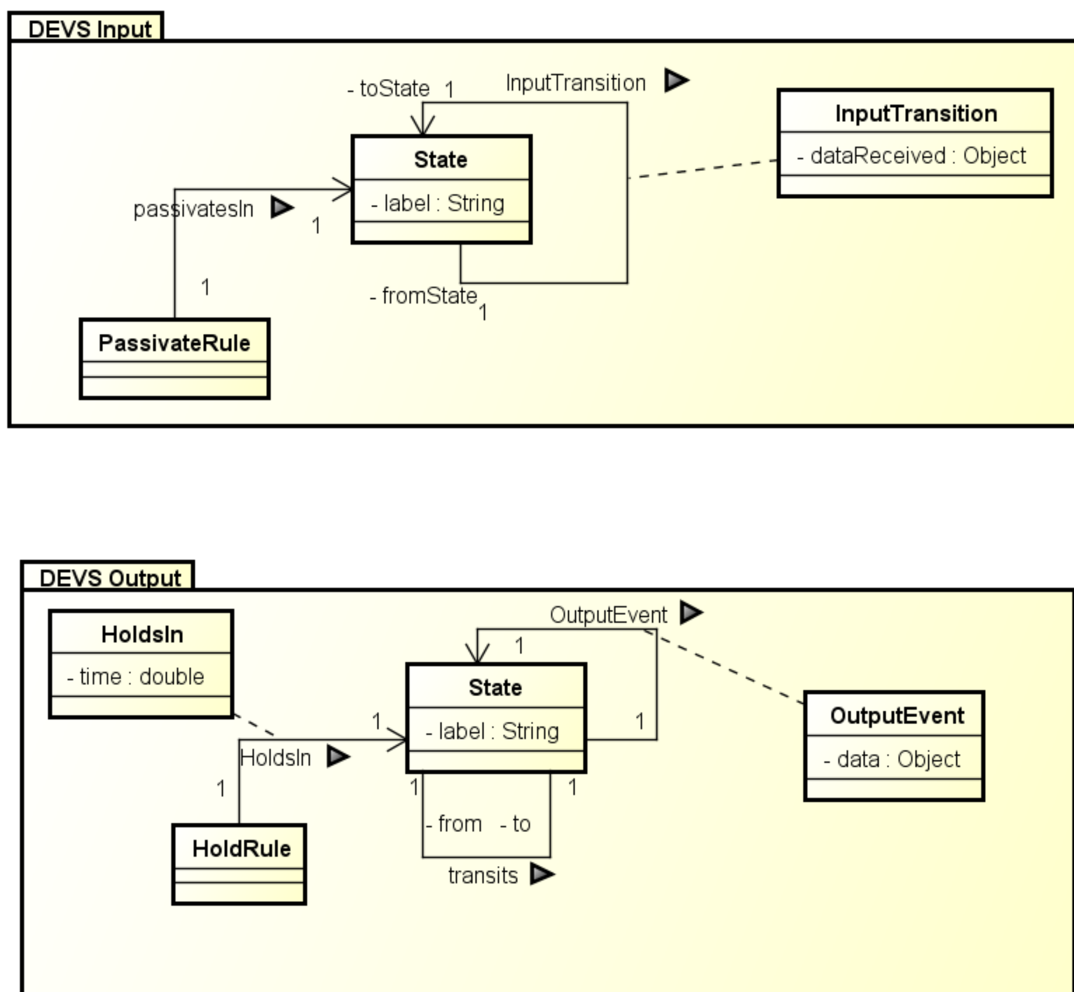

Figure 6 – Patterns expressed as diagram classes in UML.

Figure 6 depicts our patterns expressed as UML class diagrams. DEVS Input rules specify a `PassivateRule` that passivates in one and only one state, whose name is represented by a label. From this state, an `InputTransition` occurs when a pre-determined

type of data is received and causes the transition from one state to one and only one another state. DEVS Output rules specify a `HoldRule` that holds in one and only one state for a pre-determined amount of time. From that state, an output event occurs for the delivery of some data, and the execution flow transits to another state.

Statement/ Statement	PS	HS	IS	IT	OT	ET
HS	X					
IS						
IT	X					X
OT	X					X

Table 4 – Conflicts and compatible instructions in DEVS.

Table 4 shows the potential conflicts identified:

- **HS-PS**: A hold state cannot be concomitantly a passivate state, and vice versa. Specification of a state as hold and passivate might cause a conflict in a state, as such state should, at the same time, wait for some pre-specified time and also wait indefinitely. Therefore, such combination is unfeasible;
- **IT-PS**: Internal transitions are automatically triggered after an amount of time, e.g., after one second, go to another state. As such, a state should not be specified as a passivate state together with a specification of an internal transition, as it might cause a spontaneous transition at any moment;
- **IT-ET**: An internal transition cannot be concomitantly an external transition, and vice versa, since they have a different nature;
- **OT-PS**: Technically, an output transition is an internal transition. Since a passive state cannot have an internal transition, it cannot have an output transition.
- **OT-ET**: Since an internal transition cannot be an external transition it cannot be an output transition.

Such conflicts must be taken into consideration in the design of robust SoS simulations. Therefore, two patterns have emerged to encapsulate such principles: one that groups codes representing input transitions and their sub-activities involved, and another representing output transitions. Their establishment relied on the classical Gamma's structure (VLISSIDES *et al.*, 1995), i.e., recurrent problem, context, and solution. Since DEVS is based on inputs and outputs, we have established one pattern for input (Table 5) and

Name	DEVS Input
Recurrent Problem	Specification of a set of simulation instructions that characterize an input event without conflicts with other instructions.
Solution	`passivate in s<<fromState>>!` `when in s<<fromState>> and receive << dataReceived>> go to s<< toState>>!`

Table 5 – Patterns for input in DEVS simulation models.

another for output (Table 6). The context is the same for both (DEVS simulation models) and the recurrent problem changes only in function of the purpose, i.e., input or output.

DEVS Input

According to the DEVS Input pattern, an input will cause a transition when, at some state, it receives a datum. If `passivate` comes after input instruction `when`, it might cause a conflict with a `hold` of a following output instruction.

DEVS Input (Figure 6) specifies a `PassivateRule` that passivates in one and only one state, whose name is represented by a label. From this state, an `InputTransition` occurs when a pre-determined type of data is received and causes the transition to another state.

DEVS Output

Once an output occurs spontaneously (with no triggering event), the execution flow should remain in a state for one second (this time can be specified according to convenience), perform the output, and transit to the next state. If the next state causes an input, it will be subject to either a `passivate` instruction, or a new `hold`.

DEVS Output (Figure 6) specifies a `HoldRule` that holds in one and only one state for a pre-determined amount of time. From that state, an output event occurs, delivering some data, and transiting from one state to another state.

3.1.4. Generation of Coupled Models

In ASAS approach, a SoS architectural description specified in SosADL is verified against its metamodel expressed in Xtext[7] during the transformation. If the SosADL code

[7] https://eclipse.org/Xtext/

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

Name	DEVS Output
Recurrent Problem	Specification of a set of simulation instructions that characterize an output event without conflicts with other instructions.

Solution	`hold in s<<fromState>> for time 1!`
	`after s<<fromState>> output << dataType>>!`
	`from s<<fromState>> go to s<<toState>>!`

Table 6 – Output pattern for DEVS simulation models.

conforms to this metamodel, the code is used as input to an Xtend[8] script that performs the transformation mechanism and returns a functional code written in DEVS. Coupled models in DEVS specify the way constituent systems exchange data with each other to exhibit an emergent behavior. The code of such coupled models systematically specifies the entities involved in the SoS and the way they interact, i.e.,the systems that send data and those that receive them. In SosADL, SoS software architectures are modeled as coalitions. The correspondences between SosADL and DEVS are summarized in Table 25.

Table 7 – Mapping of SosADL into DEVS.

SoS concept	**Representation in SosADL**	**DEVS**
Constituent Systems	Coalition	Decomposition
Data Types	Data Type	Data Type
Gate/Connection	Gate/Connection	DEVS Port
Interfaces	Binding	Coupling
SoS Architecture	Coalition + Binding	Coupled Model

Constituent Systems. In SosADL, the list of all constituent systems that compose the software architecture of an SoS is represented by a Coalition. By definition, coalitions are alliances of constituents connected via mediators. When translated into DEVS, coalitions are mapped into a DEVS Decomposition, i.e., a statement of the coupled model that systematically lists all inner structures (e.g., systems, mediators, among others) that form the software architecture of the SoS (ZEIGLER *et al.*, 2012).

Data Types. When a communication is established between constituents and they start to interoperate, data are exchanged between them. Indeed, SoSADL relies on *typed*

[8] xtend-lang.org/

connections, i.e., connections with a specific type of data. Data types must be preserved by the transformation and properly converted into DEVS format.

Gate/Connection. Gates are a structure through which connections can be established. Since the notion of connection does not exist in DEVS, each SosADL Connection is mapped as a port in DEVS.

Interfaces. The concept of interface encapsulates the communication between two entities (in this case, systems). Consequently, a detailed analysis of an interface should contain (explicitly or not) functions *send* located in one system element, and *receive* located in another [9]. In SosADL, interfaces are specified through bindings, which correspond to the list of all combinations between output ports and input ports that establish a communication between two entities in a SoS. In DEVS, each binding is mapped into a coupling, i.e., a statement describes the way information flows between two systems in the SoS.

Sos Architecture. Finally, the software architecture of an SoS is represented as an abstract architecture in SosADL, which specifies a coalition and a set of bindings, and subsequently mapped in a coupled model, which is a set containing a decomposition and couplings.

Listing 16 depicts a SosADL code that represents the specification of a software architecture of an FMSoS. In Listing 16, the software architecture of SoS represented comprises four `sensors`, one `gateway`, and four `transmitters` (types of mediators) (Lines 4 to 12). `bindings` (Lines 13 to 23) represent the way connections between constituents and mediators are established through gates, and SoS dynamics for data transmission until a gateway. A sensor collects the `water level` through actuators, encapsulates it with the specific `location` in which the collecting was performed, and a time stamp. The sensor then transmits the data to the closest mediator, which forwards them to the next sensor, until the gateway has been reached.

```
1 sos FloodMonitoringSos is {
2   architecture FloodMonitoringSosArchitecture( ) is{
3     behavior coalition is compose {
4     sensor1 is Sensor
5     sensor2 is Sensor
6     sensor3 is Sensor
7     sensor4 is Sensor
8     gateway is Gateway
9     mediator1 is Mediator
10    mediator2 is Mediator
11    mediator3 is Mediator
12    mediator4 is Mediator
13    } binding {
14    relay gateway::notification::alert to warning::alert and
```

[9] SEBoK. Guide to the systems engineering body of knowledge, version 1.6, 2016.

```
15      relay gateway::request to request and
16      unify one { sensor1::measurement::measure }
17         to one { mediator1::fromSensors } and
18      unify one { mediator1::transmit::towardsGateway }
19         to one { sensor2::measurement::pass } and
20      unify one { sensor2::measurement::measure }
21         to one { mediator2::fromSensors } and
22      unify one { mediator2::transmit::towardsGateway }
23         to one { gateway::notification::measure } and
24      unify one { sensor3::measurement::measure }
25         to one { mediator3::fromSensors} and
26      unify one { mediator3::transmit::towardsGateway }
27         to one { sensor4::measurement::pass } and
28      unify one { sensor4::measurement::measure }
29         to one { mediator4::fromSensors} and
30      unify one { mediator4::transmit::towardsGateway }
31         to one { gateway::notification::measure }
32 }  }
```

Source code 3 – Description of an architecture of an FMSoS in SosADL.

In SosADL, a connection is specified as `system :: gate :: connection`. Indeed, the same gate can hold one or more connections. A unification is established for each pair of sensors with a mediator between them by a `unify` statement (Lines 15-23). According to such statements, an output connection `measure` from the gate `measurement` is linked to the input connection `fromSensors` of the closest mediator. A mediator gathers data from a sensor (Lines 11 to 14) and forwards them to the next sensor. Mediators have an output connection termed as `towardsGateway`. Such connections are linked to the sensors through an input connection called `pass` in the gate `measurement` to receive the data transmitted and forward them to the gateway (Lines 16, 18, 20 and 22). Lines 22 and 23 link the output connection of the mediator to the gateway connection called `measure`. In this case, a mediator mediates a constituent and the gateway. The `relay` statement establishes the communication between the SoS and external systems, connecting the notification gate of a gateway to an external connection. Each binding specified in SosADL is mapped into a coupling in DEVS. Listing 18 shows the equivalent code derived from the coalition according to the transformation rules specified in Xtend depicted in Listing 17, available in details in Anex B. Line 1 in Listing 18 shows `FloodMonitoringSoSArchitecture` is formed by the same systems specified in the SosADL code. Lines 2 to 9 show the data exchange among all systems and mediators derived from the specification of the coalition. These lines are created by iterating on the unifying statements. A line is created for each of the unifying connections specified in the SosADL model. Finally, DEVS tool converts that code into an executable simulation model.

According to Listing 18, sensors transmit data to their closest mediator (Lines 2, 4, 6, and 8), which receives them in Lines 3, 5, 7, and 9 forwards `Measure` to the next sensors.

```
1 From the top perspective, FloodMonitoringSosArchitecture is made of
2 Sensor1, Sensor2, Sensor3, Sensor4, Gateway, Mediator1, Mediator2, Mediator3,
3 and Mediator4!
4
5 From the top perspective,  Sensor1 sends Measure to Mediator1!
6 From the top perspective,  Mediator1 sends Measure to Sensor2!
7 From the top perspective,  Sensor2 sends Measure to Mediator2!
8 From the top perspective,  Mediator2 sends Measure to Gateway!
9 From the top perspective,  Sensor3 sends Measure to Mediator3!
10 From the top perspective,  Mediator3 sends Measure to Sensor4!
11 From the top perspective,  Sensor4 sends Measure to Mediator4!
12 From the top perspective,  Mediator4 sends Measure to Gateway!
```

Source code 4 – Coupled model for FMSoS generated in DEVS.

Since `Sensor2` and `Sensor4` send their data to `Mediator2` and `Mediator4`, respectively (Lines 4 and 6), the gateway is reached (Lines 5 and 9). When data arrive in the gateway, their values are tested against a pre-determined depth threshold. If they are higher, the gateway emits a flood alert. Therefore, the network of exchanged messages between constituents and the flood alert trigger indicate the SoS mission has been accomplished.

3.1.5. A Dynamic Reconfiguration Mechanism for Supporting SoS Dynamic Architectures

SoS architectures are inherently dynamic. Therefore, coupled models generated by the model transformation must provide strategies for dealing with changes that may occur in the software structure. Such changes must be well-defined and the final result must be the same whenever a change is performed, as a change should conduct the SoS to a new functional state. The literature reports well-established sets of architectural changes for single systems software architectures (CAVALCANTE; BATISTA; OQUENDO, 2015). However, when multiple interoperable systems that form a SoS are considered, a gap must still be bridged. This section provides a canonical set of dynamic changes that can affect the SoS software architectures and well-defined steps for their execution.

Dynamic reconfiguration in a SoS is based on four types of architectural changes, namely addition of constituent, deletion of a constituent, substitution of the constituent, and reorganization of the architecture. They are invoked at simulation time by the simulation user and executed by a reconfiguration controller (further explained), which will perform the necessary changes in the architecture.

Addition. Addition of a constituent into the simulation of a SoS software architecture depends on a well-defined set of steps. Firstly, a new constituent must be created and added to the simulation. It must be linked to the SoS for effectively joining it. A constituent is selected in the architecture to be linked to the new constituent. Then an appropriate

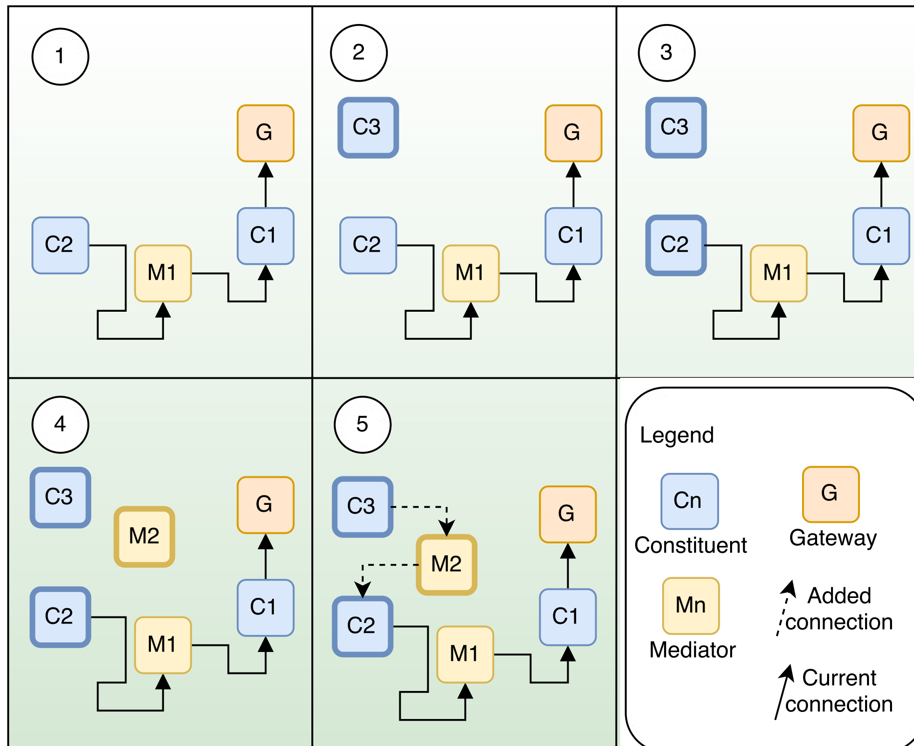

Figure 7 – Illustration of the addition of a constituent in a simulation of SoS software architecture.

mediator is instantiated and added to the simulation. Connections between constituents and the mediator are established. The following steps are performed, as shown in Figure 7:

1. The SoS simulation user issues a request to add a particular constituent to a reconfiguration controller. Scene 1 represents the state of SoS at that time. The constituent to be added is C3;

2. Constituent C3 is instantiated and added to the simulation;

3. The controller selects another constituent in the simulation to be connected to C3 (in this case the C2);

4. The controller instantiates and adds an appropriate mediator M2 to the simulation; and

5. The controller adds connections between C2 and C3, which are then mediated by M2.

Remotion. The remotion process is initiated by the user who explicitly chooses a constituent to be removed. Such a constituent sends a unique identifier to the controller, and constituents connected to it are listed. These connections are eliminated and mediators are

removed. The controller searches for constituents in the architecture that can be connected to the remaining ones. This important step maintains the SoS operation in progress. At the end of the process, the controller removes the constituent from the simulation, as described in the steps below and illustrated in Figure 8.

1. The SoS simulation user issues a remotion request for a particular constituent. Scene 1 represents the state of SoS at that time. The constituent chosen to be deleted is C1;

2. The connection between constituent 1 and the mediator that connects it with the remainder of the SoS is undone;

3. The binding between mediator 1 and the remainder of the SoS is undone and mediator 1 is selected to be deleted; and

4. Both (C1 and M1) are removed.

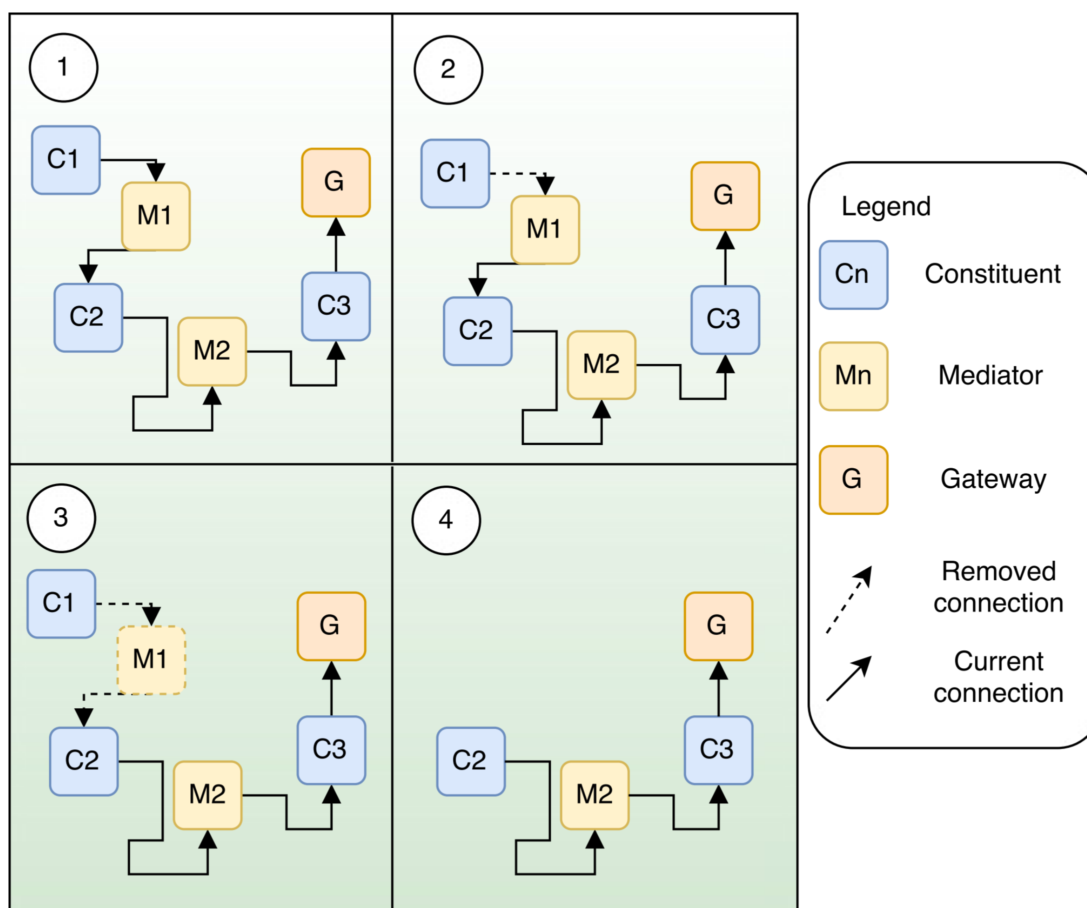

Figure 8 – Process of remotion of constituent in a simulation of a SoS software architecture.

Substitution. It is performed as a concatenation of the remotion and addition operators. Steps are followed as shown in Figure 9

1. The SoS simulation user issues a request for the removal of constituent C1;
2. The connection between constituent 1 and the mediator that connects it with the remainder of the SoS is undone;
3. The connection between mediator 1 and the remainder of the SoS is undone, mediator 1 is selected to be eliminated and C2 is chosen to connect C3;
4. Both (C1 and M1) are eliminated, and the user adds the constituent C3 which is instantiated and added to the simulation;
5. The controller instantiates and adds a compatible mediator (M2) to the simulation; and
6. The controller adds the necessary connections to connect C3 with C2. M2 starts to mediate them.

Reorganization. Architectural reorganization process consists in the complete dissolution of the architectural configuration, and reestablishment of new connections between the constituents, which leads to a new operational state of SoS, as shown in Figure 10. The controller removes all connections between the constituents and all mediators, except connections between the controller and the constituents. The controller arbitrarily chooses a constituent in the simulation (random process) and tries to establish its connections with other constituents. The process is repeated until all elements have been connected in SoS, which results in an architecture configuration different from the original one.

The SoS dynamic architecture is managed at runtime by a *Dynamic reconfiguration controller* (DRC). Figure 41 shows the way DRC interacts with the SoS architecture simulated. DRC is an artificial architectural element that manages every architectural change that occurs. It is added to the simulation to enable the simulation user to perform architectural changes at runtime. From the DEVS simulation model perspective, the reconfiguration controller is an atomic model that (i) adds constituents to the simulation, and the necessary connections and mediators, and relinks the properties of the initial architecture; (ii) removes the constituents of the simulation, connections and mediators, relinking the remaining constituents for maintaining an operational SoS architecture; (iii) removes the constituent and replaces it for another one; and (iv) reorganizes the architecture by removing all connections and mediators and thereafter establishing different mediated connections for creating a new architectural configuration while retaining the initial

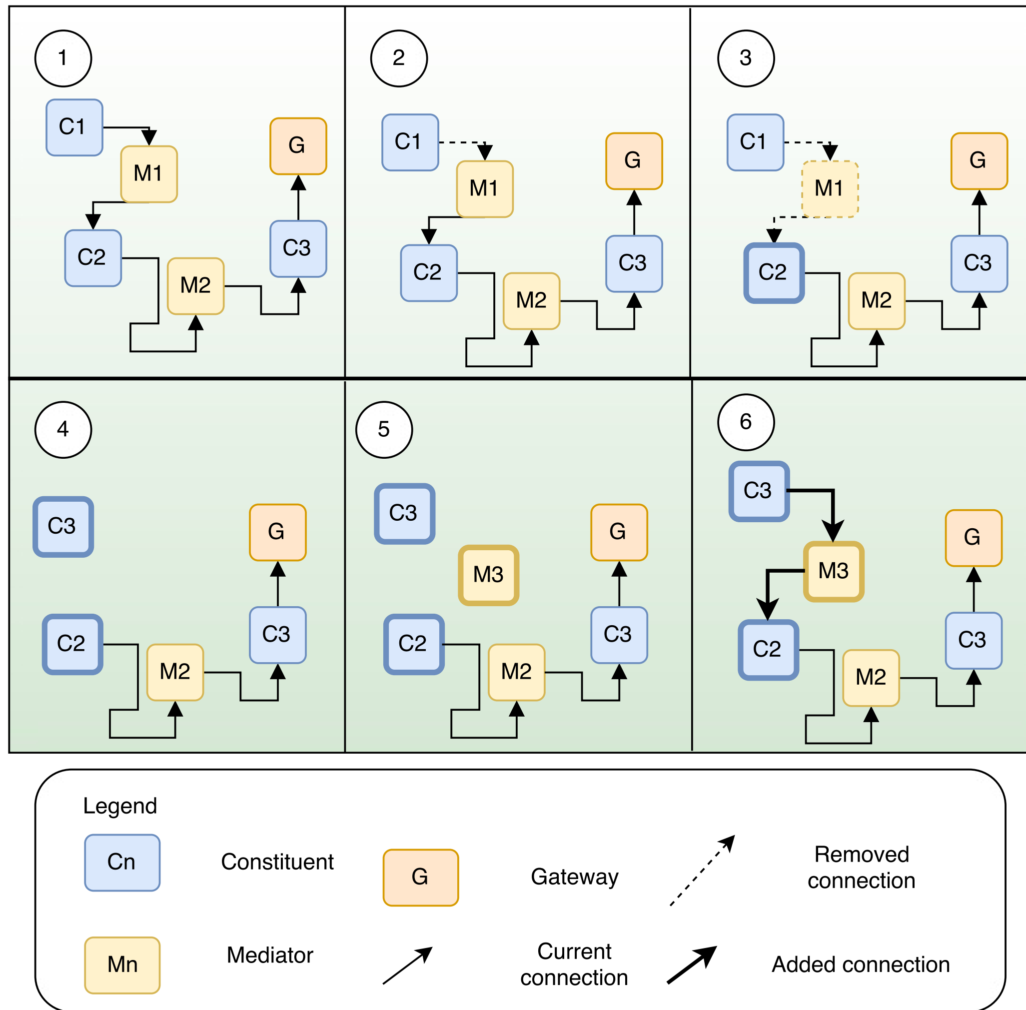

Figure 9 – Substitution of constituent in the simulation of a SoS.

architectural properties. Details on the inclusion of this element in the DEVS simulation are discussed in Appendix B.

3.2. Evaluation

ASAS was conceived to support the evaluation of the SoS software architectures functional characteristics, i.e., missions. ASAS enables architects to evaluate SoS behaviors, as well as their dynamic architecture. A case study for investigation on the feasibility of ASAS was conducted. The protocol encompasses the following steps (RUNESON; HöST, 2009): (i) Case study design (Preparation and planning for data collection), (ii) Execution (Collection of evidence), (iii) Analysis of collected data, and (iv) Reporting. Goal-Question-Metric (GQM) technique was adopted for case study (BASILI; CALDIERA; ROMBACH, 1992).

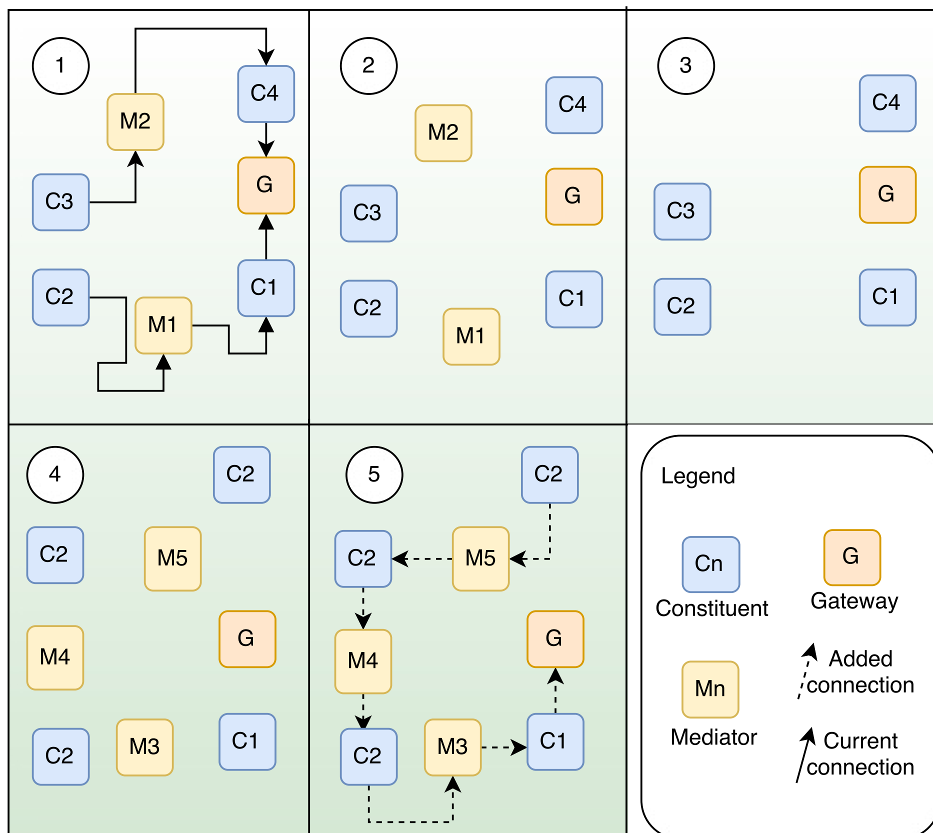

Figure 10 – Process of reorganization of architecture in the simulation of a SoS software architecture.

Goal: to assess whether ASAS approach enables the transformation, simulation, and evaluation of SoS software architectures in regards to its functionalities, while still considering dynamic architecture. The following research questions and respective set of metrics were established:

RQ1: Can the transformation successfully produce functional simulation models?

Rationale. Since the simulation model is automatically generated, soundness of the produced model must be checked. Therefore, a transformation can be considered successful if the simulation runs without errors, and its output is similar to the behavior observed in the real SoS.

Metric M1. Simulation failures: given by the number of detected failures during model simulation.

RQ2: Does ASAS support evaluation of SoS and their dynamic software architectures?

Rationale. Along the SoS operation, constituents can join or leave the SoS, which pro-

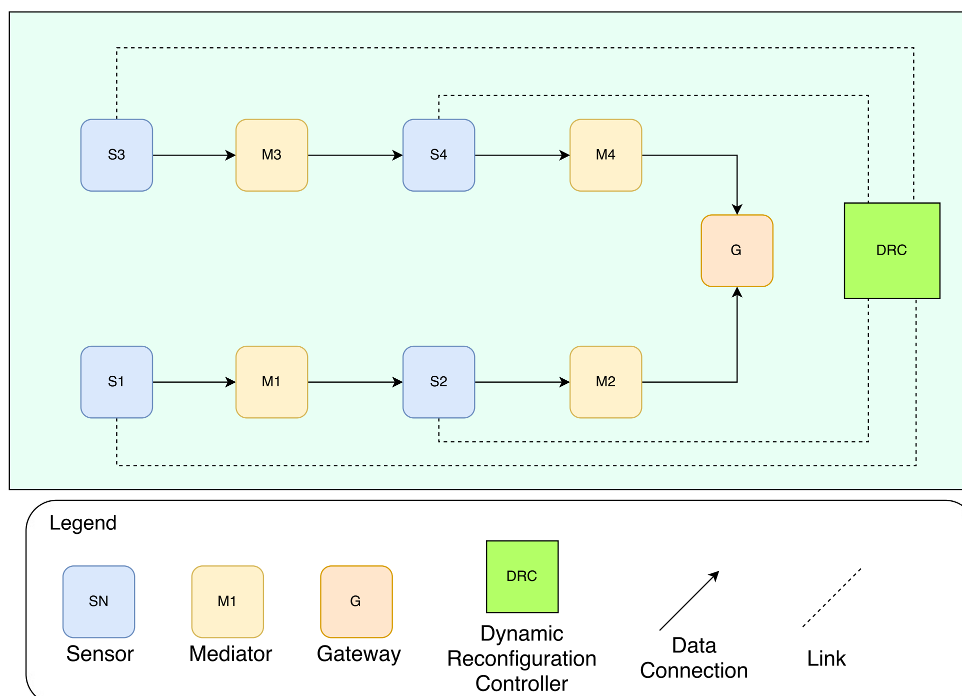

Figure 11 – An illustration of the relation between DRC and constituents simulated.

duces different architectural configurations (coalitions). This question investigates if the approach supports the comparison of different architectural configurations, and election of the one that best suits SoS needs.

Metric M2.1 Accuracy, the percentage in which the SoS operat ion is reliable in a diversity of architectural configurations for the achievement of their pre-defined missions;

Metric M2.2 Data loss, i.e., the percentage of data that does not arrive in the final destination due to data collision;

Metric M2.3 Scale: Number and diversity of constituents achieved by SoS during its simulation; and

Metric M2.4 Support for decision on architectural configuration, whether the simulation enables collection of data and analysis to decide which architectural configuration offers best results.

RQ3: Is ASAS domain-independent?

Rationale. ASAS should be applied to multiple domains to be considered a valid simulation-based evaluation approach for SoS software architectures. Therefore, this question investigates whether ASAS can be adopted to more than one domain.

Metric M3. Number of application domains: given by the number of different domains in which ASAS was successfully applied;

RQ4: Can simulation patterns be recurrently applied?
Rationale. We proposed patterns for automatically generating behaviors with no conflicting instructions for constituents simulation. Therefore, we aimed at assessing whether such patterns could be broadly reused in different solutions. This research question supports the investigation on the potential of reuse of the solutions proposed.
Metric M4. Effectiveness: given by the number of functional atomic models and state machine lines of code effectively generated for constituents in DEVS, which represent the number of times the same patterns were recurrently applied.

ASAS was evaluated in two different scenarios, namely a Flood Monitoring SoS and a Space SoS, so that data from different sources could be gathered for the drawing of conclusions[10]. Context and results are reported in the following sections.

3.2.1. Scenario 1: Flood Monitoring SoS

ASAS was evaluated in a case study on a Flood Monitoring SoS (FMSoS), which is a SoS intended to be part of a smart city. FMSoS monitors rivers crossing urban areas, which pose great danger in rainy seasons, potentially damaging property, threatening lives, and spreading diseases. It notifies possible emergency situations to residents, businesses owners, pedestrians, and drivers located near the flooding area, and governmental entities and emergency systems. Moreover, it is intended to be part of a larger SoS composed of Wireless River Sensors, Telecommunication Gateways, Unmanned Aerial Vehicles (UAVs), Vehicular Ad Hoc Networks (VANETs), Meteorological Centers, Fire and Rescue Services, Hospital Centers, Police Departments, Short Message Service Centers and Social Networks, as described in (OQUENDO, 2016c). Such SoS involves the National Center for Natural Disaster Monitoring, which monitors 1000 cities, with 4700 sensors, including 300 hydrological sensors, and 4400 rain gauges.

Step 1. Design of an SoS architecture in SosADL: FMSoS was specified via SoSADL. Its architecture was designed to be composed of five different types of constituents, as illustrated in Figure 34, and described below:

1. **smart sensors**: cyber-physical systems that monitor flood occurrences in urban areas, located on river edges;

2. **gateways**: devices that gather data from constituents and share them with external entities;

[10] R3 and R4 are both answer in Section 3.2.4 during the case study synthesis process.

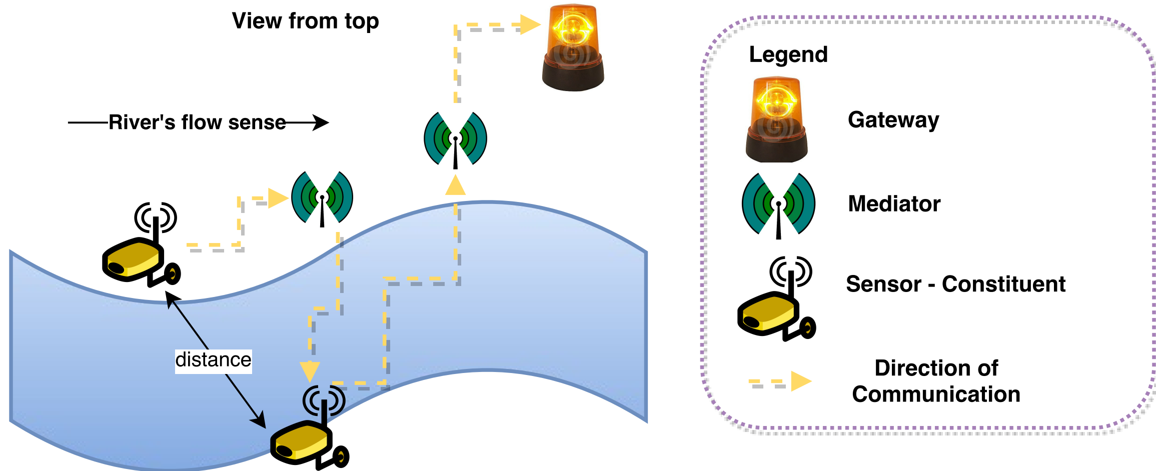

Figure 12 – A flood monitoring system-of-systems (FMSoS) (GRACIANO NETO *et al.*, 2016).

3. **crowd-sourcing systems**: mobile applications used by citizens for real-time communication of water level rising. In such systems, the danger level is a pre-defined value between one and six. One represents no risk, whilst six represents a flood effectively occurring. Human users can classify flood risk according to their observations;

Data Preparation. We chose a dataset collected by the actual FMSoS over four days, from November 23th 2015 to November 27th 2015. This interval was important because during these months a number of floods occurred. This enabled us to establish whether or not our simulation results in a diversity of situations. We established *a 4-window strategy* implemented at the gateways that receive data from constituents to confirm floods. For each set of four data that subsequently arrives, the gateway checks them. For the period studied, the river had an average rise from 35 to 50 cm, depending on the location. Thus, in this context, the threshold of a flood is defined as a rise of 100 cm or more. If at least one pair of data that arrived have both their depth levels at least 100 cm (the threshold established for that city), a flood alarm is triggered. Table 8 illustrates a numerical instance. It corresponds to real data that arrived sequentially at the gateway. Data that arrive are chronologically ordered, and pairs of data are analyzed. If at least one pair has two measures equal or greater than 100 cm, a flood is confirmed. Subsequent measures will confirm if it is an actual flood or not.

Table 8 – A sample of data collected by a sensor and sent to a gateway.

sample id	sensor	timestamp	depth (cm)
#1	S2	2015-11-23 01:58	58

Data were stored in text files and delivered by the stimuli generators along the FMSoS, feeding the simulation. These stimuli generators delivered 1,000 samples for each

sensor. Timestamps represented that each data sample was sent every five minutes for each sensor (i.e., 12 samples by hour, 288 per day, totalizing 3,47 days of data simulated). We also considered an amount of 1,000 samples for each crowd-sourcing system that is part of the SoS. We adapted our dataset so to have similar data for stimuli generators for crowd-sourcing systems. For crowd-sourcing systems, the aforementioned scale was used to classify risk between 0 and 6. Zero means that crowd-sourcing systems is not contributing to flood diagnosis. We mapped the height of water in original data to a scale of risk between 1 and 6, 1 being no risk, and 6 being flood effectively occurring. Human users can classify a risk between these values according to what he/she sees. So we could imitate how people would react and behave according to the changes in water level registered before by sensors. Then, we created a dataset corresponding to the data used to feed sensors.

Step 2. Evaluation planning: This step was conducted according to the research questions and metrics established and presented in this Section. A comprehensive data set provided by a real project that combines crowd-sourcing and sensor data for detecting floods was chosen (HORITA *et al.*, 2015) and data collected from November 23th 2015 to November 27th 2015 (four days), a period of intense rains and floods were used.

Step 3. Execution of the model transformation: After the preparation of all material, the model transformation was run and produced the corresponding DEVS models with no errors.

Step 4. Deployment: All DEVS models were accordingly deployed, and the next step was initiated.

Step 5. Simulation and architectural evaluation: During simulation, we analyzed whether it exhibited the FMSoS behavior, i.e., flood alerts whenever collected data had exceeded a predetermined threshold. 50 different architectural configurations of varied numbers of sensors, crowd-sourcing systems, and gateways and mediators dynamically appearing between the constituents were analyzed. We started with a configuration of four sensors, one gateway, and zero crowd-sourcing systems (besides the necessary mediators). Progressively, the number of sensors was increased, followed by the number of gateways and crowd-sourcing systems. The Simulation lasted 6 hours and 20 minutes.

Step 6. Analysis: Since the results of the simulation were stored in log files, the corresponding value of the aforementioned metrics could be calculated and compared for the selection of configurations with the best results.

Figure 13 shows a summary of the simulation outcome. We plotted (i) the percentage

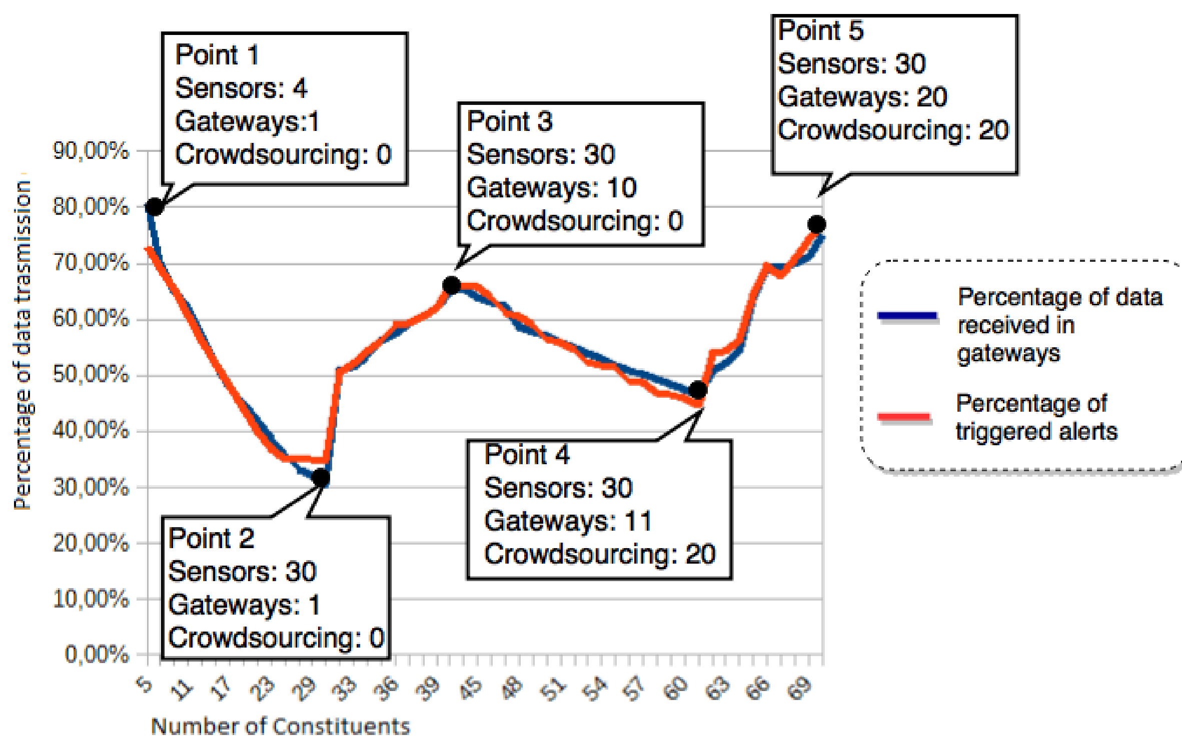

Figure 13 – Relation between percentage of data received in gateways and alerts triggered.

of 1000 data samples fed for each sensor correctly transmitted along the SoS architecture considering the variation in the number of constituents, and (ii) percentage of flood alerts triggered. The data loss increased in function of the number of sensors, and both reliability of data transmission and triggered alerts were reduced (Point 2). The number of crowd-sourcing systems was also increased for an architectural configuration of 40 constituents (Point 3), i.e., 30 sensors and 10 gateways (mediators were not considered). An increase in the number of crowd-sourcing systems increased neither the transmission rate, nor the number of alerts triggered because of the bottleneck of the gateways. The results improved again when the number of crowd-sourcing systems was fixed at 20 (Point 4), and the number of gateways was increased to 20 (Point 5), with 30 sensors, 20 gateways, and 20 crowd-sourcing systems (70 constituents, except the mediators). The rate of alerts correctly triggered was close to the rate of data effectively transmitted, therefore, when the data are correctly transmitted, the alerts follow the same trend. In this case, functional aspect is totally dependent on the operational aspect. Our evaluation was performed in a machine with an Intel core i5-3230M 2.60GHz (x64) Processor; Memory 4 GB; HD: 1TB; and Ubuntu 16.04 64 bits.

The data were analyzed according to the aforementioned metrics. Good results were achieved when FMSoS involved many constituents. However, results were not better than using only five constituents. Hence, unless a geographic area to be covered is huge, the use of

a small number of constituents can achieve the same results of the use of a large number[11]. Our approach successfully supported the automatic generation of simulation models for FMSoS specifications, and facilitated the evaluation of different architectural configurations.

Results. Gateways were the critical element for improvements in FMSoS operation. The results of a SoS architecture composed of four sensors were as good as those of 70 constituents, an support architects towards saving efforts and budget, e.g., during the acquisition of constituents for construction of a real SoS. Such a conclusion would not be drawn at the design stage without ASAS.

RQ1: Can the transformation successfully produce functional simulation models?

Answer. The simulation ran accordingly with no failures. Therefore, transformation was feasible and well-succeeded for this particular context. Further applications should be tested, however, M1 (Simulation failures, given by the number of failures detected during model simulation) currently equals 0.00%.

RQ2: Does ASAS enable architects to evaluate a SoS software architecture considering its inherent dynamics?

A flood alert (mission assessed in this case study) was accurately triggered. Therefore, our results imply a high level of confidence and feasibility. This case was carried out by a SoS architect, who interacted with the SoS simulation, triggering changes for observing the SoS behavior after the architectural change. Despite changes, SoS operation was preserved. Data acquired enable the architect to draw conclusions on the efficiency of different architectural configurations. Therefore, M2.1 (the percentage of well-succeeded SoS behaviors in a diversity of architectural configurations, achieving pre-defined missions) was 100,00% for this case.

Regarding data loss (M2.2), Figure 14 shows the results collected along the simulation execution. Different architectural configurations showed different data transmission efficiency. Figure 15 displays the statistical analysis of the results. The maximum percentage of data received was 77.48%, whilst the average was 55.95%[12]

50 (M2.3: scale) different architectural configurations with a diversity of constituents were exhibited during architectural evaluation. Indeed, metrics were collected for each different set of constituents. ASAS provided good results and enabled data collection and

[11] At least for this domain, configurations defined, and types of constituents

[12] In this simulation, no data delivery guarantee was provided. A supplementary simulation was conducted, performing the guarantee of reception. 100% of the data were received, and all the flood alerts were accordingly triggered.

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

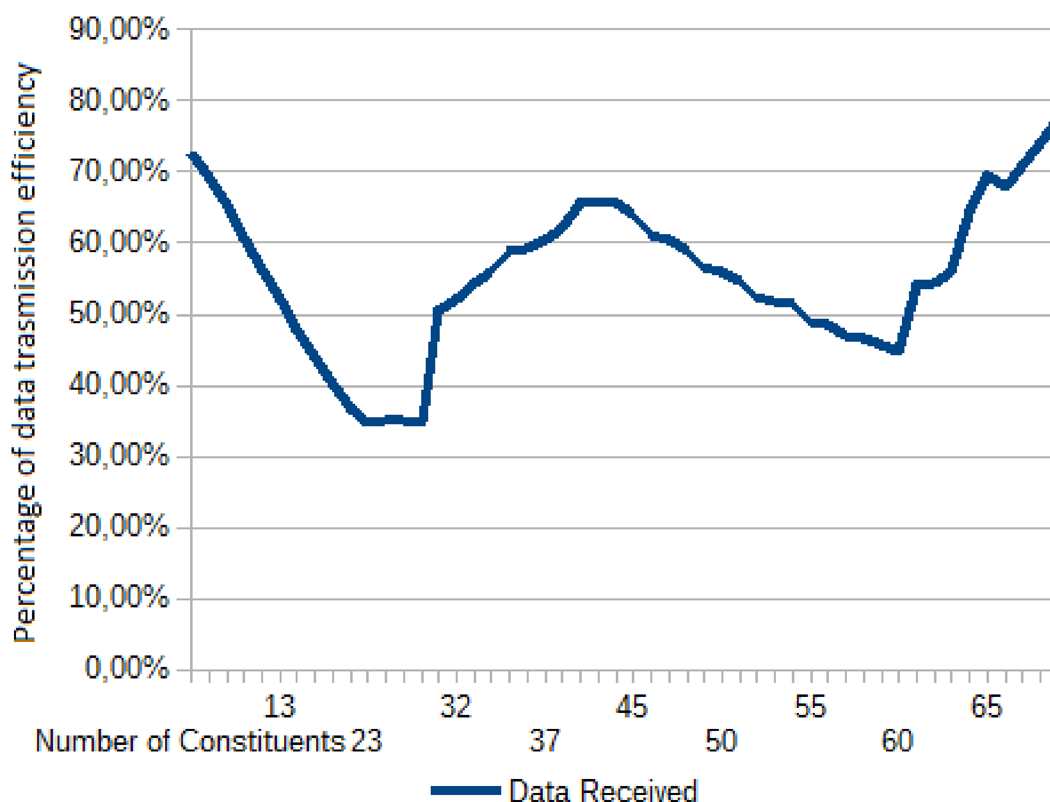

Figure 14 – Relation between data received in one or more gateways and SoS scale.

analysis and supported the decisions on architectural configurations that exhibited the best results. Hence, ASAS supported decisions on different SoS architectural configurations for 100% of the cases (M2.4: support for decision on best architectural configurations). RQ3 and RQ4 were answered as a result for the triangulation process of case studies investigation, being answered later in this section.

Threats to Validity. The threats to validity for Case 1 include the scale of our evaluation, verification of correctness of the transformation rules, and SoS topology. Our solution can scale, as scaling SoS consists in the specification of further bindings in the coalitions in SoSADL and replications of atomic models in DEVS. Regarding transformation correctness, correspondences were established between entities in both models. The results relieved the threat, once ASAS produced functional simulations in more than one situation. Further investigations of topologies and different numbers of constituents will be conducted. Due to the limited time window and suitability of the period for our purposes, the selection of data might show bias. However, as the limited period has a plurality of inputs, including sunny (dry) and rainy days, this bias is aliviated.

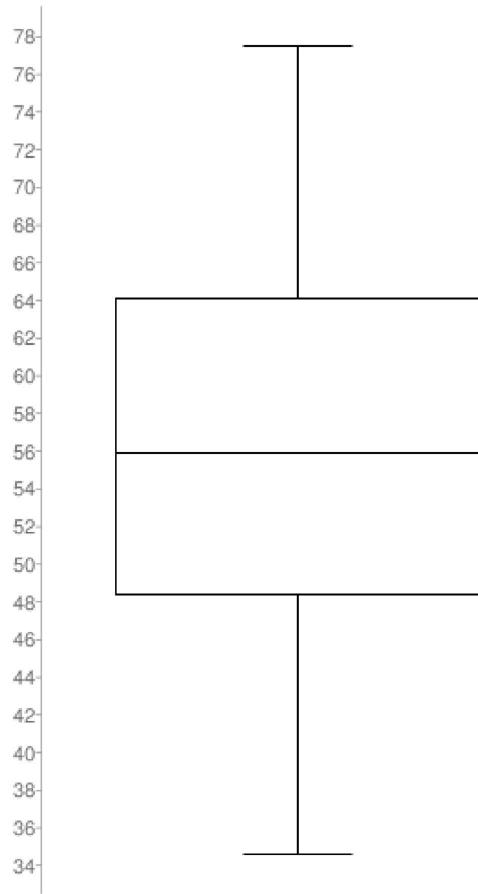

Figure 15 – Box plot for data received for 50 different architectural configurations.

3.2.2. Scenario 2: Space SoS with One Satellite

A Space SoS is a SoS composed of constituents in ground and space that accomplish missions such as telecommunication, global position (GPS), weather forecast, Earth and space observation, meteorology, resource monitoring, and military observation. Space SoS can contain approximately 800 constituents (YAMAGUTI; ORLANDO; PEREIRA, 2009). Space systems are usually divided into three main segments, namely Spatial, which is the part placed in orbit (satellites, space probes, space stations); Launcher, used for placing the space instruments and constituents in orbit (rockets, space shuttles); and Ground, which supervises satellite operations. Ground consists of a mission control system, an operation control system, ground stations and data communication networks (WERTZ; LARSON, 1999; ECSS, 2008). Each segment materializes one or more systems that have their own attributions. In systems engineering, missions are defined and constituents are articulated for a certain space mission.

Satellites are the main constituents of a space SoS. Each satellite is divided into several subsystems, namely onboard computer, power system, propulsion system, attitude

control, and communication system. Satellites are considered in two parts: payload and structural. The payload part, composed of sensors and infrared cameras, assures a system accomplishes the mission. The structural part maintains the satellite in operation. It includes a solar panel, batteries, and reaction control system. The satellite establishes contact only when it passes over the geographic location of the ground station.

The launching of a satellite into space is costly to space systems. The lauching of a CubeSat, an open source architecture of 10cm X 10cm X 10cm, for example, is estimated as $80,000 dollars. Due to such high costs and relevant potential losses, the system is considered *critic domain*.

Examples of missions for a Space SoS include (i) monitoring of the Amazon forest, taking pictures and observing deforestation; (ii) telecommunication to support Internet Worldwide and TV; (iii) scientific missions, such as study of solar behavior, and exploration of other planets; (iv) river monitoring, for example, in the case of Rio Doce ecological disasters; and (v) detection of tsunamis and hurricanes.

Step 1. Design of a SoS architecture in SosADL: The following concepts are specially important in the space domain.

- **Telemetry**: a technical name given to the information received from the status of the satellite during its passage on the ground stations;
- **Telecommand**: an operation remotely sent to satellites requiring them to perform actions, such as, capturing images or opening of the solar panel.

Space SoS is composed of the following different types of constituents:

1. **Command and Control Center (C2)**: located in São José dos Campos, it generates a telecommand and telemetry packet;
2. **Satellite**: a synchronous polar orbit satellite that generates images of the planet every 5 days.
3. **Ground Station:** located in Cuiabá, it involves reception and satellite data transfer (telemetry and telecommand), and temporarily stores image data and satellites tracking;
4. **Remote Sensing Data Center:** receives records, processes, storage, and distributes images and data from remote sensing;

Figure 16 – Illustration of a Brazilian SoS for data collection via satellites (INPE, 2017).

5. **Data Collection Platform (DCP)**: a device whose electronic sensors measure environmental variables such as precipitation, atmospheric pressure, solar radiation, temperature, air humidity, dew point, wind direction and speed, and detect variations in water bodies levels [13]. In the Brazilian Space Mission, such DCP data are automatically transferred to artificial satellites in the Earth orbit and retransmitted to ground stations to be distributed to end users, thus enabling the monitoring of large territorial extensions and remote areas.

Every mission in a Space SoS is performed according to a meta-process, called *Meta-process for Payload missions in Space SoS*, as shown in Figure 17 and follows the steps described below:

1. Remote Sensing Data Center requests payload data for Command and Control Center (C2);

[13] <http://www.simge.mg.gov.br/simge/sobre-o-simge>

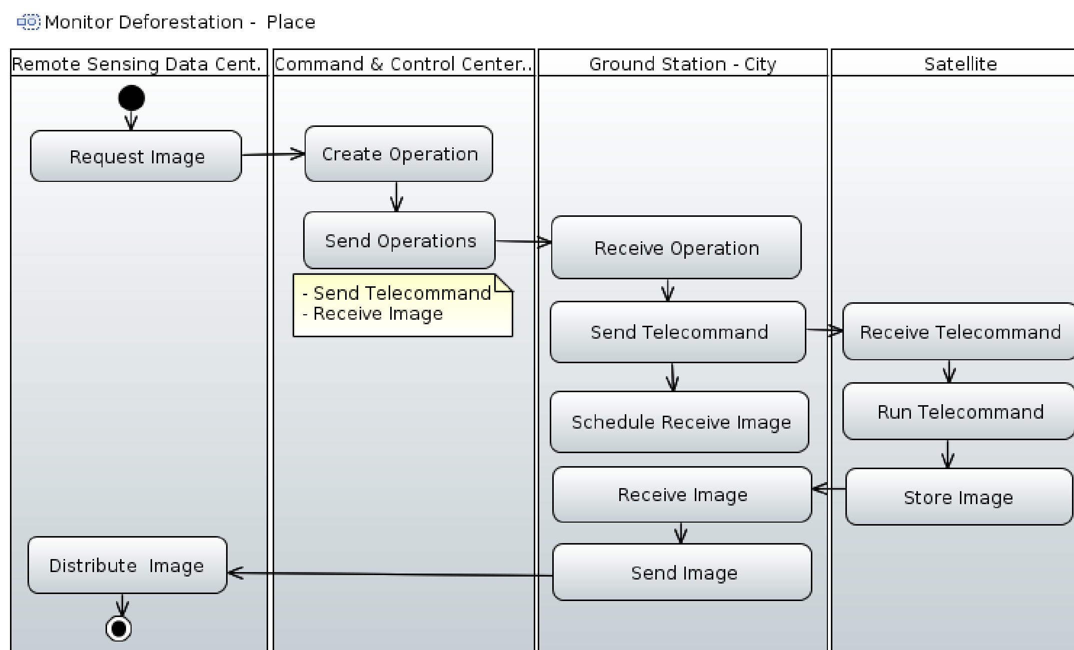

Figure 17 – Activity diagram of the business process followed by a space SoS for the monitoring of the Amazon.

2. C2 Center creates the operations(telecommand and telemetry) and schedules their execution;
3. Ground Station configures antennas and rotors;
4. Ground Station establishes links with Satellite;
5. Ground station sends remote control;
6. Satellite executes commands;
7. Satellite stores payload data;
8. Ground Station requests payload data;
9. Satellite forwards telemetry data;
10. Ground Station stores raw data;
11. Remote Sensing Data Center searches for telemetry data;
12. Remote Sensing Data Center tags and stores data;
13. Remote Sensing Data Center distributes payload data to the Mission Center.

As this is a meta-process, it is conducted by meta-activities and meta-constituents. The instantiation of a concrete process consists in the identification of the constituents and activities that replace those elements in the process.

Missions: The general mission of such Space SoS is *Data Collection*. The several types of data collection include water and deforestation index, which can be accessed online. This SoS was designed to undertake two missions, namely (i) Taking pictures (monitor) of the Amazon region, and (ii) Distribution of environmental data collected by Data Collection Platforms (DCP).

We conducted requirements elicitation meetings with an expert from the Brazilian National Institute of Space Research for modeling a software architecture of a Space SoS. The expert helped us understand the SoS structure, its main constituents, and the way they interoperate to achieve the main results expected. A small-scale SoS was modelled with only one mission to be accomplished. After we conducted this pilot study, we planned on the generation and execution of two other simulations. For both, the Space SoS was composed of one data center, one C2 center, and one ground station.

We designed an architecture in SosADL with an initial set of 126 data collection platforms (DPC), and one satellite, representing the current architecture of some Brazilian states. Figure 18 highlights the states involved. DCPs are spread in the territory of each Brazilian state to monitor environmental data. Apart from DCP stations without coordinates and/or available data, the State of São Paulo (SP) has 67 DCP stations. Minas Gerais (MG) has 48 and Rio de Janeiro (RJ) has 11, which totalizes 126 DPC stations. The states of Paraná (PR - 12 stations), Goiás (GO - 48 stations), Mato Grosso (MT - 40), and Amazonas (AM) - due to its importance (81 DPC stations) were also considered, as shown in Figure 18.

The architecture designed was comprised of one data center, a C2 center, a ground station, a satellite, and 126 DPC (SP + MG + RJ) (130 constituents, apart from mediators). DCP are located in land or water (rivers or ocean). The simulation was executed, and DCP from other states (PR, GO, MT, AM) were dynamically included until 307 platforms (a total of 311 constituents) had been reached, as illustrated in Table 9. This procedure was performed in five steps, one at a time, and originated five different architectural configurations, as shown in Table 9.

Data Preparation. The official website[14] of the Brazilian Institute of Space Research (INPE) offers a query interface that enables access to data from all DPC platforms in operation in the Brazilian territory. To use realistic data, we manually performed the

[14] <http://sinda.crn2.inpe.br/PCD/SITE/novo/site/index.php>

Table 9 – Number of DCP constituents and data for each space SoS architectural configuration.

#	States	Land DCPs	Water DCPs	Total DCPs	Data Sent by Land DCPs	Data Sent by Water DCPs	Total Data Sent
1	SP MG RJ	92	34	126	54691	35357	90048
2	SP MG RJ PR	95	43	138	56350	41256	97607
3	SP MG RJ PR GO	127	59	186	74404	52756	127160
4	SP MG RJ PR GO MT	133	93	226	76747	95229	171976
5	SP MG RJ PR GO MT AM	146	161	307	84457	158999	243457

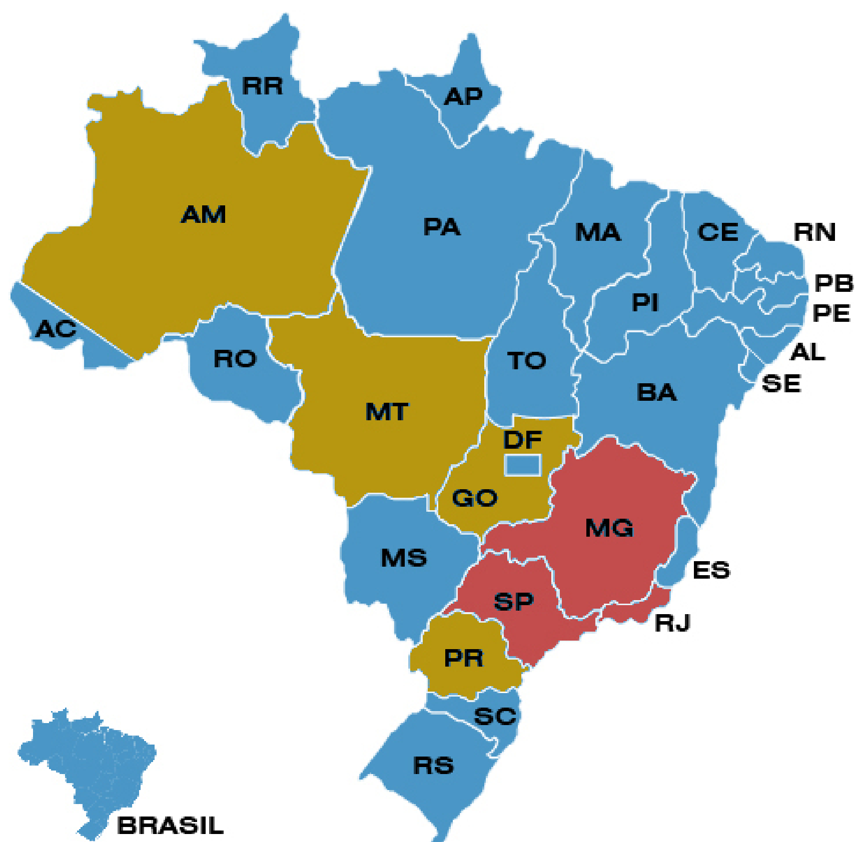

Figure 18 – Brazilian territory map adapted from (DNIT, 2017).

queries, and collected one-year data within the data availability period for each DCP platform. Such data refer to the 75 DPC stations in São Paulo state, 51 stations in Minas Gerais, and 12 in Rio de Janeiro, which totaled 138 PCD stations (constituents) that compose the initial architecture of the Space SoS.

The data used were collected by DCPs for one year. During the simulation, DCP stations of four states were progressively added. Data were separated into 73-day slots (365 days divided into five blocks). As the simulation progressed, DCP from the aforementioned states were added and the simulation was fed with corresponding data, considering the sequence shown in Table 9. The initial configuration involved DCPs from three states, and progressively increased until eight states had been reached. Data were delivered according to the remaining simulation time. In the case of Paraná, for example, 292 days (1 year - 73 days) were analyzed, as it was added only after 73 days of simulation were fed. The DCP from Goiás received 219 days of data, and so on. This was necessary to guarantee a homogeneous delivery of data between different states. Otherwise, data from São Paulo state would be over while Amazon would still be receiving remaining data.

Real data was also used to feed orbits trajectories. We gathered data from a real orbit of a Brazilian satellite that covers the territory[15].

Step 2. Evaluation planning: This study aims at assessing the behavior of the SoS architecture, checking the results as we increased the number of DCP stations until 300, exercising dynamic architecture and its impact on the functionalities provided by a SoS.

Step 3. Execution of the model transformation: Model transformation was executed with SosADL files used as input, producing DEVS simulation models as outcome.

Step 4. Deployment: Simulation models were accordingly deployed in MS4ME.

Step 5. Simulation Execution and Architectural evaluation: Due to the huge amount of data, a more powerful machine was required to conduct the study. Both simulations run in an Intel(R) Xeon(R) CPU E5-2620 v3, with 30 GB RAM, 2 TB HD, in a server running on Ubuntu 16.04.3 LTS.

Step 6. Results and Analysis. The simulation lasted 1,676 minutes (approximately 27 hours), spent 2-3 core processors, and used about 90% of each, whilst IO used the maximum processing power during the whole simulation. MS4ME console was redirected

[15] We used an app written in Python for accessing the Satellite data server [16]. A script was created to access and took this data, treat them, and write them in a file once per second, creating a realistic orbit of a real satellite (China–Brazil Earth Resources Satellite 4 (CBERS-4)).

Table 10 – Percentage of data transmitted to the satellite and simulation time.

#	Data Received from Land DCP	Data Received from Water DCP	Total Data Received	Percentage of Land Data Received	Percentage of Water Data Received	Percentage of Data Received (total)	Simulation Time (min)
1	51,942	33,143	85,085	94.97%	93.74%	94.49%	253
2	53,232	38,987	92,219	94.47%	94.50%	94.48%	278
3	69,612	49,281	118,893	93.56%	93.41%	93.50%	315
4	70,174	89,192	159,366	91.44%	93.66%	92.67%	391
5	78,380	140,291	218,671	92.80%	88.23%	89.82%	439

Table 11 – Data loss for the space SoS simulation.

#	Data Loss (Land DCP)	Data Loss (Water DCP)	Data Loss (Total)	Percentage of Data Loss (Land DCP)	Percentage of Data Loss (Water DCP)	Percentage of Data Loss (Total)
1	2,749	2,214	4,963	5.03%	6.26%	5.51%
2	3,118	2,269	5,388	5.53%	5.50%	5.52%
3	4,792	3,475	8,267	6.44%	6.59%	6.50%
4	6,573	6,037	12,610	8.56%	6.34%	7.33%
5	6,077	18,708	24,786	7.20%	11.77%	10.18%

Table 12 – Telecommands in space SoS simulation.

#	Telecommands sent	Telecommands Received	Pictures Captured	Pictures Not Captured	Pictures returned to Ground Station	Pictures not returned to Ground Station
1	4,000	4,000	3,996	4	3,095	1
2	4,000	4,000	4,000	0	3,994	6
3	4,000	4,000	3,998	2	3,995	3
4	4,000	4,000	3,999	1	3,994	5
5	4,000	4,000	4,000	0	3,998	2

to `/dev/null`, otherwise the simulation would be slow. The satellite passed over each DCP station every 201 minutes (every 3,35 hours).

Tables 10, 11, and 12 show the results of percentage of data transmitted to the satellite, percentage of data received, simulation time; data loss; and data on telecommands, respectively. No telecommand was lost.

RQ1: Can the transformation successfully produce functional simulation mod-

els?
Yes. The model transformation produced functional simulation models, with 0.00% simulation failures (M1).

RQ2: Does ASAS approach support evaluation of SoS software architectures considering its inherent dynamics?
ASAS enabled comparisons and evaluations of the entire SoS from a holistic point of view and considering different architectural configurations. Accuracy (M2.1), i.e., the percentage of effectiveness delivered by SoS operation in the different architectural configurations varied from 89.82% to 94.49%. Data losses (M2.2) increased due to the increase in the number of DCP and competition for resource (satellite), from 5.51% to 10.18%. Scale (M2.3) was also examined with variations in the number of constituents up to 311. Results achieved by different architectural configurations were also analyzed through simulations, and confirmed ASAS could support SoS architectural evaluations through simulations (M2.4).

3.2.3. Scenario 3: Space SoS with Satellites Constellation

In this scenario, we aimed at investigating the results of the use of a satellite constellation, i.e., a set of satellites used in association for the improvement in the services provided (as telecommunications). Such constellation is to be soon launched to space[17] by the Brazilian Space Agency.

We were concerned to simulate the Space SoS with all DCP stations in Brazil, increasing the number of satellites from one to six (as it will be done soon by the Brazilian Space Agency) to check (i) the level of resource competition, the percentage with which the missions were met in relation to the expected, and (ii) how the increase in the number of satellites improves the performance of a SoS as a whole, reducing the waiting time to receive a requested data from the satellite.

Step 1. Architecture design: We modelled an architecture with a C2 center, a data center, a ground station, six satellites, and 249 DCP stations (a total of 262 constituents). The orbits were defined according to a study on the constellation of satellites (CARVALHO *et al.*, 2013). Maximum and average contact times, maximum and average revisit times,

[17] On September 18, 19 and 20, University of Brasilia held the 1st BRICS Remote Sensing Satellite Constellation Forum, a meeting to bring together representatives of the BRICS space agencies - Brazil, Russia, India, China and South Africa - to discuss technical aspects related to the five countries' initiative in establishing a constellation of six (6) remote sensing satellites. Source: <https://goo.gl/mhPGtq>.

percentage of satisfactory revisits and average number of contacts per day were observed during the preparation for simulation. The configuration chosen was three orbital planes with two satellites in each plane were adopted, as recommended by Carvalho et al. (CARVALHO *et al.*, 2013), and shown in Figure 19. In their case, the revisit was less than one hour in 100% of cases.

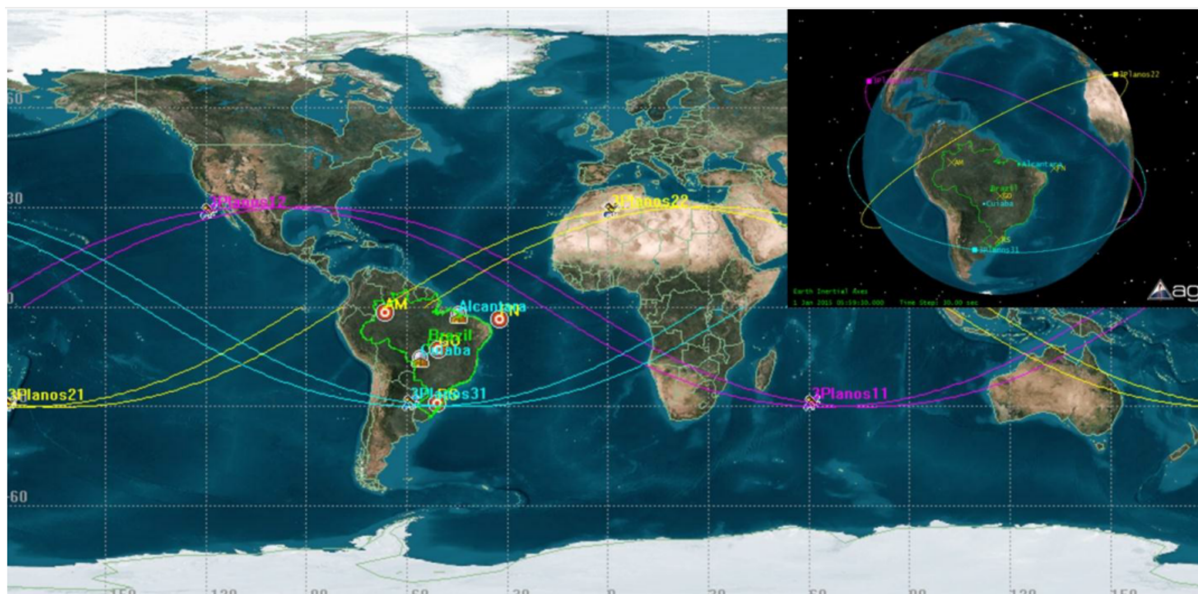

Figure 19 – Three orbital trajectories containing two satellites each (CARVALHO *et al.*, 2013).

Data Preparation. Data that fed the second simulation were prepared according to the same rationale defined for the first simulation. We obtained a data set from INPE corresponding to data collected between January 1st, 2017 and October 31st, 2017 from each DCP station in operation the entire Brazilian territory. The first SoS architectural configuration for the simulation was composed of all the 249 DCPs in operation and one satellite. A satellite was then added and another 1/6 of the dataset was fed. The same procedure was performed until six satellites had been added. Six satellites were then added for enabling the study of the Space SoS performance. Listing 21 in Appendix C shows an excerpt of the code of a satellite modelled in SosADL.

The satellites used for the second simulation were: a) Sino-Brazilian Satellite of Earth Resources (CBERS-4) (Period: 100.3 minutes); b) Data Collection Satellite 1 (SCD 1) (Period: 99.7 minutes), and c) Data Collection Satellite 2 (SCD 2) (Period: 99.7 minutes). A script was also developed for accessing an external link that provide these coordinates, which enabled a revisit period of 30 minutes[18].

[18] Each orbit contains two satellites with different movements that enable them to revisit the same DCP station each 30 minutes.

Table 13 – Space SoS architectural configurations for Constellation of Satellites.

# Architectural Configuration	Number of Satellites	Simulation time
1	1	283
2	2	287
3	3	286
4	4	285
5	5	289
6	6	292
	TOTAL	2,014 minutes (33.56 hours)

Step 2. Evaluation planning: This study aims at assessing the behavior of the SoS architecture, checking whether the results exhibited by the SoS can be improved with new satellites being deployed. This endeavor is important, as the Brazilian government intends to launch such satellite constellation in the next years. Dynamic architecture is also intended to be evaluated, increasing the number of DCP stations until 300.

Step 3. Execution of the model transformation: Model transformation was executed with SosADL files used as input, producing DEVS simulation models as outcome.

Step 4. Deployment: Simulation models were accordingly deployed in MS4ME.

Step 5. Simulation Execution and Architectural evaluation: Due to the huge amount of data, a more powerful machine was required to conduct the study. Both simulations run in a Intel(R) Xeon(R) CPU E5-2620 v3, with 30 GB RAM, 2 TB HD, in a server running on Ubuntu 16.04.3 LTS.

Step 6. Results and Analysis. ASAS supported the architectural evaluation for Space SoS according to pre-established metrics, as shown in Tables 14, 15, and 16. An increase in the number of satellites reduced the competition between DCP stations, which enabled the SoS to better perform its missions, thus reducing conflicts and concurrency between the DCP for the satellites.

The total simulation time was approximately 34 hours, as shown in Table 13. The simulation involved four types of DCP stations, in conformance with data delivered by INPE: 133 meteorological DCP stations, 36 agricultural DCP stations, 77 hydrological stations, and three PCDQagua, which totaled 249 DCP stations, distributed in 21 states as follows: 'PR': 1, 'SC': 2, 'MG': 23, 'MT': 2, 'MA': 16, 'BA': 19, 'RS': 7, 'SP': 20, 'AM': 4, 'CE': 46, 'PB': 1, 'TO': 14, 'RJ': 3, 'PA': 10, 'ES': 3, 'MS': 4, 'PE': 15, 'GO': 10, 'RN': 27, 'RO': 7, 'SE': 15.

As the data feeding was homogeneous (1/6 of the data set for each architectural configuration), data were delivered as follows: 39,895 samples of meteorological data, 61,519 samples of hydrological data, 12,780 samples of agricultural data, and 970 samples of Qagua data, which totaled 115,164 samples processed by Space SoS. Table 14 shows the percentage of such data effectively received in ground stations during the simulation, as well as the respective percentages of the amounts used to feed the simulation through stimuli generators. For all cases, an increase in the number of satellites reduced the competition between missions and DCP stations, and increased the number of data effectively transferred.

Conflicts occurred between missions, i.e., in some cases, the DCP data were not received in the satellite because it was performing another action related to an Amazon monitoring mission, as taking a photograph. In other cases, DCPs were very close geographically. Consequently, only one of them could transmit the data to the satellite. In other cases, the conflict occurred because the satellite was passing a region to take both a photograph and collect some DCP data (the same situation occurred for the telemetry). As DCP-satellite transmission is more elaborate, it requires more time to be performed. Hence, in situations that a satellite passes far from a region to be monitored or passes very fast, a lack of time to capture the photograph can occur. Analogically, the same situation can occur for telemetry. In addition, the satellite takes a photograph of a pre-established region only when it is close enough.

Table 15 shows complementary data of Table 14, i.e., the percentage of data loss. An increase in the number of satellites reduced data losses, as expected.

Table 16 shows the amount of telecommand sent to each architectural configuration, number of photographs captured and not captured, and number of photographs returned and not returned to the ground station.

Table 17 displays the percentage of accomplishment of each mission, according to mission requests shown in Table 16. ASAS accordingly supported a robust analysis of the success of each architectural configuration for the accomplishment of missions, and revealed an increase in the number of satellites was beneficial for the Space SoS, as the percentage of mission accomplishments was increased. Moreover, an increase in the number of satellites improved the trustworthiness of such SoS by increasing the precision with which Space SoS missions were achieved.

RQ1: Can the transformation successfully produce functional simulation models?

Yes. The model transformation was well-succeeded to produce simulation codes for a Space SoS with a constellation of satellites, producing functional simulation models, with 0.00%

Table 14 – Percentage of data transmitted by each architectural configuration and received in ground.

# Architectural Configuration	% Total Data Received
1	89.81%
2	91.24%
3	91.70%
4	94.85%
5	95.85%
6	97.94%

Table 15 – Percentage of data loss in satellite constellation simulation.

# Architectural Configuration	% Total Data Loss
1	10.19%
2	8.76%
3	8.30%
4	5.15%
5	4.15%
6	2.06%

Table 16 – Results of telecommands and photographs requests, taken, and returned to ground.

#	Telecomand sent	Telecommand Received	Telecommand Lost	Captured photographs	Non-Captured photographs	Photographs returned to Ground Station	Photographs not returned to Ground
1	4,000	4,000	0	3,948	52	3,948	0
2	4,000	4,000	0	3,967	33	3,965	2
3	4,000	4,000	0	3,981	19	3,980	1
4	4,000	4,000	0	3,994	6	3,994	0
5	4,000	4,000	0	4,000	0	4,000	0
6	4,000	4,000	0	4,000	0	4,000	0

Table 17 – Percentage of missions accomplished in Scenario 3.

#	% Percentage of accomplishment *Data collection.*	% Percentage of accomplishment *Obtain picture from Amazon.*
1	89.81%	98.70%
2	91.24%	99.13%
3	91.70%	99.50%
4	94.85%	99.85%
5	95.85%	100.00%
6	97.94%	100.00%

simulation failures (M1).

RQ2: Does ASAS approach support evaluation of SoS software architectures considering its inherent dynamics?
ASAS enabled comparisons and evaluations of the entire SoS from a holistic point of view and considering different architectural configurations. The best accuracies (M2.1) achieved by Space SoS were 97.94% for data collection, and 100.00% for the obtaining of pictures from the Amazon region. Data loss (M2.2) reached 2.06% of the total loss when six satellites were used. Therefore, an increase in the number of satellites significantly reduces data collision, resources competition, consequently reducing data losses. Arrangements of constituents were tried, including satellites at runtime, exercising M2.3 (scale). ASAS enabled us to evaluate six different architectural configurations, drawing conclusions to decide which architectural configuration offers best results (M2.4).

3.2.4. Synthesis

We applied *triangulation* technique (RUNESON; HöST, 2009), which involves taking different angles towards the studied object and providing a broader picture. In this study we used *Data (source) triangulation*, which consists in the use of more than one data source or collection of the same data in different occasions. Three different simulations were conducted in the context of two different SoS (one for Flood Monitoring SoS and two simulations for Space SoS). For simplicity, we will name them S1, S2, and S3. S1 is the FMSoS simulation, S2 is the first simulation for Space SoS, and S3 is the second Space SoS simulation. A synthesis of results was performed through a comparison between the data acquired from the different simulations. Triangulation enabled us on drawing broad and reliable conclusions about the effectiveness of ASAS approach to support simulation and evaluation of SoS software architectures. Figure 20 shows, respectively, data on simulation duration, maximum number of constituents, number of architectural reconfigurations, and diversity of constituents (number of different types) for each simulation. In simulation S1, we exercised dynamic architecture due to the major number of architectural changes, whilst in S2 and S3 we focused on the number and types of constituents, and the way they might affect the success of the SoS in accomplishing missions.

Simulations were performed so that metrics associated with the architecture, as accuracy and data loss could be collected. Our claim is ASAS supports architects in the capture of data such as accuracy and data loss for assessing a SoS software architecture. Loss of data is complementary to the accuracy. Generally, an increase in elements that receive data such as gateways or satellites, reduces the data loss. Therefore, no correlation

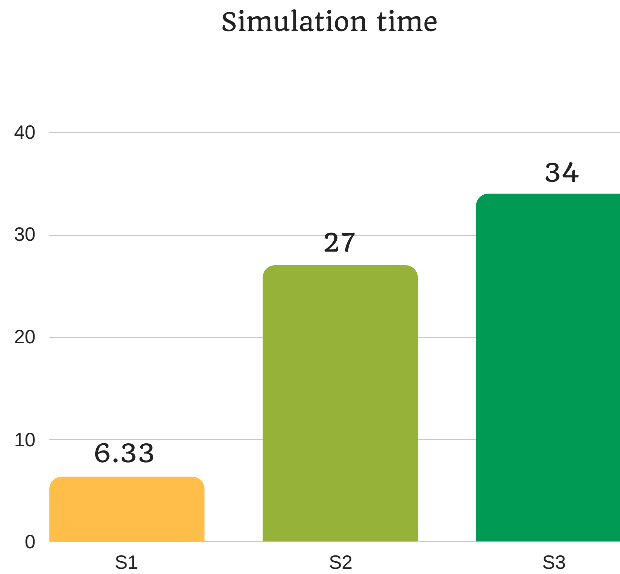

Figure 20 – Comparison between data obtained in case study: simulation time.

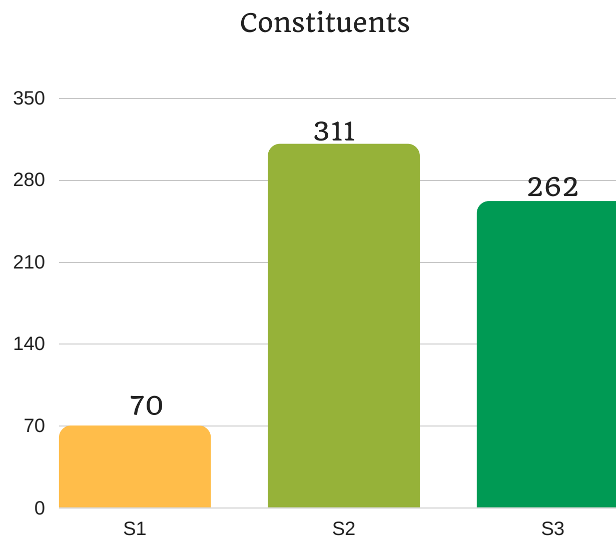

Figure 21 – Maximum number of constituents in each simulation.

between the metrics collected was investigated.

RQ1: Can the transformation successfully produce functional simulation models?

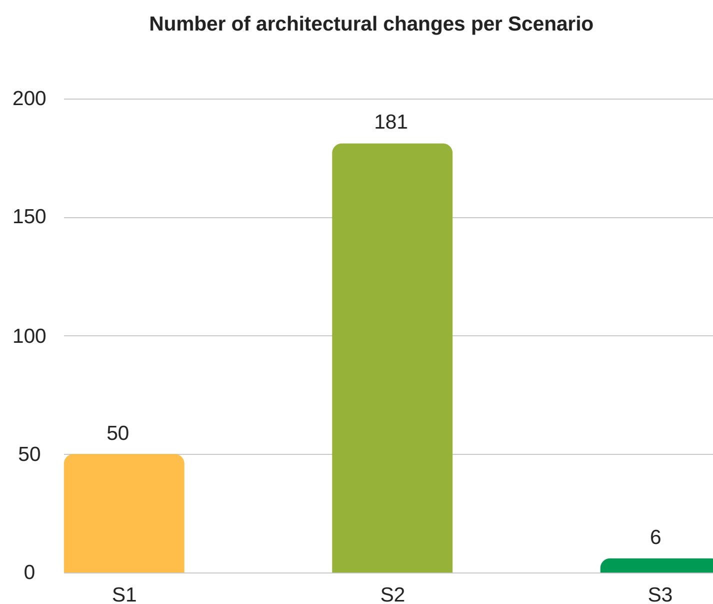

Figure 22 – Number of architectural changes at runtime.

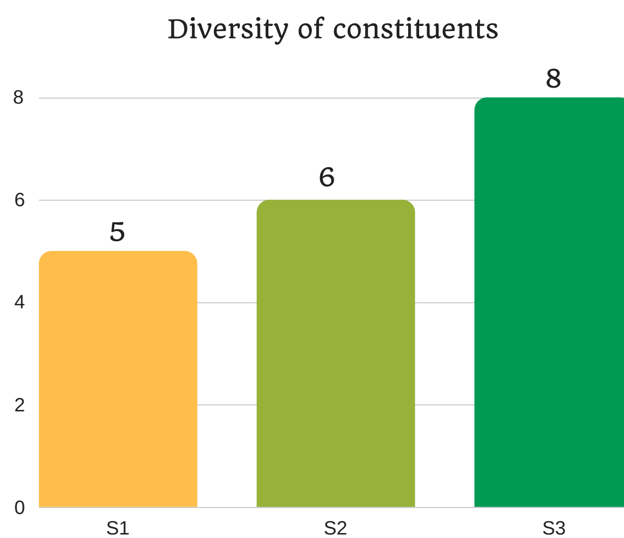

Figure 23 – Diversity of constituents.

Answer. No simulation failure was detected. Hence, this measure was of 0.00% for our case study.

RQ2: Does ASAS approach support evaluation of SoS software architectures considering its inherent dynamics?

Yes. ASAS enables observation of even unexpected results, as an equally good result with a small number of constituents and large in the case of flood monitoring; and to confirm previous predictions, such as the fact that increasing the number of satellites improves Space SoS results. Future investigations might also lead to conclusions on the impact of more satellites on other services provided by the Space SoS, such as telecommunication. We related accuracy (M2.1), data loss (M2.2) and scale (M2.3), as shown in Figure 24, and plotted the largest number achieved by each metric in the three simulations regarding each simulation (S1, S2, and S3) and their respective amounts of constituents (70, 311, and 262). We also considered the difference between mission one and two for simulation S3.

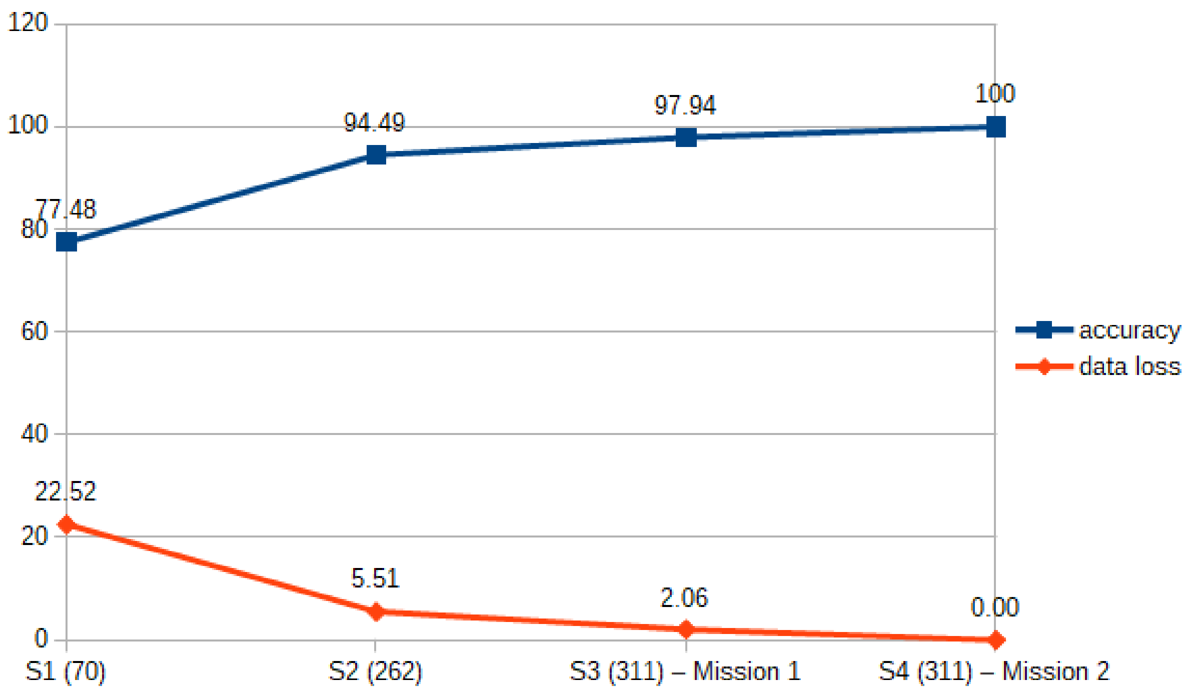

Figure 24 – Relation between accuracy and data loss.

Regarding M5 (support for decisions on architectural configurations, i.e., whether the simulation enables collection of data and analysis to decide which architectural configuration offers best results), ASAS supported decisions of the best architectural configurations according to the aforementioned metrics in all cases.

RQ3: Is ASAS approach domain-independent?

Possibly yes, according to M3 (given by the amount of different domains to which ASAS was applied). ASAS was applied to two different domains, namely flood monitoring in a smart city, and space SoS. Initial evidence indicated ASAS is domain-independent, but

further evidences are required.

RQ4: Can patterns be recurrently applied?

3487 lines of code (LOC) were produced by the patterns in Flood Monitoring SoS, including the behaviors of constituent systems and mediators. No conflicting instructions occurred and the systems ran accordingly as predicted at design-time. As the average number of lines for each pattern is 2.5 (2 for one pattern and 3 for the another pattern), patterns were applied almost 1,000 times (M4 for FMSoS = 934). Table 3.3 summarizes the data.

Table 18 – Number of lines of code produced by our patterns for FMSoS.

Model	Number of Models	Lines per model	Lines per model type
Sensor	43	24	1,032
Gateway	20	18	360
Crowd	9	22	198
CrowdGateway	3	17	51
Transmitter	43	13	559
CrowdTransmitter	9	15	135
TOTAL	127	109	2,335

In simulation 2, the patterns were also successfully applied during the model transformation for the generation of systems behaviors. Table 19 shows the number of LOC generated for each simulation. No conflicting instructions occurred and the systems ran accordingly as predicted at design-time. Under the same rationale, patterns were successfully applied almost 5,000 times (M4 for Space SoS = 4,896) for automatic generation of simulation models for software architectures of a Space SoS.

3.3. Discussion

ASAS approach exhibits robustness, as case studies were conducted in different scenarios, with a large and increasing number and diversity of constituents and architectural arrangements. It also supports collection of data and establishment of conclusions and comparisons between SoS architectural configurations. More than 15 KLOC of functional simulation code were produced (as shown in Tables and 19), and this number does not even consider code generated to describe the SoS structure itself. The following observations raised can possibly be generalized for other domains and SoS applications:

Co-existence of missions in a SoS

Table 19 – Number of lines automatically generated for simulations 1 and 2 of space SoS - NoM means Number of Models.

	Simulation 1			Simulation 2		
Model	NoM	#Lines per Model	#Lines per model type	Number	#Lines per model	#Lines per model type
CommandAndControl	1	15	15	1	15	15
DataCenter	1	14	14	1	14	14
GroundStation	1	22	22	1	22	22
Satellite	1	61	61	1	71	71
PCDAquatico	34	21	714	3	21	63
PCDTerrestre	92	21	1,932	36	21	756
PCDHidro	0	0	0	77	21	1617
PCDMet	0	0	0	133	21	2793
MediatorC2ToGround	1	9	9	1	9	9
MediatorDataCenterToC2	1	9	9	1	9	9
MediatorGroundToDataCenter	1	9	9	1	9	9
MediatorGroundToSatellite	1	26	26	1	26	26
MediatorPCDToSatellite	126	26	3,276	249	36	8,964
					TOTAL	12,240

The co-existence of missions must be accordingly planned. A conjecture that we extract is *Mediators can hold the logics to trigger a mission accomplishment.*, as illustrated in Figure 25[19] for Flood Monitoring and Space SoS. For the former case, a mediator emerges and enables data transmission in the SoS software architecture when the distance between two sensors is shorter than or equal to 50 meters. For the latter, the approximation of the satellite to the DCP station is examined by the mediators, and when it is shorter than or equal to a specific value, the data transmission is enabled. Hence, in both cases, mediators are pivotal elements for the control of data transmission and triggering of a mission. `choose` and `switch` structures in SosADL syntax enable the design of those solutions without causing non-determinism in the resulting state machine-driven constituents simulation, as shown in Listing 21, lines 73-112.

Another conclusion is *if a resource (constituent) in a SoS software architecture is part of more than one mission accomplishment, both the scheduling and the context switching must be implemented in the shared resource.* Behavior in satellites followed this rule, as they were specified using a `switch` structure the enable the alternation between missions being achieved.

[19] Credits for the images used to compose the Figure: <https://goo.gl/npTLdm>, <https://goo.gl/DCU3L7>

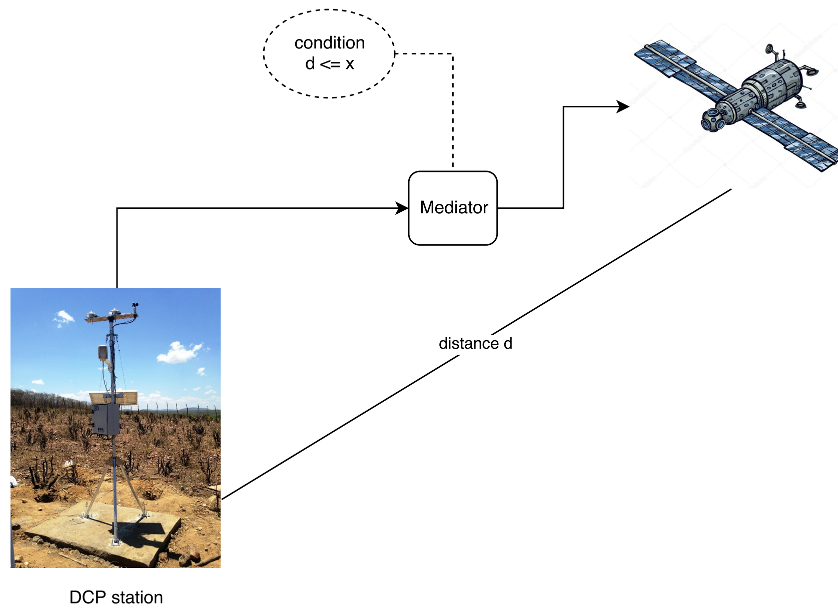

Figure 25 – An illustration of communication between DCP and a satellite.

Types of Missions

We distinguished two types of missions: a data forwarding mission, and business process-oriented mission. The former type is hold by DCP stations in Space SoS, and sensors in Flood Monitoring SoS. In neither cases, constituents were aware of the destination of the data collected and forwarded. On the other hand, we perceive that the data requisition made by the Data Center is business process oriented, i.e., the activities being accomplished are interdependent and there is (i) an order in which they are executed, (ii) a systematic separation of responsibilities, (iii) many roles being played, and (iv) many institutions involved. Moreover, the mission show a request-response nature. As such, we conjecture that those *two types of missions may co-exist* in the Space SoS specified; and *SoS domain comprehends, at least, two types of missions, namely data forwarding and business-process oriented.*. We believe another SoS probably exhibits one, both or more types of missions may co-exist in their architecture, sharing and competing for resources. Strategies must be established for resources scheduling.

Threats to Validity. The following threats were identified for cases 1 and 2: (i) scale: although it is not a problem, since our simulation can handle a larger number of constituents, it hampers data visualization and processing, as a large number of constituents is hardly visualized in a simulation and a more powerful processor would be required. Such problems are reduced as the resulting data are saved in spreadsheets to be properly analyzed; (ii) the order in which changes were performed in the architecture: The order in which

constituents were added probably would not influence the final results; (iii) the set of changes proposed for the architecture: substitution and removal of constituents were not performed. Constituents were only added in the context of this study[20]. However, in the types of SoS investigated, addition was the main dynamic architecture operator, as removal and substitution were not performed in the real cases that inspired our investigation. This threat, therefore, does not influence our results; Lastly, (iv) correctness of the transformation rules: correspondences were established between entities in both models and the resulting simulation model relieved the threat, as a solution was presented. The same model transformation was successfully applied in three different simulations for two different application domains, which reinforces the claim of the transformation correctness.

Regarding internal validity (FELDT; MAGAZINIUS, 2010), variables and metrics were assigned to the pre-established research questions. We believe the important causal relations were accordingly mentioned and examined. External validity, which comprises the extent to which is possible to generalize the findings, our results exhibit potential for generalization, as we analyzed two different domains, and exploited scale (number and diversity of constituents), besides proposing conjectures related to our findings. About construction validity, ASAS approach adopts simulations that enable architects to carefully examine the casual relationship between the the planning of our study and the observed outcome. We carefully examined the simulation traces to be sure that the observed outcome correspond to the effect we intended to measure. Conclusion validity was not considered, as we did not investigated the statistical significance of our findings.

Related work. Although model transformations from architecture descriptions to another formalism for evaluation purposes is not a new research subject (MICHAEL; RIEHLE; SHING, 2009; WANG; DAGLI, 2011; GE *et al.*, 2013; TRUBIANI *et al.*, 2013; CAVALCANTE; OQUENDO; BATISTA, 2014; ALEXANDER; NICOLAESCU; LICHTER, 2015; CAVALCANTE *et al.*, 2016), most proposals deal with architecture representation and evaluation for single systems or SoS architectures at systems-level, and do not tackle software concern. Many formalisms have been adopted to represent SoS architectures, such as UML, SySML, Colored Petri Nets (CPN), with their inherent advantages and drawbacks. ASAS is a model-based approach for simulation and evaluation of SoS software architectures using a visual approach based on simulations. As such, co-related approaches must be analyzed under the following perspectives: (i) Formalism adopted for describing SoS software architectures; (ii) Formalism adopted for SoS simulation; (iii) Adoption of MBE approaches and model transformations; (iv) Support to SoS dynamic architectures;

[20] In another study (MANZANO; GRACIANO NETO; NAKAGAWA, 2018), we exercised all the dynamic architecture operators. But this was not part of the scope of this study, specifically

and (v) Evaluated concern (functional or non-functional). Table 20 shows comparisons between related approaches regarding the aforementioned parameters.

Huynh and Osmundson proposed a systems engineering methodology for performing architecture analysis of SoS, involving process modeling with SysML, the conversion of the resulting SysML models into an executable simulation model for a tool called Extend, and the subsequent analysis via simulation results (HUYNH THOMAS; OSMUNDSON JOHN, 2006). However, the drawbacks of SySML were already discussed, besides the limitations about dynamic architectures.

Ackermann et al. dealt with the concept of software architectures in SoS, and proposed a method for architecture compliance checking, as they claim other previous approaches focused solely on structural characteristics and ignored behavioral conformance (ACKERMANN; LINDVALL; CLEAVELAND, 2009). They modeled the desired behavioral specification in UML sequence diagram notation and behaviors were extracted from the SoS implementation, and mapped for the model of the desired behavior. They explore how their approach can be applied to investigate reliability issues in SoS. However, their approach is manual (ours is automatic) and the architecture is represented in a *box-and-lines* style, which hampers precision in representation.

Michael et al. (MICHAEL; RIEHLE; SHING, 2009) introduced a mathematical model to tie the non-functional requirements of software systems to their SoS software architecture, and developed an approach for the evaluation of the quality of software architecture in light of meeting the requirements. Three levels of evaluation, namely domain reference architecture, platform-independent architecture, and platform-/technology-dependent architecture were proposed. The authors illustrate a SoS software architecture by means of a hypothetical missile defense SoS that consists of a Command, Control and Battle Management (C2BM) system, a set of sensors, and a set of airborne interceptors. They specify the SoS architecture using UML diagrams, as class and activity diagrams. However, they provided no definition for SoS software architecture, or a method for its evaluation, but only reviewed the literature on methods to be adopted or extended for the Verification and Validation (V&V) of SoS software architectures.

Griendling and Mavris investigated the development of a DODAF-based executable architecture approach to analyze SoS alternatives (GRIENDLING; MAVRIS, 2011). They adopted a discrete event simulation using a Petri Net implementation to examine the expected time to complete missions. They also adopted a combination of Microsoft Excel and MATLAB for creating an executable environment prototype. They model a SoS as a graph represented by an adjacency matrix. However, they only analyze two architectural alternatives.

Table 20 – Comparison between related approaches.

Approach (Cronological Order)	Formalism adopted for description of SoS software architectures	Formalism adopted for SoS simulation	Adoption of MBE approaches and model transformations	Support to SoS dynamic architectures	Type of concern evaluated
Huynh and Osmundson (HUYNH THOMAS; OSMUNDSON JOHN, 2006)	SySML	Executable models (Tool called Extend)	Yes	No	Non-Functional (Network Utilization and Throughput)
Ackermann et al. (ACKERMANN; LINDVALL; CLEAVELAND, 2009)	UML	None	No	No	Functional and non-functional (reliability)
Michael et al. (MICHAEL; RIEHLE; SHING, 2009)	UML	None	No	No	Functional and Non-Functional
Griendling and Mavris (GRIENDLING; MAVRIS, 2011)	Petri Nets	Petri Nets	No	No	Functional (Mission)
Ge et al. (GE *et al.*, 2013)	DoDAF views documented in SysML	ExtendSim	No	No	Functional
Xia et al.(XIA *et al.*, 2013)	UML	Simulink	Yes	No	Non-Functional
Tomson and Preden (TOMSON; PREDEN, 2013)	Agents[21]	None	No	No	Functional
Fang (FANG; DELAURENTIS; DAVENDRALINGAM, 2013)	Colored Petri Nets (CPN)	Colored Petri Nets (CPN)	No	Yes	Non-Functional (Complexity and Perfomance)
Bocciarelli and D'Ambrogio (BOCCIARELLI; D'AMBROGIO, 2014)	SySML	HLA	Yes	No	Functional
Guariniello and DeLaurentis (GUARINIELLO; DELAURENTIS, 2014a)	FNDA	None	No	Yes	Functional and Non-Functional (Security and Robustness)
Falkner et al. (CHIPRIANOV *et al.*, 2014; FALKNER *et al.*, 2016)	GraphML*	Customized Tool	Yes	No	Non-Functional (Performance)
Fuchs and Niklas (FUCHS; LINDMAN, 2014)	UML/SySML	None	Yes	No	Not clear
Vierhauser et al. (VIERHAUSER *et al.*, 2016)	Own DSL	ReMinds environment	Yes	No	Not done
Gassara et al. (GASSARA; BOUASSIDA; JMAIEL, 2017; GASSARA *et al.*, 2017)	Bigraph	None	Yes	Yes	None
ASAS Approach (GRACIANO NETO *et al.*, 2018b)	SosADL	DEVS	Yes	Yes	Functional

Ge et al. explored SoS architectural evaluation via DoDAF, using multiple views to describe a SoS architecture (GE *et al.*, 2013). They document SoS through static models based on SySML, such as activity diagrams, and activities, resources, rules, performers, and locations in a static way under DoDAF framework. They also constructed executable models using ExtendSim (formerly called Extend) as the dynamic simulation platform and ran them to check whether the parts were operating as expected. Their claim is the design of the architecture enables the exploration of alternative solutions, and possible behaviors and potentially reachable states can be predicted. Moreover, every simulation can be viewed as a possible evolution of actual SoS. Their simulation does not address dynamic architectures (as they require multiple simulations to see different architectural alternatives), and no explanation on the way their executable model was constructed is provided.

Xia et al. (XIA *et al.*, 2013) evaluated SoS architectures regarding performance and effectiveness, and adopted a model-based approach for transforming system architecture models in Simulink[22] into executable models. Their approach supports measurement of non-functional properties, such as feasibility and efficiency, but it does not consider software architectures (they work on systems level, also addressing hardware and other low-level aspects). They illustrated many SoS architectural viewpoints in conformance with DoDAF, and adopted MagicDraw as the environment for describing the architecture and UML as the basis to represent SoS architecture. They also employed ATL (JOUAULT; KURTEV, 2006)(ATLAS Transformation Languages, which is a model transformation language and toolkit proposed by ATLAS research group based on the QVT (Query/View/Transformation, a model transformation standard proposed by OMG (KURTEV, 2008; OMG, 2011))) to map architectural and Simulink models. They did not evaluate the success of a SoS in accomplishing missions, and adopted UML with all limitations. ASAS supports the evaluation of functional achievements and adopts SosADL as the formalism. Dynamic architecture is not mentioned in their study.

Tomson and Preden introduced Multi-agent Communication Environment (MACE), an agent-based framework for simulation of complex SoS that uses middleware (TOMSON; PREDEN, 2013). They describe SoS as a heterogeneous network of collaborating agents that strive to improve the performance of the resulting system by harmonizing the behaviors of individual agents. Their method to validate the behavior of a SoS adopts the simulation of the system behavior in a controlled environment. Their canonical constructs are Environment (space and time in which agents act), Agent (an autonomous entity in the environment), Mediator (an independent software layer between the agent, environment and other agents), and Connection (a direct link between agents that describes the

[22] www.mathworks.com/products/simulink/

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

communication channel). They evaluated their approach using a distributed monitoring system that detects asymmetric threats based on the work done in the scope of the European Defence Agency's program using UAV (unmanned aerial vehicle) and UGS (unmanned ground sensor). They report the simulation according to time units and information packets transferred in the SoS architecture during simulation. It is not clear how functional properties are evaluated, and dynamic architecture is not mentioned.

Fang et al. reported results on evolving SoS architectures to satisfy capability and performance objectives through adding new systems, replacing existing systems, and changing links between constituents (FANG; DELAURENTIS; DAVENDRALINGAM, 2013). They employed Colored Petri Nets (CPN) (a discrete event dynamic simulation tool) to model, simulate and evaluate the existing and evolving architectures, while still considering the cost of architecture evolutions. They included dynamic complexity as a complexity metric for SoS, and claimed an appropriate evolution choice could be achieved through the examination of the tradeoff space between complexity and performance. The authors conducted a literature review and detected five common approaches for SoS executable architecting, which are Markov chains, Petri nets, system dynamics models, mathematical graphs and Agent Based Modeling (ABM). ABM is suitable for representing an environment composed of interactive parties, however it suffers from computational workload. CPN is a formal language in low level of abstraction, while we adopt SosADL, an ADL in high-level of abstraction conceived especially for SoS software architecture context.

Bocciarelli and D'Ambrogio addressed an automatic generation of simulation from SySML to High-Level Architecture (HLA)[23] simulation (BOCCIARELLI; D'AMBROGIO, 2014). They conducted two model transformations: SysML-to-HLA model-to-model transformation, which takes the SysML-based system specification as input and yields the HLA-based simulation model as output; and a HLA-to-Code model-to-text transformation, which takes the simulation model as input and yields the code that implements the HLA-based simulation as output. A distributed simulation (DS) is performed and the results are checking whether or not the system behavior satisfies the user's requirements and constraints. According to such an evaluation, the SysML specification drives the implementation of the system. Otherwise, a system must be redesigned. However, in their approach, they generate code for distributed simulations, which demand a larger number of computer and simulation peers. This is costly than using a single computer, as we are

[23] HLA (High Level Architecture) is an IEEE standard (IEEE 1516-2010; IEEE 1516.1-2010; IEEE 1516.2-2010) providing a general architecture for the implementation of distributed simulatons (BOCCIARELLI; D'AMBROGIO, 2014), a distributed simulation (DS) formalism. HLA provides the specification of a common technical architecture for use across all classes of simulations in the US Department of Defense (DoD) (DAHMANN, 1997; IEEE, 2010).

able with ASAS. Moreover, in their approach, constituents must be known at design-time, not covering dynamic architectures, as it is required for SoS.

Guariniello and DeLaurentis evaluate and compare different architectures regarding their reliability and robustness under attack (GUARINIELLO; DELAURENTIS, 2014a; GUARINIELLO; DELAURENTIS, 2014b). They reported an ongoing study for the evaluation of SoS architectures by a tool for Functional Dependency Network Analysis (FDNA) in the aerospace SoS domain. They evaluated the impact of cyberattacks, delivering, as outcome, a number that represents the variation in the operability of the directly affected constituent system, and identifying the critical systems and the critical links with respect to their impact on the overall behavior when cyberattacks occur. The authors claim a future improvement will use an agent based model test bed to validate the inputs required by the methods to analyze specific problems. They do not adopt simulations, which can hamper the SoS dynamics visualization.

Falkner et al. (FALKNER *et al.*, 2016) proposed a change of emphasis from SoS specifications to executable models for the purposes of performance prediction. They developed an environment called MEDEA, which is a MDE-based *system execution environment* that supports evaluation and performance prediction of SoS. The methodology underpinning MEDEA follows a performance analysis and prediction process, which consists of three phases, namely modelling, execution (simulation), and performance analysis and evaluation. The process is guided by formulating a performance question, such as *What is the utilization of constituent UAV?*. Th authors model SoS under many views using GraphML, a platform-independent language that supports modelling of interfaces, behavior, and workload of SoS and its constituents. Despite they adopt GraphML to represent SoS architectures, not exactly the SoS software architecture. MEDEA uses the Jenkins continuous integration environment to automate the code generation, compilation, deployment and execution to ensure simulation, reliability and repeatability. Authors evaluate different architectural configurations, but not as a result of changes from another past one. A definition of SoS performance is not given, but they offer a number that measures the utilization related to each constituent to achieve a mission. However, the proposal shows limitations related to: (i) description of SoS architectures (not SoS software architecture), (ii) emulation (not simulation) of performance models above existing middleware and hardware to support early performance evaluation within multiple deployment scenarios, and (ii) lack of evaluation support for dynamic and functional properties.

Fuchs and Niklas reported the state-of-the-practice techniques and technologies used in modeling and simulation of SoS in the European Space Agency (ESA) (FUCHS; LINDMAN, 2014). They model SoS architectures under many views (using UML and

SysML) through MagicDraw. Profiles are generated from the Ecore metamodel, involving UPDM, and the accompanying standards, SysML and SoaML. The profiles are used in the application of ESA-specific stereotypes to standard UML entities in MagicDraw. The general approach taken in the ESA-Architecture Framework (ESA-AF) is based on the Eclipse framework and uses of its inherent extensibility, and diagramming functionality is based on the Eclipse Graphical Modeling Framework (GMF). It is nor clear how they deal with SoS architecture dynamics, as SySML offer only static models.

Vierhauser et al. developed a tool called ReMinds (REquirements Monitoring INfrastructure for Diagnosing Systems of Systems), which is a flexible framework for the development of monitoring solutions that cover different systems forming an SoS (VIERHAUSER *et al.*, 2016). They externalized challenges for monitoring system-of-systems architectures, which include (i) monitoring a SoS at different layers and levels of granularity, (ii) monitoring across different systems with respect to checking global SoS properties, different constituent systems and their interaction, (iii) monitoring of different technologies, (iv) monitoring systems with different speeds, (v) diversity of system requirements and monitors, and (vi) performance of the monitoring solution. They created an entire framework for SoS monitoring from scratch based on Java and C++ and developed a DSL on top of a Java-based incremental checker (no name was given) implemented by Xtext and Xtend. Hence, their framework is co-related to MS4ME environment, for example, with more functionalities. However, they do not work with software architecture concepts, and dynamic architectures, and do not evaluate SoS.

Gassara et al. (GASSARA; BOUASSIDA; JMAIEL, 2017; GASSARA *et al.*, 2017) introduced a tool that supports modeling of SoS architectures through Bigraphs, following the research of Wachholder and Stary (WACHHOLDER; STARY, 2015; STARY; WACHHOLDER, 2015). Bigraphs (Bigraphical Reactive Systems(BRS) (MILNER, 2009)) is a formal/mathematical theory for modeling systems. Authors claim it has also been applied to capture software architectures and modeling applications for context-aware systems and ubiquitous computing environments; and enables the modeling of SoS constituents through their structural and behavioral characteristics. They use GMTE (a tool for graph transformation and matching (HANNACHI *et al.*, 2013)) and model air cargo SoS and its constituents by agents of a certain structure and behavior. The approach also addresses structural modifications in architecture. However, no evaluation method or simulation has been established.

Apart from SoSADL, several other notations have been used for expressing SoS architectures, e.g. (GUESSI *et al.*, 2015) UML[24] (semi-formal), SysML[25] (semi-formal),

[24] http://www.uml.org/
[25] http://sysml.org/

and CML[26] (formal). Since UML and SySML are general-purpose languages, they lack support for the specification and validation of dynamic properties of SoS. On the other hand, CML is a formal language especially conceived for SoS formal specification within the context of Comprehensive Modelling for Advanced Systems of Systems (COMPASS) alliance. However, CML does not focus on emergent behaviors (FITZGERALD *et al.*, 2013).

Transformations from other models to DEVS have also been proposed (CETINKAYA; VERBRAECK; SECK, 2012; GONZALEZ *et al.*, 2015; HU *et al.*, 2014b). However, no approach has supported the transformation of SosADL descriptions into DEVS models. Considering that, we have established a model transformation approach that takes SosADL models of the software systems constituting a SoS and produces DEVS simulation models. None of the aforementioned approaches cover the requirements addressed in ASAS, as discussed.

Patterns for SoS simulations. Other proposals have explored patterns for simulation in DEVS, however under distinct perspectives (CETINKAYA; VERBRAECK, 2011; HAMRI; MESSOUCI; FRYDMAN, 2013; HAMRI; BAATI, 2010; JÉRON *et al.*, 2008; SCHULZ; EWING; ROZENBLIT, 2000). Cetinkaya and Verbraeck established an approach for the management of models and metamodels in the simulation engineering (CETINKAYA; VERBRAECK, 2011). They listed a set of properties model transformation rules should maintain to produce reliable simulations. However, they neither tackle SoS context, nor externalize patterns for the conception of a simulation. Hamri et al. (2010) present a specific catalogue of design patterns for DEVS context that addresses problems such as (i) selection of a method based on type of target and type or value of one other variable without hardwiring the selection as a conditional statement or (ii) Changes in constituent behavior, depending on its internal state, without hardwired multi-part conditional code. However, they do not provide details on how to group DEVS instructions for designing constituents behavior, avoiding conflicts between them. Hamri et al. (2013) report behavioral design patterns to design DEVS behaviors to supply DEVS designers with software engineering techniques (HAMRI; MESSOUCI; FRYDMAN, 2013). However, their patterns are only related to state changes. Differently from out proposal, they provide no grouping of instructions as a set of patterns. Jéron et al. investigated prediction of occurrences of a pattern in a partially-observed discrete-event system. They consider a pattern a set of event sequences modeled by a finite-state automaton. The occurrences of the pattern are predictable if any of them are inferred before the pattern is completely executed. They proposed an off-line algorithm that verifies the property of predictability,

[26] http://www.compass-research.eu/approach.html

but did not address patterns for the conception of simulation, rather than for the automatic identification of patterns in DEVS simulations. Shulz et al. also present a mapping involving DEVS (SCHULZ; EWING; ROZENBLIT, 2000). They argue that the DEVS formalism is more expressive than that of StateCharts and present a mapping of the two system modeling formalisms promises to combine the benefits of formally well-defined models and a sound tool implementation, as we do. However, they do not externalize any pattern applied in this model transformation as we do. Finally, Petitdemange et al. established a solution based on patterns for reconfiguration of SoS software architectures (PETITDEMANGE; BORNE; BUISSON, 2016). Despite involving SoS domain and reconfiguration, no patterns were created for simulation.

3.4. Final Remarks

This chapter introduced ASAS, an approach conceived as a joint effort of two research groups[27] that supports the simulation-based evaluation of functional concerns of SoS software architectures. We explained details on the transformation that maps SoS software architecture descriptions documented in SosADL in DEVS simulation models, and how the resulting models can be used to evaluate different architectural configurations SoS can assume along its life cycle. ASAS enables software architects to identify one or more configurations that yield best results towards achieving the trustworthiness expected from SoS operating in critical domains (GRACIANO NETO, 2017; GRACIANO NETO *et al.*, 2018b). The model transformation adopted in ASAS materializes the SosADL operational semantics, defined in (OQUENDO, 2016a), complementing SosADL models with executable models of SoS software architecture.

Among the contributions from the advances reported in this chapter, we can cite:

1. **A model-based approach to produce simulations of SoS software architecture**: ASAS enables software architects to automatically produce simulation models with exogenous dynamic reconfiguration, i.e., one of the architectural elements being simulated (also automatically created) is responsible for the management of reconfiguration actions at runtime. Simulations (i) represent the SoS inherent dynamics, (ii) offer a visual approach, (iii) a more precise control over the topology of the architecture, and (iv) how communication is established between constituents. This approach advances the precedent ones by harmonizing static and dynamic views of SoS architectural documentation, enriching the existent approaches based only on static or dynamic models. Moreover, ASAS focuses on SoS software architectures,

[27] SofTware ARchitecture Team (START/ICMC-USP) and ArchWare (IRISA/UBS)

isolating the software view from hardware and operational concerns (inherent to system engineering approaches), abstracting low-level issues, and automating the production of simulation models from a specification of SoS software architectures in a high level of abstraction;

2. **A method to evaluate SoS inherent dynamics and architectural alternatives**: ASAS comprises a method to support the assessment of different architectural configurations that can emerge at runtime. Architectural decisions can be drawn considering a pre-established set of metrics, enabling the selection of the best architectural configurations, which is a contribution over the past approaches, especially regarding SoS software architectures;

3. **A means for the validation of SoS behaviors (including emergent ones)**: ASAS, by means of the simulation models automatically produced, comprises a method that allows an architect to evaluate the SoS software architecture about the functionalities intended to be offered. Besides predicted behaviors, ASAS also supports prediction of SoS behaviors in case of non-predicted failures and exceptions. SoS behaviors are deliberately and intentionally designed (BOARDMAN; SAUSER, 2006), i.e., the SoS engineer is the major player for creatively exploring the functionalities delivered by the constituents, assembling them for innovative purposes. Validation activity consists in the checking of the conformance between the missions specified at the requirements level and the corresponding emergent behaviors that accomplish such missions (IEEE, 2012; IEEE Computer Society, 2014). However, due to the nature of such behaviors, a validation approach for SoS software architectures requires a dynamic viewpoint that externalizes emergent behaviors. It should support software architects to predict and validate desired emergent behaviors. The following four major categories of techniques for validating software architectures[28]: scenario-based, simulation-based, mathematical/logical-based, and experience-/metric-based. Considering the dynamic nature of SoS software architectures, a simulation-based approach is undoubtedly the appropriate one[29]. Simulation-based approaches have supported the validation of dynamic properties for SoS (NIELSEN *et al.*, 2015; MITTAL; RAINEY, 2015; MICHAEL; RIEHLE; SHING, 2009; SAUSER; BOARDMAN; VERMA, 2010; ZEIGLER *et al.*, 2012; WACHHOLDER; STARY, 2015; FRANÇA; TRAVASSOS, 2016). Such approaches (MICHAEL *et al.*, 2011; FRANÇA; TRAVASSOS, 2016; WACHHOLDER; STARY, 2015; XIA *et al.*, 2013): (i) support the validation of expected emergent behaviors, (ii) empower the observations of unex-

[28] Dobrica and Niemela discuss further details about validation methods for software architecture (DOBRICA; NIEMELE, 2002; MICHAEL; RIEHLE; SHING, 2009; MICHAEL *et al.*, 2011).
[29] Nielsen et al. deeply discuss simulation approaches for SoS (NIELSEN *et al.*, 2015).

pected emergent behaviors; (iii) enable the prediction of errors, diagnosing them and permitting corrections; and (iv) provide a visual and dynamic viewpoint, reproducing stimuli the system can receive from an operational environment. Moreover, ASAS possibly enables the exploration of non-predicted behaviors that may emerge at runtime due to some specific set of stimuli. Emergence is a term to denote the study of emergent behaviors in a range of situations. Emergent behavior is a particular SoS characteristic triggered by the reception of stimulus and data exchanged between the constituents, and between constituents and their environment (GRAHAM, 2013). Such behaviors are an holistic phenomena that occurs through a certain number of interactions among the constituents that produce a global result that could not be delivered from any one of them in isolation.

4. **Patterns for the creation of DEVS SoS simulations**: Another contribution refers to patterns to automatically derive excerpts of code representing behaviors of constituents that form a SoS simulation. This solution was recurrently applied during the exploration of different cases and showed potential for reuse. The idea can be generalized and externalized for automatic generation of any discrete-event based simulation formalism, and the model transformation itself can be reused in other contexts;

5. **Operational Semantics and Animation for SosADL**: Executable models demand an operational semantics, i.e., a structured description of the expected result of each construct of the source language, realizing an entire framework that drives how a model will represent the dynamics of a system. The model transformation conceived also comprises a definition of the operational semantics for SosADL, and as a consequence, originates executable (simulation) models equivalent to the source models. Indeed, simulations are animations for SosADL models. Animations can help one to better understand modeled behaviors. Novices and experienced developers can benefit from the visualization of modeled behaviors provided by model animators. Model animation can provide quick visual feedback to novice modelers and help them identify improper uses of modeling constructs. Experienced modelers can use model animation to understand designs created by other developers better and faster (FRANCE; RUMPE, 2007). π-Calculus for SoS specifies the operational semantics of SosADL (any language with an operative semantics is executable) (OQUENDO, 2016b). ASAS contributes by defining a denotational executable semantics with DEVS that conforms to the operational semantics defined in π-Calculus for SoS for the subset of SosADL that was implemented. ASAS offers the first execution system for SosADL. Model transformations generate simulations, which comprise

the executable semantics for SoSADL, accordingly supporting implementation and validation of SoS operation at design-time.

6. **Domain Independence**: One of the concerns evaluated in our case study was the domain independence of our solution. We evaluated ASAS using two orthogonally different domains, namely space and urban flood monitoring. For both, the model transformation successfully produced functional simulation models that supported the evaluation of SoS software architectures regarding their functional requirements. Therefore, we claim our approach can be applied to other domains.

Results of this research can open possibilities for the emergence of new research branches and several applications in the forthcoming years. Among such possibilities, we can cite:

1. **Industrial simulation and co-simulation**: Despite the success of ASAS for predicting SoS behaviors and evaluating SoS software architectures regarding their effectiveness to offer functionalities, the evaluation of some non-functional properties require a more industrial approach. *Co-simulation* is the name given to the practice of combining two or more simulators for reliably representing real conditions to which a system can be submitted in order for the evaluation of specific properties, as river, sea, wind, and rain simulators, electricity and automotive simulators, or network protocols and cyber attacks simulators (GOMES *et al.*, 2017). ASAS shall be extended towards an automatic generation for other simulators, or even to fit the combination of multiple simulators;

2. **Continuous Value Delivery for SoS**: Most SoS undergo periodic rearchitecting. However, this does not necessarily occur at the 'speed of need' (RICCI *et al.*, 2013). *Value robustness* is the ability of a system to deliver stakeholder value in the face of the dynamic world in which the system operates over its lifespan. Sustaining stakeholder value delivery in an operational system is a continual and difficult challenge (ROSS; RHODES, 2015). ASAS can also be extended to analyze such deliverable value;

3. **Simulation-based Software Engineering for SoS**: Software Engineering is the application of systematic principles to the planning, design, development, testing, implementation, and maintenance of software-based systems (TANIR, 2017). Due to particular characteristics of SoS and their importance, researchers have investigated on ways of properly engineering software for SoS, which has created the so-called new area of interest Software Engineering for Systems-of-Systems (SESoS) (GUESSI *et al.*, 2015; CALINESCU; KWIATKOWSKA, 2010; NAKAGAWA *et al.*, ; DRIRA;

OQUENDO, 2015). On the other hand, a branch of Software Engineering has been founded on simulation-based studies (FRANÇA; TRAVASSOS, 2012; FRANÇA; TRAVASSOS, 2015; TANIR, 2017), as simulations enable an early anticipation of inconsistencies at design-time, preventing propagation of errors for the final product. ASAS can also foster the development of this branch research using other simulations formalisms and tools, as many of them have been proposed (TENDELOO; VANGHELUWE, 2017), and strategies can be established to use simulations as source of evidence and validity, benchmarking SoS software models, and advancing verification and validation towards a simulation-based sofware engineering;

4. **Multiple missions (deadlocks and resource competition)**: We investigated how a small set of missions can share a finite set of constituents to accomplish SoS missions. However, in a larger context, the concurrency of many missions being accomplished by a finite set of constituents can lead to deadlocks, bottlenecks, and other recognized problems from the domain of multiple communicating processes. Solutions and strategies that deal with such issues must be investigated and designed;

5. **Methodology for the Validation of SoS Behaviors (including emergent ones)**: ASAS offers a visual approach for the evaluation of SoS architecture behaviors. However, when the number of constituents and functionalities offered by SoS increases, this visualization is no longer trivial. Then, signals that characterize the emergent behaviors and conclusion of behaviors foreseen in SoS at large scale must be defined so that the conformance between missions and their corresponding emergent behaviors can be checked (GRACIANO NETO, 2016). Therefore, a robust and reliable methodology must be established for the validation of emerging behaviors, and ASAS is an approach upon which this advance can be built.

6. **Visual monitoring of SoS dynamics**: ASAS supports the simulation of SoS from an operational perspective, i.e., simulation provides a mechanism to visualize how data is exchanged between the constituents during the accomplishment of missions. However, DEVS simulators do not use to support the simulation of constituents movements, for example. A future work shall provide an extension of ASAS to automatically provide a visual simulation of constituents movements over their environments. This would be particularly useful to study dynamics of a constellation of satellites over a territory, or to simulate the movement and interaction of autonomous cars in a smart city.

Next chapter presents a strategy to reestablish the consistency between simulation models and architectural models through a reverse engineering transformation approach.

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

CHAPTER

4

BACK-SOS: A MODEL-BASED APPROACH FOR RECONCILIATION BETWEEN DESCRIPTIVE AND PRESCRIPTIVE MODELS OF SYSTEMS-OF-SYSTEMS SOFTWARE ARCHITECTURES

As SoS progresses its operation, architectural changes result in several different architectural configurations at runtime. Frequent reconfigurations can quickly degrade the quality of the SoS architecture as it further deviates from its initial prescriptive architecture. This chapter presents Back-SoS, a model-based approach that supports the verification of the conformance between architectural configurations to be executed at runtime and their prescriptive architecture. Back-SoS is complementary to ASAS as, whilst ASAS (presented in Chapter 3) enables us to evaluate many different architectural configurations that arise at runtime regarding their impact on SoS operation, Back-SoS enables the architect to realign, during design-time, the conformance between the prescriptive SoS architectural model with its corresponding runtime version (the descriptive architecture). We demonstrated the feasibility of this approach in an urban Flood Monitoring SoS (FMSoS) and concluded that Back-SoS could bring important support to avoid architecture drift and, as a consequence, to improve the quality of SoS and their architectures.

4.1. Presentation of Back-SoS Approach

Back-SoS prescribes a *backward* model transformation that reconciles consistency between the intended architecture and runtime models. In our approach, SoS execution models are represented using DEVS, which is a notation that supports representation of SoS at runtime, besides supporting architectural reconfigurations (ZEIGLER *et al.*, 2012). In turn, SoS prescriptive models are documented in SosADL (OQUENDO, 2016c). For the context of this chapter, we solve the problem of restoring the alignment between concrete architecture and architectural instance. Other gaps to be bridged and challenges regarding this research area are mentioned in Section 4.3.

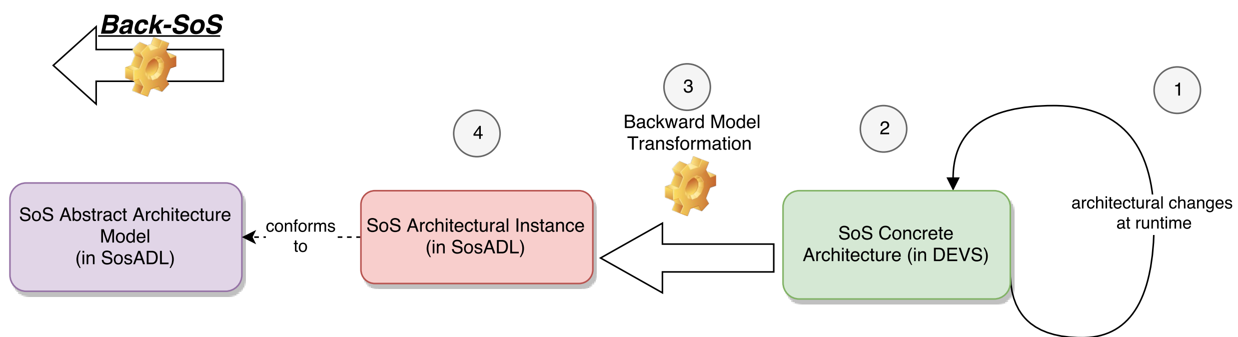

Figure 26 – Model discovery mechanism for SoS software architectures.

Back-SoS is illustrated in Figure 26, and consists of the following steps:

Step 1. Model recovery activity and model discovery activity: After SoS simulation starts, a number of architectural configurations can arise. Changes can take place for a number of reasons, which potentially cause inconsistencies between current SoS architectural configuration and models that document its intended architecture. Every change that occurs at SoS architectural level triggers a model recovery and discovery mechanism. Model recovery for SoS software architectures comprises the identification of the architectural elements that are currently in operation at the SoS architecture. After recovery is performed, model discovery is performed, querying the simulation about the existing links between architectural elements (constituents and mediators) after the change is completed. Step 1 produces a DEVS model as an output. As changes can be beneficial or harmful for SoS behaviors, the conclusion of model discovery mechanism triggers the following step;

Step 2. Architectural evaluation: As a SoS can exhibit a diversity of coalitions, a set of parameters is chosen by the software architect for assessing the novel architectural configuration, obtained by changes, in regards to the precedent ones (HOFMEISTER *et al.*,

2007). These parameters can include behaviors that are intentionally designed for a SoS and triggered by specific stimuli (GRACIANO NETO *et al.*, 2017), or non-functional properties, such as performance. Then, an architectural analysis activity takes place (GRACIANO NETO, 2016; GRACIANO NETO, 2017; GRACIANO NETO *et al.*, 2018b). Architects evaluate the new architectural configuration according to the set of parameters chosen, and classify them in beneficial, neutral, or harmful for SoS operation. After evaluating each of them, architects can design strategies for maintaining beneficial SoS architectural configurations in spite of SoS dynamic architecture, and perform the following step;

Step 3. Reconciliation: Between several architectural configurations, SoS architects select one that exhibited best results, such as performance, and functional response. Then, the chosen architectural configuration is submitted to reverse transformation, being updated on the source prescriptive SoS architectural model.

Step 4. Architectural consistency checking: After the prescriptive model (architectural instance part) is updated, mechanisms must check whether the new version of the concrete SoS architectural description (propagated to the current architectural instance) still conforms to the original SoS abstract architecture. If yes, the processes is finished until a new change at runtime occurs. If not, a new architectural configuration is sought and reverse transformed until a right one emerges to match the original abstract architecture.

Step 1 requires a characterization of changes that can cause deviations in a SoS software architectures, and the creation of a model recovery and discovery mechanism to automatically recover SoS architectural configuration elements, and link them to materialize the current architectural configuration (also achieving a model discovery mechanism for SoS). Hence, Section 4.1.1 reports the characterization of architectural drifts that we created and used to establish model recovery and discovery mechanisms (discussed in Section 4.1.2) for SoS architectures at runtime. Step 2 (architectural evaluation) is briefly discussed in Section 4.1.3, as it is presented in Chapter 3. Section 4.1.4 details the reverse model transformation used in Step 3 (Reconciliation), and Section 4.1.5 discusses mechanisms to check architectural consistency in Step 4.

4.1.1. Architectural Drift in SoS architectures

In SoS domain, an *architectural deviation* is any architectural change that violates the SoS prescriptive architectural model, i.e., the architecture initially intended for a SoS. Hence, a *SoS architectural drift* occurs due to the introduction of any new design decisions (architectural deviation) in a descriptive model (constituents, mediators, or interfaces)

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

that were not predicted at the SoS prescriptive architecture. Actually, deviations often occur in SoS architectures, as any new constituent that join a SoS will change its current architectural configuration. Changes can be beneficial or harmful. But, as software architects are concerned to build architectures that offer the best results for their clients, beneficial architectural configurations are intended to be propagated to the official architectural documentation materialized by the SoS prescriptive architectural model. Hence, preventing such an architectural drift requires reestablishment of consistency and synchronization between the descriptive SoS current model and the correspondent prescriptive model documented using an ADL.

Cavalcante et al. defined a canonical set of architectural changes that can arise at any type of dynamic architecture, which are (CAVALCANTE; BATISTA; OQUENDO, 2015): *insertion of architectural element*, *removal of architectural element*, *substitution of architectural element*, and *rearrangement of the entire architecture* establishing, hence, new connections. In SoS, the basic architectural elements are constituent systems, mediators, and coalitions (OQUENDO, 2016c; NIELSEN *et al.*, 2015). Then, these aforementioned architectural changes could be rewritten for SoS domain as:

- insertion of new constituent,
- removal of constituent,
- substitution of constituent, and
- coalition rearrangement.

This group/set of architectural changes could also be applied to mediators that form the SoS architecture. As a matter of fact, considering that all constituent systems interoperate with others solely by means of a mediator, any of these operations on constituents cause identical operations on their respective mediators, i.e., deleting a constituent necessarily requires removal of mediators attached to it; and adding a constituent also requires adding mediators to connect the new constituent, allowing its participation in the coalition.

4.1.2. Model Recovery and Discovery Mechanism at SoS Concrete Architectural Level

For simulation purposes and for our context, model recovery and discovery are performed at the same step. Java objects are available in MS4ME environment to support access to the simulation model that represents the running SoS. Figure 27 illustrates a

simulation of a small instance of a SoS composed by smart sensors, mediators, and gateways, and mechanisms to reconfigure and discover models in DEVS simulations. The strategy that we designed consists of adding two additional structures in the SoS architecture simulation: one is dynamic architecture controller, which supports dynamic architecture during SoS simulation in DEVS, and a model discovery mechanism (MDM), which accesses the simulation at runtime to map the current SoS architecture to SosADL. When an architectural change occurs or is required in a SoS simulation, the SoS dynamic architecture controller performs such change (for instance, removal of a constituent). After that, the controller calls the model discovery mechanism. MDM accesses the Java simulation object managed by MS4ME, reads the coupled model, and maps back the simulation model to SoSADL model, updating the original specification.

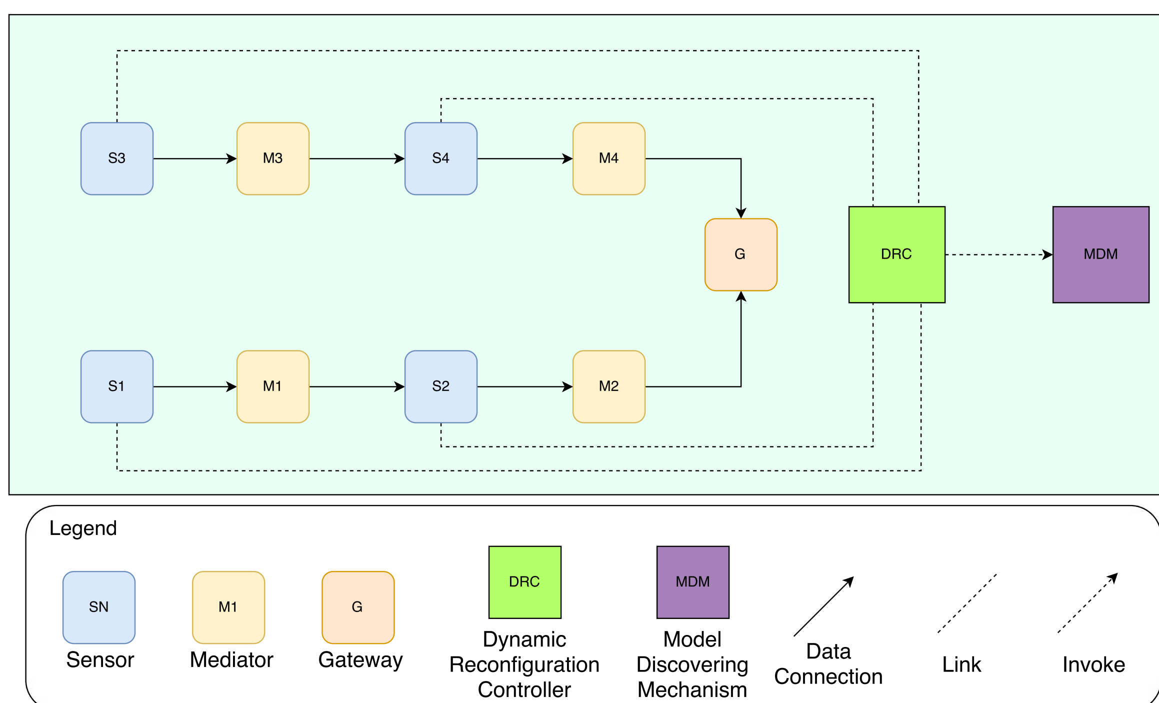

Figure 27 – Model discovery mechanism for SoS software architectures.

Listing 5 presents excerpts of code written in XTend that automatically generates a mechanism for model discovery in DEVS SoS simulations. Listing 5 shows a method called `mSyncModel`, which is responsible to deploy a Java method (`syncModel`) within the Model Discovery Mechanism (MDM) to write the SosADL model that corresponds to the current running architectural configuration. Line 6 creates the file in which the new SosADL model will be written. Line 8 writes the definition of SoS gates (as Lines 1-12 in Listing 7). Lines 10-13 write the definition of constituents within the coalition specification (Lines 13-21 in Listing 7). Lines 16-18 writes the bindings, which correspond to all combinations between output ports and input ports that establish a communication between two systems (Lines

23-39 in Listing 7).

```
1 def private String mSyncModel(){
2     return '''
3     add additional code
4     <%
5       private void syncModel(){
6         BufferedWriter writer = new BufferedWriter(new FileWriter("Model/"+
   ↪ getName().replace("DRC","")+"REC"+Integer.toString(syncCount)+".sosadl
   ↪ "));
7
8         writer.write(aux1SyncModel());
9
10        for(AtomicModelImpl e : constituents){
11          writer.write("      " + e.getName() + " is " + getTypeCons(e));
12          writer.newLine();
13        }
14        writer.write("}");
15        writer.write("binding{");
16        for(Connection e : connections){
17          writer.write("      unify one{ " + e.outClass.getName()+"::"+e.
   ↪ outPort.getName()"} to one {" + e.inClass.getName()+"::"+e.inPort.
   ↪ getName()});
18                  }
19      }
20    %>!
21    '''
22  }
```

Source code 5 – Model discovery mechanism.

4.1.3. Architectural Evaluation

Step 2, architectural evaluation is performed using an approach termed ASAS (A Model-Based Approach to Simulate and Evaluate Software Architectures of Systems-of-systems) (GRACIANO NETO *et al.*, 2018b), initially defined in Chapter 3. ASAS approach supports SoS simulation and architectural evaluation of different configurations of SoS software architectures. After a simulation model for a SoS software architecture is produced, such model is executed, and analysis of functional and non-functional properties is performed to choose between the diverse architectural configurations that emerge at runtime due to changes.

This step consists of the following parts:

- Producing a simulation model for a SoS software architecture;
- Monitoring simulation execution;

- Assessing SoS architecture according to a pre-defined set of metrics.

ASAS methodology is applied herein as a means for supporting the evaluation of the new architectural configuration that emerges from Step 1 (model recovery and discovery), assessing whether it exhibits better results than the precedent one. If yes, reconciliation step is triggered to be performed.

4.1.4. Reconciling Descriptive and Prescriptive SoS Software Architectures

To establish Back-SoS, we adopted mappings between DEVS and SosADL metamodels, as presented in Table 25. In DEVS, a coupled model specifies how constituent systems exchange data between themselves. The code of such coupled models systematically specifies which entities are involved in the SoS and how they interoperate, i.e., which systems send data and which systems receive such sent data. In SosADL, SoS software architectures are modeled by a concept termed coalition. Coalitions represent a temporary alliance among constituents that can be dynamically formed at runtime to fulfill the SoS mission through emergent behaviors. Mediators, in turn, are architectural elements concerned with establishing communication between two or more constituents) (WIEDERHOLD, 1992; INVERARDI; TIVOLI, 2013).

Table 21 – Mapping between DEVS and SosADL.

SoS concept	**DEVS**	**SosADL**
Set of Constituent Systems	Decomposition	Coalition
Data Types	Data Type	Data Type
Gate/Connection	DEVS Port	Gate/Connection
Interfaces	Coupling	Binding
SoS Architecture	Coupled Model	Coalition + Binding

Set of Constituent Systems. They become a Decomposition, i.e., a statement of the coupled model that systematically lists all the inner structures (e.g., systems, mediators, among others) that form the SoS software architecture (ZEIGLER *et al.*, 2012).

Data Types. Constituents exchange data according to a pre-defined data type. Hence, data types are preserved and properly converted into DEVS data types format.

Gate/Connection. SosADL Connections are mapped into DEVS ports.

Interfaces. Interfaces encapsulate the communication between two systems. In SosADL, interfaces are specified through bindings, which correspond to all combinations between

output ports and input ports that establish a communication between two systems. In DEVS, each one of the bindings is mapped into a coupling, i.e., a statement that tells how information flows between a pair of two specific systems in the SoS.

Sos Architecture. Finally, the SoS software architecture is represented as a concrete architecture in SosADL, which specifies a coalition and a set of bindings, and mapped subsequently in a coupled model, which is a set of a decomposition and couplings.

For exemplification purposes, our following listings present excerpts of real code of a software architecture of a Flood Monitoring and Emergency Response SoS (FMSoS). Flood Monitoring and Emergency Response SoSs address the problem of flash floods, which raise critical harms in different countries over rainy seasons. This becomes particularly critical in cities that are crossed by rivers (OQUENDO, 2016a). This SoS involves the National Center for Natural Disaster Monitoring, which monitors 1000 cities, with 4700 sensors, including 300 hydrological sensors, and 4400 rain gauges. We will use a subset of this Flood Monitoring and Emergency Response SoS, which is itself an SoS, i.e., the Urban River Monitoring System, henceforth, FMSoS (GRACIANO NETO *et al.*, 2017). Such SoS includes wireless river sensors, telecommunication gateways, unmanned aerial vehicles (UAVs), Vehicular Ad Hoc Networks (VANETs), Meteorological Centers, Fire and Rescue Services, Hospital Centers, Police Departments, Short Message Service Centers and Social Networks. For this context, we focus on smart sensors, which are fixed smart cyberphysical systems monitoring flood occurrences in urban areas, located on river edges. In that SoS, such sensors are responsible by collecting data from the river (such as the water level), and forwarding such data through mediators until gateways, that can deliver data for specific purposes of public authorities.

Listing 6 shows a coupled model that represents the current state of a SoS simulation that can be discovered and originate a correspondent SoS software architecture specification in SosADL. Sensors in Listing 6 transmit data to their closest mediator (Lines 2, 4, 6, and 8). These mediators receive such data in Lines 3, 5, 7, and 9 forward them to the neighbor sensors. Since `Sensor2` and `Sensor4` sent their data to `Mediator2` and `Mediator4` respectively (Lines 4 and 6), the gateway was reached (Lines 5 and 9). When these data arrive at the gateway, their values are tested against a pre-determined depth threshold. If they are higher, the gateway emits a flood alert. Thus, the network of exchanged messages between constituents and the flood alert trigger indicate that the SoS mission, i.e., producing flood alerts, has been accomplished by these constituent system.

```
1 From the top perspective, WnsMonitoringSosArchitecture is made of Sensor1,
    ↪ Sensor2, Sensor3, Sensor4, Gateway, Mediator1, Mediator2, Mediator3,
    ↪ and Mediator4!
2 From the top perspective,  Sensor1 sends Measure to Mediator1!
```

```
3 From the top perspective,  Mediator1 sends Measure to Sensor2!
4 From the top perspective,  Sensor2 sends Measure to Mediator2!
5 From the top perspective,  Mediator2 sends Measure to Gateway!
6 From the top perspective,  Sensor3 sends Measure to Mediator3!
7 From the top perspective,  Mediator3 sends Measure to Sensor4!
8 From the top perspective,  Sensor4 sends Measure to Mediator4!
9 From the top perspective,  Mediator4 sends Measure to Gateway!
```

Source code 6 – Coupled model for FMSoS generated in DEVS.

Listing 7 presents a SoS architectural specification in SosADL. A SoS architectural specification has a set of gates (Lines 3-10), and a coalition (Lines 12-22). Coalition is the name given for the arrangement of constituents that form a specific architectural configuration. As such, a coalition has a set of constituents, and bindings, i.e., their interfaces and how constituents are connected to each other (Lines 22-32). Listing 7 shows the result achieved by applying Back-SoS in a DEVS simulation model, obtaining a SosADL model as output, as shown in Listing 6.

```
1  sos MoniteringSos is {
2    architecture MonitoringSosArchitecture( ) is{
3      \\gates declaration hidden
4
5      behavior coalition is compose {
6        sensor1 is Sensor
7        sensor2 is Sensor
8        sensor3 is Sensor
9        sensor4 is Sensor
10       gateway is Gateway
11       mediator1 is Mediator
12       mediator2 is Mediator
13       mediator3 is Mediator
14       mediator4 is Mediator
15     } binding {
16       unify one { sensor1::measurement::measure } to one { mediator1::
       ↪ transmit::measure } and
17       unify one { mediator1::transmit::measure } to one { sensor2::
       ↪ measurement::measure } and
18       unify one { sensor2::measurement::measure } to one { mediator2::
       ↪ transmit::measure } and
19       unify one { mediator2::transmit::measure } to one { gateway::
       ↪ notification::measure } and
20       unify one { sensor3::measurement::measure } to one { mediator3::
       ↪ transmit::measure} and
21       unify one { mediator3::transmit::measure } to one { sensor4::
       ↪ measurement::measure } and
22       unify one { sensor4::measurement::measure } to one { mediator4::
       ↪ transmit::measure} and
23       unify one { mediator4::transmit::measure } to one { gateway::
       ↪ notification::measure }
24     }
```

```
25    }
26 }
```

Source code 7 – Coalition specified in SosADL.

Once the SoS architectural instance (SosADL description of a SoS software architecture intance) is reconciled with current SoS concrete architecture (DEVS simulation model), it is necessary to check whether the novel architectural instance is still adherent to the original abstract architecture. This activity is explained in the next section.

4.1.5. Mechanisms to Check Conformance between Abstract and Concrete Software Architectures

A SoS abstract architecture comprises a set of the types of constituents that can compose a SoS software architecture. Hence, an abstract architecture specification involves only the candidate type of constituents that are allowed to join and be part of a SoS, and the potential connections that can be established between them.

Listing 8 depicts an example of an abstract architecture of a SoS documented in SosADL (OQUENDO, 2016a). As shown in such listing, a coalition may involve possibly many sensor constituents, exactly one gateway constituent and possibly many transmitter mediators. The abstract architecture specification does not specify which constituent systems will exist at runtime, but simply which are the possible systems that may exist and which are the required conditions for forming a coalition among the systems identified at runtime to participate in the SoS (OQUENDO, 2016a).

```
1  architecture WnsMonitoringSosArchitecture() is {...
2    behavior coalition is compose {
3      sensors is sequence{Sensor}
4      gateway is Gateway
5      transmitters is sequence{Transmitter}
6    } binding {...
7       forall{isensor1 in sensors, isensor2 in sensors
8         suchthat
9         exits{itransmitter in transmitters
10          suchthat
11            (isensor1 <> isensor2) implies
12          unify one{itransmitter::fromSensors}
13            to one{isensor1::measurement::measure}
14              and unify one{itransmitter::towardsGateway}
15              to (one{isensor2::measurement::pass}
16             xor unify one{itransmitter::towardsGateway}
17              to one{gateway::notification::measure}
18        }
19     // multiplicities are 'one', 'none',
20     // 'lone' (none or one),
```

```
21      // 'any' (none or more),
22      // 'some' (one or more), 'all'
23      }
24 } guarantee {...}
```

Source code 8 – A SoS abstract architecture specified in SosADL.

A mechanism to check conformance between concrete architectural instances of a SoS and its associated abstract architecture can be manual or automatic. Essentially, checking the conformance of an architectural instance to an abstract architecture involves assessing whether there is any type of constituent that was not predicted and that currently composes the SoS architecture, and whether there exist, in the current architectural instance, any association between two types of constituents that was not originally specified.

4.2. Evaluation

We conducted a case study to investigate whether Back-SoS model transformation is feasible, i.e., if it is possible to update the SoS architectural prescriptive model considering the current descriptive SoS architectural model.
Case study execution plan: We evaluated Back-SoS approach using FMSoS. Rivers that cross urban areas represent great danger to the population in raining seasons, often causing flash-floods that may damage properties, businesses, and spread diseases. The FMSoS used to evaluate our approach is composed of smart sensors that use software to monitor the occurrences of flooding in urban areas, and crowdsourcing systems that enable the population to communicate threats of flood while they walk or move in the city. The FMSoS used can trigger a single emergent behavior: *flood alert*. Smart sensors are scattered along the river at a regular distance and their communication is mediated by transmitters between them. Data collected by each sensor are transmitted to a gateway that can emit an alarm for the public authorities when it detects a flooding event. Crowdsourcing systems are installed in citizen mobiles and can also communicate floods or increasing in the water level. **Case study execution plan**: We evaluated Back-SoS approach using FMSoS. Rivers that cross urban areas represent great danger to the population in raining seasons, often causing flash-floods that may damage properties, businesses, and spread diseases. The FMSoS used to evaluate our approach is composed of smart sensors that use software to monitor the occurrences of flooding in urban areas, and crowdsourcing systems that enable the population to communicate threats of flood while they walk or move in the city. The FMSoS used can trigger a single emergent behavior: *flood alert*. Smart sensors are scattered along the river at a regular distance and their communication is mediated by transmitters between them. Data collected by each sensor are transmitted to a gateway that can

emit an alarm for the public authorities when it detects a flooding event. Crowdsourcing systems are installed in citizen mobiles and can also communicate floods or increasing in the water level. **Case study execution plan**: We evaluated Back-SoS approach using FMSoS. Rivers that cross urban areas represent great danger to the population in raining seasons, often causing flash-floods that may damage properties, businesses, and spread diseases. The FMSoS used to evaluate our approach is composed of smart sensors that use software to monitor the occurrences of flooding in urban areas, and crowdsourcing systems that enable the population to communicate threats of flood while they walk or move in the city. The FMSoS used can trigger a single emergent behavior: *flood alert*. Smart sensors are scattered along the river at a regular distance and their communication is mediated by transmitters between them. Data collected by each sensor are transmitted to a gateway that can emit an alarm for the public authorities when it detects a flooding event. Crowdsourcing systems are installed in citizen mobiles and can also communicate floods or increasing in the water level.

Scenario: A simulation model was produced by a DEVS expert from an initial SoS architecture model. As simulation proceeds, SoS architecture changes due to addition, removal, and substitution of constituents, besides rearrangement of the entire architecture. To make the SoS software architecture specification sustainable, it is important to maintain the runtime architectural configuration synchronized with the source architectural specification. Then, we established the following research question:

Research Question: How efficient is Back-SoS approach to maintain the synchronization between a SoS concrete architecture at runtime and its original architectural instance model?

Rationale: As we established a reverse model-based approach to reconcile descriptive and prescriptive SoS architectural models, we want to assure that our approach is successful in its intended purpose in regards to the relations between concrete architecture and architectural instance[1]. This research question evaluates the correctness of the model transformation to generate the expected outcome.

Metrics: The following metrics were established to evaluate our approach:

M1 - Correctness: Number of resulting SoS prescriptive architectural instance models correctly extracted from the simulation model. Correctness is assessed via manual inspection.

M2 - Mutation coefficient: Aiming at assessing the degree in which changes were being performed across the architectural reconfigurations, we defined a **mutation coefficient (*mc*)**, i.e., a rate that represents the similarity between the original model in SosADL and

[1] Correspondences and automatic matching and reconciliation between architectural instance and abstract architecture at prescriptive model level is not matter of investigation for this case study.

the current version obtained from the current architectural configuration originated by the reverse transformation. This coefficient measures how many lines are different between two models, and divide by the total amount of lines of the larger model. The more similar these models are, closer to one is the coefficient value. The more distant these models are, closer to zero is this value. This metric was established to ensure that there was significant difference between the initial architectural configuration and those arising from the dynamic architecture.

Method: To perform such investigation, we followed Back-SoS steps described in Section 4.1, which involves 1) model recovery and discovery after changes on SoS simulation model, 2) architectural evaluation, 3) reconciliation, and 4) architectural consistency checking.

Reporting

Context Settle: We specified an initial version of a FMSoS software architecture with a configuration structured with four sensors, four mediators, and one gateway, as illustrated in Figure 28. Considering a first architectural version of a FMSoS software architecture, we performed a well-defined set of changes. We added one sensor per time (and the corresponding mediators), until reaching a set of 30 different sensors in the SoS architecture, resulting in 26 different architectural configurations. After that, we increased the number of gateways until reaching ten. Next, one crowdsourcing gateway was added. Lastly, crowdsourcing systems were added until 20, and crowdsourcing gateways until 9. At total, 65 different architectural configurations were obtained due to architectural changes.

We performed a set of 65 architectural changes in FMSoS simulation. Each change originated a new architectural configuration of the simulated SoS as outcome. As such, we performed the procedures below for each one of the architectural configurations. It is important to highlight that architectural evaluation is not the focus of the approach reported herein. Hence, since the aim of this evaluation is to assess the model recovery, discovery, and reconciliation mechanisms, architectural evaluation was not performed and every change performed in the SoS simulation model was propagated to the SosADL model in order to exercise the aforementioned mechanisms.

Our investigation involved seven SoS software architects: one that designed and implemented the model discovery, recovery, and reverse transformation solutions (SA1); one that specified the initial architectural configuration in DEVS simulation models (SA2); one to execute and monitor changes in the SoS simulation model during its execution and to report on model recovery and discovery mechanism (SA3), one to perform the reverse model transformation execution (SA4), and three SoSADL experts to perform a

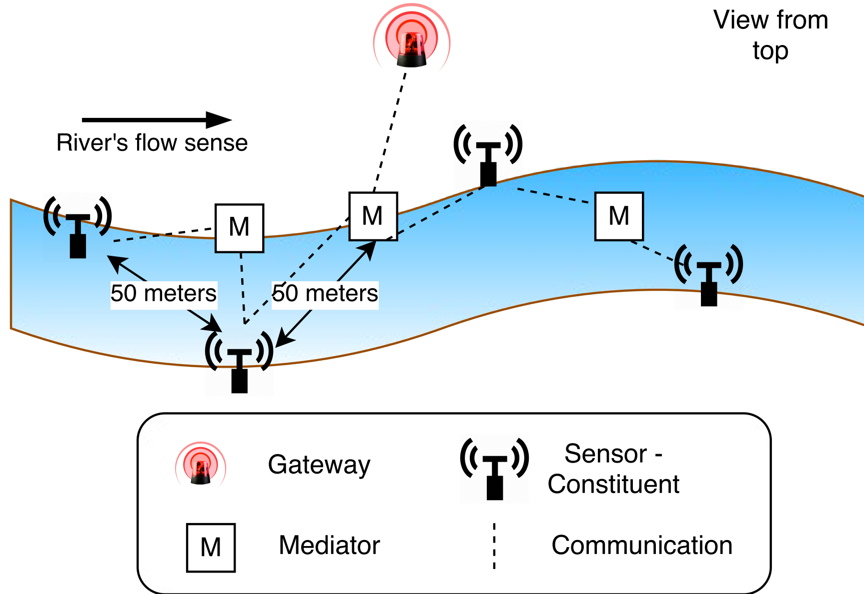

Figure 28 – An illustration of the initial state of the FMSoS architectural configuration.

peer-review on the SosADL models originated from the reverse transformation (SA5, SA6, and SA7). We report below the results obtained during the conduction of the well-defined steps.

Step 1. Model recovery and discovery: SA1, SA2, and S3 worked together on this activity. SA1 and SA2 observed the FMSoS simulation while SA3 was performing changes in the simulation model. Every time a change was performed, a new architectural configuration emerged as a result. SA1 and SA2 observed if the FMSoS maintained its operation despite the change performed. As the dynamic reconfiguration was linked to the MDM mechanism, every time a change was performed, the MDM mechanism was triggered to be aware of the new architectural configuration acquired by such SoS. SA3 monitored the outcomes of such process. Model recovery mechanism addressed the identification of the current operational architectural elements at the SoS architecture. After recovery was performed, model discovery mechanism was performed, querying the simulation about the existing links between architectural elements (constituents and mediators) after the change is completed. As an outcome, a new DEVS simulation model is produced as output, being submitted to the reverse transformation;

Step 2. Architectural evaluation: This step is intended to be performed before architectural reconciliation. However, as the aim of this evaluation was to exercise MDM mechanisms and reverse transformation, this step is suppressed from this evaluation;

Step 3. Reconciliation: As model recovery and discovery was completed, and SA1,

SA2, and SA3 agreed about the correctness of results from Step 1, SA4 performs the reverse transformation. This step materializes the reconciliation between DEVS current model and SosADL prescriptive architectural instance. Currently, an entire new SosADL model is produced as outcome, conducting to step 4;

Step 4. Architectural consistency checking: SA4, SA5, and SA6 perform a manual architectural consistency checking. After the prescriptive model (architectural instance) documented in SosADL is created as outcome of reverse transformation, software architects check whether the new version of the concrete SoS architectural description (propagated to the current architectural instance) still conforms to the original SoS abstract architecture. If yes, the processes is finished until a new change at runtime occurs. If not, a new architectural configuration is sought and reverse transformed until a right one emerges to match the original abstract architecture.

Table 22 – Part of the results collected during the case study.

#	Sensors	Gateways	CS	mc
1	5	1	0	0.88
2	6	1	0	0.80
3	7	1	0	0.74
4	8	1	0	0.69
5	30	2	0	0.22
6	30	5	0	0.21
33	30	10	8	0.17
65	30	10	20	0.12

The procedure described above was performed 65 times, as 65 architectural changes were performed during this investigation. Results are discussed, as follows.

Results. Table 22 shows the results that document changes between some of the architectural configurations (difference of only one sensor)[2]. Each line shows the number of the architectural configuration, the amount of sensors, gateways, and crowdsourcing systems (CS) that form such SoS architecture, and the mutation coefficient (**M2**).

A single constituent model for representing a sensor in a simulation (in DEVS) has around 70 lines of code (LoC). A mediator holds 53 LoC, a gateway has 57, and a crowdsourcing system has 65 LoC. Mediators for crowdsystems have 39 LoC, and gateways for this crowd systems hold 49 LoC. Adding the amount of the code necessary to specify their interoperability links (30, for the first architectural configuration), a simple simulation

[2] Due to space reasons, the complete list of architectural changes, architectural configuration, and mutation coefficient are externally available at <>.

of a the first architectural configuration shown in Table 22 has 579 LoC; while the last architectural configuration, with 30 sensors, 30 mediators, 10 gateways, 20 crowdsourcing systems, 20 crowd mediators and 10 crowd gateways, and 411 LoC for interoperating them reach the amount of 7,241 LoC. On the other hand, considering SosADL, A single constituent model for representing a sensor in a simulation has 69 lines of code (LoC). A mediator holds 49 LoC, a gateway has 53, a crowdsourcing system has 65 LoC, 39 LoC for crowd mediators and 49 LoC for crowd gateways. Adding the amount of the code necessary to specify their interoperability links materializing the architectural description (77 LoC in first configuration, and 376 LoC in last one), the last configuration reaches more than 7,000 LoC. Back-SoS approach was well-succeeded for dealing with this large amount of LoC, with well-succeeded reverse transformations.

```
1  sos MonitoringSos is {
2    architecture MonitoringSosArchitectureRec1() is{
3      //identical code
4      behavior coalition is compose {
5        //identical code
6        mediatorRec1 is Mediator
7        sensorRec1 is Sensor
8      } binding {
9        //identical code
10       unify one { sensorRec1::location::coordinate } to one
           { mediatorRec1::location::coordinate} and
11       unify one { sensorRec1::measurement::measure } to one
           { mediatorRec1::transmit::measure } and
12       unify one { mediatorRec1::transmit::measure} to one {
           Sensor1::measurement::measure} and
13       unify one { sensorRec1::location::coordinate} to one {
             mediatorRec1::location::coordinate }
14     }
15   }
16 }
```

Figure 29 – Excerpt of a FMSoS architecture restored through reverse transformation (part of the code is hidden for the reader convenience).

After the architectural changes were performed, we manually inspected the resulting models. We concluded that for all reverse transformation uses, our model transformation was effective to automatically generate a SosADL model totally adherent to what was expected as a result, i.e., an equivalent form of the current SoS architectural configuration at runtime that is in conformance with the SoS abstract architecture.

Figure 29 shows the resulting code obtained through applying the reverse model transformation to the instance number 2 (#2) shown in Table 22. Over the initial simulation model (#1), a new sensor was added, creating a new architectural configuration. As the dynamic controller is triggered to change the architecture, the Model Discovery Mechanism is called to extract an updated SosADL model version, which is termed `WnsMonitoringSosArchitectureREC1.sosadl`. Figure 29 depicts such SosADL resulting code highlighting the parts that were added as result of the reverse transformation.

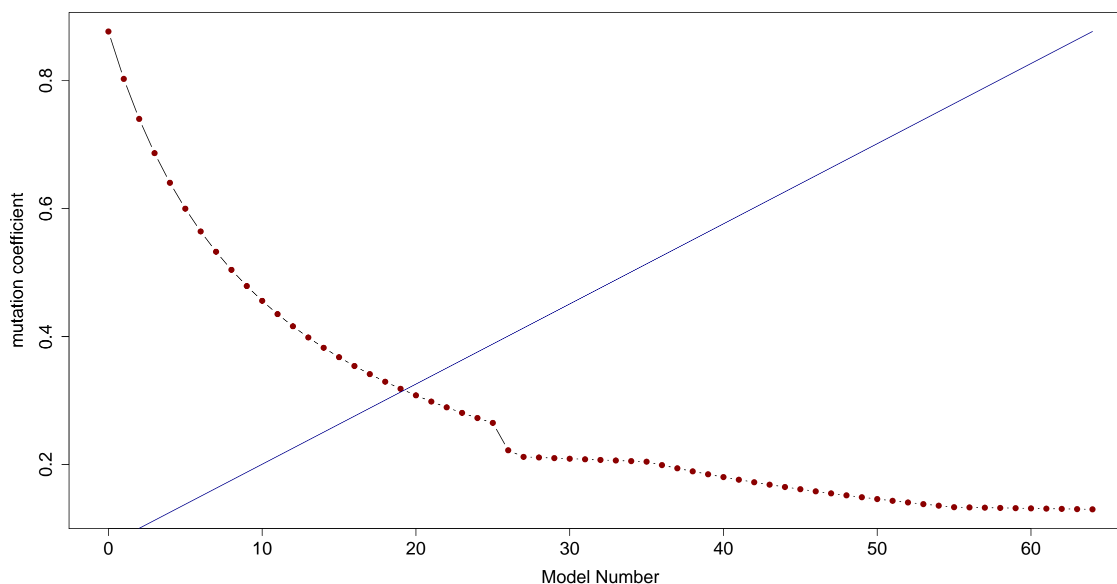

Figure 30 – Relation between number of constituents and mutation coefficient.

Moreover, we also plotted a chart to show the relation between the number of constituents and the differences between the resulting SoS architectural model in regards to the initial version. This chart is shown in Figure 30. X axis shows the identifier of the architectural model, whilst y-axis shows the mutation coefficient for each architectural instance. The lower the mutation coefficient, the greater the change of the model than the current one. The downward curve shows mutation coefficient number decreasing contrasting with line that expresses increasing in the number of constituents. In parallel, Figure 31 depicts a box plot that represents the average of mutation coefficients. Outliers are almost 1 and almost 0, showing configurations one and eight in Table 22. The average is close to 0.2. As this is a value close to 0, this means that the mutation rate was predominantly high, showing consistent differences between the initial architectural configuration (that conformed to the initial version of the prescriptive model) and the resulting architectures derived from the dynamic architecture reconfiguration at runtime.

Hence, we can affirm that M1 (correctness), for this case study, was 100%, despite the high degree of mutation achieved by the 65 different architectural configurations obtained during the study. This means that for all reverse transformations executed over the 65 different architectural models obtained from the MDM execution, correct SosADL models were obtained and their correctness was attested by a manual cross-reviewed inspection conducted by three software architects. Then, we can assume and answer the arisen RQ, affirming that Back-SoS approach is well-succeeded to maintain synchronization between a SoS architectural instance and its corresponding concrete architecture at runtime.

We discuss our findings as follows.

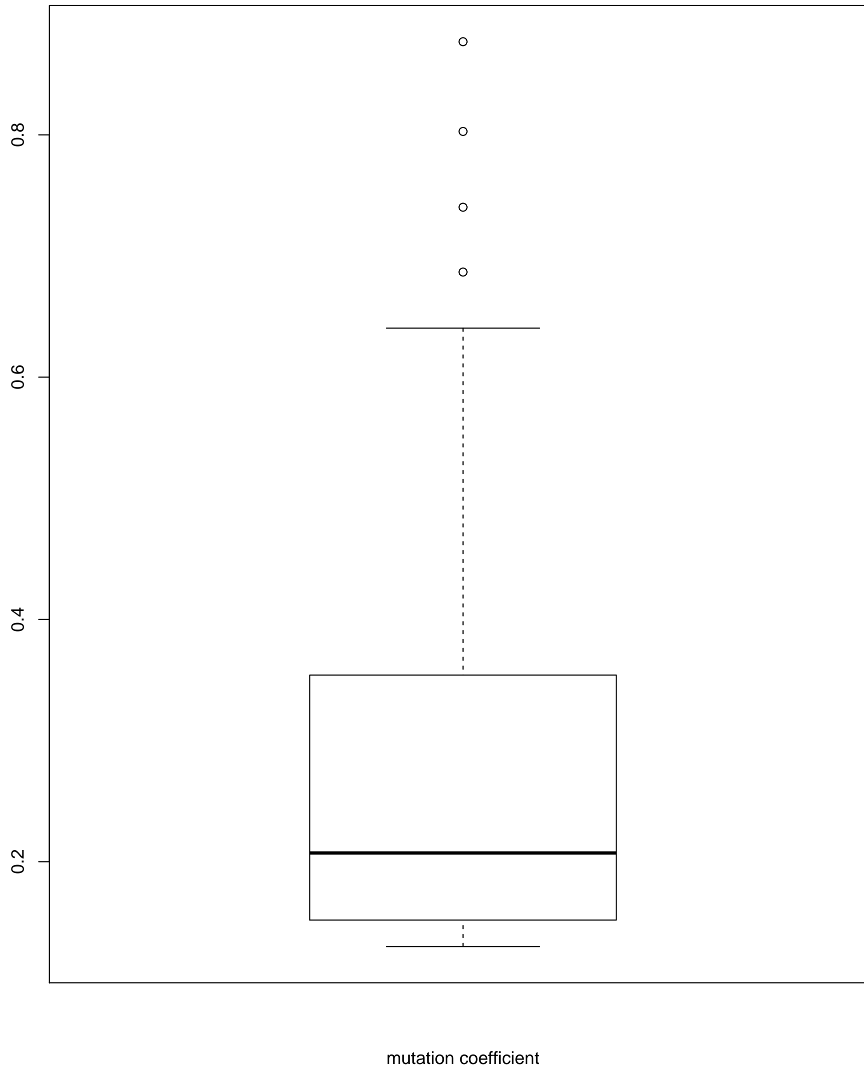

Figure 31 – Average of mutation coefficient samples.

4.3. Discussion

In our investigation, we exercised addition of constituents, as Back-SoS approach is concerned to model recovery, discovery, and reconciliation mechanisms. *A priori*, established mechanisms are not even aware of the operation that was performed to cause an architectural change and trigger their action, as such mechanisms are only concerned to support updating of SoS architectural specification. As a matter of fact, the change performed to the SoS architecture at runtime was not an issue for Back-SoS approach,

as the reverse model transformation only took the current configuration of the runtime architecture and generated, a corresponding SosADL architectural model, as outcome. However, further investigation still must be performed on other operators.

In our approach, Back-SoS is intended to be used when the result of an architectural evaluation over a SoS change shows that the new configuration is potentially beneficial by offering new features, new business exploration opportunities, or even resource savings or better performance, such as faster response time or reduced energy costs to perform an action. Due to that, we have included in our approach a step of architectural evaluation, since we believe that reverse transformations will not be performed deliberately without any criteria. Moreover, not all changes need to be or will be propagated to the architectural specification. Actually, behavior correctness is supposed to be assured using an architectural evaluation approach. In addition, in previous investigations (GRACIANO NETO, 2016; GRACIANO NETO *et al.*, 2016; GRACIANO NETO *et al.*, 2017; GRACIANO NETO *et al.*, 2018b), we show how the architectural evaluation of a SoS can be conducted using simulations as a platform. However, since the SoS architectural evaluation is not the focus of this chapter or the assessment reported herein (apart from space constraints), we have not explicitly included this step when reporting our results.

Indeed, we did not solve all the problems related to architectural erosion in SoS software architectures. We make up important contributions on such topic and, in parallel, we open a novel research branch for software engineering of SoS. Deviations in SoS are very common due to the high dynamism expected of this type of system, especially when considering systems such as smart cities, in which cars and people can enter and leave the city carrying their constituents, and making the architecture highly dynamic. In cases where it will be worth evaluating whether changes are beneficial or maleficent for SoS operation, an in which changes must be propagated to the architectural documentation, we offer insights of solutions, gaps that must be bridged, and mechanisms to model recovery, discovery, and reconciliation via reverse model transformation. Considering this, we can not disregard the advances and contributions brought by our approach and reported in this chapter, despite the numerous challenges and gaps that still remain.

Threats to Validity. Threats to the validity of our results include: (i) scale (external validity and generalization potential), which we considered it was covered, as we extended our results to 65 different architectural configurations, reaching 120 constituents in a same SoS model. Further investigations with more constituents must be carried out; (ii) bias in construction validity: the mutation coefficient was established as a means of comparing the differences between the seed architectural model and those ones resulting from the architectural changes. The selection of such metric could be seen as a threat.

However, we established this metric inspired on data clusters method to group data and see differences between groups. As we treat architectural models as groups of data, this threat is relieved by the solid results that are often acquired from the adoption of such approach; (iii) bias during inspection is also a threat, as we performed manual inspections in all the resulting models of the reverse transformation. However, this threat was relieved by a careful and long inspection, guaranteeing that errors and tiring did not bring any mistake for our analysis. Besides, we also applied Back-SoS to regenerate the simulation model from the architecture recovered in every resulting model. This also relieved this threat, generating functional simulation models in all the cases; (iv) transformation correctness: this is a recurrent problem in model-based software engineering solutions. To relieve this threat, we carefully checked the transformation rules established, and whether the correspondences between models were being preserved. Then, this threat was reduced; finally, (v) architectural changes set: we only performed the addition of constituents in this study. Substitution, removal, and rearrangement were not investigated.

Related Work. In 2014, a Systematic Literature Review (SLR) was carried out to investigate how model-based practices have been applied to engineer SoS (GRACIANO NETO *et al.*, 2014). From that SLR, two studies reported reverse transformation approaches. In one of them, Tu et al. (TU; ZACHAREWICZ; CHEN, 2011) reported an ongoing research and illustrated how transformations can be used only to generate constituents software code (not for SoS software architecture as a whole) and how to apply model-based software engineering (MBSE) to perform reverse engineering over legacy enterprise information systems, with coverage of interoperability issues, forming a class of SoS known as Systems-of-Information Systems (SoIS) (YAHIA *et al.*, 2009). Bruneliere et al. (BRUNELIERE *et al.*, 2010) also report a reverse engineering that is executed by a model discovery mechanism, motivating an automatic update of the source model in execution time for software-intensive SoS. Barbi et al. discuss the generation of configuration and deployment code, and models from code as well, and is the only one study which really performs both kinds of transformations (forward and backward), but only these configuration and deployment aspects are generated (BARBI *et al.*, 2012). However, for all the aforementioned related work, their notion of SoS does not include dynamic architecture, emergent behavior, abstract architectures, and other essential features of software-intensive SoS.

Other approaches, including manual and bidirectional approaches such as graph transformations, could be used to solve the problem of synchronizing SoS architectural specifications (prescriptive architectures) and their correspondent runtime configurations (descriptive architectures). However, manual approaches would suffer from the low, repet-

itive, and error-prone task of manually transforming the current runtime architectural configuration to SoS architectural specification, besides checking conformance between models. In turn, Bidirectional transformations (BXs) provide a mechanism for maintaining consistency between two related models, one referred to as the source and the other as the view (CHENEY *et al.*, 2017). A bidirectional transformation consists of a pair of transformations (a forward and a backward) that satisfy round-trip engineering laws. Bidirectional programming languages have been designed to aid the user in writing bidirectional transformations, with which the programmer only needs to write one program that can be interpreted either as get or put, and the two interpretations are guaranteed to be well-behaved (KO; ZAN; HU, 2016). However, such approach still presents some pitfalls, as it is not totally mature yet, specially considering applying that for large scale context as SoS applications domain.

De Silva and Balasubramaniam (SILVA; BALASUBRAMANIAM, 2012) presented a comprehensive survey on how to control software architecture erosion and techniques for architectural restoration, covering methods for recovering and reconciling an eroded architecture with its intended architecture. They highlight techniques and tools for reconciling software architectures of single systems, such as (i) adaptation of the running software architecture to the first one by semi-automated changes; (ii) refactorings, and pattern-based approaches; and (iii) architecture discovery, which consists of extracting the architectural model from a running system. However, they highlight that their survey did not identified architecture discovery methods that are specifically geared towards addressing architecture erosion. Moreover, no one technique is mentioned for SoS context.

Other options of ADL and simulation formalisms were also analyzed about the suitability to represent SoS and to bridge the gap between descriptive and prescriptive models in SoS. However, current ADL lack mechanisms to capture uncertainty, dynamism, and potentially undesired behaviors that can emerge from SoS architecture configurations (GRACIANO NETO *et al.*, 2014), which can hamper the prediction and guaranteeing of SoS correct operation. Languages often adopted to describe SoS architectures, such as UML[3], SysML[4], and CML[5], lack expressiveness for describing SoS architectures, with drawbacks that can difficult SoS architectural specifications, as they do not support, in particular, partial description of constituents not known at design time. Other initiatives have proposed approaches that use model transformations from an architectural model (SysML, HLA, DoDAF[6], π-ADL) to some simulation formalism (Go language, Simulink[7])

[3] UML, http://www.uml.org/
[4] SysML, http://sysml.org/
[5] CML, http://www.compass-research.eu/approach.html
[6] DoDAF, US Department of Defense Architecture Framework, 2010.
[7] Simulink, www.mathworks.com/products/simulink/

(CAVALCANTE; BATISTA; OQUENDO, 2015; FALKNER *et al.*, 2016; XIA *et al.*, 2013; ZEIGLER *et al.*, 2012). However, these approaches do not support: (i) SoS software architecture specification (CAVALCANTE; BATISTA; OQUENDO, 2015); (ii) dynamic architecture and constituents not known at design time (FALKNER *et al.*, 2016); and (iii) the concept of SoS software architecture with all the necessary details to guarantee precision in representation (XIA *et al.*, 2013; ZEIGLER *et al.*, 2012). SosADL was recently conceived to support specification of SoS architecture descriptions (OQUENDO, 2016c). SosADL is an executable language. However, SosADL models demand an execution mechanism that interpret the operational semantics of such language. The establishment of a mapping between SoS descriptive and prescriptive models enables such mechanism by automatically generating a denotational executable semantics based on DEVS that conforms to the original Pi-Calculus for SoS operational semantics for the subset of SosADL that we worked on.

Contributions of our work. Back-SoS brings the following contributions.

1. **A characterization of architectural drift in SoS**: Architectural deviations, drift, and erosion has been broadly exploited in software engineering literature, including model recovery, discovery, and reconciliation mechanisms. However, for SoS context, this discussion is still scarce. Then, we make a contribution on characterizing such elements in regards to SoS software architectures;

2. **A list of architectural changes that may cause drifts**: in Back-SoS approach, we envisioned the types of changes that can cause deviations in a SoS, creating a first taxonomy for this that can be reused in forthcoming research;

3. **A Model recovery and discovery for SoS**: We also create a model-based mechanism that automatically recovery architectural elements from a SoS runtime model, and creates, as outcome, a model that represents the SoS current configuration. This can also be seen as a contribution, as the mechanism can be replicated and extended for other contexts;

4. **A reconciliation mechanism**: We also report the creation and evaluation of a reconciliation mechanism. We implemented it through the use of a reverse model transformation, that takes the result from model discovery approach, and produces an updated version of the SoS software architecture documented in SosADL; and

5. **Results of an evaluation**: We also report results reported on the conduction of a case study performed using part of the mechanisms prescribed in Back-SoS approach;

6. **A process to deal with architectural drift in SoS software architectures**: well-defined steps communicated in Section 4.1 can also be seen as a contribution, as it is technology agnostic, and could be adapted for any model recovery, discovery, and reconciliation mechanism that can be chosen for SoS context.

Challenges for SoS Architecture Drift. We raise some open issues and challenges that must be still addressed in the next years about SoS architecture drift, erosion, degradation, and reestablishment of consistency, as follows:

1. **Mechanisms to edit only the parts that need to be updated in concrete SoS architectural model:** Currently, architectural reconciliation mechanisms are often concerned to restore, in the prescriptive model, only the parts that have been effectively changed at runtime. This is specially important for systems that are too large. In our approach, we generate, as outcome of the reverse transformation, an entire SosADL model that corresponds to the architectural specification (we do not perform reverse engineering of the constituent codes, but only from the specification of how they interoperate). Further research must be conducted to deal with this issue and enable automatic partial editing of SoS architectural prescriptive models;

2. **Multiple concurrent changes:** In a real SoS, many concurrent changes can occur in the SoS architecture. Further investigation must be conducted to deal with the scheduling of these changes and how this will impact on architectural recovery, discovery, evaluation, and reconciliation.

3. **Transient changes**: Changes can occur due to failures or attacks, and self-healing mechanisms can be established for restoring SoS operational integrity. In such cases, changes are not permanent, and model recovery and discovery should not be necessarily triggered. Such topics must be further exploited, and strategies shall be provided to deal with security, self-healing, and transient changes in SoS software architecture.

4. **Open SoS and Discovery of models for non-predicted constituents**. SoS are inherently open systems. As such, architectural changes could be possible to occur involuntarily, i.e., not necessarily started by a human codifier that adds a new rule not prescribed in the original architectural description, but as a result of the possibility of new constituents joining the SoS. Therefore, it is necessary to define operations to be performed to reestablish the conformity between the concrete model and the abstract SoS model after the concrete model has been restored in relation to the runtime architecture.

5. **Model comparison mechanisms**. In MBSE approaches, model comparison is the procedure of comparing two models to support, between other functionalities, version controlling in a models repository and models maintenance, enabling that only parts that were changed are updated in an original source model. Currently, our approach only conserves parts of SosADL models that did not change in regards to the original version that created the simulation models used as target to apply Back-SoS. Model comparison mechanisms must be established for SoS specification and simulation languages.

6. **Reestablishment of architectural configurations**. Architectural changes can originate a defective SoS architectural configuration. Self-adaptive mechanisms must be established to undo changes, recovering, at runtime, a functional SoS architecture;

7. **Consistency with SoS's initial abstract architecture.** In Section 4.1.5, we discuss the requirements for supporting the reestablishment of consistency between SoS descriptive architecture (simulation model) and SoS architectural instance at prescriptive level, and the conditions that characterize and effective architectural deviation between both levels of abstraction in SoS prescriptive model. In our evaluation, conformance between SoS architectural instance and SoS abstract architecture was manually performed by experts. However, we envision a potential to automate such task. Mechanisms to check conformance between the SoS architectural instance models and the SoS abstract architecture model must be established, assuring architectural consistency, and avoiding SoS architectural erosion, even in abstract level;

8. **Techniques for Hierarchic SoS**. SoS are often hierarchic and have different characteristics to system of components. Systems can also be members of multiple other systems. Hence, discovery mechanisms must be expanded and also address the multiple levels of hierarchy that can be exhibited by a SoS at runtime;

9. **Model discovery mechanisms for an Operational SoS.** Once we have been able to perform a reverse transformation from simulation to SoS architectural specification, the next steps proceed towards doing it for an operational SoS. An correspondent animated SoS linked to the operational SoS (represented similarly to a simulation and visualizable in a screen) could add to monitor architectural changes and restoration of SoS architectural consistency via the same transformation.

4.4. Final Remarks

This chapter presented Back-SoS, a reverse model transformation approach for supporting the checking of the consistency between descriptive and prescriptive SoS architectural models. As some architectural configurations that emerge at runtime can categorically affect and hamper the SoS entire performance (GRACIANO NETO *et al.*, 2018b), it is important to establish strategies to keep beneficial SoS architectural configurations, maintaining both models (descriptive and prescriptive ones) synchronized to avoid problems that come from architectural drifts and erosion. After beneficial architectural configurations are identified, our approach also supports maintaining a synchronization between such configurations and the prescriptive model. This procedure could avoid architectural degradation and problems emerged from it, undoubtedly consisting of a novel investigation area of utmost importance. This work is probably the first one that deals with architectural drifts in SoS software architectures; therefore, there is still a lot of work to be done. Besides the reverse transformation, we also externalized an approach to deal with model discovery in SoS simulations, and a list of challenges for the future. We hope our insights contribute for the forthcoming research and for fostering the development of reliable, useful SoS that have been recurrently found in critical application domains.

CHAPTER 5

STIMULI-SOS: A MODEL-BASED APPROACH FOR AUTOMATIC CREATION OF STIMULI GENERATORS IN SIMULATIONS OF SOFTWARE ARCHITECTURES OF SYSTEMS-OF-SYSTEMS

Simulations can contribute to guarantee SoS trustworthiness. They consist of a recurrent approach in SoS Engineering to anticipate failures early in SoS life cycle. Simulations externalize how the whole SoS behaves at runtime (WACHHOLDER; STARY, 2015; MITTAL; RAINEY, 2015; ZEIGLER *et al.*, 2012; GRACIANO NETO *et al.*, 2014). To be reliable, a simulation must faithfully reproduce the conditions under which a SoS operates. These conditions must involve SoS surrounding environment (such as rain and temperature) and constituents operational conditions (such as battery level and GPS location) (ZEIGLER *et al.*, 2012; VANGHELUWE, 2008). A manual approach can fail to reproduce the real frequency of such stimuli, since an expert would have to simultaneously inform inputs for all constituents at runtime until the end of the simulation. Moreover, a manual approach to generate inputs for such simulation can be costly. For example, to reproduce SoS dynamics, for each unit of time, each constituent in the simulation must be fed. A stimulus is often delivered to a constituent system through a user interface interaction. For each stimulus, one user interaction is needed. Considering a SoS formed by six constituents, if each one of them requires one stimulus by unit of time, after only 100 units of time, 600 interactions (such as clicks) need to be performed. Thus, the effort

needed to feed a simulation with a greater number of constituents or for a longer period of time is extremely high, making this approach unfeasible to simulate real SoS.

In this scenario, stimuli generators can support SoS simulation. They consist of a virtual simulation entity responsible for playing the role of the environment, delivering input to a SoS (SANCHEZ-MONTANES; KONIG; VERSCHURE, 2002). However, manually coding of such stimuli generator is equivalently not feasible. Stimuli generators are domain-dependent and totally adherent to the environment modeling, which is itself challenging for SoS development (RAJKUMAR *et al.*, 2010; SELIC, 2012; HEHENBERGER *et al.*, 2016). For example, if we want to simulate a reactive system (such as a temperature sensor), it is important to predict a subset of stimuli that it can receive in order to establish how it will react to them. This entity should encompass details such as the scale in which it will work (celsius, fahrenheit, kelvin, or another scale), a range of acceptable values (from -50 to 60 degrees celsius, for example), the description of the data as a data structure (with value and type), instances that could be received, and frequency in which it must be delivered. Additionally, its development is costly, as it requires writing additional simulation code, often in a lower abstraction level, such as state machines, ports, inputs and outputs details. Aiming to reduce costs associated to the engineering of a stimuli generator, we can explore the possibility of automating its creation, hence supporting: (i) the prediction of the surrounding environment dynamics; and (ii) an anticipation of possible events and natural phenomena that could hamper SoS correct operation.

In context, it is noteworthy to pose the following research question: *How is it possible to automatically obtain a functional stimuli generator that reproduces environmental conditions to the simulation of a SoS*? To answer this question, in this chapter we present a model-based derivation approach for automatically producing stimuli generators to feed a SoS simulation at runtime. In this approach, architectural descriptions play the role of input model as they inherently store information about expected inputs and outputs of the SoS, supporting environmental modeling. We evaluate our approach with regard to its correctness/reliability in automatically producing stimuli generators for the simulation of a real SoS that monitors flash floods risk in a river that crosses urban areas. Results of this study reveal that our approach is reliable and capable of deriving stimuli generators that conforms with the expected inputs that must be received by simulated constituents, and that effectively triggers the SoS simulation.

This chapter presents *Stimuli-SoS*, a model-based approach to support the creation of stimuli generators to be used in the simulation of SoS. Stimuli-SoS takes advantage of software architecture descriptions for automating the creation of such generators. Specifically, this approach transforms SosADL into dynamic models expressed in DEVS.

We carried out a case study in which *Stimuli-SoS* was used to automatically produce stimuli generators for a simulation of a Flood Monitoring SoS.

5.1. Presentation of Stimuli-SoS

Stimuli-SoS is a model-based approach established to automatically derive stimuli generators for SoS simulations. For each distinct type of constituent in a SoS, a dedicated stimuli generator is created. Architectural models often bring some sort of environment description (ISO, 2011). To establish the basis of Stimuli-SoS, we decided to use a SoS architectural model to derive stimuli generators for a SoS simulation via a model transformation.

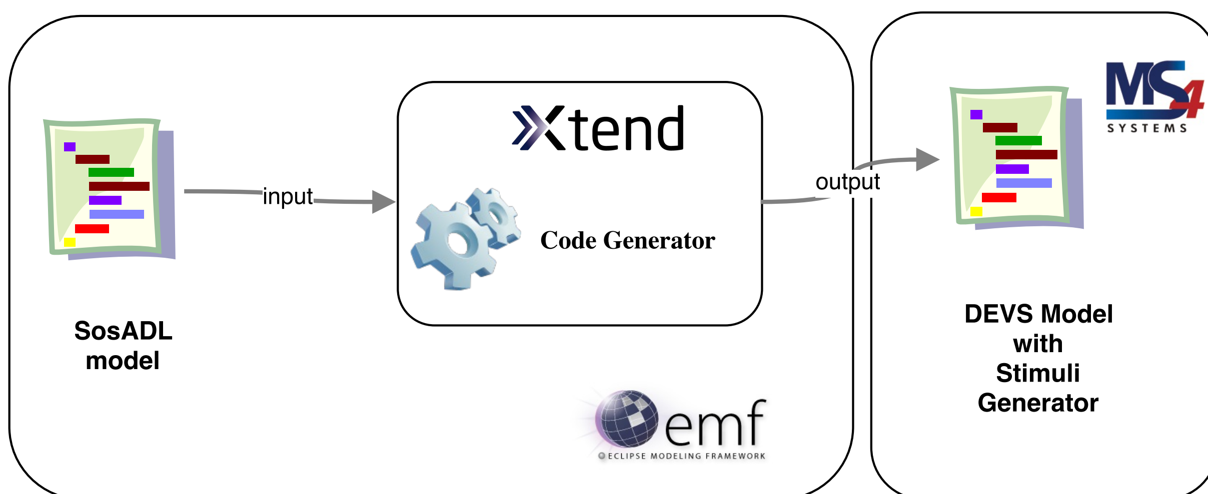

Figure 32 – SoSADL2DEVS transformation (GRACIANO NETO, 2017).

5.1.1. A Systematic Approach to Derive Stimuli Generators

We established Stimuli-SoS as a systematic approach based on well-defined activities. The systematic approach involves a reference workflow to derive stimuli generators. Figure 33 shows the proposed workflow which is represented through an UML activity diagram using SPEM[1] stereotypes. Each activity is developed by a SoS architect. The result of the execution is the generation of work products. Our approach consists of the following activities:

1. **Specification of SoS software architecture**: In the first activity, an architectural description of the SoS software architecture is specified using SoSADL. The work

[1] SPEM - Software & Systems Process Engineering Metamodel: http://www.omg.org/spec/SPEM/

products delivered are used as input for the next activity. Environment modeling is a sub-activity performed in this step;

2. **Automatic derivation of stimuli generators**: This activity comprises running the model transformation, receiving SoSADL work products as inputs, and delivering DEVS files as outputs, including the stimuli generators;

3. **Inclusion of stimuli generators in the target simulation model**: After delivering the DEVS files, they must be included in a project that is deployed in MS4ME tool to support the launching and execution of the simulation; and

4. **Execution and monitoring of simulation**: This activity uses the stimuli generator and collects data from the simulation to observe emergent behaviors, to perform statistical analysis and to collect evidence for validation and verification of properties of the SoS software architecture.

5.1.2. Model Transformation

All SoSADL elements must be traduced to DEVS to create a functional simulation. In SoSADL, there is a special type of connection called **environment**, which abstracts interaction of a SoS with the surrounding environment, emitting outputs to the environment, or receiving stimuli from it, e.g., when the system is a sensor. However, there are no straightforward elements in DEVS to automatically produce environment stimuli. We decided to harmonize both formalisms (SoSADL and DEVS) through a model transformation. Such transformation enables the creation of stimuli generators that deliver the expected inputs the constituents wait to perform transitions and to start their execution.

Listing 9 shows an excerpt of a code in SoSADL that depicts part of the specification of one constituent: in this case, a sensor. Some parts are hidden because they do not influence in the derivation of stimuli generation. It is possible to see, for example, 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`, which is used to receive the level of battery available. A connection in SoSADL has a name and a data type that can be transferred through that communication channel. Then, when a connection is specified with the environment modifier, it actually models what is expected to be received from the environment and the data type expected. Each type of constituent requires a different stimuli generator. Then, such architectural model is used as an input for a model transformation that collects the set of environment connections, extracting the data type, and creating one respective output state transition for each one

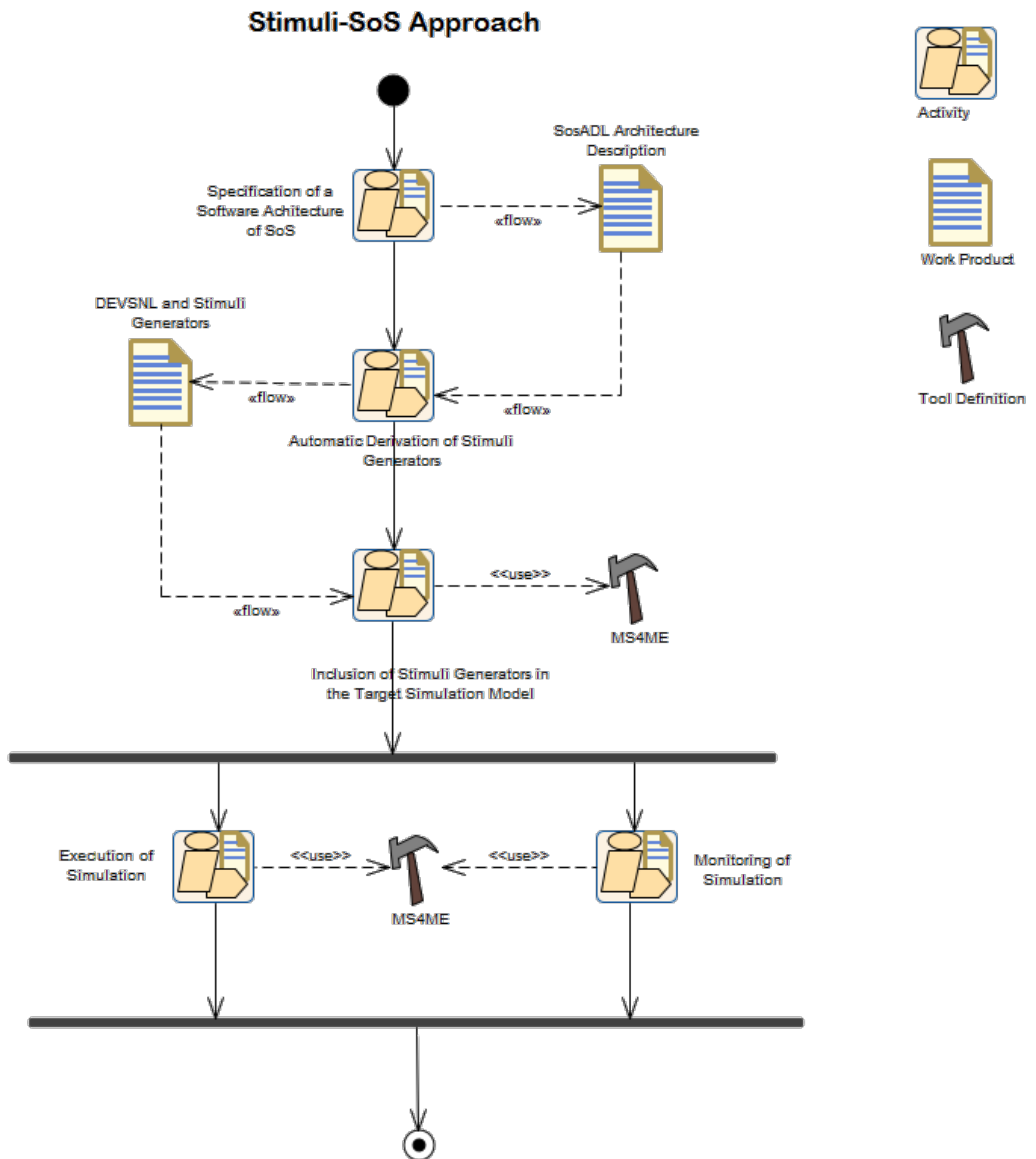

Figure 33 – Stimuli-SoS workflow.

of them. These state transitions are assembled in sequence to form an entire state diagram that will drive the stimuli generator operation. Then, each state transition will deliver one of the expected data to the correspondent constituent whose architectural specification model was used to create that stimuli generator. Each stimuli generator is associated to a data flow that receives data from a textual file. That file holds the data that feeds the

constituent. Then, these data are read from the text file and sent to constituent, triggering the simulation to run. This happens in a periodic and constant rhythm so as to keep the simulation running.

```
1 //'with' imports declarations suppressed
2 // Description of Sensor as a System Abstraction
3 library WsnSensor is {
4   system Sensor( lps:Coordinate ) is {
5   // Declaration of local types hidden
6   gate measurement is {
7       connection pass is in { MeasureData }
8       environment connection sense is out { MeasureData }
9   }
10
11     gate energy is {
12       environment connection threshold is in { Energy }
13       environment connection power is in { Energy }
14     }
15    }
16 }
```

Source code 9 – A specification of a sensor in SoSADL.

The following steps are followed by the transformation chain that produces stimuli generators:

1. All connections of all the constituents are mapped into a specification format and saved in a text file;

2. Connections are read from the text file and analyzed. Such connections are parsed from the architectural description of the SoS to be in the following format: `measurement::sense;RawData-true`. This first part is the name of the gate in which the connection has been specified. The second part is the name of the connection. The third part represents the data type that can be transferred across that communication channel. The last part of each connections descriptions is a boolean: it has a `true` value if the connection is of the type `environment` and `false` if it is not. The transformation algorithm searches for environment connections. Each connection specified as an `environment` connection produces one transition in the specification of the state diagram in the resulting stimuli generator. Hence, the stimuli 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.

Listings available externally[2] show and bring explanations about the Xtend code that materializes the model transformation. We evaluate our approach as follows.

[2] https://goo.gl/vPbKcL

5.2. Evaluation

To investigate the reliability of Stimuli-SoS approach, we performed a case study using a Flood Monitoring and Emergency Response SoS.

5.2.1. Scenario Description

We specified a FMSoS architecture with 42 sensors, 9 crowdsourcing systems, and 18 drones, following the model shown in Figure 34. Each drone has its own base (18 drone bases), and transmits the information collected through its own 3G gateway (a gateway that will be in the vicinity). 18 gateways are spread along the river boards. Mediators were produced as much as necessary to mediate these constituents, and 20 gateways were also used to receive these transmissions.

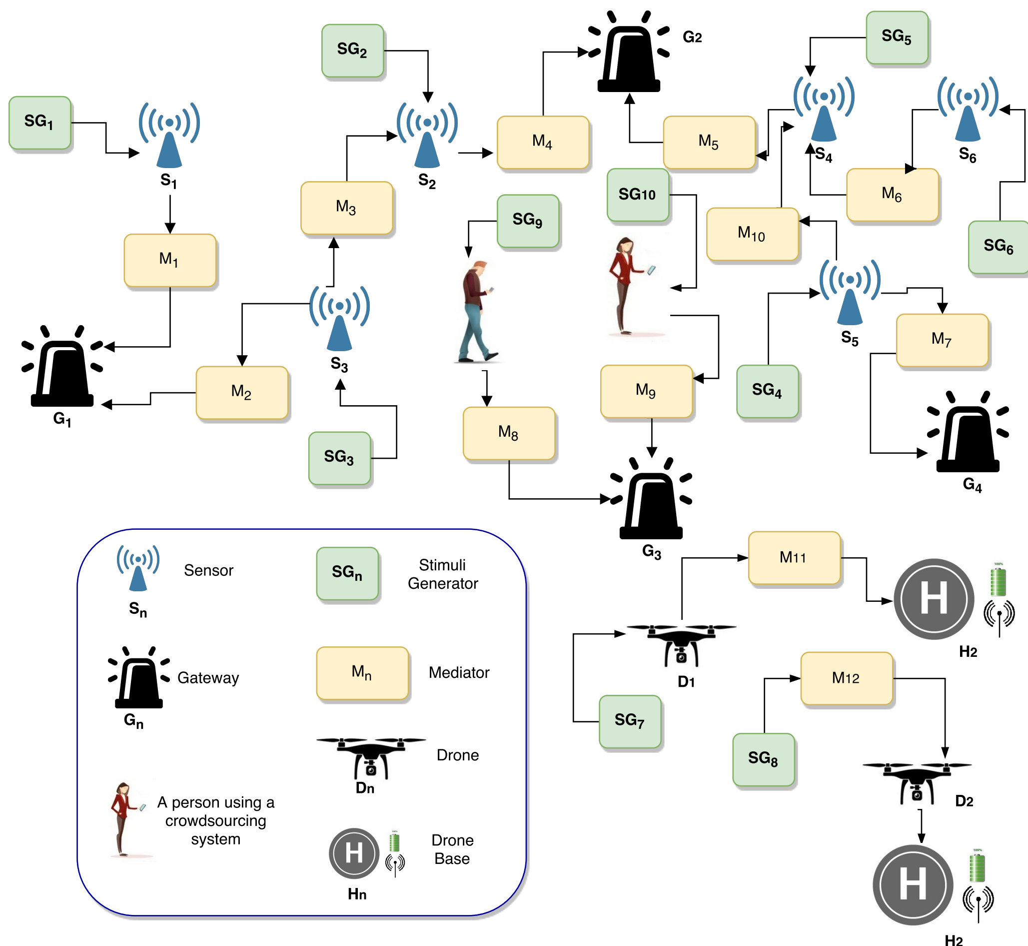

Figure 34 – A flood monitoring system-of-systems (FMSoS) Architecture.

FMSoS monitors occurrences of floods in an urban area. Rivers cross the city and, when the rains are intense, floods frequently occur, causing losses, damage, and imminent danger for the population. FMSoS is composed by five different types of constituents:

1. **smart sensors**, which are fixed embedded systems monitoring flood occurrences in urban areas, located on river edges;

2. **gateways**, which gather data from constituents and share them with other systems;

3. **crowdsourcing systems**, which are mobile applications used by citizens for real-time communication of water level rising;

4. **drones**, which are UAVs also concerned to complement sensors observations by monitoring the river water level while they fly over it, sending pictures if some change in the water level occurs; and

5. **drone bases**, which are fixed basis from where drones departure, and for where they come back to recharge battery, and transmit their data.

Our FMSoS is concerned with one specific mission: *emitting flood alerts to public authorities that can draw strategies to protect the population*. It consists of a collaborative SoS, with no a central authority that orchestrates the constituents functionalities to accomplish missions. Data are gathered in gateways, analyzed according to flood risk, and a status (alert or no alert) is transmitted to public authorities. Figure 35 gives an illustration of FMSoS deployed in a river[3]. 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. Data collected by sensors are transmitted until reaching the gateway. Besides, drones fly on the river and return to their bases to recharge and eventually communicate with gateways to alert about a flood threat. In parallel, people that walk close to the river can also contribute by communicating that water level is increasing if they perceive this happening. In case of flood, gateways emit alarms for public authorities. Authorities cross data coming from all the constituents to draw a conclusion of an imminent flood, taking decisions to protect population.

FMSoS exhibits the following characteristics (MAIER, 1998; OQUENDO, 2016a)[4]:

- **Operational independence of the constituents**: Each constituent (sensor, crowdsourcing system, or drone) operates in a way that is independent of other constituents, as they belong to different city councils and have different missions in the region of São Carlos;

[3] Credits for the images used to compose the figure: http://goo.gl/TTOlAa, http://goo.gl/QCUAKY, http://goo.gl/a9Y0Dw, https://goo.gl/rFkYJ6, https://goo.gl/8YojYj, https://goo.gl/XyWEZw, https://goo.gl/VpftdV, https://goo.gl/dfMPLl.

[4] Moreover, our example scenario also covers constituents heterogeneity, autonomy, and SoS scale, characteristics that are commonly assigned to SoS, as well (JAMSHIDI, 2009).

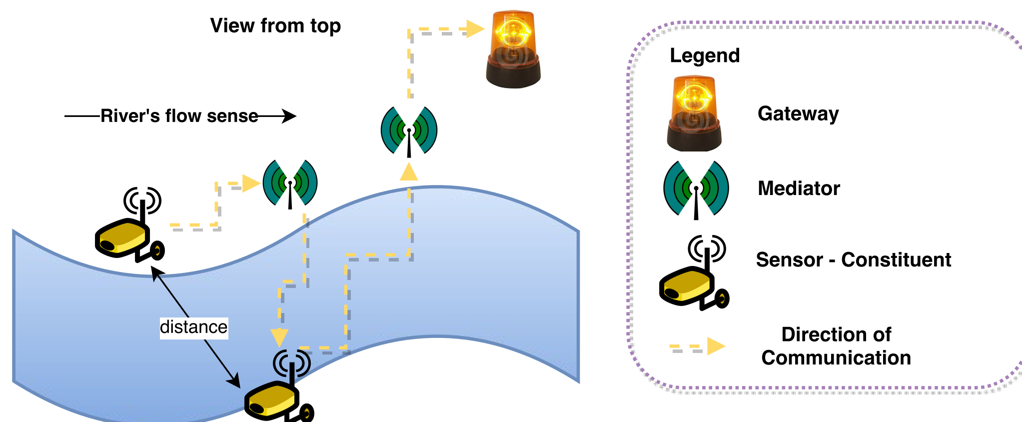

Figure 35 – An illustration of part of a FMSoS.

- **Managerial independence of constituents**: A diversity of stakeholders and enterprises might independently own, deliver, and manage different constituents that compose FMSoS. Moreover, each constituent has its own management strategy for transmission vs. energy consumption and will act under the authority of the different city councils;

- **Distribution**: All the constituents interoperate through a communication network;

- **Evolutionary Development**: SoS evolves as a consequence of changes in the configuration or functionality of constituents; and

- **Emergent behavior**: One unique constituent could not deliver a flood alert by itself. For instance, if only one sensor, or crowdsourcing system or drone performs its activities in an urban area, it could not notify a flood on time, being not effective. It might emit a false alert, since the flood could be limited to another place. Hence, the flood alert is result of the interoperability among a diversity of constituents working in cooperation, spread along the riverbank.

For each one of the constituent types, a specific type of stimuli generator was automatically produced using our model transformation approach. For each constituent type, connections were specified in SoSADL with the `environment` modifier to support the automatic derivation of stimuli generators. Mediators do not need a stimuli generator as they receive stimuli from other constituents and they do not have environment connections. We discuss the rationale behind each one of the environment connections for each type of constituent, as follows:

- **Smart sensors**: battery (power level), coordinate (GPS location), water level, and power threshold.

Rationale: they receive battery level and power threshold, and a coordinate to start their work. After that, the stimuli generator will deliver water level to them, imitating the data obtained by sensors to verify if the SoS will detect possible floods;

- **Gateways**: battery (power level), coordinate (GPS location), and power threshold.
  Rationale: The gateway (materialized by an industrial computer linked to the internet) provides the base station for collecting these measures and processing them, possibly warning the risk of imminent flood (OQUENDO, 2016a). Data are transmitted from other constituents and gathered in gateways. Hence, only power level, coordinate, and power threshold are necessary.

- **crowdsourcing systems**: battery (power level), coordinate (GPS location), visual perception, and power threshold.
  Rationale: crowdsourcing systems are apps that enable population to communicate a possible flood threat by interacting with mobile. It is possible to communicate the risk level and to send pictures to show the situation. These systems do not interact with environment, but with humans. Hence, operational aspects are documented as environment issues (power level, coordinate, and power threshold), and a specified behavior enables citizen to send information according to a pre-defined danger scale and pictures that endorse their perception (ALBUQUERQUE *et al.*, 2017). However, this perception also represents the environment. Hence, we defined in SoSADL that the danger level is a pre-defined value (between 1 and 6, 1 being no risk, and 6 being flood effectively occurring) that can be classified by the human user according to what he/she sees. Figure 36 shows a real picture of a human dummy painted in river wall in front of USP. People use it as a reference to classify the flood risks according to the aforementioned levels. In turn, Figure 37 shows how the numbers and the co-related water level appear in the mobile app so that a person can classify the risk. Looking at the painting available in Figure 36, it can classify the risk according to the scale available in Figure 37, and send to gateways. Then, an environment connection called `perception` was defined in SoSADL specification, enabling that these pre-defined data can be sent according to what the user selects. Then, it is still possible to automatically create a stimuli generator that delivers these data;

- **Drones**: battery (power level), coordinate (GPS location), water depth, and power threshold, image, and distance flown.
  Rationale: Most professional radio control systems reach 2km of radius extension. A drone has an average autonomy range of 10 minutes. After that, it is required to come back and recharge its battery. Its average speed is 16 meters per second. Hence, it can fly 5 minutes to go, and return in the next 5 minutes to recharge. Then, he can

fly a route of 2400 meters one way, and 2400 meters back. As the Monjolinho River, where we are applying the case study, has an extension of 43 kilometers, it will take 18 drones to individually cover 2.4 km each. We will call the drone connection as `water depth`, since it measures the height of water differently from the sensor. In addition, a connection called `image` will be responsible for taking a photo of the place that has altered water height and send via 3G to responsible authorities, initiating the alert. Photos are taken only when the water depth exceeds a given threshold. For measurement purposes, the `flown distance` will also be delivered constantly to the drone, within the limit of 2.4 kilometers. The `GPS position` is also delivered constantly by the stimulus generator, changing its values over time, to simulate the autonomous movement.

- **Drone basis**: battery (power level), coordinate (GPS location), and power threshold. **Rationale**: This is the radio control basis, for where drones come back to recharge battery. Only its own battery level, coordinate, and power threshold are necessary to model its environment of interest.

5.2.2. Case Study Protocol

The case study was conducted according to the following steps (RUNESON; HöST, 2009): (i) Case study design (Preparation and planning for data collection); (ii) Execution (Collection of evidence); (iii) Analysis of collected data, and (iv) Reporting.

Scenario: Our case study consists of a Flood Monitoring SoS (FMSoS) concerned to monitor a river that crosses an urban area, aiming to detect potential flash floods, i.e., floods that can occur quickly with huge damage and risk for population. It consists of the description of part of a SoS already in operation in São Carlos, Brazil, monitoring the Monjolinho river that crosses the urban area and that causes recurrent flash floods, causing damage and losses.The goal of this case study is:

Goal: to evaluate with regard to its correctness if stimuli generators automatically produced are able to trigger and feed a simulation until the end of its execution.
Rationale. Our approach was designed to support simulations of SoS software architectures by automatically producing stimuli generators. As such, we claim that to be reliable, a simulation must reproduce the conditions under which a SoS operates, considering both its surrounding environment (such as rain and temperature) and constituents operational conditions (such as battery level and GPS location). Then, our evaluation is based on the success of our approach to support automatic production of stimuli generators that can (i)

Figure 36 – A real picture of a human dummy used to classify floods risk.

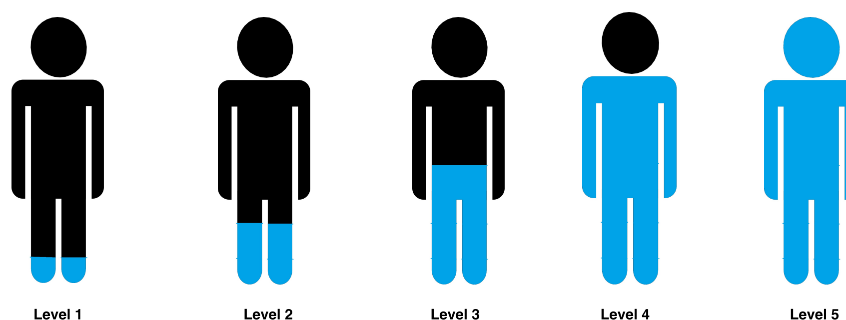

Figure 37 – Water level with a human dummy.

reproduce the surrounding environment and constituents operational conditions, and (ii) maintain the simulation running until the end of data input. Considering that we use a

software architecture description as the basis to produce three different types of stimuli generators. If the software architecture is faithfully described and the generation method is correct, the stimuli generators created will address our claims. Then, we established the following research question: *How is it possible to automatically obtain a functional stimuli generator that reproduces environmental conditions to the simulation of a SoS*? To answer this question, we established a model-based approach that produces such stimuli generators from a SoS software architecture description, and established the following goal to the case study (that matches our first research question). From this general goal, we derived the following research questions with their respective metrics:

RQ1. Are the (automatically created) stimuli generators functional?
Rationale. This question establishes whether or not the stimuli generators automatically generated are functional, that is, if they can work into the context of a simulation after deployed, exactly how they were created, without any manual intervention or modification.
Metric - Success fee: percentage of data correctly delivered to the correspondent constituent, considering the amount of that data that is intended to be delivered.

RQ2. Is the stimuli generator capable of triggering a simulation correctly?
Rationale. The simulation only starts when the correct stimuli are received by the constituents and they start their operation, making the entire SoS operate. This research question evaluates if the simulation starts correctly.
Metric - Efficiency: A participant observes if the simulation is successfully triggered by the stimuli received during its entire execution cycle.

RQ3. Is the stimuli generator capable of supporting an entire simulation execution correctly?
Rationale. The aim of a stimuli generator is supporting a simulation with a continuous emission of stimuli that keep the simulation running.
Metric - Number of problems during simulation execution: given by the proportion of errors during simulations compared to the total execution of the simulation.

Research Instruments

We used a FMSoS to collect all data used in the simulation. We adopted Eclipse Modeling Framework (EMF) as the platform to develop SoSADL models based on Xtext. Xtend is the transformation language, MS4ME[5] is the simulation platform, and DEVS (in particular, a DEVS dialect called DEVS) is the formalism for running the generated

[5] http://www.ms4systems.com/pages/ms4me.php

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

simulation models.

Data preparation

We obtained data collected by the sensors that are under supervision of a Brazilian entity responsible for monitoring natural disasters (Brazilian Center for Monitoring and Warnings of Natural Disasters - CEMADEN)(HORITA *et al.*, 2015). These data were parsed, and stored in a text file. Stimuli generator are fed with them, emitting them to the simulation, stimulating it until the end of the execution. The input of data triggers the constituents operation, cause their interoperability, reach the gateway, are processed creating new data that correspond to positive or negative flood alerts. We also collect these data to analyze results.

We chose a large sample of data collected by the real FMSoS from November 23th 2015 to December 31th 2015. This interval was important because during these months a number of floods occurred. This enabled us to establish whether or not our simulation results in a diversity of situations. We sent 1000 samples for each sensor, being sent every 5 minutes, and 1000 for crowdsourcing systems. Considering that we only had data to feed sensors in a simulation, we adapted them so to have similar data for stimuli generators for crowdsourcing systems and drones. For crowdsourcing systems, the aforementioned scale was used to classify risk between 0 and 6. So we could simulate how people would react and behave due to the changes in water level registered before by sensors. Then, we created a dataset correspondent to the data used to feed sensors. This dataset is available externally[6]. For drones, we used 5000 drone data, since the drone receives every 500 meters a measurement and 2500 meters flown every 5 minutes, totalizing this amount for the entire days that we consider in our sample.

Analysis procedures of collected data.

Stimuli-SoS approach is concerned to the automatic production of stimuli generators. Hence, we need to evaluate if the stimuli generators automatically produced (i) conform to an expected structure of a DEVS model that send stimuli to a simulation, and (ii) are functional, correctly delivering data to the respective constituents that wait for their stimuli. Thus, a quantitative analysis can be adopted to (i) measure the correctness and similarity of stimuli generators to the expected form of a functional DEVS atomic model that deliver data, (ii) evaluate if the stimuli generator keep its operation, delivering data along the entire simulation cycle, and (iii) evaluate if the simulation is correctly triggered and maintained in operation until the end of the input procedure. Hence, we adopted a

[6] http://www.inf.ufg.br/ valdemarneto/journalMaterials/stimuli-sos.html

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

quantitative analysis in our case study (RUNESON; HöST, 2009). We follow a systematic approach divided in well-established steps, reporting the collection and measurement of pre-defined expected data, observing and measuring the scenario via simulation according to pre-defined metrics, and drawing conclusions from these results to answer the research questions established.

Reporting

We report our results based on the steps systematically followed to achieve the derivation of the stimuli generators for FMSoS constituents. A video shows a summary of the entire procedure[7].

1) SoS Software Architecture Specification

The specification of the software architectural description of the FMSoS was conceived as a joint work between SofTware ARchitecture Team (START/ICMC) at University of São Paulo and ArchWare (IRISA) at University of South Britanny, in France. Such specification was conducted by a team of four people using SoSADL language. This step was accomplished after two months of work, and received four iterations to perform refinements on the SoSADL syntax, to cover some gaps that were not identified before, and to refine the software architectural description itself until reaching an acceptable format. We specified an FMSoS architecture with 42 sensors, 9 crowdsourcing systems, and 18 drones, as described in Figure 34. Such specification was validated by a peer-review procedure composed by the SoSADL creator and other SoS experts. The complete SoSADL architecture specification is available externally[8].

2) Automatic Derivation of Stimuli Generators

After the accomplishment of the first step, the automatic derivation step was conducted. The software architectural description produced in step 1 was used as input for this step, being processed by the model transformation script, delivering a stimuli generator for sensors that compose the FMSoS. At this step, a distinct stimuli generator is produced for each distinct type of constituent. In FMSoS case, three types of stimuli generator are conceived: one stimuli generator for sensors, another one for crowdsourcing systems, and another for drone system. The transformation runs and delivers the code in two seconds. The products of this activity (the stimuli generators themselves) were evaluated using the metrics defined in RQ1 (similarity and correctness).

[7] https://vimeo.com/220144774
[8] https://goo.gl/xk5h3z

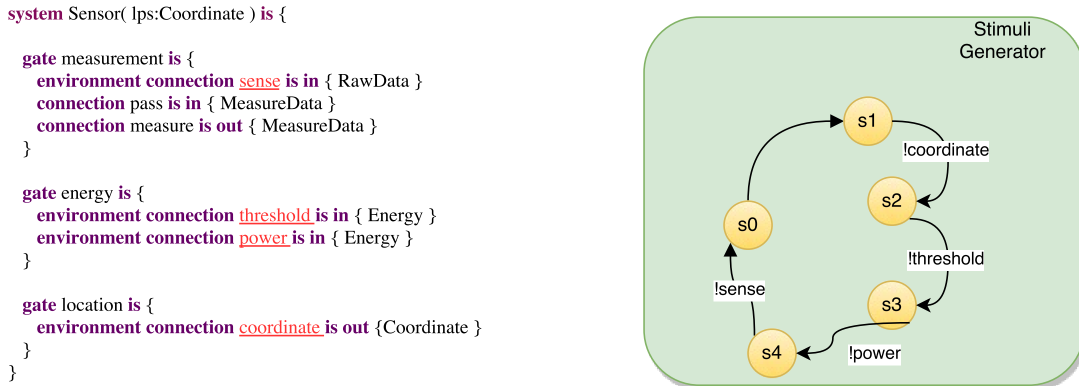

Figure 38 – Illustration of how an automaton is derived from SoSADL system specification to create a functional stimuli generator.

Figure 38 illustrates how an automaton is derived from SoSADL system specification to create a functional stimuli generator. 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 atomic models for each one of the constituent types[9]. In turn, derivation of the stimuli 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, as shown in the code available in Figure 38. Some parts are hidden since they do not influence in the discussion of stimuli 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. Connection `sense` is that one responsible to receive raw data, i.e., the water level from that is being measured from the river by sensor actuators. Lastly, connection `coordinate` receives GPS coordinate from the sensor GPS.

SoSADL models are analyzed by the transformation algorithm, searching for environment connections. Each connection specified as an `environment` connection (underlined in Figure 38) produces one transition in the specification of the state diagram in the resulting stimuli generator. Hence, the stimuli generator consists of a special type of model that has a continuous behavior (a behavior materialized as a loop) to emit stimuli by output state transitions, starting and maintaining the SoS operation. Figure 38 depicts a state diagram equivalent that is created with state transitions created to deliver each of one of the connection data types underlined. It delivers the aforementioned data, and comes

[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.

back to the state s0, forming a loop that continuously offer stimuli for SoS simulation. Order is not important, as constituents are only triggered when the data received matches the input data expected in the state transition in which its operation is at that moment.

```
1 generates output on coordinate!
2 generates output on threshold!
3 generates output on power!
4 generates output on sense!
5
6 to start hold in s1 for time 1!
7 from s0 go to s1!
8 after s1 output coordinate!
9 from s1 go to s2!
10 hold in s2 for time 1!
11 after s2 output threshold!
12 from s2 go to s3!
13 hold in s3 for time 1!
14 after s3 output power!
15 from s3 go to s4!
16 after s4 output sense!
17 from s4 go to s0!
```

Source code 10 – DEVS code for a stimuli generator.

Listing 15 shows the code in DEVS that specifies part of the stimuli generator produced using our approach. The stimuli generator is created not only with the automaton that guides its operation, but also with specification of ports, data types, and all the apparatus necessary to make it executable and to enable the execution of the target simulation (some parts are hidden for the reader convenience). In Listing 15, the stimuli generator has four output ports (Lines 1 to 4) that delivers the collection of the geographic positions (coordinate), power threshold, power level (battery energy) and the water level sensed by sensors.

3) Inclusion of Stimuli Generators in the Target Simulation Model

After the automatic derivation, the stimuli generator must be deployed in the simulation code specified in DEVS and already deployed in MS4ME environment. This step consists of moving the stimuli generator file to the simulation project in MS4ME environment. MS4ME environment automatically generates a Java file that corresponds to the execution entity of each stimuli generator. The SoS architectural description in DEVS is also adapted to include stimuli generators, and to specify that they must emit data to their correspondent constituents, that is, those that hold environment connections that were used as input to produce the respective stimuli generators. Figure 34 illustrates an example of FMSoS architecture during simulation. Mediators enable transmission of data received by sensors from stimuli generators until the nearest gateway. This activity was evaluated by checking if, after deployed, the simulation become executable.

4) Simulation Execution and Monitoring

The simulation took for six hours and twenty minutes (6.20h) in Processor Intel core i5-3230M 2.60GHz (x64), with 4 GB of RAM Memory, HD of 1TB, and running Ubuntu 16.04 64 bits. The data corresponds to 38 days of monitoring data from the Monjolinho River. Data were stored in text files and delivered by the stimuli generators along the FM-SoS, feeding the simulation. This step was evaluated according to the metrics established in research questions two and three (efficiency and number of problems during simulation execution).

We established *a four-window strategy* implemented at the gateways that receive data from constituents to confirm floods. For each four data that subsequently arrives, the gateway checks it by pairs (three possible combination of pairs of four data that arrives). If at least one pair that arrived have both their depth levels equals to or major than 100 cm (the threshold established for that city), a flood alarm is triggered. Experts remarked that one sensor could trigger a false alarm due to the possibility of sediment accumulation, which can increase the measured collected in a location, but that does not represent a flood. Hence, taking pairs was considered a valid strategy. Table 23 illustrates an example. It corresponds to real data that arrived sequentially at the gateway. Each four data that arrive are chronologically ordered, and pairs of data given by (S2,S3), (S1,S3), and (S3,S4) are analyzed. If at least one of the pairs has two measures equal or greater than 100 cm, a flood is confirmed. We did allow the sum of four measures that generate false alarms (for example, S1=90cm, S2=90cm, S3=90cm, S4=130cm). This can represent an increasing in the level of water, but not a flood. Subsequent measures will confirm if it is an actual flood or not. After all the data were analyzed, we compared our results to the original results to evaluate the confidence of the automatically generated simulation.

However, it is possible to remark in Table 23, data do not arrive in order. Hence, if a flood occurs, S1 will be the first to increase its measure, followed by S2, S3, and S4 respectively. Thus, a false negative can occur due to the delay to arrive at the gateway and possible losses of data. One possible situation occurs when only one transmit data with more than 100 cm to the gateway, because it was the last one in that sequence of four measures, but the other measures, even if not 100 cm yet, can have already slightly increased, indicating a possible a flood coming. To avoid this, a new test was done: we added both measures of the other combinations that were not checked in the first cycle to avoid false diagnostics. For example, considering Table 23, we obtain the values of depth from (S2+S3), (S2+S4), and (S1+S4).

After the simulation terminated we analyzed the perceptions from the observation and answered the research questions, as follows. Figure 39 shows the biggest averages of

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

Table 23 – A sample of data sent by sensors.

sensor	timestamp	depth(cm)	alert
S2	2015-11-23 01:58	58	**NO**
S1	2015-11-23 02:03	56	
S3	2015-11-23 02:03	54	
S4	2015-11-23 02:03	57	

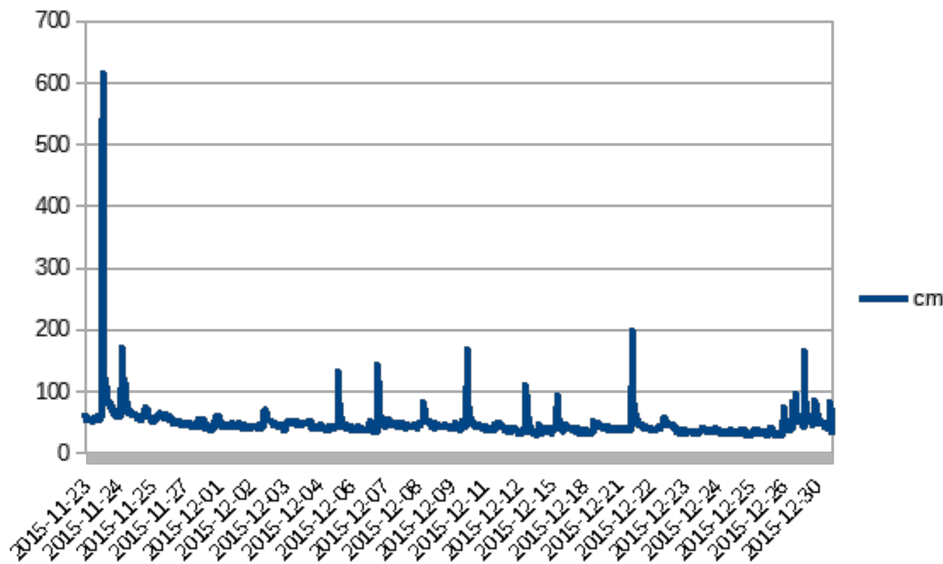

Figure 39 – Monitoring depth of water in simulation during data processing.

depth of water reached in each of the days analyzed. Considering that the four-window strategy exhibits a threat of flood, days November 23th, and December 21st and 30th are the most relevant. Other moments exhibit values bigger than 100 cm, but only as a momentary occurrence. The graphic enables us to analyze that the stimuli generator was capable of delivering the data continuously during all the simulation execution. Next we discuss the answers to the research questions.

RQ1. Are the (automatically created) stimuli generators functional?

Yes. The stimuli generators were analyzed by a specialist that agreed that it contains all necessary structures to deliver the expected behavior. Moreover, we observed their behavior during the simulation execution, and the data that arrived in the gateways. 100% of the data were correctly delivered to the simulation.

RQ2. Are the stimuli generators capable of triggering a simulation correctly?

Yes. We followed the entire cycle of operation of the simulation. The stimuli generators were capable of receiving the input data from the database and generating the

expected stimuli for the constituents, triggering the SoS interoperability. Hence, the stimuli generator was well-succeeded. Three types of stimuli generators were derived from the specifications in SoSADL, as available externally[10]. For all of them, they were able to receive data stored in text files and deliver them to the simulation.

RQ3. Is the stimuli generator capable of supporting an entire simulation execution correctly?

During the entire cycle, the generator stimulated the simulation, completing the operation cycles of this SoS. The group of samples were grouped in four measures each to give a flood alert or not, depending on the analysis of the data. We followed the same strategy implemented in the real gateway: from four data that arrives, if two are above the threshold, the flood alert is triggered. 29 flood alerts occurred along some hours of flood (considering that each group of four data received that whose sum was more than the threshold triggered the alert). During the considered period, besides one effective flood (November 23th), in which the level of water arrived at almost seven meters, two other real threats of flood occurred on December 7th and 21st. With no failures, the stimuli generator was capable of supporting the simulation during its entire cycle of operation.

5.3. Discussion

Our solution promotes the automatic production of stimuli generators from architectural descriptions of SoS. It can create a distinct stimuli generator for each distinct type of constituent that forms a SoS. We applied the same methodology to produce three different types of functional stimuli generators: one for a smart sensor, one for crowdsourcing system, and another one for a drone, diversifying our data sources and characterizing the required multiple source of datas of a case study.

Considering our context, requirements elicitation for Flood Monitoring SoS was a joint effort between several institutions, such as ICMC/USP, IRISA/UBS, CEMADEN, and Franhoufer Institute. They elaborated the requirements, describing the surrounding environment for such SoS. After that, we used this document as an input to create the architectural description in SoSADL. Despite the environment being highly dynamic, a SoS is concerned only with a subset of the possible stimuli that can be received. Stimuli set completeness is a SoS requirements engineering issue. Considering a Flood Monitoring SoS as an example, data used in our example (such as water level) are relevant and enough to draw conclusions about a possible flood and the respective flood alert (as this is the

[10] https://goo.gl/xk5h3z

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

intended purpose for this SoS). Dynamics of the environment is handled by the sensors actuators themselves (at hardware level). Oscillations in values are passed to software-level, received and processed to draw actions. Then, we believe that the success of our approach relies on the effectiveness of a SoS to deal with the set of environment data delivered to it (even if this data set is specific and restrict). Moreover, at simulation level, we can observe how SoS will behave considering specific data received. If we notice that data delivered are not sufficient to draw conclusions, a new study can be conducted to increase data expected in order to increase precision of floods detection. However, considering the data set that we worked on and these types of constituents, SoS was able to detect all the flood threats that effectively occurred. Hence, we can conjecture that the stimuli set completeness is acceptable.

We also addressed scale, heterogeneity, and autonomy, which are important concerns for SoS. About scale, we run an example with 69 constituents, without considering gateways, mediators, and drone bases. For all constituent types, stimuli generators were automatically derived and worked correctly. About heterogeneity, we used five different types of constituents. About autonomy, all of these constituents exhibit their own structure, behavior, and independent operation.

It is important to highlight on how much the adoption of stimuli generators reduces the manual work of the SoS simulation. To perform this work manually, considering that each of the stimuli would consist of a click. Each click demands a reasoning from the human analyst. If each click needs 10 seconds to be decided and executed by a human, in a sample of 1000 data for each crowdsourcing system, 1000 for each sensor, and 5000 for each drone, this would result in an amount of 141,000 samples to be entered by the human user into an architecture of 42 sensors, 9 crowdsourcing systems, and 18 drones. Therefore, this work would require 1,410,000 seconds, which amounts to almost 392 hours of work, or almost 50 days of work in 8-hour days. Our procedure needed little more than 6 hours to run.

Contributions. The contributions of our approach are listed as follows:

- **Productivity**: We claim that Stimuli-SoS contributes to the productivity in the SoS engineering. Using our approach, we simulated 38 days in little more than six hours. The effort necessary to correctly simulate the activities of a human to reproduce real data accordingly would be significantly larger than using our solution, as discussed earlier. Thus, our approach is almost 65 times more productive than a manual approach, considering the architecture we used.

- **Reuse**: Programming the model transformation to automatically produce a stimuli

generator by one specialist with integral dedication took five days of work (a total of approximately 40 hours). Despite learning curve associated to DEVS modeling, Xtend programming, and domain-specific knowledge to adapt model transformations, the model transformation can be reused in a myriad of other domains. Producing a stimuli generator for each type of constituent of a SoS takes the same amount of time. The same specialist that produced the model transformation also produced an operational stimuli generator to realize the final format that should be achieved.

- **Model-Based Engineering for SoS**: Application of model-based methods in SoS engineering is still at the beginning (STEINHOGL, 2015). Moreover, a recent study reveals that MBE has been adopted for the development of SoS (around 60% of included studies in a systematic mapping applied MBE for development of SoS (LANA *et al.*, 2016)). MBE has been applied in SoS context for managing systems complexity, developing candidate architectures, and verifying design decisions early in the development process. Thus, we believe that our approach contributes to SoS software engineering by establishing a novel model-based approach to support SoS simulation and environment modeling. The automatic generation of stimuli generator for simulation of software architectures of SoS purposes is a contribution for Software Engineering for SoS and SoSE, as these techniques were broadly adopted for hardware benchmarking, but rarely applied in software engineering, in particular, Software Engineering for SoS;

- **Environment Modeling**: Environment modeling is an emerging issue, not only for simulation domain or SoS domain, but also for modern software engineering as a whole (DAVID *et al.*, 2013; IQBAL; ARCURI; BRIAND, 2015). It is important to improve techniques and methods to model the surrounding environment in which systems will be deployed, predicting situations that could not be dealt with effectively without this type of modeling, and preventing failures not envisioned before. These are vital issues as SoS becomes increasingly autonomous and ubiquitous, working on domains such as flood monitoring (OQUENDO, 2016c) and crisis management (SANTOS *et al.*, 2015).

- **Stimuli-SoS Workflow**: Stimuli generators are produced using a SoS software architecture description as input, following well-defined systematic steps that achieve the production of functional stimuli generators deployed in a simulation. The proposed workflow is also a contribution of our work, as it exhibits potential to be reproduced in other scenarios and contributes by prescribing how to produce this type of simulation structure.

Threats to Validity. Considering conclusion validity, we conjecture that it is not a remarkable threat for this study, since we do not have a statistical strength in our conclusions and we do not compare our approach with others, but use an exploratory study to draw our conclusions and justify our claims. Considering internal validity, we can raise the strategy to divide the data received by the gateway in a four window strategy. As this is the number of constituents, we do not perform remarkable partitions in data. Hence, we consider that this is not a significant threat. Considering construction validity, we draw our conclusions based on an approach that was systematically followed to automatically derive stimuli generators. Hence, we more observe than we measure. Further quantitative studies must be carried out to compare other forthcoming generations for different domains and that one we worked on here. Considering external validity, we run a case study in which, using approach, three different types of stimuli generators were produced, each one for a different type of system: a crowdsourcing system, a drone, and a sensor. As such, we increased our sources of evidence, even considering that all of them work in a same single simulation. Further investigation must be carried out, but there is some potential to application in other domains and generalization.

Regarding other threats, we can mention the possibility of failures if the SoS architect does not qualify the environment connections in SoSADL with the keyword `environment`. If it occurs, simulation can fail because 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 in larger contexts and applications are still required. Our approach was evaluated in regards to its success to support automatic production of stimuli generators that can correctly (i) reproduce the surrounding environment and constituents operational conditions. Considering that we use a software architecture description as the basis to produce stimuli generators, if the software architecture is not faithfully described, the stimuli generators created can not be correctly produced. We relieved this threat by submitting the software architecture description to a specialist. Another threat to validity is the correctness of the model transformation. To minimize the impact of this threat, a specialist conducted a manual inspection and carefully evaluated if each transformation rule produced exactly the expected output considering each input given.

Related Work. Recent studies have investigated the adoption of simulation in software engineering (FRANÇA; TRAVASSOS, 2016), and simulation has certainly been applied for SoS development (GRACIANO NETO *et al.*, 2014; XIA *et al.*, 2013; BOGADO; GONNET; LEONE, 2014). Additionally, initiatives have invested in the simulation of software architectures, but not specifically for software architectures of SoS, such as SysML

(OMG, 2017), MatLab/Simulink (MATLAB, 2010), Palladio [11] (BECKER; KOZIOLEK; REUSSNER, 2009), Bogado et al. (BOGADO; GONNET; LEONE, 2014) and Alexander et al. (ALEXANDER; NICOLAESCU; LICHTER, 2015). Other initiatives invested in modeling and simulating SoS, but with no support for software architecture concept (ZEIGLER *et al.*, 2012).

The development of stimuli generators for simulation purposes is not a new trend (YANG *et al.*, 2012; AL-HASHIMI, 1995; KITCHEN; KUEHLMANN, 2007). Initiatives have investigated the adoption of stimuli generators for hardware benchmarking. For example, Al-Hashimi (AL-HASHIMI, 1995) describes the use of stimuli generators to produce digital input signals for simulation purposes of analogic-digital systems. Kitchen and Kuehlmann present an approach to stimulate simulations of hardware with a stimuli generator that performs a random generation of input stimuli that obey a set of declaratively specified input constraints. Rahman and Lombigit (RAHMAN *et al.*, 2014) describe the development of a software that systematically generates stimulus required for code simulation (functional and timing) of new digital processors in gamma spectroscopy system. Yang et al. (YANG *et al.*, 2012) adopts simulations for verification purposes to evaluate the correctness of System-on-Chips. They apply stimuli generator to offer a broader coverage of test cases aiming to confirm the correctness of the chip operation. Thus, they do not work on top of software architectures, automating only the generation of the stimuli but not the infrastructure that will forward stimuli to the simulation.

For simulations in the context of SoS software engineering and software architecture, only few works have investigated stimuli generators. Table 24 compares closest related approaches considering the following six characteristics addressed by our approach:

1. Description of SoS Software Architectures: Does the highlighted approach adopt some formalism to describe SoS software architectures?

2. Simulation of SoS Software Architectures: Does the approach support simulation of SoS Software Architectures?

3. Environment Modeling: Does this approach adopt some type of environment modeling for simulation purposes?

4. Environment simulation: Does the approach adopt some type of environment simulation?

5. Adoption of Stimuli Generator: Does the approach adopt stimuli generator as the technique to inject inputs into the simulation;

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

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

6. Automatic derivation of Stimuli Generator: Does the approach prescribe some type of automatic derivation or mechanisms to stimulate a simulation?

Table 24 – Comparison between co-related approaches.

Approach	1	2	3	4	5	6
DEVS (ZEIGLER *et al.*, 2012)	No	No	Yes	Yes	Yes	No
Kewley et al. (KEWLEY *et al.*, 2008)	No	No	No	No	No	No
Soyez (SOYEZ *et al.*, 2014)	No	No	Yes	No	No	No
Stimuli-SoS	Yes	Yes	Yes	Yes	Yes	Yes

DEVS is a well-established formalism for simulating SoS in virtual environments (ZEIGLER *et al.*, 2012). DEVS deals with the system architecture, i.e., a simulation model in DEVS considers software and hardware aspects of all the constituents that compose a SoS, and for the SoS itself. DEVS takes into account several important characteristics of software architectures, such as data types, constituent systems (represented as atomic models), constituent behaviors (expressed as labeled input diagrams), SoS dynamics and how constituent exchange data (coupled models), events, and the overall organization of such constituents. However, it does not preserve the architectural details of SoS software architecture specification and relies on a low-level abstraction formalism, as discussed before.

Kewley et al. claim that constituents should be simulated by isolated simulations, and that such simulations should be *federated*, that is, they should interoperate in a synergistic way to form the whole simulation of a SoS (KEWLEY *et al.*, 2008). They adopt a framework called SySHub to play the role of *glue* that enables federations of models to support SoS simulation. However, they do not work on the level of software architecture (simulation or representation), despite the fact that they consider Distributed Interactive Simulation (DIS) and High Level Architecture (HLA) as potential architectural and interoperability methods for description and federated simulations of SoS (IEEE, 2010). However, even these notations do not tackle the concepts we address in our approach related to software architecture. They consider environment modeling as a potential forthcoming contribution of the SySHub system. However, we did not find continuity at this research topic or more recent papers that report supporting environmental modeling in SySHub context. Therefore, automatic derivation of stimuli generator is not currently covered in that approach.

Soyez et al. propose an agent-based tool to support modeling of static and dynamic aspects of SoS (SOYEZ *et al.*, 2014). Their formalism is based on the multi-level agent-based model IRM4MLS, which allows the representation of multiple entities that can interoperate at different levels, i.e., a constituent can be itself an SoS, hence supporting different levels of granularity (MORVAN; VEREMME; DUPONT, 2011). To evaluate their approach, they implemented a co-simulation of a directed SoS coping with a reconfiguration problem in the domain of Intelligent Autonomous Vehicles. Despite the use of co-simulation, they do not provide any evidence of concerns with the notion of software architecture, nor automatic code generation or stimuli generators. They support the modeling of environment and claim that their formalism is suitable for simulation. However, there is no evidence strengthening their claim.

Considering these previous works, there is a gap regarding the automatic derivation of stimuli generators based on software architectural descriptions of SoS. Our approach bridges these gaps and contributes by advancing the state of the art about simulation of software architectures of SoS.

By the nature of SoS, environments are only partially known at design-time (OQUENDO, 2016a). It is important to emphasize that our approach is to generate stimuli for simulation, not to automatically create the data to be used in the simulation. A prototype of the data is created that is functional, but there is no technique for creating data that is reliable to reality. Currently, this type of data is collected from other sources, and inserted via Java code into the body of the stimulus generator. Nonetheless, there is an important contribution towards environmental modeling in SoS engineering. In this stage of the contribution, we automatically create a virtual entity for the simulation capable of delivering the data in a rhythmic rhythm, imitating the surrounding environment from the data provided to the stimulus generator to feed the simulation. In a next step, we intend to invest in the automatic creation of these data by a more accurate description of the environment. Next section brings final remarks and potential for future research.

5.4. Final Remarks and Forthcoming Steps

This chapter presented Stimuli-SoS, an approach to systematically and automatically derive stimuli generators to support the execution of simulation of SoS. We established the following research question to be answered: *How is it possible to automatically obtain a functional stimuli generator that reproduces environmental conditions to the simulation of a SoS*? We concluded that the stimuli generators automatically created:

1. conform expected format. The transformation derived is what was expected;

2. were capable of receiving input data from the database and generating expected stimuli for the constituents, triggering the SoS;

3. were capable of correctly supporting an entire simulation execution; and

4. reproduce the environmental conditions of an SoS to become simulations functional without manual intervention.

Stimuli-SoS approach successfully produced a functional stimuli generator, which was thereby able to trigger the execution of a SoS architecture simulation. The stimuli generator correctly forwarded data to the simulation, which was able to reproduce 29 flood alerts triggered by the SoS during a flooding event. In particular, Stimuli-SoS is almost 15 times more productive than a manual approach to produce data for this simulation. From another perspective, we noticed that for our context, the effort necessary to manually create a stimuli generator for a particular SoS could require approximately the same effort that we invested to develop the model transformation that we adopt in our approach, with the additional benefit that the transformation can be reused for a myriad of other domains. Our approach succeeded in automatically deriving a functional stimuli generator that can reproduce environmental conditions for simulating an SoS. In particular, we presented new contributions regarding productivity and automation for the use of a model-based approach in SoS engineering.

Potential applications and forthcoming investigation can be conducted relying on the advances produced by our research. Co-simulation, for instance, is an important but significant challenge. It exhibits the potential to establish a communication between industrial simulators. However, even for the context of simulation of single subsystems that compose a whole monolithic system, co-simulation is still matter of investigation (GOMES, 2016; SCHWEIZER; LU; LI, 2016). Stimuli generators have the potential to be the interface that enables receiving the injection of values from industrial simulators. The automatic derivation of these stimuli generators from software architectural descriptions of SoS with support for environment modeling may enhance the fidelity of the code generated and the proximity with the environmental modeling provided by industrial simulators.

Simulations have been recognized as source of empirical evidences for software engineering (FRANÇA; TRAVASSOS, 2016). Hence, the adoption of our approach can leverage the research on empirical software engineering supported by simulations. Adopting our approach can aid in the automation of simulation-based studies, deriving stimuli generators to be applied during the simulation operation.

Stimuli generators materialize an infrastructure to support Verification, Validation and Testing (VV&T) activities in an automated way (ANAND *et al.*, 2013). They can be

applied to benchmark a SoS, working as a platform for VV&T of SoS. Each transition in an atomic model can work as a test case, and data can be provided by external files that hold test cases that are automatically generated by a testing tool (ANAND *et al.*, 2013; KOREL, 1990). Moreover, VV&T for SoS is currently a challenging research issue and point of investigation in Software Engineering for SoS (LANA *et al.*, 2016).

Our approach also exhibits a potential to become an architectural pattern for modeling of simulations. As stimulating a simulation is a recurrent problem, we can establish a stimuli generator as a systematic and repetitive solution that can be adapted according to the context in which it will be applied. Simulation is an important branch of Software Engineering for SoS. It exhibits a remarkable potential to be largely adopted in Software Engineering for SoS in the forthcoming years. Then, investigating potentials of automation in the coverage of tests and correctness of operation is paramount to avoid damages, losses, and financial problems that could be brought by an SoS deployed with errors. We believe that our approach can contribute to leverage the degree of trustworthiness delivered by an SoS.

CHAPTER

6

CONCLUSIONS

During this PhD project, an infrastructure was developed to support the functional evaluation of SoS software architectures and to answer the research question *How can we evaluate SoS software architectures at design-time?*. Such infrastructure consists of (i) a solution for automatic generation of simulations for SoS software architectures with support for dynamic reconfiguration, (ii) representation of the environment and automatic generation of stimuli generators, and (iii) a means to reestablish consistency between the runtime architecture and the original SoS architectural documentation. This infrastructure supported an analysis of the impact of the SoS dynamic architecture on functionalities provided. Such analysis were performed for two different domains (Flood Monitoring and Space) using several simulations automatically obtained.

6.1. Solutions

To answer RQ1 (How can the evaluation of SoS architectures be supported) and RQ2 (How can SoS dynamics be anticipated and predicted at design-time?), in chapter 3 we proposed ASAS and described how it enabled us to evaluate SoS operation in all the simulations produced and executed. We could measure and evaluate the success fee achieved by the Flood Monitoring SoS and by Space SoS to accomplish their missions, despite their inherent dynamic architectures, and the percentage of success with which they were able to offer their functionalities (GRACIANO NETO *et al.*, 2018b). We evaluated those architectures regarding to 1) the percentage of success with which a SoS achieves one or more missions considering its dynamic architecture and uncertainty, 2) its inherent capacity to maintain its operation in course, despite all these impacting factors, and 3) the percentage of success with which the data are transported and delivered in the

final destination to be used and communicated. Moreover, we were also able to analyze results achieved by different architectural configurations, successfully predicting which architectural configuration was best for the context of a FMSoS, and confirm that increasing the number of satellites would improve the services provided by a Space SoS. We also provided a set of simulation instructions as patterns, which enabled us to automatically generate functional simulations with no conflicting specification rules (GRACIANO NETO *et al.*, 2018c).

As SoS continuously changes its architectural arrangement due to its inherent dynamics, to answer RQ3 (How can SoS architectural description be continually consistent with its runtime configuration, despite its inherent dynamics?), in Chapter 4, we presented Back-SoS, which consists of an approach that supports the synchonization between current SoS architectural configuration and the actual SoS architectural description in SosADL (GRACIANO NETO *et al.*, 2018a). We proposed the concept of architectural drift for SoS software architectures, and how it can be dealt with.

Finally, to answer RQ4 (How can the surrounding environment be modelled for a SoS simulation purpose?), we established Stimuli-SoS (GRACIANO NETO *et al.*, 2016; GRACIANO NETO *et al.*, 2017), a model-based solution to automatically derive stimuli generators from a SoS software architectural description specified in SosADL. Stimuli generators are used in simulations of SoS software architectures to make them executable, reproducing the real conditions in which those architectures should be deployed, playing the role of the SoS surrounding environment, continuously delivering stimuli to feed the SoS simulation, maintaining the SoS operation, and supporting the prediction of SoS reaction due to a diversity of stimuli that it can receive.

6.2. Limitations

This section describes limitations of this thesis and how these can be addressed in future research.

1. **SosADL grammar was not fully implemented.** Some language constructs such as protocols, properties, and ask/tell operators have not been exploited. Protocols and properties have not been mapped to DEVS because part of these properties is described in other parts of a SoS architectural model, and because the focus of the solution of this thesis is on the reliability under a SoS functional perspective. Ask/tell are synchronous constructions that allow SoS to tell the environment around its functionalities and to obtain from this environment functionalities available from

other constituents, establishing new connections with them, and potentially providing new functionalities.

2. **Correctness of the simulation model is related to the correctness of the architectural model.** Since we use a model transformation that maps SoSADL to DEVS, simulation reliability is tied to the correctness of the SoS architectural specification. In this case, some type of inspection or evaluation must be performed by a specialist before basing the conclusions on the simulation. After all, the quality of the conclusions can be affected if the architectural specification is imprecise. We conducted peer-reviewing during SoS architectural specifications to avoid inaccuracy in specification. Moreover, our method enables monitoring SoS dynamics at runtime. Hence, simulations serve as a proof of concept for architectural specifications.

3. **SosADL is an ADL, but not a ML (modeling language).** Despite the advances made by SosADL, a complementary model is still needed to support architects with a visual evaluation of the SoS model. Currently, this is only possible through simulation, i.e., there is no visual architectural description prior to the simulation. In this sense, a modeling language can be constructed (based on UML Profile, for example), to use the formal foundation of the language, and provide an alternative, more intuitive and agile way of visualizing SoS architecture configurations and their dynamics.

6.3. Possible Extensions and Future Work

Several opportunities of research emerge to further the achievements of this thesis, described as follows.

1. **Simulation-based SoS Software Testing**: Simulations can support empirical evaluation in Software Engineering (FRANÇA; TRAVASSOS, 2012; FRANÇA; TRAVASSOS, 2013; FRANÇA; TRAVASSOS, 2015; FRANÇA; TRAVASSOS, 2016). However, when simulations are used for the purpose of evaluating architectures (regarding their functional aspect), simulations must exercise as many as possible scenarios to which a SoS can be submitted. Otherwise, the simulation will not be reliable as an evaluation method. In this sense, a SoS software engineering test approach can be designed to help the simulation cover as many scenarios as possible, increasing the reliability of the designed solution. The need for exhaustion of the scenarios can raise branches like test case set generation and mutation analysis for this specific situation. Testing strategies for SoS simulation models must be established, as it has been proposed for embedded systems in Simulink formalism

(BARESI; DELAMARO; NARDI, 2017), and are still scarce for DEVS formalism (LI *et al.*, 2011);

2. **SoS Architectural Evaluation Method through Non-Functional Properties with Monitoring via Simulation**: As a complementary branch for co-simulation, ASAS approach can also be expanded to involve architectural evaluation about non-functional properties. A valuable outcome for software architecture community can be automatic trade-off analysis, i.e., given the non-functional properties and the percentage in which they are achieved by the SoS, how varying constituents can impact on these values, considering the optimization functions that drive such trade-off analysis.

3. **Solutions were created based on simulations.** Simulations are pivotal elements in the development of SoS, especially when considering the human integrity risks to which SoS users will be subjected, and the costs of acquiring a SoS and its constituent systems. However, when SoS applications start to be built and deployed in the real world, adaptations in the proposed methods will need to be performed to achieve the same results, such as generating functional constituent code, or performing automatic discovery of constituents and their available functionalities, and updating the architectural description.

4. **Trustworthiness for SoS**: Trustworthiness requires, in addition to the operation correctness, also the safety and security. This thesis covers only correctness in the sense that the simulation allows to validate whether the SoS is correct in terms of emergent behaviors. Security has also been dealt with for SoS context (HACHEM *et al.*, 2016). Safety must also be tackled, and the association of the threefold SoS trustworthiness.

5. **SoS Acquisition**: Cost is perhaps the primary driver to decide whether to build a SoS or to create a new specialized system (JOHNSON, 2015). ASAS enables to evaluate the performance of different arrangements of constituents. For example, in a case study that we conducted, we concluded that using a small number of constituents could achieve the same results than using a large number of constituents. Owing to such information, it is possible to anticipate which constituents are effectively necessary to build a SoS, and predict the budget necessary to acquire them. Hence, ASAS can also be extended to have a finantial prediction branch;

6. **Systems-of-Information Systems (SoIS) modeling and evaluation**: A SoIS is a set of interoperable Information Systems (IS) that are combined to achieve some broader business value and/or to exploit some business opportunity (CARLSSON;

STANKIEWICZ, 1991; BRESCHI; MALERBA, 1997; SALEH; ABEL, 2015; MAJD; MARIE-HÉLÈNE; ALOK, 2015). A SoIS is formed from a specific cluster of the firms, technologies, and industries involved in the generation and diffusion of new technologies and in the knowledge flow that takes place among them. Under this perspective, SoIS have a strong business nature (GRACIANO NETO; OQUENDO; NAKAGAWA, 2017). As SoIS are often formed by constituent systems that are managed and operated by independent organizations and they can cooperate to accomplish inter-organizational missions and, as a consequence, inter-organization business processes. This raises the need of establishing representation and management strategies to support concepts such as sequence of activities and interdependence between roles and goals (GRACIANO NETO *et al.*, 2017). ASAS must also be extended to deal with such specific types of SoS.

Une approche dirigée par les simulations à base de modèles pour concevoir les architectures de systèmes-des-systèmes à logiciel prépondérant Valdemar Vicente Graciano Neto 2019

REFERENCES

ABRAHAO, S.; INSFRAN, E. Evaluating software architecture evaluation methods: An internal replication. In: **21st International Conference on Evaluation and Assessment in Software Engineering (EASE 2017)**. Karlskrona, Sweden: ACM, 2017. p. 144–153. Citation on page 6.

ACKERMANN, C.; LINDVALL, M.; CLEAVELAND, R. Towards behavioral reflexion models. In: **20th IEEE International Conference on Software Reliability Engineering (ISSRE 2009)**. Piscataway, NJ, USA: IEEE Press, 2009. p. 175–184. Citations on pages 93 and 94.

AL-HASHIMI, B. **The Art of Simulation Using PSpice: Analog and Digital**. 1st. ed. Boca Raton, FL, USA: CRC Press, Inc., 1995. Citations on pages 30, 31, and 154.

ALBUQUERQUE, J. P. de; HORITA, F. E. A.; DEGROSSI, L. C.; ROCHA, R. dos S.; ANDRADE, S. C. de; RESTREPO-ESTRADA, C.; LEYH, W. Leveraging volunteered geographic information to improve disaster resilience: Lessons learned from agora and future research directions. In: CAMPELO, C. E. C.; BERTOLOTTO, M.; CORCORAN, P. (Ed.). **Volunteered Geographic Information and the Future of Geospatial Data**. Hershey, USA: IGI Global, 2017. p. 158–184. Citation on page 140.

ALEXANDER, P.; NICOLAESCU, A.; LICHTER, H. Model-based evaluation and simulation of software architecture evolution. In: **10th International Conference on Software Engineering Advances (ICSEA 2015)**. Barcelona, Spain: IARIA, 2015. p. 153 – 156. Citations on pages 92 and 154.

ALLEN, R.; DOUENCE, R.; GARLAN, D. Specifying and analyzing dynamic software architectures. In: ASTESIANO, E. (Ed.). **Fundamental Approaches to Software Engineering**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. p. 21–37. Citations on pages 25 and 26.

ALLEN, R.; GARLAN, D. A formal basis for architectural connection. **ACM Transactions on Software Engineering Methodology**, ACM, New York, NY, USA, v. 6, n. 3, p. 213–249, Jul. 1997. Citation on page 41.

ALVAREZ, J. L.; METSELAAR, H.; AMIAUX, J.; CRIADO, G. S.; VENANCIO, L. M. G.; SALVIGNOL, J.-C.; LAUREIJS, R. J.; VAVREK, R. Model-based system engineering approach for the Euclid mission to manage scientific and technical complexity. In: **SPIE Astronomical Telescopes + Instrumentation (SPIE 2016)**. Edinburgh, United Kingdom: Society of Photo-Optical Instrumentation Engineers (SPIE), 2016. p. 1–19. Citation on page 21.

ANAND, S.; BURKE, E. K.; CHEN, T. Y.; CLARK, J.; COHEN, M. B.; GRIESKAMP, W.; HARMAN, M.; HARROLD, M. J.; MCMINN, P. *et al.* An orchestrated survey of

methodologies for automated software test case generation. **Journal of Systems and Software**, Elsevier, v. 86, n. 8, p. 1978–2001, 2013. Citations on pages 157 and 158.

ANDERSSON, J.; LEMOS, R.; MALEK, S.; WEYNS, D. Software engineering for self-adaptive systems. In: CHENG, B. H.; LEMOS, R.; GIESE, H.; INVERARDI, P.; MAGEE, J. (Ed.). Berlin, Heidelberg: Springer-Verlag, 2009. chap. Modeling Dimensions of Self-Adaptive Software Systems, p. 27–47. Citation on page 25.

ANDRÉN, F.; STRASSER, T.; ROHJANS, S.; USLAR, M. Analyzing the need for a common modeling language for smart grid applications. In: **11th IEEE International Conference on Industrial Informatics (INDIN 2013)**. Bochum, Germany: IEEE, 2013. p. 440–446. Citation on page 37.

ANDRÉN, F. P.; STRASSER, T. I.; KASTNER, W. Engineering Smart Grids: Applying Model-Driven Development from Use Case Design to Deployment. **ENERGIES**, MDPI AG, Basel, Switzerland, v. 10, n. 3, p. 1–33, 2017. Citations on pages 33, 37, and 38.

ANDREWS, Z.; PAYNE, R.; ROMANOVSKY, A.; DIDIER, A.; MOTA, A. Model-based development of fault tolerant systems of systems. In: **7th IEEE International Systems Conference (SysCon 2013)**. Orlando, FL, USA: IEEE, 2013. p. 356–363. Citations on pages 23 and 35.

BADDOUR, R.; PASPALIARIS, A.; HERRERA, D. SCV2: A model-based validation and verification approach to system-of-systems engineering. In: **10th System of Systems Engineering Conference (SoSE 2015)**. San Antonio, TX, USA: IEEE, 2015. p. 422–427. Citation on page 35.

BAGDATLI, B.; MAVRIS, D. Use of high-level architecture discrete event simulation in a system of systems design. In: **2nd IEEE Aerospace Conference (Aero 2012)**. Big Sky, MT, USA: IEEE, 2012. p. 1–13. Citation on page 29.

BALASUBRAMANIAN, K.; SCHMIDT, D. C.; MOLNAR, Z.; LEDECZI, A. System integration using model-driven engineering. In: TIAKO, P. F. (Ed.). **Designing Software-Intensive Systems: Methods and Principles**. USA: IGI Global, 2009. Citation on page 18.

BALCI, O. Principles of simulation model validation, verification, and testing. **Transactions of the Society for Computer Simulation International**, Society for Computer Simulation International, San Diego, CA, USA, v. 14, n. 1, p. 3–12, 1997. Citation on page 5.

BALDWIN, W. C.; SAUSER, B.; CLOUTIER, R. Simulation approaches for system of systems: Events-based versus agent based modeling. **Procedia Computer Science**, Elsevier, Amsterdam, Netherlands, v. 44, n. 1, p. 363 – 372, 2015. Citations on pages 29 and 30.

BANKS, J. Introduction to simulation. In: **31st Conference on Winter Simulation: Simulation - a Bridge to the Future - Volume 1 (WSC 1999)**. Phoenix, Arizona, USA: ACM, 1999. p. 7–13. Citation on page 30.

BARBI, E.; CANTONE, G.; FALESSI, D.; MORCIANO, F.; RIZZUTO, M.; SABBATINO, V.; SCARRONE, S. A model-driven approach for configuring and deploying systems of systems. In: **7th International Conference on System of Systems Engineering (SoSE 2012)**. Genova, Italy: IEEE, 2012. p. 214–218. Citations on pages 18, 19, 24, 32, 34, 35, 39, and 124.

BARESI, L.; DELAMARO, M. E.; NARDI, P. A. Test oracles for simulink-like models. **Automated Software Engineering**, Springer, New York, NY, USA, v. 24, n. 2, p. 369–391, 2017. Citation on page 162.

BARTON, P. I.; PANTELIDES, C. C. Modeling of combined discrete/continuous processes. **AIChE Journal**, American institution of Chemical Engineers, New York, NY, USA, v. 40, n. 6, p. 966–979, 1994. Citation on page 31.

BASILI, V.; CALDIERA, G.; ROMBACH, H. D. **Software modeling and measurement: the Goal/Question/Metric paradigm**. College Park, Maryland, USA, 1992. Citation on page 63.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. 3rd. ed. Indianapolis, Indiana, USA: Addison-Wesley Professional, 2012. Citations on pages 2, 22, and 43.

BATISTA, T. Challenges for sos architecture description. In: **First International Workshop on Software Engineering for Systems-of-Systems (SESoS 2013)**. Montpellier, France: ACM, 2013. p. 35–37. Citations on pages 18 and 26.

BAY, J. S. Recent advances in the design of distributed embedded systems. In: **Aerosense 2002**. Orlando, FL, USA: The International Society for Optical Engineering, 2002. p. 36–45. Citations on pages 18 and 35.

BECKER, S.; KOZIOLEK, H.; REUSSNER, R. The palladio component model for model-driven performance prediction. **Journal of Systems and Software**, Elsevier Science Inc., New York, NY, USA, v. 82, n. 1, p. 3–22, 2009. Citation on page 154.

BELLOIR, N.; CHIPRIANOV, V.; AHMAD, M.; MUNIER, M.; GALLON, L.; BRUEL, J.-M. Using Relax Operators into an MDE Security Requirement Elicitation Process for Systems of Systems. In: **8th European Conference on Software Architecture Workshops (ECSAW 2014)**. Vienna, Austria: ACM, 2014. p. 32:1–32:4. Citations on pages 35 and 36.

BETTINI, L. **Implementing Domain-Specific Languages with Xtext and Xtend**. Birmingham, United Kingdom: Packt Publishing, 2013. Citation on page 31.

BETTINI, L.; STOLL, D.; VOLTER, M.; COLAMEO, S. Approaches and tools for implementing type systems in xtext. In: CZARNECKI, K.; HEDIN, G. (Ed.). **Software Language Engineering**. Berlin, Germany: Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 7745). p. 392–412. Citation on page 38.

BOARDMAN, J.; SAUSER, B. System of systems - the meaning of 'of'. In: **1st IEEE/SMC International Conference on System of Systems Engineering (SoSE 2006)**. Los Angeles, USA: IEEE, 2006. p. 1–6. Citations on pages 3 and 101.

BOCCIARELLI, P.; D'AMBROGIO, A. Model-driven method to enable simulation-based analysis of complex systems. In: **Modeling and Simulation-Based Systems Engineering Handbook**. Boca Raton, Florida, USA: CRC Press, 2014. p. 119–148. Citations on pages 94 and 96.

BOEHM, B. A view of 20th and 21st century software engineering. In: **28th International Conference on Software Engineering (ICSE 2006)**. Shanghai, China: ACM, 2006. p. 12–29. Citation on page 1.

BOGADO, V.; GONNET, S.; LEONE, H. Modeling and simulation of software architecture in discrete event system specification for quality evaluation. **Simulation**, Society for Computer Simulation International, San Diego, CA, USA, v. 90, n. 3, p. 290–319, 2014. Citations on pages 30, 153, and 154.

BOSCH, J. **Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach**. New York, USA: Addison-Wesley, 2000. Citations on pages 28 and 30.

BRESCHI, S.; MALERBA, F. Sectoral innovation systems: technological regimes, schumpeterian dynamics, and spatial boundaries. **Systems of innovation: Technologies, institutions and organizations**, Pinter London, p. 130–156, 1997. Citation on page 163.

BROWN, K.; CHIPKEVICH, M.; BAMBERGER, R.; HUANG, T.-T.; MATTIES, M.; REEVES, J.; ROUFF, C. Model-based design for affordability of a netted intelligence, surveillance, and reconnaissance concept. **Johns Hopkins APL Technical Digest (Applied Physics Laboratory)**, v. 33, n. 1, p. 23–36, 2015. Citation on page 35.

BRUNEAU, J.; CONSEL, C. Diasim: a simulator for pervasive computing applications. **Software: Practice and Experience**, John Wiley & Sons, Ltd, New Jersey, USA, v. 43, n. 8, p. 885–909, 2013. Citation on page 30.

BRUNELIERE, H.; CABOT, J.; JOUAULT, F.; MADIOT, F. Modisco: A generic and extensible framework for model driven reverse engineering. In: **25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)**. Antwerp, Belgium: ACM, 2010. p. 173–174. Citation on page 124.

BRYANS, J.; FITZGERALD, J.; PAYNE, R.; KRISTENSEN, K. Maintaining emergence in systems of systems integration: a contractual approach using SysML. In: **INCOSE International Symposium**. Las Vegas, USA: INCOSE, 2014. p. 166–181. Citation on page 35.

BRYANS, J.; FITZGERALD, J.; PAYNE, R.; MIYAZAWA, A.; KRISTENSEN, K. Sysml contracts for systems of systems. In: **9th International Conference on System of Systems Engineering (SoSE 2014)**. Adelaide, Australia: IEEE, 2014. p. 73–78. Citation on page 35.

BRYANS, J.; PAYNE, R.; HOLT, J.; PERRY, S. Semi-formal and formal interface specification for system of systems architecture. In: **7th IEEE International Systems Conference (SysCon 2013)**. Orlando, FL, USA: INCOSE, 2013. p. 612–619. Citations on pages 18, 19, 23, 35, and 36.

CALINESCU, R.; KWIATKOWSKA, M. Software engineering techniques for the development of systems of systems. In: CHOPPY, C.; SOKOLSKY, O. (Ed.). **Foundations of Computer Software. Future Trends and Techniques for Development**. Berlin, Germany: Springer Berlin Heidelberg, 2010, (Lecture Notes in Computer Science, v. 6028). p. 59–82. Citations on pages 17, 35, 103, and 104.

CANOVAS, J.; MOLINA, J. An architecture-driven modernization tool for calculating metrics. **IEEE Software**, IEEE Computer Society, Los Alamitos, CA, USA, v. 27, n. 4, p. 37–43, 2010. Citation on page 16.

CARLE, P.; KERVARC, R.; CUISINIER, R.; HUYNH, N.; BEDOUËT, J.; RIVIÈRE, T.; NOULARD, E. Simulation of Systems of Systems. **AerospaceLab**, Onera Scientific Information Department, Palaiseau, France, n. 4, p. p. 1–10, 2012. Citations on pages 3 and 29.

CARLSSON, B.; STANKIEWICZ, R. On the nature, function and composition of technological systems. **Journal of Evolutionary Economics**, Springer, Berlin, Germany, v. 1, n. 2, p. 93–118, 1991. Citation on page 163.

CARVALHO, M. J. M. de; LIMA, J. S. dos S.; JOTHA, L. dos S.; AQUINO, P. S. de. CONASAT - Constellation of Nano Satellites for Environmental Data Collection (in portuguese). In: **16th Brazilian Symposium on Remote Sensing**. Foz do Iguaçu, Brazil: National institution of Space Research, 2013. p. 9108–9115. Citations on pages 19, 80, and 81.

CAVALCANTE, E.; BATISTA, T. V.; OQUENDO, F. Supporting dynamic software architectures: From architectural description to implementation. In: **12th Working IEEE/IFIP Conference on Software Architecture (WICSA 2015)**. Montreal, QC, Canada: IEEE, 2015. p. 31–40. Citations on pages 26, 43, 59, 108, and 126.

CAVALCANTE, E.; OQUENDO, F.; BATISTA, T. Architecture-Based Code Generation: From π-ADL Architecture Descriptions to Implementations in the Go Language. In: AVGERIOU, P.; ZDUN, U. (Ed.). **8th European Conference on Software Architecture (ECSA 2014)**. Viena, Austria: Springer International Publishing, 2014, (Lecture Notes in Computer Science, v. 8627). p. 130–145. Citations on pages 22 and 92.

CAVALCANTE, E.; QUILBEUF, J.; TRAONOUEZ, L.; OQUENDO, F.; BATISTA, T.; LEGAY, A. Statistical model checking of dynamic software architectures. In: **10th European Conference on Software Architecture (ECSA 2016)**. Copenhagen, Denmark: Springer, 2016. p. 185–200. Citations on pages 22 and 92.

CERRUDO, C. **Keeping Smart Cities Smart: Preempting Emerging Cyber Attacks in U.S. Cities**. Washington D.C., USA, 2015. Citation on page 1.

CETINKAYA, D.; VERBRAECK, A. Metamodeling and model transformations in modeling and simulation. In: **44th Winter Simulation Conference (WSC 2011)**. Phoenix, Arizona: IEEE, 2011. p. 3048–3058. Citation on page 99.

CETINKAYA, D.; VERBRAECK, A.; SECK, M. D. Model Transformation from BPMN to DEVS in the MDD4MS Framework. In: **8th Symposium on Theory of Modeling and**

Simulation - DEVS Integrative M&S Symposium (TMS/DEVS '12). Orlando, USA: Society for Modeling and Simulation International (SCS), 2012. p. 1–6. Citation on page 99.

CHALMERS, D. J. Strong and weak emergence. In: DAVIES, P.; CLAYTON, P. (Ed.). **The Re-Emergence of Emergence**. Oxford, United Kingdom: Oxford University Press, 2006. Citation on page 3.

CHENEY, J.; GIBBONS, J.; MCKINNA, J.; STEVENS, P. On principles of least change and least surprise for bidirectional transformations. **Journal of Object Technology**, v. 16, n. 1, p. 3:1–31, 2017. Citation on page 125.

CHIPRIANOV, V.; FALKNER, K.; SZABO, C.; PUDDY, G. Architectural Support for Model-Driven Performance Prediction of Distributed Real-Time Embedded Systems of Systems. In: AVGERIOU, P.; ZDUN, U. (Ed.). **8th European Conference on Software Architecture (ECSA 2014)**. Vienna, Austria: Springer, 2014. p. 357–364. Citation on page 94.

CHOI, B. K.; KANG, D. **Modeling and Simulation of Discrete Event Systems**. 1st. ed. Hoboken, New Jersey, USA: Wiley Publishing, 2013. Citation on page 30.

CICCHETTI, A.; RUSCIO, D. D.; ERAMO, R.; PIERANTONIO, A. Automating co-evolution in model-driven engineering. In: **12th International IEEE Enterprise Distributed Object Computing Conference (EDOC 2008)**. Munich, Germany: IEEE Computer Society, 2008. p. 222–231. Citation on page 16.

COOK, T. S.; DRUSINKSY, D.; SHING, M. T. Specification, validation and run-time monitoring of soa based system-of-systems temporal behaviors. In: **3th IEEE International Conference on System of Systems Engineering (SoSE 2007)**. San Antonio, USA: IEEE, 2007. p. 1–6. Citation on page 23.

COSTA, C.; PÉREZ, J.; CARSÍ, J. Á. Dynamic adaptation of aspect-oriented components. In: SCHMIDT, H. W.; CRNKOVIC, I.; HEINEMAN, G. T.; STAFFORD, J. A. (Ed.). **Component-Based Software Engineering**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 49–65. ISBN 978-3-540-73551-9. Citation on page 25.

CZARNECKI, K.; FOSTER, J.; HU, Z.; LAMMEL, R.; SCHURR, A.; TERWILLIGER, J. F. Bidirectional transformations: A cross-discipline perspective. In: PAIGE, R. F. (Ed.). **Theory and Practice of Model Transformations**. Berlin, Germany: Springer Berlin Heidelberg, 2009, (Lecture Notes in Computer Science, v. 5563). p. 260–283. Citation on page 17.

CZARNECKI, K.; HELSEN, S. Classification of model transformation approaches. In: **OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture (GTCMDA 2003)**. Anaheim, USA: ACM, 2003. Citations on pages 17 and 38.

DAHMANN, J.; MARKINA-KHUSID, A.; DOREN, A.; WHEELER, T.; COTTER, M.; KELLEY, M. Sysml executable systems of system architecture definition: A working example. In: **11th Annual IEEE International Systems Conference (SysCon 2017)**. Montreal, Canada: IEEE, 2017. p. 1–6. Citation on page 23.

DAHMANN, J. S. High level architecture for simulation. In: **First International Workshop on Distributed Interactive Simulation and Real Time Applications (DIS-RTA 1997)**. Eilat, Israel, Israel: IEEE, 1997. p. 9–14. Citation on page 96.

DAHMANN, J. S.; JR., G. R.; LANE, J. A. Systems engineering for capabilities. **CrossTalk Journal - The Journal of Defense Software Engineering**, v. 21, n. 11, p. 4–9, 2008. Citation on page 2.

DAVID, O.; II, J. A.; LLOYD, W.; GREEN, T.; ROJAS, K.; LEAVESLEY, G.; AHUJA, L. A software engineering perspective on environmental modeling framework design: The object modeling system. **Environmental Modelling & Software**, v. 39, p. 201 – 213, 2013. Thematic Issue on the Future of Integrated Modeling Science and Technology. Citation on page 152.

DICKERSON, C.; VALERDI, R. Using relational model transformations to reduce complexity in SoS requirements traceability: Preliminary investigation. In: **5th International Conference on System of Systems Engineering (SoSE 2010)**. Loughborough, United Kingdom: IEEE, 2010. p. 1–6. Citation on page 35.

DNIT. **DNIT - National Department of Transportation Infrastructure**. 2017. <https://goo.gl/Byb6wn>. Last Access: December 2017. Citations on pages 19 and 77.

DOBRICA, L.; NIEMELE, E. A survey on software architecture analysis methods. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 28, n. 7, p. 638–653, Jul. 2002. Citations on pages 5, 6, 28, and 101.

DRIRA, K.; OQUENDO, F. Special issue on advanced architectures for the future generation of software-intensive systems. **Future Generation Computer Systems**, Elsevier, v. 47, n. 10, p. 60–61, 2015. Citations on pages 103 and 104.

ECLIPSE. **Acceleo**. 2012. Available: <http://www.eclipse.org/acceleo/>. Citation on page 31.

ECSS. **ECSS Space Engineering - Ground systems and operations**. Noordwijk, The Netherlands, 2008. Citation on page 72.

FALCONE, A.; GARRO, A.; TAYLOR, S.; ANAGNOSTOU, A.; CHAUDHRY, N.; SALAH, O. Experiences in simplifying distributed simulation: The HLA development kit framework. **Journal of Simulation**, v. 11, n. 3, p. 208–227, 2017. Citation on page 29.

FALKNER, K.; SZABO, C.; CHIPRIANOV, V.; PUDDY, G.; RIECKMANN, M.; FRASER, D.; ASTON, C. Model-driven performance prediction of systems of systems. **Software & Systems Modeling**, Springer, v. 15, n. 3, p. 1–27, 2016. Citations on pages 6, 22, 29, 31, 32, 94, 97, and 126.

FANG, Z.; DELAURENTIS, D.; DAVENDRALINGAM, N. An approach to facilitate decision making on architecture evolution strategies. **Procedia Computer Science**, v. 16, n. Supplement C, p. 275 – 282, 2013. Citations on pages 94 and 96.

FARCAS, C.; FARCAS, E.; KRUEGER, I.; MENARINI, M. Addressing the integration challenge for avionics and automotive systems - from components to rich services. **Proceedings of the IEEE**, v. 98, n. 4, p. 562–583, 2010. Citation on page 18.

FELDT, R.; MAGAZINIUS, A. Validity threats in empirical software engineering research-an initial survey. In: **International Conference on Software Engineering and Knowledge Engineering (SEKE 2010)**. Redwood City, San Francisco Bay, USA: KSI Research Inc. and Knowledge Systems institution Graduate School, 2010. p. 374–379. Citation on page 92.

FIRESMITH, D. **Profiling Systems Using the Defining Characteristics of Systems of Systems (SoS)**. Pittsburgh, Pennsylvania, 2010. Citation on page 26.

FISCHER, N.; SALZWEDEL, H. Overcoming the Generation Gap in aircraft designs with executable specifications. In: **31st IEEE/AIAA Digital Avionics Systems Conference (DASC 2011)**. Sydney, Australia: IEEE, 2011. p. 1–10. Citation on page 18.

FITZGERALD, J.; FOSTER, S.; INGRAM, C.; LARSEN, P. G.; WOODCOCK, J. **Model-based Engineering for Systems of Systems: the COMPASS Manifesto**. Cambrigde, United Kingdom, 2013. Citations on pages 2 and 99.

FITZGERALD, J.; LARSEN, P. G.; WOODCOCK, J. Foundations for model-based engineering of systems of systems. In: AIGUIER, M.; BOULANGER, F.; KROB, D.; MARCHAL, C. (Ed.). **Complex Systems Design & Management**. Switzerland: Springer, 2014. p. 1–19. Citation on page 5.

FOSTER, H.; KINGDOMHIJA, A. M.; ROSENBLUM, D. S.; UCHITEL, S. Specification and analysis of dynamically-reconfigurable service architectures. In: WIRSING, M.; HÖLZL, M. (Ed.). **Rigorous Software Engineering for Service-Oriented Systems: Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing**. Berlin, Heidelberg: Springer, 2011. p. 428–446. Citation on page 40.

FRANÇA, B. B. N. de; TRAVASSOS, G. H. Reporting guidelines for simulation-based studies in software engineering. In: **16th International Conference on Evaluation & Assessment in Software Engineering (EASE 2012)**. Ciudad Real, Spain: IET, 2012. p. 156–160. Citations on pages 104 and 161.

______. Are we prepared for simulation based studies in software engineering yet? **CLEI Electronic Journal**, v. 16, n. 1, 2013. Citation on page 161.

______. Simulation based studies in software engineering: A matter of validity. **CLEI Electronic Journal**, v. 18, n. 1, 2015. Citations on pages 104 and 161.

______. Experimentation with dynamic simulation models in software engineering: planning and reporting guidelines. **Empirical Software Engineering**, v. 21, n. 3, p. 1302–1345, 2016. Citations on pages 28, 101, 153, 157, and 161.

FRANCE, R.; RUMPE, B. Model-driven development of complex software: A research roadmap. In: **Future of Software Engineering (FOSE 2007)**. Minneapolis, USA: IEEE, 2007. p. 37–54. Citations on pages 17, 18, 28, 29, 31, and 102.

FUCHS, J.; LINDMAN, N. Using modeling and simulation for system of systems engineering applications in the european space agency. In: RAINEY, L. B.; TOLK, A. (Ed.). **Modeling and Simulation Support for System of Systems Engineering Applications**. Hoboken, New Jersey, USA: John Wiley & Sons, Inc, 2014. p. 303–336. Citations on pages 94 and 97.

GARLAN, D.; MONROE, R. T.; WILE, D. **ACME: An Architecture Description Interchange Language**. Pittsburgh, USA, 1997. Citation on page 26.

GASSARA, A.; BOUASSIDA, I.; JMAIEL, M. A tool for modeling sos architectures using bigraphs. In: **32nd Symposium on Applied Computing (SAC 2017)**. Marrakech, Morocco: ACM, 2017. p. 1787–1792. Citations on pages 24, 39, 94, and 98.

GASSARA, A.; RODRIGUEZ, I. B.; JMAIEL, M.; DRIRA, K. A bigraphical multi-scale modeling methodology for system of systems. **Computers & Electrical Engineering**, v. 58, n. Supplement C, p. 113 – 125, 2017. Citations on pages 24, 94, and 98.

GE, B.; HIPEL, K. W.; LI, L.; CHEN, Y. A data-centric executable modeling approach for system-of-systems architecture. In: **7th International Conference on System of Systems Engineering (SoSE 2012)**. Genova, Italy: IEEE, 2012. p. 368–373. Citation on page 35.

GE, B.; HIPEL, K. W.; YANG, K.; CHEN, Y. A data-centric capability-focused approach for system-of-systems architecture modeling and analysis. **Systems Engineering**, Wiley Subscription Services, Inc., A Wiley Company, v. 16, n. 3, p. 363–377, 2013. Citations on pages 92, 94, and 95.

GEZGIN, T.; ETZIEN, C.; HENKLER, S.; RETTBERG, A. Towards a Rigorous Modeling Formalism for Systems of Systems. In: **15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW 2012)**. Shenzhen, China: IEEE, 2012. p. 204–211. Citations on pages 34 and 35.

GOKHALE, A.; BALASUBRAMANIAN, K.; KRISHNA, A. S.; BALASUBRAMANIAN, J.; EDWARDS, G.; DENG, G.; TURKAY, E.; PARSONS, J.; SCHMIDT, D. C. Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. **Science of Computer Programming**, v. 73, n. 1, p. 39–58, Sep. 2008. Citations on pages 17, 18, 19, 23, 24, 35, and 36.

GOMES, C. Foundations for continuous time hierarchical co-simulation. In: **ACM Student Research Competition at MODELS**. Saint Malo, France: CEUR, 2016. p. 7–13. Citations on pages 31 and 157.

GOMES, C.; THULE, C.; BROMAN, D.; LARSEN, P. G.; VANGHELUWE, H. Co-simulation: State of the art. **CoRR**, abs/1702.00686, 2017. Citation on page 103.

GONCALVES, M. B.; CAVALCANTE, E.; BATISTA, T.; OQUENDO, F.; NAKAGAWA, E. Y. Towards a conceptual model for software-intensive system-of-systems. In: **IEEE International Conference on Systems, Man, and Cybernetics (SMC 2014)**. San Diego, USA: IEEE, 2014. p. 1605–1610. Citation on page 1.

GONZALEZ, A.; LUNA, C.; DANIELE, M.; CUELLO, R.; PEREZ, M. Towards an automatic model transformation mechanism from UML state machines to DEVS models. **CLEI Electronic Journal**, v. 18, n. 2, 2015. Citation on page 99.

GRACIANO NETO, V. V. Validating emergent behaviors in systems-of-systems through model transformations. In: **ACM Student Research Competition at MODELS**. Saint Malo, France: CEUR, 2016. p. 1–6. Citations on pages 19, 13, 24, 35, 37, 104, 107, and 123.

_____. A model-based approach towards the building of trustworthy software-intensive systems-of-systems. In: **39th International Conference on Software Engineering Companion (ICSE-C 2017)**. Buenos Aires, Argentina: IEEE Press, 2017. p. 425–428. Citations on pages 19, 20, 37, 49, 100, 107, and 133.

GRACIANO NETO, V. V.; CAVALCANTE, E.; HACHEM, J. E.; SANTOS, D. S. On the interplay of business process modeling and missions in systems-of-information systems. In: **IEEE/ACM Joint 5th International Workshop on Software Engineering for Systems-of-Systems and 11th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (JSOS 2017)**. Buenos Aires, Argentina: IEEE, 2017. p. 72–73. Citation on page 163.

GRACIANO NETO, V. V.; GARCÉS, L.; GUESSI, M.; PAES, C.; MANZANO, W.; OQUENDO, F.; NAKAGAWA, E. Y. ASAS: An approach to simulate and evaluate systems-of-systems software architectures. **Journal of Software and Systems**, X, n. Y, p. 1 – 20, 2018. Submitted. Citation on page 13.

_____. ASAS: An approach to support simulation of smart systems. In: **51st Hawaii International Conference on System Sciences (HICSS 2018)**. Big Island, Hawaii, USA: IEEE, 2018. p. 5777–5786. Citations on pages 12, 13, 35, 37, 38, 94, 100, 107, 110, 123, 129, and 159.

GRACIANO NETO, V. V.; GUESSI, M.; OLIVEIRA, L. B. R.; OQUENDO, F.; NAKAGAWA, E. Y. Investigating the model-driven development for systems-of-systems. In: **8th European Conference on Software Architecture Workshops (ECSAW 2014)**. Vienna, Austria: ACM, 2014. p. 22:1–22:8. Citations on pages 6, 8, 13, 19, 22, 31, 40, 124, 125, 131, and 153.

GRACIANO NETO, V. V.; MANZANO, W.; GARCÉS, L.; GUESSI, M.; ; OLIVEIRA, B.; VOLPATO, T.; NAKAGAWA, E. Y. Back-SoS: Towards a model-based approach to address architectural drift in systems-of-systems. In: **The 33rd ACM/SIGAPP Symposium On Applied Computing (SAC 2018)**. Pau, France: ACM, 2018. p. 1–3. Citations on pages 13 and 160.

GRACIANO NETO, V. V.; MANZANO, W.; GARCÉS, L.; GUESSI, M.; OLIVEIRA, B.; VOLPATO, T.; NAKAGAWA, E. Y. Realigning descriptive and prescriptive architectural models of systems-of-systems. **IEEE Systems Journal**, X, n. Y, p. 1 – 20, 2018. Submitted. Citations on pages 12 and 13.

GRACIANO NETO, V. V.; MANZANO, W.; ROHLING, A.; VOLPATO, T.; NAKAGAWA, E. Y. Externalizing patterns for simulation in software engineering of systems-of-systems. In: **ACM/SIGAPP Symposium On Applied Computing (SAC 2018)**. Pau, France: ACM, 2018. p. 1–8. Citations on pages 12, 13, and 160.

GRACIANO NETO, V. V.; OQUENDO, F.; NAKAGAWA, E. Y. Systems-of-Systems: Challenges for Information Systems Research in the Next 10 Years. In: **Big Research Challenges in Information Systems in Brazil (2016-2026) at Brazilian Symposium on Information Systems GRANDSI-BR/SBSI**. Florianópolis, Brazil: SBC, 2016. p. 1–3. Citation on page 4.

_____. Smart Systems-of-Information Systems: Foundations and an Assessment Model for Research Development. In: _____. **Grand Challenges in Information Systems for the Next 10 years**. Porto Alegre, Brazil: Brazilian Computer Society, 2017. p. 1–12. Citation on page 163.

GRACIANO NETO, V. V.; PAES, C. E.; GARCÉS, L.; GUESSI, M.; OQUENDO, F.; NAKAGAWA, E. Y. Stimuli-SoS: A model-based approach to derive stimuli generators in simulations of software architectures of systems-of-systems. **Journal of the Brazilian Computer Society**, v. 23, n. 1, p. 13:1–13:22, 2017. Citations on pages 12, 13, 35, 107, 112, 123, and 160.

GRACIANO NETO, V. V.; PAES, C. E. B.; OQUENDO, F.; NAKAGAWA, E. Y. Supporting simulation of systems-of-systems software architectures by a model-driven derivation of a stimulus generator. In: **9th Workshop on Distributed Development, Software Ecosystems and Systems of Systems (WDES 2016)**. Maringá, Brazil: SBC, 2016. p. 61–70. Citations on pages 19, 12, 13, 30, 31, 67, 123, and 160.

GRAHAM, B. **Nature's Patterns - Exploring Her Tangled Web**. New York, USA: FreshVista, 2013. Citations on pages 3, 5, and 102.

GRAY, J.; RUMPE, B. Models in simulation. **Software & Systems Modeling**, v. 15, n. 3, p. 605–607, 2016. Citation on page 51.

GREENYER, J.; RIEKE, J. Applying advanced tgg concepts for a complex transformation of sequence diagram specifications to timed game automata. In: **4th International Conference on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011)**. Budapest, Hungary: Springer-Verlag, 2012. p. 222–237. Citation on page 17.

GRIENDLING, K.; MAVRIS, D. N. Development of a dodaf-based executable architecting approach to analyze system-of-systems alternatives. In: **32nd Aerospace Conference (AeroConf 2011)**. Big Sky, USA: IEEE, 2011. p. 1–15. Citations on pages 24, 93, and 94.

GROGAN, P.; WECK, O. de. Infrastructure System Simulation Interoperability Using the High-Level Architecture. **IEEE Systems Journal**, IEEE, PP, n. 99, p. 1–12, 2015. Citations on pages 35 and 36.

GUARINIELLO, C.; DELAURENTIS, D. Communications, information, and cyber security in systems-of-systems: Assessing the impact of attacks through interdependency

analysis. **Procedia Computer Science**, v. 28, n. Supplement C, p. 720 – 727, 2014. 2014 Conference on Systems Engineering Research. Citations on pages 94 and 97.

_____. Integrated analysis of functional and developmental interdependencies to quantify and trade-off ilities for system-of-systems design, architecture, and evolution. **Procedia Computer Science**, v. 28, n. Supplement C, p. 728 – 735, 2014. 2014 Conference on Systems Engineering Research. Citation on page 97.

GUESSI, M.; GRACIANO NETO, V. V.; BIANCHI, T.; FELIZARDO, K. R.; OQUENDO, F.; NAKAGAWA, E. Y. A systematic literature review on the description of software architectures for systems of systems. In: **30th Symposium On Applied Computing (SAC 2015)**. Salamanca, Spain: ACM, 2015. p. 1433–1440. Citations on pages 1, 6, 22, 23, 28, 40, 41, 98, 103, and 104.

GUESSI, M.; OQUENDO, F.; NAKAGAWA, E. Y. Checking the architectural feasibility of systems-of-systems using formal descriptions. In: **11th System of Systems Engineering Conference (SoSE 2016)**. Kongsberg, Norway: IEEE, 2016. p. 1–6. Citations on pages 35, 37, and 46.

GURGEL, A.; MACIA, I.; GARCIA, A.; STAA, A. von; MEZINI, M.; EICHBERG, M.; MITSCHKE, R. Blending and reusing rules for architectural degradation prevention. In: **13th International Conference on Modularity (Modularity 2014)**. Lugano, Switzerland: ACM, 2014. p. 61–72. Citations on pages 7 and 27.

GURP, J. van; BOSCH, J. Design erosion: problems and causes. **Journal of Systems and Software**, v. 61, n. 2, p. 105 – 119, 2002. Citation on page 27.

HACHEM, J. E.; PANG, Z. Y.; CHIPRIANOV, V.; BABAR, A.; ANIORTÉ, P. Model driven software security architecture of systems-of-systems. In: **23rd Asia-Pacific Software Engineering Conference (APSEC 2016)**. Hamilton, New Zealand: IEEE, 2016. p. 89–96. Citation on page 162.

HAMRI, M.; MESSOUCI, R.; FRYDMAN, C. Discrete event design patterns. In: **1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS 2013)**. Montreal, Canada: ACM, 2013. p. 349–354. Citation on page 99.

HAMRI, M. E.; BAATI, L. On using design patterns for devs modeling and simulation tools. In: **4th Spring Simulation Multiconference (SpringSim 2010)**. Orlando, Florida: Society for Computer Simulation International (SCSI), 2010. p. 121:1–121:9. Citation on page 99.

HANNACHI, M. A.; RODRIGUEZ, I. B.; DRIRA, K.; HERNANDEZ, S. E. P. Gmte: A tool for graph transformation and exact/inexact graph matching. In: KROPATSCH, W. G.; ARTNER, N. M.; HAXHIMUSA, Y.; JIANG, X. (Ed.). **9th IAPR-TC Workshop on Graph-based Representations (GbR 2013)**. Vienna, Austria: Springer Berlin Heidelberg, 2013. p. 71–80. Citation on page 98.

HASSELBRING, W.; REUSSNER, R. Toward trustworthy software systems. **Computer**, v. 39, n. 4, p. 91–92, April 2006. Citations on pages 11 and 12.

HAUSE, M. Model-based system of systems engineering with updm. In: **20th Annual International Symposium of the International Council on Systems Engineering (INCOSE 2010)**. Chicago, Illinois, USA: INCOSE, 2010. v. 1, p. 580–594. Citation on page 24.

_____. The unified profile for DoDAF/MODAF (UPDM) enabling systems of systems on many levels. In: **4th IEEE International Systems Conference (SySCon 2010)**. Xiamen, China: IEEE, 2010. p. 426–431. Citation on page 24.

HEHENBERGER, P.; VOGEL-HEUSER, B.; BRADLEY, D.; EYNARD, B.; TOMIYAMA, T.; ACHICHE, S. Design, modelling, simulation and integration of cyber physical systems: Methods and applications. **Computers in Industry**, v. 82, p. 273 – 289, 2016. Citation on page 132.

HELDAL, R.; PELLICCIONE, P.; ELIASSON, U.; LANTZ, J.; DEREHAG, J.; WHITTLE, J. Descriptive vs prescriptive models in industry. In: **ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)**. Saint-Malo, France: ACM/IEEE, 2016. p. 216–226. Citation on page 27.

HELLESTRAND, G. Engineering safe autonomous mobile systems of systems using specification (model) based systems architecture amp; engineering. In: **7th IEEE International Systems Conference (SysCon 2013)**. Orlando, FL, USA: IEEE, 2013. p. 599–605. Citations on pages 34 and 35.

HIMSS. **Definition of Interoperability**. Chicago, USA, 2013. Available at: <http://www.himss.org/library/interoperabilitystandards/what-is>. Last Access: August 2017. Citation on page 1.

HOFMEISTER, C.; KRUCHTEN, P.; NORD, R. L.; OBBINK, H.; RAN, A.; AMERICA, P. A general model of software architecture design derived from five industrial approaches. **Journal of Systems and Software**, Elsevier, v. 80, n. 1, p. 106–126, 2007. Citation on page 107.

HOLDEN, T.; DICKERSON, C. A ROSETTA framework for live / synthetic aviation tradeoffs: Preliminary Report. In: **8th International Conference on System of Systems Engineering (SoSE 2013)**. Wailea-Makena , HI, USA: IEEE, 2013. p. 218–223. Citations on pages 35 and 37.

HORITA, F. E.; ALBUQUERQUE, J. P. de; DEGROSSI, L. C.; MENDIONDO, E. M.; UEYAMA, J. Development of a spatial decision support system for flood risk management in Brazil that combines volunteered geographic information with wireless sensor networks. **Computers & Geosciences**, v. 80, p. 84 – 94, 2015. Citations on pages 68 and 144.

HU, J.; HUANG, L.; CAO, B.; CHANG, X. Executable Modeling Approach to Service Oriented Architecture Using SoaML in Conjunction with Extended DEVSML. In: **11th IEEE International Conference on Services Computing (SCC 2014)**. Anchorage, Alaska, USA: IEEE, 2014. p. 243–250. Citations on pages 35 and 43.

_____. Extended DEVSML as a Model Transformation Intermediary to Make UML Diagrams Executable. In: **26th International Conference on Software Engineering**

and Knowledge Engineering (SEKE 2014). Vancouver, Canada: Knowledge Systems institution, 2014. Citation on page 99.

______. SPDML: Graphical Modeling Language for Executable Architecture of Systems. In: **1st International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC 2014)**. Shangai, China: IEEE, 2014. p. 248–255. Citation on page 35.

HU, J.; HUANG, L.; CHANG, X.; CAO, B. A Model Driven Service Engineering Approach to System of Systems. **8th Annual IEEE Systems Conference (SysCon 2014)**, IEEE, Ottawa, Ontario, Canada, p. 136–145, 2014. Citation on page 35.

HUYNH THOMAS, V.; OSMUNDSON JOHN, S. **A Systems Engineering Methodology for Analyzing Systems of Systems Using the Systems Modeling Language (SysML)**. Monterey, CA, USA, 2006. Available at: https://calhoun.nps.edu/handle/10945/40375. Citations on pages 93 and 94.

ICS-CERT. **The Future of Smart Cities: Cyber-physical Infrastructure Risk**. USA, 2015. Citation on page 2.

IEEE. IEEE Standard for Modeling and Simulation High Level Architecture (HLA) - Framework and Rules. **IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)**, p. 1–38, Aug 2010. Citations on pages 35, 36, 96, and 155.

______. IEEE Standard for System and Software Verification and Validation. **IEEE Std 1012-2012**, May 2012. Citation on page 101.

IEEE Computer Society. **Software Engineering Body of Knowledge (SWEBOK) - V3**. EUA: Angela Burgess, 2014. Available: <http://www.swebok.org/>. Citation on page 101.

INCOSE. **The Guide to the Systems Engineering Body of Knowledge (SEBoK)**. San Diego, CA, 2016. Citations on pages 1, 2, 5, 6, and 30.

INPE. **Sistema Integrado de Dados Ambientais**. : Instituto Nacional de Pesquisas Espaciais, 2017. <http://sinda.crn2.inpe.br/PCD/SITE/novo/site/index.php>. Accessed: 2017-12-30. Citations on pages 19 and 74.

INVERARDI, P.; TIVOLI, M. Automatic synthesis of modular connectors via composition of protocol mediation patterns. In: **35th International Conference on Software Engineering (ICSE 2013)**. San Francisco, CA, USA: IEEE, 2013. p. 3–12. Citations on pages 48 and 111.

IQBAL, M. Z.; ARCURI, A.; BRIAND, L. Environment modeling and simulation for automated testing of soft real-time embedded software. **Software & Systems Modeling**, v. 14, p. 483–524, 2015. Citation on page 152.

ISO. ISO/IEC/IEEE Systems and software engineering – Architecture description. **ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)**, p. 1–46, Dec 2011. Citations on pages 1, 2, 22, 43, and 133.

JAMSHIDI, M. System of systems engineering - new challenges for the 21st century. **IEEE Aerospace and Electronic Systems Magazine**, v. 23, n. 5, p. 4–19, May 2008. Citation on page 1.

_____. **System of Systems Engineering - Innovations for 21st century**. Hoboken, New Jersey, USA: Wiley, 2009. Citations on pages 1 and 138.

JÉRON, T.; MARCHAND, H.; GENC, S.; LAFORTUNE, S. Predictability of Sequence Patterns in Discrete Event Systems. In: **17th IFAC World Congress (IFACWC 2008)**. Seoul, South Korea: Elsevier, 2008. p. 537–543. Citation on page 99.

JOHNSON, S. B. System health management. In: RAINEY, L. B.; TOLK, A. (Ed.). **Modeling and Simulation Support for System of Systems Engineering Applications**. Hoboken, New Jersey, USA: Wiley, 2015. p. 131–144. Citation on page 162.

JOUAULT, F.; KURTEV, I. Transforming Models with ATL. In: **Satellite Events at the MoDELS 2005**. Montego Bay, Jamaica: Springer-Verlag, 2006. p. 128–138. Citation on page 95.

KASSAB, M.; NEILL, C.; LAPLANTE, P. State of practice in requirements engineering: contemporary data. **Innovations in Systems and Software Engineering**, Springer, London, UK, p. 235–241, 2014. Citation on page 4.

KAZMAN, R.; SCHMID, K.; NIELSEN, C.; KLEIN, J. Understanding patterns for system of systems integration. In: **8th International Conference on System of Systems Engineering (SoSE 2013)**. Maui, Hawaii, USA: IEEE, 2013. p. 141–146. Citation on page 18.

KEWLEY, R.; COOK, J.; GOERGER, N.; HENDERSON, D.; TEAGUE, E. Federated simulations for systems of systems integration. In: **40th Winter Simulation Conference (WSC 2008)**. Miami, FL, USA: The institution for Operations Research and the Management Sciences, 2008. p. 1121–1129. Citation on page 155.

KITCHEN, N.; KUEHLMANN, A. Stimulus generation for constrained random simulation. In: **9th IEEE/ACM International Conference on Computer-Aided Design (ICCAD '07)**. San Jose, California: IEEE Press, 2007. p. 258–265. Citations on pages 30, 31, and 154.

KLEIN, J.; VLIET, H. van. A systematic review of system-of-systems architecture research. In: **9th International ACM Sigsoft Conference on Quality of Software Architectures (QoSA 2013)**. Vancouver, Canada: ACM, 2013. p. 13–22. Citation on page 23.

KLEPPE, A. G.; WARMER, J.; BAST, W. **MDA Explained: The Model Driven Architecture: Practice and Promise**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. Citation on page 16.

KO, H.-S.; ZAN, T.; HU, Z. Bigul: A formally verified core language for putback-based bidirectional programming. In: **2nd Workshop on Partial Evaluation and Program Manipulation (PEPM 2016)**. St. Petersburg, USA: ACM, 2016. p. 61–72. Citations on pages 34 and 125.

KOREL, B. Automated software test data generation. **IEEE Transactions on software engineering**, IEEE, v. 16, n. 8, p. 870–879, 1990. Citation on page 158.

KRUCHTEN, P. The 4+1 view model of architecture. **IEEE Software**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 12, n. 6, p. 42–50, Nov. 1995. Citation on page 4.

KUMAR, R.; KHAN, S. A.; KHAN, R. A. Revisiting software security: durability perspective. **International Journal of Hybrid Information Technology (SERSC)**, v. 8, n. 2, p. 311–322, 2015. Citations on pages 11 and 12.

KURTEV, I. State of the Art of QVT: A Model Transformation Language Standard. In: SCHÜRR, A.; NAGL, M.; ZÜNDORF, A. (Ed.). **3th International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007) - Revised Selected and Invited Papers**. Kassel, Germany: Springer Berlin Heidelberg, 2008. p. 377–393. Citation on page 95.

LAMSWEERDE, A. V. Goal-oriented requirements engineering: A guided tour. In: IEEE. **9th International Requirements Engineering Conference (RE 2001)**. Toronto, Canada, 2001. p. 249–262. Citation on page 21.

LANA, C. A.; SOUZA, N. M.; DELAMARO, M. E.; NAKAGAWA, E. Y.; OQUENDO, F.; MALDONADO, J. C. Systems-of-systems development: Initiatives, trends, and challenges. In: **XLII Conferencia Latinoamericana de Informática (CLEI 2016)**. Valparaiso, Chile: IEEE Press, 2016. p. 1–10. Citations on pages 8, 19, 31, 40, 152, and 158.

LANG, U.; SCHREINER, R. Model driven security accreditation (MDSA) for agile, interconnected it landscapes. In: **1st ACM Workshop on Information Security Governance (WISG 2009)**. Chicago, Illinois, USA: ACM, 2009. p. 13–22. Citation on page 19.

LARA, J. de; GUERRA, E. Formal support for model driven development with graph transformation techniques. In: **2nd Track on Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones (DSDM 2005)**. Denver, Colorado: CEUR Workshop Proceedings, 2005. Citation on page 17.

LEMOS, R. d.; GUERRA, P. A. d. C.; RUBIRA, C. M. F. A fault-tolerant architectural approach for dependable systems. **IEEE Software**, v. 23, n. 2, 2006. Citation on page 25.

LEMOS, R. de; GACEK, C.; ROMANOVSKY, A. ICSE 2002 workshop on architecting dependable systems. In: **24th International Conference on Software Engineering (ICSE 2002)**. Orlando, Florida: ACM, 2002. p. 673–674. Citations on pages 5 and 25.

LEVENDOVSZKY, T.; KARSAI, G.; MAROTI, M.; LEDECZI, A.; CHARAF, H. Model reuse with metamodel-based transformations. In: GACEK, C. (Ed.). **7th International Conference on Software Reuse (ICSR 2002)**. Austin, TX, USA: Springer Berlin Heidelberg, 2002. p. 166–178. Citation on page 16.

LEWIS, C.; SMITH, R.; BEAULIEU, A. A model driven framework for n-version programming. In: **5th IEEE International Systems Conference (SysCon 2011)**. Montreal, QC, Canada: IEEE, 2011. p. 59–65. Citation on page 19.

LI, X.; VANGHELUWE, H.; LEI, Y.; SONG, H.; WANG, W. A testing framework for devs formalism implementations. In: **2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium (TMS-DEVS 2011)**. Boston, Massachusetts: Society for Computer Simulation International, 2011. p. 183–188. Citation on page 162.

LUCKHAM, D. C.; VERA, J. An event-based architecture definition language. **IEEE Transactions on Software Engineering**, v. 21, n. 9, p. 717–734, 1995. Citation on page 26.

MAGEE, J.; KRAMER, J. Dynamic structure in software architectures. **SIGSOFT Software Engineering Notes**, ACM, New York, NY, USA, v. 21, n. 6, p. 3–14, 1996. Citation on page 26.

MAIER, M. W. Architecting principles for systems-of-systems. **Systems Engineering**, v. 1, n. 4, p. 267–284, 1998. Citations on pages 1, 2, 3, 5, 18, and 138.

MAJD, S.; MARIE-HÉLÈNE, A.; ALOK, M. An architectural model for system of information systems. In: CIUCIU, I.; PANETTO, H.; DEBRUYNE, C.; AUBRY, A.; BOLLEN, P.; VALENCIA-GARCÍA, R.; MISHRA, A.; FENSEL, A.; FERRI, F. (Ed.). **13th On the Move to Meaningful Internet Systems: OTM 2015 Workshops**. Rhodes, Greece: Springer International Publishing, 2015. p. 411–420. Citation on page 163.

MANZANO, W.; GRACIANO NETO, V. V.; NAKAGAWA, E. Y. Dynamic-SoS: An approach for the simulation of system-of-systems dynamic architectures. **The Computer Journal**, Oxford Academic, Oxford, UK, X, n. Y, p. 1 – 16, 2018. Submitted. Citations on pages 12 and 92.

MATLAB. **version 7.10.0 (R2010a)**. Natick, Massachusetts: The MathWorks Inc., 2010. Citation on page 154.

MEDVIDOVIC, N.; TAYLOR, R. N. Software architecture: foundations, theory, and practice. In: ACM. **32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)**. Cape Town, South Africa: IEEE, 2010. p. 471–472. Citation on page 25.

MEINKE, K. Learning-based testing of cyber-physical systems-of-systems: A platooning study. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, v. 10497 LNCS, p. 135–151, 2017. Citation on page 32.

MENS, T.; GORP, P. V. A taxonomy of model transformation. **Electronic Notes in Theoretical Computer Science (ENTCS)**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, v. 152, n. 1, p. 125–142, Mar. 2006. Citation on page 16.

MENS, T.; GORP, P. V.; KARSAI, G.; VARRÓ, D. Applying a model transformation taxonomy to graph transformation technology. In: KARSAI, G.; TAENTZER, G. (Ed.). **4th International Workshop on Graph and Model Transformations (GraMot 2005)**. Tallinn, Estonia: Elsevier, 2005. p. 143–159. Citation on page 17.

MICHAEL, J. B.; DRUSINSKY, D.; OTANI, T. W.; SHING, M.-T. Verification and validation for trustworthy software systems. **IEEE Software**, IEEE Computer Society, Los Alamitos, CA, USA, v. 28, n. 6, p. 86–92, 2011. Citations on pages 5, 11, 12, 28, 31, and 101.

MICHAEL, J. B.; RIEHLE, R.; SHING, M. T. The verification and validation of software architecture for systems of systems. In: **4th IEEE International Conference on System of Systems Engineering (SoSE 2009)**. Albuquerque, USA: IEEE, 2009. p. 1–6. Citations on pages 5, 6, 28, 31, 92, 93, 94, and 101.

MILLER, J.; KINGDOMERJI, J. M. **MDA Guide**. Needham, USA, 2003. Citation on page 16.

MILNER, R. **The Space and Motion of Communicating Agents**. 1st. ed. New York, NY, USA: Cambridge University Press, 2009. Citation on page 98.

MITTAL, S.; MARTIN, J. R. Model-driven systems engineering for netcentric system of systems with DEVS unified process. In: **45th Winter Simulation Conference (WSC 2013)**. Washington DC, USA: Society for Modeling and Simulation International, 2013. p. 1140–1151. Citations on pages 18, 19, 23, and 24.

MITTAL, S.; RAINEY, L. Harnessing Emergence: The Control and Design of Emergent Behavior in System of Systems Engineering. In: **47th Summer Simulation Multi-Conference (SummerSim 2015)**. Chicago, Illinois: SCSI, 2015. p. 1–10. Citations on pages 3, 5, 28, 101, and 131.

MITTAL, S.; ZEIGLER, B. P.; MARTIN, J. L. R.; SAHIN, F.; JAMSHIDI, M. Modeling and simulation for systems of systems engineering. In: JAMSHIDI, M. (Ed.). **System of Systems Engineering**. Washington, USA: John Wiley & Sons, Inc., 2008. p. 101–149. Citations on pages 28 and 29.

MOHAMMADI, N. G.; PAULUS, S.; BISHR, M.; METZGER, A.; KÖNNECKE, H.; HARTENSTEIN, S.; WEYER, T.; POHL, K. Trustworthiness attributes and metrics for engineering trusted internet-based software systems. In: **Third International Conference on Cloud Computing and Services Science (CLOSER 2013)**. Aachen, Germany: Springer International Publishing, 2014. p. 19–35. Citations on pages 4, 11, and 12.

MOISESCU, M. A.; SACALA, I. S.; DUMITRACHE, I.; REPTA, D. Cyber physical systems based model-driven development for precision agriculture. In: **7th Symposium on Model-driven Approaches for Simulation Engineering (Mod4Sim 2017)**. Virginia Beach, Virginia: Society for Computer Simulation International, 2017. p. 6:1–6:11. Citation on page 29.

MOOIJ, A. J.; VOORHOEVE, M. Specification and generation of adapters for system integration. In: LAAR, P. van de; TRETMANS, J.; BORTH, M. (Ed.). **Situation Awareness with Systems of Systems**. Berlin, Germany: Springer, 2013. p. 173–187. Citations on pages 35 and 39.

MORDECAI, Y.; ORHOF, O.; DORI, D. Model-based interoperability engineering in systems-of-systems and civil aviation. **IEEE Transactions on Systems, Man, and Cybernetics: Systems**, PP, n. 99, p. 1–12, 2017. Citation on page 24.

MORVAN, G.; VEREMME, A.; DUPONT, D. Irm4mls: The influence reaction model for multi-level simulation. In: **11th International Workshop on Multi-Agent-Based Simulation (MABS 2010)**. Toronto, Canada: Springer Berlin Heidelberg, 2011. p. 16–27. Citation on page 156.

NAKAGAWA, E. Y.; GONCALVES, M.; GUESSI, M.; OLIVEIRA, L. B. R.; OQUENDO, F. The state of the art and future perspectives in systems of systems software architectures. In: **First International Workshop on Software Engineering for Systems-of-Systems (SESoS 2013)**. Montpellier, France: ACM, 2013. p. 13–20. Citation on page 22.

NAKAGAWA, E. Y.; OQUENDO, F.; AVGERIOU, P.; SANTOS, R. **Joint 5th International Workshop on Software Engineering for SoS and 11th Workshop on Distributed Software Development, Software Ecosystems, and Systems-of-Systems - Colocated with ICSE 2017 - Buenos Aires, Argentina**. Online. Available at: http://sesos-wdes-2017.icmc.usp.br/. Last Access: January 2018. Citations on pages 103 and 104.

NAMI, M.; SURYN, W. Software trustworthiness: Past, present and future. In: YUAN, Y.; WU, X.; LU, Y. (Ed.). **International Conference on Trustworthy Computing and Services (ISCTCS 2012)**. Beijing, China: Springer Berlin Heidelberg, 2013. p. 1–12. Citations on pages 4, 5, 11, and 12.

NEEMA, S.; BAPTY, T.; KINGDOMOS, X. K.; NEEMA, H.; SZTIPANOVITS, J.; KARSAI, G. Model based integration and experimentation of Information Fusion and C2 Systems. In: **12th International Conference on Information Fusion (FUSION 2009)**. Seattle, USA: IEEE, 2009. p. 1958–1965. Citations on pages 19 and 33.

NIELSEN, C. B.; LARSEN, P. G.; FITZGERALD, J.; WOODCOCK, J.; PELESKA, J. Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 48, n. 2, p. 18:1–18:41, Sep. 2015. Citations on pages 2, 5, 8, 19, 22, 28, 31, 101, and 108.

OMG. **Meta Object Facility (MOF) Core Specification Version 2.0**. Object Management Group, 2006. (OMG Available Specification). Available: <http://www.omg.org/cgi-bin/doc?formal/2006-01-01>. Citation on page 16.

OMG. **Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1**. 2011. Available: <http://www.omg.org/spec/QVT/1.1/>. Citation on page 95.

OMG. **UML: Unified Modeling Language**. 2015. Available at: <http://www.omg.org/spec/UML/About-UML/>. Access: January 2018. Citation on page 15.

OMG. **The OMG Systems Modeling Language (SySML) Version 1.5**. 2017. Available at: http://www.omgsysml.org/. (ISO/IEC 19514:2017, Information technology – Object management group systems modeling language (OMG SysML))Last Access: December 2017. Citations on pages 15, 23, and 154.

OPHEL, J.; OPHEL, J. An introduction to the high-level language standard ml. In: LAUER, P. E. (Ed.). **Functional Programming, Concurrency, Simulation and Automated Reasoning**. Berlin, Germany: Springer Berlin Heidelberg, 1993, (Lecture Notes in Computer Science, v. 693). p. 47–70. ISBN 978-3-540-56883-4. Citation on page 16.

OQUENDO, F. π-ADL: An Architecture Description Language Based on the Higher-order Typed π-calculus for Specifying Dynamic and Mobile Software Architectures. **SIGSOFT Software Engineering Notes**, ACM, New York, NY, USA, v. 29, n. 3, p. 1–14, 2004. Citations on pages 15, 26, and 41.

______. Formally Describing the Software Architecture of Systems-of-Systems with SosADL. In: **11th Annual System of Systems Engineering (SOSE 2016)**. Kongsberg, Norway: IEEE, 2016. p. 1–6. Citations on pages 25, 30, 41, 42, 47, 100, 112, 114, 138, 140, and 156.

______. π-Calculus for SoS: A Foundation for Formally Describing Software-intensive Systems-of-Systems. In: **11th IEEE System of Systems Engineering Conference (SoSE 2016)**. Kongsberg, Norway: IEEE, 2016. p. 1–6. Citations on pages 25, 42, 46, and 102.

______. Software architecture challenges and emerging research in software-intensive systems-of-systems. In: **10th European Conference on Software Architecture (ECSA 2016)**. Copenhagen, Denmark: Springer, 2016. p. 3–21. Citations on pages 2, 25, 66, 106, 108, 126, and 152.

______. Architecturally describing the emergent behavior of software-intensive system-of-systems with sosadl. In: **12th System of Systems Engineering Conference (SoSE 2017)**. Waikoloa, USA: IEEE, 2017. p. 1–6. Citation on page 3.

OQUENDO, F.; LEGAY, A. Formal Architecture Description of Trustworthy Systems-of-Systems with SosADL. **ERCIM News**, ERCIM, n. 102, 2015. Citations on pages 4 and 48.

OREIZY, P. *et al.* Issues in modeling and analyzing dynamic software architectures. In: **International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA 1998)**. Villa Favorita, Italy: ACM, 1998. p. 54–57. Citations on pages 25 and 26.

PAVON, J.; GOMEZ-SANZ, J.; PAREDES, A. The SiCoSSySApproach to SoS engineering. In: **6th International Conference on System of Systems Engineering (SoSE 2011)**. Irvine, CA, USA: IEEE, 2011. p. 179–184. Citations on pages 23 and 39.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. **SIGSOFT Software Engineering Notes**, ACM, New York, NY, USA, v. 17, n. 4, p. 40–52, Oct. 1992. Citation on page 27.

PETITDEMANGE, F.; BORNE, I.; BUISSON, J. Assisting the evolutionary development of sos with reconfiguration patterns. In: **10th European Conference on Software Architecture Workshops (ECSAW 2016)**. Copenhagen, Denmark: ACM, 2016. p. 9. Citation on page 100.

PICCOLBONI, L.; PRAVADELLI, G. Simplified stimuli generation for scenario and assertion based verification. In: **15th Latin American Test Workshop (LATW 2014)**. Fortaleza, Brazil: IEEE, 2014. p. 1–6. Citations on pages 30 and 31.

PLAZA, S. M.; MARKOV, I. L.; BERTACCO, V. Toggle: A coverage-guided random stimulus generator. In: **International Workshop on Logic and Synthesis (IWLS 2007)**. San Diego, CA, USA: IEEE, 2007. p. 351–357. Citations on pages 30 and 31.

PLOTKIN, G. D. Structural operational semantics the origins of structural operational semantics. **The Journal of Logic and Algebraic Programming**, v. 60, p. 3 – 15, 2004. Citations on pages 25 and 46.

RAHMAN, N. A. A.; RAMLI, A. R.; LOMBIGIT, L.; ABDULLAH, N. A.; KHALID, M. A. H. Stimulus generation technique for code simulation of fpga based gamma spectroscopy system. In: **International Nuclear Science, Technology & Engineering Conference 2013 (iNuSTEC2013)**. Kuala Lumpur, Malaysia: AIP Publishing, 2014. v. 1584, n. 1, p. 77–83. Citations on pages 30, 31, and 154.

RAJKUMAR, R. R.; LEE, I.; SHA, L.; STANKOVIC, J. Cyber-physical systems: The next computing revolution. In: **47th Design Automation Conference (DAC 2010)**. Anaheim, USA: ACM, 2010. p. 731–736. Citation on page 132.

RAMOS, A.; FERREIRA, J.; BARCELO, J. Lithe: An agile methodology for human-centric model-based systems engineering. **IEEE Transactions on Systems, Man, and Cybernetics: Systems**, v. 43, n. 3, p. 504–521, May 2013. Citations on pages 5 and 6.

RAMOS, A. L.; FERREIRA, J. V.; BARCELO, J. Model-based Systems Engineering: An Emerging Approach for Modern systems. **IEEE Transactions On Systems Man Cybernetics Part C-applications Rev.**, v. 42, n. 1, p. 101–111, 2012. Citations on pages 5, 6, 19, and 40.

RICCI, N.; RHODES, D. H.; ROSS, A. M.; FITZGERALD, M. E. Considering alternative strategies for value sustainment in systems-of-systems. In: **7th IEEE International Systems Conference (SysCon 2013)**. Orlando, FL, USA: IEEE, 2013. p. 725–730. Citation on page 103.

ROAD2SOS. **Road2SoS Project - Roadmaps for Systems-of-Systems Engineering**. 2013. <http://road2sos-project.eu/cms/front_content.php>. Last Access: July 2016. Citation on page 1.

ROMAY, M. P.; CUESTA, C. E.; FERNáNDEZ-SANZ, L. On self-adaptation in systems-of-systems. In: **1st International Workshop on Software Engineering for Systems-of-Systems (SESoS 2013)**. Montpellier, France: ACM, 2013. p. 29–34. Citation on page 26.

ROSS, A. M.; RHODES, D. H. An approach for system of systems tradespace exploration. In: RAINEY, L. B.; TOLK, A. (Ed.). **Modeling and Simulation Support for System of Systems Engineering Applications**. Hoboken, New Jersey, USA: Wiley, 2015. p. 75–98. Citation on page 103.

RUNESON, P.; HöST, M. Guidelines for conducting and reporting case study research in software engineering. **Empirical Software Engineering**, Kluwer Academic Publishers, Hingham, MA, USA, v. 14, n. 2, p. 131–164, Apr. 2009. Citations on pages 8, 63, 85, 141, and 145.

SALEH, M.; ABEL, M.-H. Information Systems: Towards a System of Information Systems. In: **7th International Conference on Knowledge Management and Information Sharing (KMIS 2015)**. Lisbonne, Portugal: Springer, 2015. p. 193–200. Citation on page 163.

SANCHEZ-MONTANES, M. A.; KONIG, P.; VERSCHURE, P. F. M. J. Learning sensory maps with real-world stimuli in real time using a biophysically realistic learning rule. **IEEE Transactions on Neural Networks**, v. 13, n. 3, p. 619–632, May 2002. Citations on pages 30 and 132.

SANTOS, D. S.; OLIVEIRA, B.; ; GUESSI, M.; OQUENDO, F.; DELAMARO, M.; NAKAGAWA, E. Y. Towards the evaluation of system-of-systems software architectures. In: **8th Workshop on Distributed Development, Software Ecosystems and Systems of Systems (WDES 2014)**. Maceió, Brazil: SBC. p. 53 – 57. Citation on page 30.

SANTOS, D. S.; OLIVEIRA, B. R. N.; DURAN, A.; NAKAGAWA, E. Y. Reporting an experience on the establishment of a quality model for systems-of-systems. In: **The 27th International Conference on Software Engineering and Knowledge Engineering (SEKE 2015)**. Pittsburgh, PA, USA: Knowledge Systems institution, 2015. p. 304–309. Citation on page 152.

SAUSER, B.; BOARDMAN, J.; VERMA, D. Systomics: Toward a Biology of System of Systems. **IEEE Transactions on Systems, Man, and Cybernetics**, v. 40, n. 4, p. 803–814, 2010. Citations on pages 28 and 101.

SCHULZ, S.; EWING, T. C.; ROZENBLIT, J. W. Discrete event system specification (devs) and statemate statecharts equivalence for embedded systems modeling. In: **7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2000)**. Edinburgh, Scotland: IEEE, 2000. p. 308–316. Citations on pages 99 and 100.

SCHURR, A.; NAGL, M.; ZUNDORF, A. (Ed.). **Applications of Graph Transformations with Industrial Relevance**. Berlin: Springer-Verlag, 2008. Citation on page 17.

SCHWEIZER, B.; LU, D.; LI, P. Co-simulation method for solver coupling with algebraic constraints incorporating relaxation techniques. **Multibody System Dynamics**, v. 36, n. 1, p. 1–36, 2016. Citation on page 157.

SELIC, B. The pragmatics of model-driven development. **IEEE Software**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 20, n. 5, p. 19–25, Sep. 2003. Citations on pages 16 and 18.

_____. MDE Basics with a UML Focus. In: BERNARDO MARCO, C. V. P. A. (Ed.). **Formal Methods for Model-Driven Engineering**. Bertinoro, Italy: Springer, 2012. p. 1. Citations on pages 15, 40, and 132.

SENDALL, S.; KOZACZYNSKI, W. Model transformation: The heart and soul of model-driven software development. **IEEE Software**, v. 20, n. 5, p. 42–45, 2003. Citations on pages 18 and 31.

SILVA, E.; BATISTA, T.; CAVALCANTE, E. A mission-oriented tool for system-of-systems modeling. In: **3th International Workshop on Software Engineering for Systems-of-Systems (SESoS 2015)**. Florence, Italy: IEEE, 2015. p. 31–36. Citation on page 21.

SILVA, E.; CAVALCANTE, E.; BATISTA, T. Refining missions to architectures in software-intensive systems-of-systems. In: **IEEE/ACM Joint 5th International Workshop on Software Engineering for Systems-of-Systems and 11th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (JSOS 2017)**. Buenos Aires, Argentina: IEEE, 2017. p. 2–8. Citations on pages 21, 35, 37, and 38.

SILVA, E.; CAVALCANTE, E.; BATISTA, T.; OQUENDO, F.; DELICATO, F. C.; PIRES, P. F. On the characterization of missions of systems-of-systems. In: **8th European Conference on Software Architecture Workshops (ECSAW 2014)**. Vienna, Austria: ACM, 2014. p. 26:1–26:8. Citations on pages 1 and 21.

SILVA, E.; CAVALCANTE, E.; BATISTA, T.; OQUENDO, F. Bridging missions and architecture in software-intensive systems-of-systems. In: **21st International Conference on Engineering of Complex Computer Systems (ICECCS 2016)**. Dubai, United Arab Emirates: IEEE, 2016. p. 201–206. Citation on page 21.

SILVA, L. de; BALASUBRAMANIAM, D. Controlling software architecture erosion: A survey. **Journal of Systems and Software**, Elsevier Science Inc., New York, NY, USA, v. 85, n. 1, p. 132–151, Jan. 2012. Citations on pages 7, 27, and 125.

SOYEZ, J.-B.; MORVAN, G.; KINGDOMI, R. M.; DUPONT, D. A Multilevel Agent-Based Approach to model and simulate Systems of Systems. In: **InTraDE project final Workshop**. Lille, France: USTL- Lagis Polytech Lille Cite Scientifique, 2014. Citations on pages 155 and 156.

STARY, C.; WACHHOLDER, D. System-of-systems support - a bigraph approach to interoperability and emergent behavior. **Data & Knowledge Engineering**, v. 105, n. C, p. 155 – 172, 2015. Citation on page 98.

STEINBERG, D.; BUDINSKY, F.; PATERNOSTRO, M.; MERKS, E. **EMF: Eclipse Modeling Framework**. 2. ed. Boston, MA: Addison-Wesley, 2009. Citation on page 38.

STEINHOGL, W. Trustworthy systems of systems - a prerequisite for the digitalization of industry. **ERCIM News**, v. 2015, n. 102, p. 1–2, 2015. Citations on pages 4, 8, and 152.

STEVENS, P. A landscape of bidirectional model transformations. In: LAMMEL, R.; VISSER, J.; SARAIVA, J. a. (Ed.). **Generative and Transformational Techniques in Software Engineering II**. Berlin, Germany: Springer Berlin Heidelberg, 2008, (Lecture Notes in Computer Science, v. 5235). p. 408–424. Citation on page 17.

______. Bidirectional model transformations in QVT: semantic issues and open questions. **Software & Systems Modeling**, Springer-Verlag, v. 9, n. 1, p. 7–20, 2010. Citation on page 17.

STOY, J. E. **Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory**. Cambridge, MA, USA: MIT Press, 1977. Citation on page 46.

SUN, Y.; DEMIREZEN, Z.; KINGDOMMAN, T. L.; MERNIK, M.; GRAY, J. Model Transformations Require Formal Semantics. In: LAWALL, J.; RÉVEILLÈRE, L. (Ed.). **2nd International Workshop on Domain-Specific Program Development (DSPD 2008)**. Nashville, United States: ACM, 2008. p. 5. Citation on page 31.

TANIR, O. Simulation-based software engineering. In: MITTAL, S.; DURAK, U.; ÖREN, T. (Ed.). **Guide to Simulation-Based Disciplines: Advancing Our Computational Future**. Berlin, Germany: Springer International Publishing, 2017. p. 151–166. Citations on pages 103 and 104.

TAYLOR, R. N.; MEDVIDOVIC, N.; DASHOFY, E. M. **Software Architecture - Foundations, Theory, and Practice**. Hoboken, New Jersey, USA: Wiley, 2010. Citations on pages 26 and 27.

TENDELOO, Y. V.; VANGHELUWE, H. An evaluation of DEVS simulation tools. **Simulation**, v. 93, n. 2, p. 103–121, 2017. Citations on pages 28, 29, and 104.

TERRA, R.; VALENTE, M. T.; CZARNECKI, K.; BIGONHA, R. S. Recommending refactorings to reverse software architecture erosion. In: **16th European Conference on Software Maintenance and Reengineering (CSMR 2012)**. Szeged, Hungary: IEEE Computer Society, 2012. p. 335–340. Citation on page 7.

TOLK, A.; DIALLO, S.; TURNITSA, C. Applying the levels of conceptual interoperability model in support of integratability, interoperability and composability for system-of-systems engineering. **Journal of Systemics, Cybernetics and Informatics**, International institution of Informatics and Cybernetics (IIIC), Winter Garden, USA, v. 5, n. 5, p. 65 – 74, 2007. Citation on page 35.

TOMSON, T.; PREDEN, J. Simulating system of systems using mace. In: **15th International Conference on Computer Modelling and Simulation (United KingdomSim 2013)**. Cambridge, United Kingdom: IEEE, 2013. p. 155–160. Citations on pages 29, 94, and 95.

TRUBIANI, C.; MEEDENIYA, I.; CORTELLESSA, V.; ALETI, A.; GRUNSKE, L. Model-based performance analysis of software architectures under uncertainty. In: **9th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA 2013)**. Vancouver, Canada: ACM, 2013. p. 69–78. Citation on page 92.

TU, Z.; ZACHAREWICZ, G.; CHEN, D. Harmonized and reversible development framework for HLA based interoperable application. In: **Symposium on Theory of Modeling and Simulation: DEVS Integrative M&S Symposium (TMS-DEVS 2011)**. Boston, MA, USA: Society for Computer Simulation International, 2011. p. 51–8. Citations on pages 19, 34, and 124.

VALERDI, R.; ROSS, A. M.; RHODES, D. H. A Framework for Evolving System of Systems Engineering. **Crosstalk**, v. 20, n. 10, p. 28–30, 2007. Citations on pages 26 and 27.

VANGHELUWE, H. Foundations of modelling and simulation of complex systems. **Electronic Communications of the EASST**, v. 10, 2008. Citations on pages 28, 29, 30, and 131.

VIERHAUSER, M.; RABISER, R.; GRUNBACHER, P.; SEYERLEHNER, K.; WALLNER, S.; ZEISEL, H. Reminds : A flexible runtime monitoring framework for systems of systems. **Journal of Systems and Software**, v. 112, n. Supplement C, p. 123 – 136, 2016. Citations on pages 94 and 98.

VLISSIDES, J.; HELM, R.; JOHNSON, R.; GAMMA, E. Design patterns: Elements of reusable object-oriented software. **Reading: Addison-Wesley**, v. 49, n. 120, p. 11, 1995. Citations on pages 51 and 54.

WACHHOLDER, D.; STARY, C. Enabling emergent behavior in systems-of-systems through bigraph-based modeling. In: **6th International Conference on Systems of Systems Engineering (SoSE 2011)**. San Antonio, TX, USA: IEEE, 2015. p. 334–339. Citations on pages 5, 24, 28, 98, 101, and 131.

WANG, R.; DAGLI, C. H. Executable system architecting using systems modeling language in conjunction with colored petri nets in a model-driven systems development process. **Systems Engineering**, Wiley, v. 14, n. 4, p. 383–409, 2011. Citation on page 92.

WERMELINGER, M. Towards a chemical model for software architecture reconfiguration. **IEE Proceedings - Software**, IET, v. 145, n. 5, p. 130–136, 1998. Citation on page 26.

WERTZ, J. R.; LARSON, W. J. **Space mission analysis and design**. 3rd illustrated edition. ed. Portland, USA: Microcosm Publishing, 1999. Citation on page 72.

WEYNS, D.; ANDERSSON, J. On the challenges of self-adaptation in systems of systems. In: **the First International Workshop on Software Engineering for Systems-of-Systems**. Montpellier, France: ACM, 2013. p. 47–51. Citation on page 26.

WIEDERHOLD, G. Mediators in the architecture of future information systems. **Computer**, v. 25, n. 3, p. 38–49, March 1992. Citations on pages 25, 42, 48, and 111.

WORTMANN, A.; COMBEMALE, B.; BARAIS, O. A systematic mapping study on modeling for industry 4.0. In: **ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017)**. Saint Malo, France: Springer, 2017. p. 281–291. Citation on page 40.

XIA, X.; WU, J.; LIU, C.; XU, L. A model-driven approach for evaluating system of systems. In: **8th International Conference on Engineering of Complex Computer Systems (ICECCS 2013)**. Singapore: IEEE, 2013. p. 56–64. Citations on pages 22, 28, 94, 95, 101, 126, and 153.

YAHIA, E.; YANG, J.; AUBRY, A.; PANETTO, H. On the move to meaningful internet systems: OTM 2009 Workshops. In: MEERSMAN, R.; HERRERO, P.; DILLON, T. (Ed.). Vilamoura, Portugal: Springer Berlin Heidelberg, 2009. chap. On the Use of Description Logic for Semantic Interoperability of Enterprise Systems, p. 205–215. Citation on page 124.

YAMAGUTI, W.; ORLANDO, V.; PEREIRA, S. Sistema brasileiro de coleta de dados ambientais: status e planos futuros (Brazilian system of environmental data collection: status and future plans). **10th Simpósio Brasileiro de Sensoriamento Remoto (SBSR 2009)**, INPE, Foz do Iguacu, Brazil, v. 14, p. 1633–1640, 2009. Citation on page 72.

YANG, S.; WILLE, R.; GROBE, D.; DRECHLER, R. Coverage-driven stimuli generation. In: **15th Euromicro Conference on Digital System Design (ECDSD 2012)**. Izmir, Turkey: IEEE, 2012. p. 525–528. Citations on pages 30, 31, and 154.

YUN, W.; SHIN, D.; BAE, D. H. Mutation analysis for system of systems policy testing. In: **IEEE/ACM Joint 5th International Workshop on Software Engineering for Systems-of-Systems and 11th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (JSOS 2017)**. Buenos Aires, Argentina: IEEE, 2017. p. 16–22. Citation on page 32.

ZAN, T.; PACHECO, H.; HU, Z. Writing bidirectional model transformations as intentional updates. In: **36th International Conference on Software Engineering (ICSE Companion 2014)**. Hyderabad, India: ACM, 2014. p. 488–491. Citation on page 17.

ZAPATA, F.; AKUNDI, A.; PINEDA, R.; SMITH, E. Basis path analysis for testing complex system of systems. **Procedia Computer Science**, v. 20, n. Supplement C, p. 256 – 261, 2013. Complex Adaptive Systems. Citation on page 31.

ZEIGLER, B. P.; KIM, T. G.; PRAEHOFER, H. **Theory of Modeling and Simulation**. 2nd. ed. Orlando, FL, USA: Academic Press, Inc., 2000. ISBN 0127784551. Citation on page 46.

ZEIGLER, B. P.; SARJOUGHIAN, H. S.; DUBOZ, R.; SOULI, J.-C. **Guide to Modeling and Simulation of Systems of Systems**. London, United Kingdom: Springer-Verlag London, 2012. Citations on pages 22, 28, 29, 30, 41, 47, 51, 56, 101, 106, 111, 126, 131, 154, 155, and 206.

ZUNIGA-PRIETO, M.; GONZALEZ-HUERTA, J.; INSFRAN, E.; ABRAHAO, S. Dynamic reconfiguration of cloud application architectures. **Software: Practice and Experience**, Wiley Online Library, Hoboken, New Jersey, USA, v. 48, n. 2, p. 327–344, 2018. Citation on page 25.

APPENDIX

A

LIST OF PUBLICATIONS

During the PhD, 23 authored and co-authored papers were published by the candidate. From them, nine were totally related to the thesis, and other 14 are indirect contributions. From the nine, six are full papers (FP): two conferences, one student research competition, one doctoral symposium, two workshops; one is a full journal article (JA); and two are workshop short papers (SP). It is important to highlight that the paper entitled *Supporting Simulation of Systems-of-Systems Software Architectures by a Model-Driven Derivation of a Stimulus Generator* received a best paper award. From the other contributions, five were full conferences and workshop papers, six short conference and workshop papers, two book chapters (BC), and one technical report (TR). All of these papers were totally related to advances in software engineering for SoS. Five other authored and co-authored article journals are already submitted to evaluation at the thesis submission time.

The list of publications is displayed, as follows:

Publications related to the solutions developed in the thesis:

1. (FP) **Graciano Neto, V. V.**; Guessi, M.; Oliveira, L. B. R.; Oquendo, F.; Nakagawa, E. Y. *Investigating the Model-Driven Development for Systems-of-Systems.* In: Proceedings of the 8th European Conference on Software Architecture Workshops, Vienna, Austria, 2014, ACM, p. 22:1–22:8.
2. (SP) **Graciano Neto, V. V.**; Guessi, M. ; Oliveira, L. B. R. ; Garcés, L. ; Nakagawa, E. Y. ; Oquendo, F. . *A Conceptual Map of Model-Driven Development for Systems-of-Systems.* In: 9th Brazilian Workshop on Distributed Software Development, Software Ecosystems and Systems-Systems (WDES 2015), Belo Horizonte, Brazil, 2015, SBC, p. 89–92.

3. (FP) **Graciano Neto, V. V.**; Paes, C. E. B. ; Oquendo, F. ; Nakagawa, E. Y. *Supporting Simulation of Systems-of-Systems Software Architectures by a Model-Driven Derivation of a Stimulus Generator*. In: 11th Brazilian Workshop on Distributed Development, Software Ecosystems, and Systems-of-Systems (WDES 2016), Maringá, Brazil, 2016, SBC, p. 61–70 (Best paper award).

4. (FP) **Graciano Neto, V. V.**. *Validating Emergent Behaviors in Systems-of-Systems through Model Transformations*. In: ACM Student Research Competition@MODELS, Saint Malo, France, 2016, CEUR, p. 1–6.

5. (JA) **Graciano Neto, V. V.**; Paes, C. E. B. ; Garcés, L. ; Guessi, M. ; Manzano, W. ; Oquendo, F. ; Nakagawa, E. Y. . *Stimuli-SoS: a model-based approach to derive stimuli generators for simulations of systems-of-systems software architectures*. Journal of the Brazilian Computer Society, v. 23, p. 1-22, 2017. Springer.

6. (FP) **Graciano Neto, V. V.**. *A Model-Based Approach Towards the Building of Trustworthy Software-Intensive Systems-of-Systems*. In: IEEE/ACM 39th International Conference on Software Engineering Companion (ICSEC), Buenos Aires, Argentina, 2017, IEEE, p. 425–428.

7. (FP) **Graciano Neto**, V. V.; Garcés, L.; Guessi, M.; Paes, C.; Manzano, W.; Oquendo, F.; Nakagawa, E. *ASAS: An Approach to Support Simulation of Smart Systems*. In: 51st Hawaii International Conference on System Sciences (HICSS 2018), Hawaii's Big Island, USA, 2018, IEEE, p. 5777–5786.

8. (FP) **Graciano Neto, V. V.**; Manzano, W.; Rohling, A.; Volpato, T., and Nakagawa, E.. *Externalizing Patterns for Simulation in Software Engineering of Systems-of-Systems*. In: The 33rd ACM/SIGAPP Symposium On Applied Computing (SAC 2018), Pau, France, 2018, ACM, p. 1–8.

9. (SP) **Graciano Neto, V. V.**; Manzano, W.; Garcés L.; Guessi, M.; Oliveira, B.; Volpato, T., and Nakagawa, E.. *Back-SoS: Towards a Model-based Approach to Address Architectural Drift in Systems-of-Systems*. In: The 33rd ACM/SIGAPP Symposium On Applied Computing (SAC 2018), Pau, France, 2018, ACM, p. 1–3.

Publications related to the scientific domain of the thesis:

1. (FP) Guessi, M., **Graciano Neto, V. V.**, Bianchi, T., Felizardo, K. R., Oquendo, F., Nakagawa, E. Y. *A Systematic Literature Review on the Description of Software Architectures for Systems of Systems*. In: The 30rd ACM/SIGAPP Symposium On Applied Computing (SAC 2015), Salamanca, Spain, 2015, ACM, p. 1433–1440.

2. (SP) **Graciano Neto, V. V.**, Garcés, L., Guessi, M., Oliveira, L. B. R., Oquendo, F.. *On the Equivalence between Reference Architectures and Metamodels*. In: 1st International Workshop on Exploring Component-based Techniques for Constructing Reference Architectures (CobRA 2015), Montréal, Canada, 2015, IEEE, p. 21–24.

3. (SP) **Graciano Neto, V. V.**; Garcés, L. ; Boscarioli, C. ; Nakagawa, E. Y. . *Investigating Issues of Human-Computer Interaction for Systems-of-Systems*. In: 9th Brazilian Workshop on Distributed Software Development, Software Ecosystems and Systems-Systems (WDES 2015), Belo Horizonte, Brazil, 2015, SBC, p. 99–100.

4. (FP) PAES, C. E. B. ; **Graciano Neto, V. V.** ; Oquendo, F. ; Nakagawa, E. Y. *Experience Report and Challenges for Systems-of-Systems Engineering: A Real Case in the Brazilian Defense Domain*. In: 10th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (WDES 2016), Maringá, Brazil, 2016, SBC, p. 41–50.

5. (SP) **Graciano Neto, V. V.**; Oquendo, F. ; Nakagawa, E. Y. *Systems-of-Systems: Challenges for Information Systems Research in the Next 10 Years*. In: Big Research Challenges in Information Systems in Brazil (2016-2026) - Brazilian Symposium on Information Systems (GranDSI-BR/SBSI 2016), Florianópolis, Brazil, 2016, SBC, p. 1-3.

6. (SP) Basso, F. ; Oliveira, T. ; Werner, C. ; **Graciano Neto, V. V.**; Oquendo, F.; Nakagawa, E. Y. *Criteria for Description of MDE Artifacts*. In: 10th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (WDES 2016), Maringá, Brazil, 2016, SBC, p. 80–84.

7. (BC) **Graciano Neto, V. V.**; Oquendo, F.; Nakagawa, E. Y. *Smart Systems-of-Information Systems: Foundations and an Assessment Model for Research Development*. In: Renata Araujo; Rita Maciel; Clodis Boscarioli. (Editors). I GranDSI-BR - Grand Research Challenges in Information Systems in Brazil 2016-2026. 1ed. Porto Alegre: SBC, 2017, v. 1, p. 13–24.

8. (SP) **Graciano Neto, V. V.**; Cavalcante, E.; Hachem, J. E.; Santos, D. S. *On the Interplay of Business Process Modeling and Missions in Systems-of-Information Systems*. In: IEEE/ACM Joint 5th International Workshop on Software Engineering for Systems-of-Systems and 11th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (JSOS), Buenos Aires, Argentina, 2017, ACM, p. 72–73.

9. (FP) Nakagawa, E. Y.; Dias, D.; Horita, F.; Affonso, F.; Abdalla, G., Duarte, I., Felizardo, K., Garcés, L., Oliveira, L., Gonçalves, M., Allian, A., Morais, M., Guessi, M., Silva, N., Bianchi, T., Volpato, T. **Graciano Neto, V. V.**, Zani, V., Manzano, W., Oliveira, B., Sena, B., Paes, C., Lana, C., Feitosa, D.; et al. *Software architecture and reference architecture of software-intensive systems and systems-of-systems.* In: 11th European Conference on Software Architecture Companion (ECSA Companion 2017), Canterbury, UK, ACM, p. 4–8.

10. (SP) Manzano, W. ; **Graciano Neto, V. V.**; Nakagawa, E. Y. *Simulation of Systems-of-Systems Software Architectures with Dynamic Reconfiguration Support* (In Portuguese). In: International Symposium on Scientific Initiation of USP (SIICUSP 2017), São Carlos, Brazil, 2017, USP, p. 1–1.

11. (BC) **Graciano Neto, V. V.**; Santos, R. P. ; Araujo, R. . *Systems-of Information Systems and Software Ecosystems: Concepts and Applications* (In Portuguese). In: Bruno Zarpelão; Joaquim Uchôa; Heitor Costa; Juliana Greghi. (Org.). Information Systems: SBSI Minicourses 2017. 1ed. Lavras: UFLA, 2017, v. 1, p. 22–41.

12. (FP) **Graciano Neto, V. V.**; Costa, S. L.; Loja, L. F. B. ; Oliveira, J. L. . *Web Enterprise Information Systems Engineering - A Path and the Road Ahead* (Invited Paper). In: Seminar on Research and Development of Web-Centric Computational Platforms (SPDPCCWeb 2017), São João Del Rei, Brazil, , 2017, UFSJDelRei, p. 1–8.

13. (FP) **Graciano Neto, V. V.**; Araujo, R. ; Santos, R. P. *New Challenges in the Social Web: Towards Systems-of-Information Systems Ecosystems.* In: Workshop on Aspects of the Human-Computer Interface in the Social Web (WAIHCWS 2017), Joinville, Brazil. 2017, CEUR Workshop Proceedings, p. 1–12.

14. (TR) **Graciano Neto, V. V.**; Garcés, L. M.; Lana, C.; Boscarioli, C.; Fortes, R.; Nakagawa, E.. *Human-Computer Interaction in a System-of-Systems for Treatment of the Elderly with Parkinson's Disease* (In Portuguese), 2017. Technical Report. ICMC, p. 1–132.

Submitted:

1. (JA) **Graciano Neto, V. V.**; Garcés, L.; Guessi, M.; Paes, C.; Manzano, W.; Oquendo, F.; Nakagawa, E. *A Round-Trip Engineering Approach for Software-Intensive Systems-of-Systems.* Special Issue on "Model Driven Engineering and Reverse Engineering: Research and Practice" in Elsevier - Journal of Systems and Software (JSS), 2018.

2. (JA) **Graciano Neto, V. V.**; Garcés, L.; Guessi, M.; Paes, C.; Manzano, W.; Oquendo, F.; Nakagawa, E. *ASAS: An Approach to Evaluate Functional Requirements in Software-Intensive Systems-of-Systems*. Journal of Software: Practice and Experience, 2018.

3. (JA) Manzano, Wallace; **Graciano Neto, V. V.**; Nakagawa; E. Y. *Dynamic-SoS: An Approach to Support System-of-Systems Dynamic Architectures Simulation*. The Computer Journal, 2018, p. 1–16.

4. (JA) Paes, C. E. B.; **Graciano Neto, V. V.**; Moreira, T.; Nakagawa, E. Y. *Conceptualization of a System-of-Systems in the Defense Domain: An Experience Report in the Brazilian Scenario*. IEEE Systems Journal, 2018, p. 1–13.

5. (JA) Rohling, A. J.; **Graciano Neto, V. V.**; Ferreira, M. G. V.; dos Santos, W. A.; Nakagawa, E. Y. *A Reference Architecture for Satellites Control Systems*. International Journal of Aerospace Engineering, 2018.

APPENDIX

B

SPECIFICATION AND DETAILS ON TRANSFORMATION OF SOSADL MODELS INTO DEVS MODELS

For the reader convenience, this chapter presents details on how a model transformation was specified to map SosADL models into DEVS models. For didactic reasons, such transformation is discussed below in two parts: generation of atomic models, and generation of coupled models. Examples are discussed in regards to a Flood Monitoring SoS.

B.0.1. Generation of atomic models

Listing 11 shows a simplified code of mediator specified in SosADL. This code is mapped into an atomic model written in DEVS depicted in Listing 13. The transformation is performed by the code specified in Xtend available in Listing 12. In Listing 11, data types are defined on Lines 2-6. Duties (in this context, only a name for the designation of gates and mediators) with their respective connections are defined on Lines 8-16. Behavior of the mediator is specified between Lines 18 and 23, and shows that a mediator (i) receives constituents coordinates (Lines 19 and 20), (ii) receives data from the sensors (Line 22) and (iii) forward such data towards a gateway (Line 23). This sequence of actions is performed in a loop.

```
1  mediator Transmitter( distancebetweengates:Distance ) is {
2    datatype Abscissa
3    datatype Ordinate
4    datatype Coordinate is tuple { x:Abscissa, y:Ordinate }
5    datatype Depth
6    datatype Measure is tuple { coordinate:Coordinate, depth:Depth }
```

```
7
8     duty transmit is {
9       connection fromSensors is in { Measure }
10      connection towardsGateway is out { Measure }
11    }
12
13    duty location is {
14      connection fromCoordinate is in { Coordinate }
15      connection toCoordinate is in { Coordinate }
16    }
17
18    behavior transmitting is {
19      via location::fromCoordinate receive coordinate
20      via location::toCoordinate receive coordinate
21      repeat {
22        via transmit::fromSensors receive measure
23        via transmit::towardsGateway send measure
24      }
25    }
26  }
```

Source code 11 – Code in SosADL for a mediator.

Lines 2-6 in Listing 11 represent the definition of data types in SosADL. Data types are transformed by Lines 1-16 in Listing 12 to produce Lines 1-22 in Listing 13. Lines 8-16, which specify the connections and duties (gates) of a mediator in SosADL in Listing 11, are transformed by Lines 18-23 (more specifically, Lines 21 and 27) in Listing 12 to produce Lines 24-27 in Listing 13, whereas Lines 18-23, which represent the behavior of a mediator specified in SosADL in Listing 11, are transformed by Lines 23-24 and 29-31 in Listing 12 to produce Lines 24-42 in Listing 13.

```
1 def compile(DataTypeDecl d)'''
2     <<IF !d.datatype.isADT>>
3         A <<d.name>> has a value!
4         the range of <<d.name>>'s value is Integer!
5         use <<d.name.toFirstLower>> with type <<d.name>>!
6     <<ELSE>>
7         <<d.name.toFirstUpper>> has <<d.datatype.compile>>
8         <<IF (d.datatype as DataType) instanceof TupleType>>
9         <<var e = (d.datatype as TupleType)>>
10        <<FOR p : e.fields>>
11        the range of <<d.name>>'s <<p.name>> is <<(p as FieldDecl).type.
   compile>>!
12        <<ENDFOR>>
13        use <<d.name.toFirstLower>> with type <<d.name>>!
14        <<ENDIF>>
15        <<ENDIF>>
16 '''
17
18 def compile(Element e) {
19 if ((connection.type == INPUT) or (action.type == RECEIVE)) {
20   if(e instanceOf Connection) ports += '''
```

```
21    accepts input on <<connectionName.toFirstUpper>> with type <<dataReceived.
      type>>!'''
22
23    if(e instanceOf Action) transitions += '''passivate in s<<fromState>>!
24    when in s<<fromState>> and receive <<dataReceived>> go to s<<toState>>!'''
25
26 }else if((connection.type == OUTPUT) or (action.type == SEND)){
27    if(e instanceOf Connection) ports += '''generates output on <<this.
      connection.typeName>>
28    with type <<dataSent.type>>!'''
29
30    if(e instanceOf Action) transitions += '''hold in s<<fromState>> for time
      1!
31    after s<<fromState>> output <<this.connection.typeName>>!
32    from s<<fromState>> go to s<<toState>>!
33    '''
34    }
35 }
```

Source code 12 – Transformation code specified in Xtend.

Regarding the transformation of data types, Xtend code in Listing 12 establishes the following strategy: if the type of data is not an ADT (Line 2), by default, it will be converted to an Integer type (Lines 3-5). Xtend accesses the name of the data type available in the Abstract Syntax Tree and substitutes it in the appropriate places in the template of DEVSNL code (Line 3, Listing 12). In DEVSNL, a simple type has a value (as specified in Line 3), and tfhe range of this type (in case of simple types) is Integer (Line 4). A variable of such a type is declared in another statement to be used for processing purposes (Line 5). Statements in DEVSNL end with an exclamation mark. Conversely, if the data type specified in SosADL is a tuple (such as Coordinate (Line 4), and Measure (Line 6) in Listing 11), then Lines 7-15 (Listing 12) are executed. In this case, a type declaration in DEVS receives the name of the data type in SosADL, and their fields through an iteration structure (Lines 10-12 in Listing 12), and lines, such as Lines 11-14 in Listing 13, are generated.

Lines 18-31 in Listing 12 buffer the definitions of ports and state transitions in DEVSNL to be printed at the end of the process (this detail is hidden from the code presented). If the transformation consists of an input transition, the code in Lines 19-25 (Listing 12) are executed. If it consists of an output transition, the Lines 26-31 (Listing 12) are executed. Since the specification of a state diagram in DEVSNL follows a declarative style, the order of the instructions generated in DESVNL does not matter. The concepts of duty and gate are suppressed in DEVSNL. Therefore, the Xtend code takes the name of the connections in SosADL and uses them (Lines 21 and 27, Listing 12) to create ports in DEVSNL. The type of data received or sent are used in the typification of the data to be transmitted in a DEVS port (Lines 21 and 27, Listing 12). `Receive` instructions in

SosADL create receive transitions in the form of Lines 23-24 in Listing 12, whereas `send` instructions in SosADL generate three lines of code in DEVSNL - one that holds in a state for time 1 second (this time was established as default), another that produces the output of some data, and another that performs the state transition to a next state (Lines 29-31). Some of the variables are global variables, and their declaration was hidden.

```
1 A Distance has a value!
2 the range of Distance's value is Integer!
3 use distance with type Distance!
4
5 A Abscissa has a value!
6 the range of Abscissa's value is Integer!
7 use abscissa with type Abscissa!
8 A Ordinate has a value!
9 the range of Ordinate's value is Integer!
10 use ordinate with type Ordinate!
11 Coordinate has x and y!
12 the range of Coordinate's x is Abscissa!
13 the range of Coordinate's y is Ordinate!
14 use coordinate with type Coordinate!
15
16 A Depth has a value!
17 the range of Depth's value is Integer!
18 use depth with type Depth!
19 Measure has coordinate and depth!
20 the range of Measure's coordinate is Coordinate!
21 the range of Measure's depth is Depth!
22 use measure with type Measure!
23
24 accepts input on FromCoordinate with type Coordinate!
25 accepts input on ToCoordinate with type Coordinate!
26 accepts input on FromSensors with type Measure!
27 generates output on Measure with type Measure!
28
29 to start hold in s0 for time 1!
30 hold in s0 for time 1!
31 from s0 go to s1! //Unobservable
32 passivate in s1!
33 when in s1 and receive Coordinate go to s2!
34 passivate in s2!
35 when in s2 and receive Coordinate go to s3!
36 passivate in s3!
37 when in s3 and receive Measure go to s4!
38 hold in s4 for time 1!
39 after s4 output Measure!
40 from s4 go to s5!
41 hold in s5 for time 1!
42 from s5 go to s3! //Unobservable
```

Source code 13 – An atomic model for a Mediator generated in DEVSNL.

Each statement of a behavior in SosADL becomes one or more 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, with no input or output event. Listing 14 illustrates a behavior called `sensing` of a smart sensor, which senses the depth of the water in a river and is a constituent within the context of FMSoS. Line 2 is an assignment statement in which a coordinate called `lps` is assigned to a variable called `sensorcoordinate`. In Line 3, the `lps` assigned to a `sensorcoordinate` in Line 2 is sent via a connection called a `coordinate` that has a gate called a `location`. Line 6 depicts a `repeat` statement in which the `powerlevel` is received via a connection called `power` in a gate called `energy`. The system receives the battery level to test whether the energy is enough to perform the instructions. Line 8 tests if the power level is above an established threshold.

```
1 behavior sensing is {
2   value sensorcoordinate is Coordinate = lps
3   via location::coordinate send sensorcoordinate
4   via energy::threshold receive powerthreshold
5
6   repeat {
7       via energy::power receive powerlevel
8       if( powerlevel > powerthreshold ) then {
9           choose {
10            via measurement::sense receive data
11            via measurement::measure send tuple{
12               coordinate = lps, depth = data::convert( ) }
13          } or {
14             via measurement::pass receive data
15             via measurement::measure send data
16          } //end choose
17      } // end if
18  } //end repeat
19 } //end sensing
```

Source code 14 – Code in SosADL for a behavior called *sensing* of a smart sensor.

Within the scope of the conditional statement, a `choose` statement (Lines 9 to 16) is triggered if the condition returns a `true` value. System will behave depending on the stimulus received: if it receives a data delivered by its own sensor, Line 10 is executed, and the data is received via a connection called `sense` at the gate called `measurement`. This data is forwarded to the closest mediator towards the gateway via the connection `measure` (Lines 11-12). If the sensor receives a data collected by another sensor and transmitted across the SoS, Line 14 is executed. The data are received via a connection called `pass` at the gate measurement and forwarded via the connection called `measure` at the same gate (Line 15).

A mapping from a SosADL behavior for a DEVS code produces a state diagram. To illustrate this procedure, we depict a state diagram in Figure 40 that corresponds to

the behavior `sensing` in Listing 14. A line of code in SosADL (henceforth, a SosADL statement) can be converted to one or more state transitions in a labeled state diagram to produce a DEVS code. Line 2, for example, is directly mapped into one transition between two states ($s_0 \Rightarrow s_1$) (Figure 40). Line 4 expresses a `receive` assignment, with an analogous treatment ($s_2 \Rightarrow s_3$). This is an output event, and as such, originates a labeled output transition ($s_1 \Rightarrow s_2$). The other one-line statements (we term them `action` statements) in line 4, lines 10 and 11, and 14 and 15 follow the same rationale.

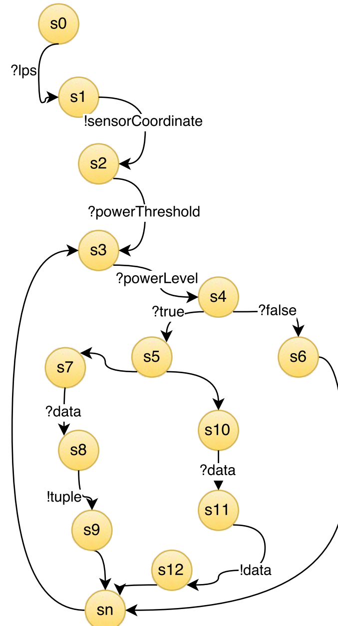

Figure 40 – A labeled state diagram corresponding to a *sensing* behavior extracted from SosADL code.

As every statement within a `repeat` command must be repeated, a new state called $\mathtt{s}_n$ is created for amalgamating the execution flow at the end of the loop, and transiting it from $\mathtt{s}_n$ to $\mathtt{s}_3$, creating a loop. Line 7 depicts the transition `?powerLevel`. Two transitions are created: one for a `true` value returned by the statement, and another for `false` values, i.e., when the battery level is not enough to perform the behavior. Transition to $\mathtt{s}_5$ is created for `?true`, and the transition to $\mathtt{s}_6$ is created for `?false`. Since no `else` statement exists, the flow is forwarded to $\mathtt{s}_n$ spontaneously, i.e., without receiving or sending anything, converging the execution flow. The same procedure is followed for the transformation of simple assignments, and two execution flows are created and chosen depending on which event occurs first. Two flows are created for each set of statements, and finally, the flows are amalgamated again by spontaneous transitions in a state previously created by the `repeat` statement ($\mathtt{s}_n$), and the transformation for the behavior is finalized. A transition ($\mathtt{s}_n$ to $\mathtt{s}_3$) restarts the execution.

```
1 passivate in s0!
```

```
2 when in s0 and receive lps go to s1!
3 hold in s1 for time 1!
4 after s1 output sensorCoordinate!
5 from s1 go to s2!
6 when in s2 and receive powerThreshold go to s3!
7 from sn go to s3!
8 when in s3 and receive powerLevel go to s4!
9 when in s4 and receive true go to s5!
10 when in s5 and receive dataSensed go to s8!
11 hold in s8 for time 1!
12 after s8 output tuple!
13 from s8 go to s9!
14 from s9 go to sn!
15 when in s5 and receive dataPassed go to s11!
16 hold in s11 for time 1!
17 after s11 output data!
18 from s11 go to s12!
19 from s12 go to sn!
20 when in s4 and receive false go to s6!
21 from s6 go to sn!
```

Source code 15 – State diagram code corresponding to a Smart sensor behavior generated in DEVSNL.

Listing 15 shows the mapping of the `sensing` behavior of Listing 14 and its equivalent state diagram in Figure 40 into DEVSNL. Each transition produces a transition in DEVSNL. Input transitions correspond to one line of code such as `when in s0 and receive lps go to s1!`. Output transitions produce three lines of code, of which one holds the flow in the state for a pre-defined time, another produces the output, and a final one produces the transition itself, as in Lines 2, 3, and 4 in Listing 15. Finally, transitions created only for moving the flow from one state to another produce one line in DEVSNL, as Line 19 in Listing 15. Transitions `s5 to s7` and `s5 to s10` are substituted during the transformation to DEVSNL by `s5 to s8` and `s5 to s11`, respectively, for the avoidance of non-determinism.

B.0.2. Generation of coupled models

In ASAS approach, a SoS architectural description specified in SosADL is verified against its metamodel expressed in Xtext[1] during the transformation. If the SosADL code conforms to this metamodel, the code is used as input to an Xtend[2] script that realizes the transformation mechanism and returns a functional code written in DEVS. Coupled models in DEVS specify the way constituent systems exchange data with each other to exhibit an emergent behavior. The code of such coupled models systematically specifies

[1] https://eclipse.org/Xtext/
[2] xtend-lang.org/

which entities are involved in the SoS and how they interact, i.e., which systems send data and which systems receive these data. In SosADL, SoS software architectures ares modeled as coalitions. The correspondences between SosADL and DEVS are summarized in Table 25.

Table 25 – Mapping of SosADL into SES/DEVS

SoS concept	**SosADL**	**SES/DEVS**
SoS	Coalition	Decomposition
Data Types	Data Type	Data Type
Gate/Connection	Gate/Connection	DEVS Port
Interfaces	Binding	Coupling
SoS Architecture	Coalition + Binding	Coupled Model

Constituent Systems. In SosADL, the list of all constituent systems that compose the software architecture of an SoS is represented by a Coalition. By definition, coalitions are alliances of constituents connected via mediators. When translated into DEVS, coalitions are mapped into a DEVS Decomposition, i.e., a statement of the coupled model that systematically lists all the inner structures (e.g., systems, mediators, among others) that form the software architecture of the SoS (ZEIGLER *et al.*, 2012).

Data Types. When a communication is established between constituents, and they start to interoperate, data are exchanged between them. Indeed, SoSADL relies on *typed connections*, i.e., connections with a specific type of data. Data types must be preserved by the transformation and properly converted into DEVS format.

Gate/Connection. Gates are a structure through which connections can be established. Since the notion of connection does not exist in DEVS, each SosADL Connection is mapped as a port in DEVS.

Interfaces. The concept of interface encapsulates the communication between two entities (in this case, systems). Consequently, a detailed analysis of an interface should contain (explicitly or not) functions *send* located in one system element, and *receive* located in another [3]. In SosADL, interfaces are specified through bindings, which correspond to the list of all combinations between output ports and input ports that establish a communication between two entities in a SoS. In DEVS, each one of the bindings is mapped into a coupling, i.e., a statement describes the way information flows between two systems in the SoS.

Sos Architecture. Finally, the software architecture of an SoS is represented as an abstract architecture in SosADL, which specifies a coalition and a set of bindings, and

[3] SEBoK. Guide to the systems engineering body of knowledge, version 1.6, 2016.

subsequently mapped in a coupled model, which is a set containing a decomposition and couplings.

Listing 16 depicts a SosADL code that represents the specification of a software architecture of an FMSoS. In Listing 16, the software architecture of SoS represented comprises four `sensors`, one `gateway`, and four `transmitters` (types of mediators) (Lines 4 to 12). `bindings` (Lines 13 to 23) represent the way connections between constituents and mediators are established through gates, and SoS dynamics for data transmission until a gateway. A sensor collects the `water level` through actuators, encapsulating it with the specific `location` in which the collecting was performed, and with a time stamp. The sensor then transmits the data to the closest mediator, which forwards it to the next sensor, until the gateway has been reached.

```
1 sos FloodMonitoringSos is {
2   architecture FloodMonitoringSosArchitecture( ) is{
3     behavior coalition is compose {
4     sensor1 is Sensor
5     sensor2 is Sensor
6     sensor3 is Sensor
7     sensor4 is Sensor
8     gateway is Gateway
9     mediator1 is Mediator
10    mediator2 is Mediator
11    mediator3 is Mediator
12    mediator4 is Mediator
13    } binding {
14    relay gateway::notification::alert to warning::alert and
15    relay gateway::request to request and
16    unify one { sensor1::measurement::measure }
17       to one { mediator1::fromSensors } and
18    unify one { mediator1::transmit::towardsGateway }
19       to one { sensor2::measurement::pass } and
20    unify one { sensor2::measurement::measure }
21       to one { mediator2::fromSensors } and
22    unify one { mediator2::transmit::towardsGateway }
23       to one { gateway::notification::measure } and
24    unify one { sensor3::measurement::measure }
25       to one { mediator3::fromSensors} and
26    unify one { mediator3::transmit::towardsGateway }
27       to one { sensor4::measurement::pass } and
28    unify one { sensor4::measurement::measure }
29       to one { mediator4::fromSensors} and
30    unify one { mediator4::transmit::towardsGateway }
31       to one { gateway::notification::measure }
32 }  }
```

Source code 16 – Description of an architecture of an FMSoS in SosADL.

In SosADL, a connection is specified as `system :: gate :: connection`. Indeed, the same gate can hold one or more connections. An unification is established for each pair

of sensors with a mediator between them by a `unify` statement (Lines 15-23). According to such statements, an output connection `measure` from the gate `measurement` is linked to the input connection `fromSensors` of the closest mediator. A mediator gathers data from a sensor (Lines 11 to 14) and forwards it to the next sensor. Mediators have an output connection termed as `towardsGateway`. Such connections are linked to the sensors through an input connection called `pass` in the gate `measurement`, which enables it to receive the data transmitted and forward them to the gateway (Lines 16, 18, 20 and 22). Lines 22 and 23 link the output connection of the mediator to the gateway connection called `measure`. In this case, a mediator mediates a constituent and the gateway. The `relay` statement establishes the communication between the SoS and external systems, connecting the notification gate of a gateway to one external connection.

Listing 17 provides the rules for the transformation of a specification of an FMSoS software architecture into a coupled model in DEVS. It depicts three transformation rules: one that receives a SosADL type called `ArchitectureDecl` as input, one that receives an `ArchBehaviorDecl` as input, and one that compiles the `Unify` statements. Lines 1 to 5 produce the first line of the DEVS code which declares a Decomposition. It takes the name of the architecture, puts it in upper case, and delivers the remaining part to next transformation rules[4]. In the second transformation rule (Lines 7 to 20), a list of the systems that compose the architecture is enumerated in the DEVS target model (Lines 12 to 15), thus completing the DEVS Decomposition. Bindings are compiled in the next transformation rule (Lines 21 to 50) invoked in Line 17. The compilation of the unifications, i.e., the specification of the data exchanged between systems involved in the SoS, proceeds as follows: both sender and the receiver names are required for the documentation of the communication between systems in DEVS. They are separated from the data available in the unifications, using the `::` as a marker that splits the String (Lines 23 to 33). However, the type of data transferred between two systems in the SoS must be known so that the simulation code can be specified. This data is not available in the architecture specification in SosADL, but it is available in the specification of the constituents and mediators in SosADL.

Line 35 hides a code that opens a file containing a specification of the connections and their respective data types. The code in Lines 37 to 44 compares the name each pair gate-connection with the gates and connections specified in the coalition, inferring the type of data that they transmit. This data is assigned to the variable `data` when it is found (Line 42). Line 46 shows the format of the output String, with sender, receiver, and data, and Line 49 prints the result. The transformation rule for `Unify` is called as many times

[4] These transformation rules were structured as presented for separation of concerns, reuse, and modularization purposes

```
1 def compile(ArchitectureDecl a) {
2     var String result  =
3     '''From the top perspective, <<a.name.toFirstUpper>>
4     is made of <<a.behavior.compile>>'''
5
6     result
7   }
8
9 def compile(ArchBehaviorDecl a){
10
11   var int size = a.constituents.size
12   var int cont = 0
13   var result  = ''''''
14   for (Constituent c: a.constituents){
15      cont++
16      result +=
17     '''<<IF (cont == size)>> and <<c.name.toFirstUpper>>
18     <<ELSE>><<c.name.toFirstUpper>>, <<ENDIF>>'''
19   }
20   result +='''!'''
21     result += '''<<a.bindings.compile>>'''
22
23   result
24   }
25      override def compile(Unify u){
26
27     var String sender = u.connLeft.compile.toString()
28     val String[] vector = sender.split('::')
29     var int firstIndex = sender.indexOf("::")
30     var String connectionSender = sender.substring(firstIndex+2,sender.length
   )
31     sender = vector.get(0)
32
33     var String receiver = u.connRight.compile.toString()
34     val String[] vector2 = receiver.split('::')
35
36     receiver = vector2.get(0)
37     var String data = "";
38
39     //Code Hidden: Reads connections from a file.
40
41     val String[] vectorConnections = data.split("-")
42     for(String s: vectorConnections){
43       val String[] vectorAux = s.split(";")
44       var String connectionName = vectorAux.get(0).replace(" ","")
45       if(connectionSender.compareTo(connectionName)==0){
46         data = vectorAux.get(1).toFirstUpper
47       }
48     }
49
50     var String result = '''
51       From the top perspective,  <<sender.toFirstUpper>>
52       sends <<data>> to <<receiver.toFirstUpper>>!
53     '''
54     result
55   }
```

Source code 17 – Transformation rules specified in Xtend for the transformation of a SosADL model into DEVS model.

```
1 From the top perspective, FloodMonitoringSosArchitecture is made of
2 Sensor1, Sensor2, Sensor3, Sensor4, Gateway, Mediator1, Mediator2, Mediator3,
3 and Mediator4!
4
5 From the top perspective,  Sensor1 sends Measure to Mediator1!
6 From the top perspective,  Mediator1 sends Measure to Sensor2!
7 From the top perspective,  Sensor2 sends Measure to Mediator2!
8 From the top perspective,  Mediator2 sends Measure to Gateway!
9 From the top perspective,  Sensor3 sends Measure to Mediator3!
10 From the top perspective,  Mediator3 sends Measure to Sensor4!
11 From the top perspective,  Sensor4 sends Measure to Mediator4!
12 From the top perspective,  Mediator4 sends Measure to Gateway!
```

Source code 18 – Coupled Model for FMSoS generated in DEVS.

as there are lines of unifications in the specification of the binding. Each binding specified in SosADL is mapped into one coupling in DEVS. Listing 18 shows the equivalent code derived from the coalition using the transformation rules specified in Xtend depicted in Listing 17. Line 1 shows that the `FloodMonitoringSoSArchitecture` is formed by the same systems specified in the SosADL code. Lines 2 to 9 show the data exchange among all systems and mediators derived from the specification of the coalition. These lines are created by iterating on the unifying statements. One line is created for each of the unifying connections specified in the SosADL model. Finally, DEVS tool converts that code into a simulation model that is executable.

Considering Listing 18, sensors transmit data to their closest mediator (Lines 2, 4, 6, and 8). Then, these mediators receive these data in Lines 3, 5, 7, and 9 forward `Measure` to the next sensors. Since `Sensor2` and `Sensor4` sent their data to `Mediator2` and `Mediator4` respectively (Lines 4 and 6), the gateway is already reached (Lines 5 and 9). When these data arrive in the gateway, their values are tested against a pre-determined depth threshold. If they are higher, the gateway emits a flood alert. Thus, the network of exchanged messages between constituents and the flood alert trigger indicate that the SoS mission, i.e., producing flood alerts, has been accomplished by these constituent system.

B.0.2.1. Dynamic reconfiguration controller structure

Dynamic reconfiguration controller is an artificial architectural element that manages every architectural change that occurs. It is added to the simulation to support the simulation user to perform architectural changes at runtime. From the DEVS simulation model perspective, the reconfiguration controller is an atomic model, that: (i) adds constituents to the simulation, also adding the necessary connections and mediators, maintaining the properties of the initial architecture; (ii) removes the constituents of the simulation, connections and mediators, relinking the remaining constituents to maintain

an operational SoS architecture; (iii) substitutes the constituents of the simulation for another constituent by removing the constituent and replacing it to the architecture being simulated; and (iv) reorganizes the architecture by removing all connections and mediators and thereafter establishing different mediated connections to create a new architectural configuration while retaining the initial architectural properties.

```
1 add library
2 <%
3  import com.ms4systems.devs.core.
    model.impl.CoupledModelImpl;
4  import com.ms4systems.devs.core.
    model.AtomicModel;
5 %>!
6
7 accepts input on Remove with type
     AtomicModelImpl!
8 accepts input on AddSensor!
9 accepts input on AddGateway!
10 accepts input on
    ReorganizeArchitecture!
11
12 use constituents with type
    ArrayList<AtomicModelImpl>!
13 use connections with type
    ArrayList<Connection>!
14 use toRemove with type
    AtomicModelImpl!
15 use auxRecConst with type
    ArrayList<AtomicModelImpl>!
16 use flagAddData with type boolean
    !
17 use originalConstituents with
    type ArrayList<AtomicModelImpl
    >!
18 use simulationTime with type long
    !
19 use contRecSensor with type int!
20 use contRecGateway with type int!
21 use contRecTransmitter with type
    int!
```

Source code 19 – Dynamic reconfigurator controller structure.

For addition of a constituent, it is necessary to send a signal to the controller to add it to the simulation. When an addition is invoked by the controller, mediators are also created to establish the communication between the existing constituents and the new one that is being added to the SoS. Stimuli generators also can be added during this process to feed the new constituents with the data necessary to trigger its interaction within the SoS simulation.

For the removal, a signal is sent to the constituent to be removed. As a response, this constituent sends a reference of its own simulation object to the controller, enabling direct access so that it can be removed. Mediators and connections that communicate

with it are also removed. If necessary, new connections and mediators can be added to reestablish the path inside the SoS architecture that enable the communication with other constituents that communicated with the constituent removed.

Substitution of constituent is a sequence of both removal and addition. In turn, for the reorganization of the architecture, a signal is sent to the controller, which removes all the connections and mediators between the constituents. After that, the controller creates new connections and mediators between constituents to raise a new functional architectural configuration.

Adding Support to Dynamic Reconfiguration through a Model Transformation. All SoSADL elements are mapped to DEVS to create a functional simulation. Transformation rules automatically create the dynamic reconfiguration controller and add it in the simulation model. This controller holds and makes available to the simulation user all the dynamic architecture operators. The canonical changes are addition and removal, i.e., fundamental operations that are bases for any type of change. Replacement of constituent was performed as a sequence of removal and addition. Reorganization was also implemented, leading to deconstruction of the entire architecture, being rebuilt again for a new architectural arrangement.

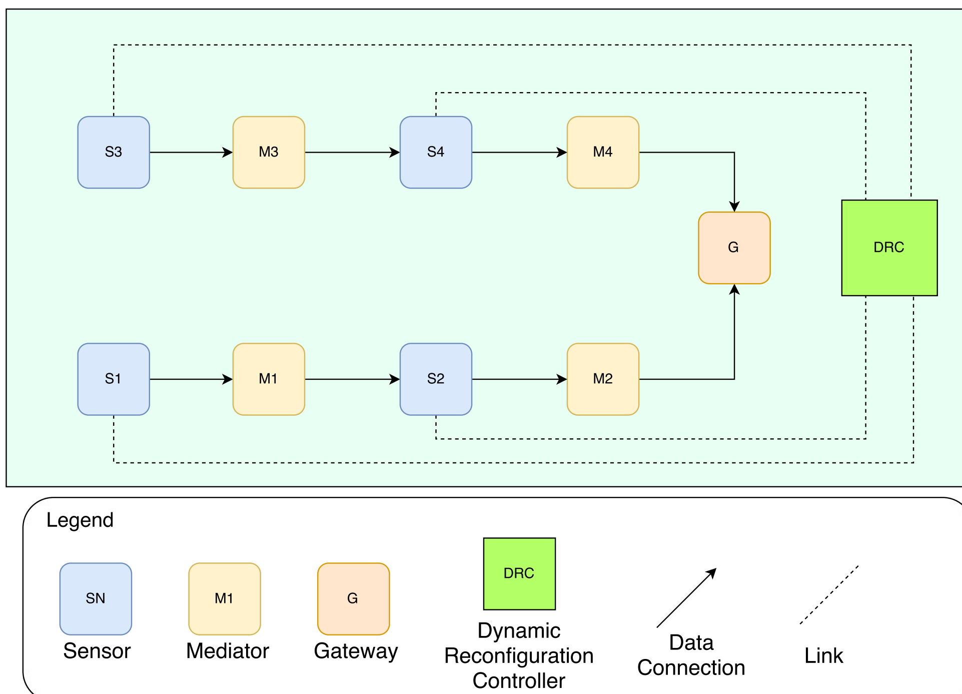

Figure 41 – An illustration of the relation between Dynamic Reconfiguration Controller (DRC) and constituents being simulated.

The model transformation generates three main elements related to dynamic

architecture into the target simulation model, which are:

- **Dynamic reconfiguration controller**: it consists of an atomic model that manages all the changes in the simulation, as shown in Figure 41. For that it manages the connections and mediators between the constituents, so that the new arrangement still remains consistent with the original architecture;
- **Identification flags**: In DEVS, single elements that compose a SoS are atomic models. Mediators and all the types of constituents are handled as the same type of entity. As consequence, it could be possible to put a sensor to play the role of a mediator, i.e., a sensor mediating other sensors, which is not desired as it hampers architectural precision. Hence, two identification flags are inserted into all the atomic models: one to check if the system is a mediator or not (boolean), and another one that is the constituent type name, such as Sensor, Transmitter, or Gateway.
- **Connections of all the constituents with the dynamic reconfiguration controller in the coupled model**: This is necessary to enable the controller to communicate with all the constituents and remove them if necessary.

```
1  def public String dynamicStructure() {
2  if (sfile instanceof SystemDecl || sfile instanceof MediatorDecl) {
3   return '''
4   <<addTransitions>>
5
6   <<removeConstituent>>
7
8   <<addFlags>>
9   '''
10 }else if (sfile instanceof ArchitectureDecl) {
11  createFile
12  return '''
13  <<addCouplingsToArchitecture>>
14  '''
15
16 }else
17  return ''''''
18 }
```

Source code 20 – A transformation excerpt that supports generation of DEVS simulation of SoS software architecture with support to dynamic reconfiguration.

Listing 20 shows an excerpt of code of the model transformation[5]. This method specified in Xtend was created to add dynamic reconfiguration support for all the existing elements of the simulation. The addition of the support to the reconfiguration is done after the compilation of the SoSADL models. All elements of the concrete architecture in

[5] Some parts of the code are hidden for the reader convenience.

SoSADL are iterated, and before each compiled file is created, each element receives some code snippets referring to the communication with the dynamic reconfiguration controller, the identification flags, and the connections that support the controller operations on the simulation model. If the SoSADL architectural element is a mediator or system, they will be handled in a similar way. If the input for the model transformation is an architecture, all the required bindings will be added. In Listing 20, Line 2 shows the transformation code that checks whether `sfile`, which is the model being compiled, is a system or a mediator. If it is one between both options, the method performs the procedures of lines 4-8, where in line 4 will be added external transition so that the model can receive the signal to it be removed, in line 6 output transitions so that it can send its reference to the controller to request for its removal and in line 8 its identification flags. Otherwise, if the element being analyzed is an architecture, all the necessary associations will be added to the coupled model (lines 10 to 16). Next we discuss the protocol and results of our evaluation.

APPENDIX

C

A SATELLITE SPECIFIED IN SOSADL

```
1 library SatelliteAmazonia3 is {
2
3   system SatelliteAmazonia3( lps:Coordinate ) is {
4
5     datatype Image is tuple { name:String, extension:String, content:Binary }
6     datatype CollectedImages is sequence { Image }
7
8     datatype Telecommand is tuple { id:integer, date:Calendar, orbitId:
    integer,
9       name:String, instruction:integer, coordinateToBeMonitored:Coordinate
10    }
11    datatype Binary
12    datatype Orbit
13    datatype Power
14    datatype SatelliteHeight
15    datatype SatelliteTemperature
16    datatype Latitude is Double
17    datatype Longitude is Double
18    datatype SatellitePosition  is tuple { x:Latitude, y:Longitude }
19    datatype Coordinate is tuple { x:Latitude, y:Longitude }
20    datatype Establish
21    datatype Distance  {
22      \\ function that calculates distance hidden for the reader convenience.
23      }
24    }
25
26    \\ More data types definition - hidden for the reader convenience.
27
28
29      gate satelliteState is {
30      environment connection power is in { Power }
31      environment connection orbit is in { Orbit }
32      environment connection temperature is in { Integer }
33      environment connection height is in { Integer }
34    }
35
```

```
36
37      gate operation is {
38        connection telecommand is in { Telecommand }
39      }
40
41
42      gate camera is {
43        environment connection image is in { Image }
44      }
45
46      gate telemetry is {
47        connection telemetry is out { Image }
48
49      }
50
51      gate location is {
52        environment connection coordinateSatellite is in { SatellitePosition }
53        connection coordinate is out { SatellitePosition }
54      }
55
56      gate establish is {
57        connection establishConnection is in {Establish}
58        connection establishConnectionGS is in {Establish}
59      }
60
61      gate notification is {
62        connection terrestrialMeasure is in { TerrestrialData }
63        connection aquaticMeasure is in { AquaticData }
64      }
65
66      behavior main is {
67        value telecommand1 : Telecommand = any
68        value powerThreshold : Power = 20 //battery threshold in 20 percent.
69        value image1 : Image = any
70        value powerNow : Power = any
71        value distanceMax:Distance = 5
72
73        repeat{
74
75          via satelliteState::power receive powerNow
76          if (powerNow > powerThreshold) then {
77            value powerNow = powerNow - 10
78          } else {
79            value powerNow = 100
80          }
81
82          choose {
83            via establish::establishConnectionGS receive establish
84            if(establish = 1) then {
85              via operation::telecommand receive telecommand
86            }
87          } or {
88            via establish::establishConnection receive establish
89            if(establish = 1) then {
```

```
90              choose {
91                via notification::terrestrialMeasure receive terrestrialData
92                do terrestrialDataBuffer::append(terrestrialData)
93              } or {
94                via notification::aquaticMeasure receive aquaticData
95                do aquaticDataDataBuffer::append(aquaticData)
96              }
97            }
98          } or {
99            via location::coordinateSatellite receive satellitePosition
100           via location::coordinateSatellite send satellitePosition
101         }
102         if(distance(satellitePosition, telecommand::coordinateToBeMonitored)
103                                       <= distanceMax) then {
104           via camera::image receive image1
105           via telemetry::telemetry send image1
106           via camera::image receive image1
107           do collectedImages::append(image1)
108           via telemetry::telemetry send image1
109         }
110       }
111     }
112   }
113 }
```

Source code 21 – Excerpt of a satellite modelled in SosADL.

ICMC
USP
SÃO CARLOS