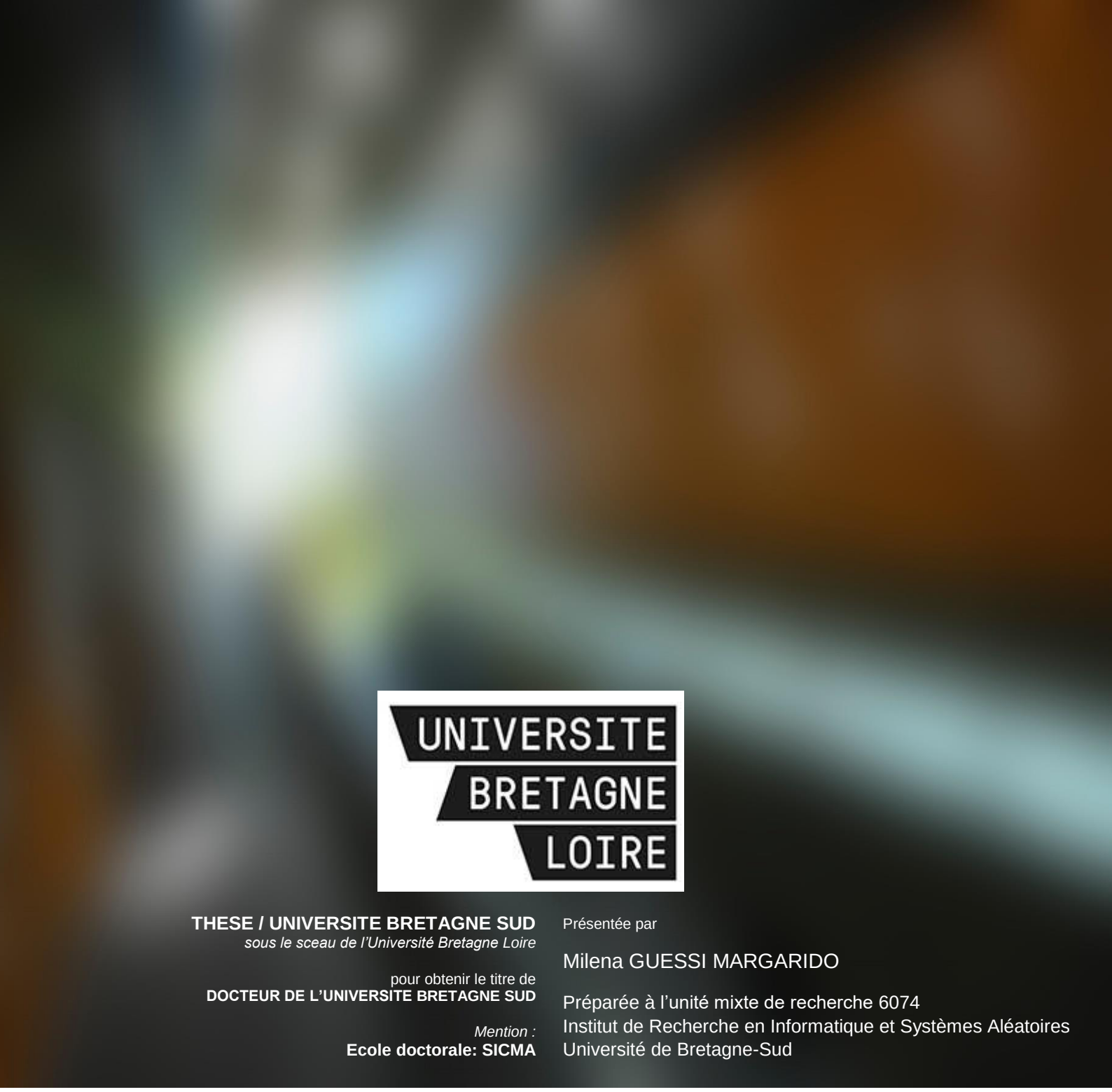
Synthesis of software architectures for systems-of-systems : an automated method by constraint solving

Milena Guessi Margarido



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

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

Mention :
Ecole doctorale: SICMA

Présentée par

Milena GUESSI MARGARIDO

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

Synthèse d'architectures logicielles pour systèmes-de-systèmes : une méthode automatisée par résolution de contraintes

Thèse soutenue le 27 septembre 2017, devant le jury composé de :

Mme. Claudia Maria Lima WERNER
Professeur des Universités, COPPE - Université Fédérale de Rio de Janeiro (UFRJ), Brésil
/ Rapporteur

M. Carlos Enrique Cuesta QUINTERO
Maître de Conférences HDR, ETSII - Université Roi Juan Carlos (URJC), Madrid, Espagne
/ Rapporteur

M. Paris AVGERIOU
Professeur des Universités, IMCS - Université de Groningue, Pays-Bas
/ Examinateur

Mme. Itana Maria de Souza GIMENES
Professeur des Universités, DIN - Universidade Estadual de Maringá (UEM), Brésil
/ 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 de Bretagne Sud, Vannes, France
/ Directeur de thèse

Milena Guessi Margarido

Synthesis of software architectures for systems-of-systems: an automated method by constraint solving

Doctoral dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC- USP and to the Université de Bretagne-Sud – UBS, in partial fulfillment of the requirements for the degrees of the Doctorate Program in Computer Science and Computational Mathematics (ICMC-USP) and PhD (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 (UBS, France)

USP – São Carlos
November 2017

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

M327s

Margarido, Milena Guessi
Synthesis of software architectures for systems-of-systems: an automated method by constraint solving / Milena Guessi Margarido; orientadora Elisa Yumi Nakagawa; coorientador Flavio Oquendo. - São Carlos - SP, 2017.
175 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, 2017.

1. Software architecture. 2. Architectural Description. 3. Systems-of-systems. 4. SoS. I. Nakagawa, Elisa Yumi, orient. II. Oquendo, Flavio, coorient. III. Título.

Milena Guessi Margarido

Síntese de arquiteturas de software para sistemas-de-sistemas: um método automatizado por resolução de restrições

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP e à Université de Bretagne-Sud – UBS, como parte dos requisitos para obtenção dos títulos de Doutora em Ciências – Ciências de Computação e Matemática Computacional (ICMC-USP) e PhD (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: Profa. Dra. Elisa Yumi Nakagawa (ICMC-USP, Brasil)
Orientador: Prof. Dr. Flavio Oquendo (UBS, França)

USP – São Carlos
Novembro de 2017

To my husband Paulo

Acknowledgements

This thesis is the result of many years of work. Through the years, many people contributed, directly and indirectly, for achieving this result.

To my advisor, Prof. Elisa, for sharing her wisdom with me and teaching me so much throughout these years. To my co-advisor, Prof. Flavio, for having me in his research group at Université de Bretagne-Sud and also sharing his expertise.

To all my professors, fellow researchers, students, and staff from University of São Paulo, in particular to Valdemar, Lina, Cristiane, Ana, Brauner, Lucas, Thiago, Daniel, and Laís, whose friendship was so joyful throughout my studies.

To researchers, colleagues, and staff from Université de Bretagne-Sud, in particular members of the ArchWare group, Gersan Moguerou, Jérémy Buisson, Franck Petitdemange, Jean Quilbeuf, and Everton Cavalcante.

To the São Paulo Research Foundation (FAPESP), for the financial support to this research (N. 2012/01646-4, 2014/24290-5).

To my husband, Paulo, for staying by my side at each step of this journey. He has made everything worth the while. I would also like to thank him for giving me the strength to face each new challenge.

To my parents, Valdir and Marilda, for always being supportive of my life long dream of becoming an educator. Also, to my sister and brother in-law, Vanessa and Leandro, whom have always been kind to me.

To my in-laws, Edson and Lélia, and my brothers and sisters in-law, Fábio and Tatiana, Gabriel and Maidy, and Maria Rita and Guilherme, for adopting me in their family. To my niece and nephew, Beatriz and Davi, for bringing joy and laughter into my life.

To all my extended family, whom have been very supportive through the roughest patch. In particular, I would like to thank my cousins, Priscila and Rodrigo, and Joseane and Romiyoshi, my aunt and uncle Regina e Reinaldo, for being there for me specially when I was living abroad. To my family's children, Gabriela, Melissa, Guilherme, and Mirela, for giving me even more reasons to persevere in my path.

Last, but not least, to my friends in France and Brazil, whose friendship always helped to brighten the darkest of the days. In particular, I would like to thank my dear friends Pamela Carreño Medrano and Arthur Brenaut. I hope our paths meet again soon.

“It’s the time you spent on your rose
that makes your rose so important”
(Antoine de Saint-Exupéry)

Abstract

Systems-of-Systems (SoS) encompass diverse and independent systems that must cooperate with each other for performing a combined action that is greater than their individual capabilities. In parallel, architecture descriptions, which are the main artifact expressing software architectures, play an important role in fostering interoperability among constituents by facilitating the communication among stakeholders and supporting the inspection and analysis of the SoS from an early stage of its life cycle. The main problem addressed in this thesis is the lack of adequate architectural descriptions for SoS that are often built without an adequate care to their software architecture. Since constituent systems are, in general, not known at design-time due to the evolving nature of SoS, the architecture description must specify at design-time which coalitions among constituent systems are feasible at run-time. Moreover, as many SoS are being developed for safety-critical domains, additional measures must be placed to ensure the correctness and completeness of architecture descriptions. To address this problem, this doctoral project employs SoSADL, a formal language tailored for the description of SoS that enables one to express software architectures as dynamic associations between independent constituent systems whose interactions are mediated for accomplishing a combined action. To synthesize concrete architectures that adhere to one such description, this thesis develops a formal method, named Ark, that systematizes the steps for producing such artifacts. The method creates an intermediate formal model, named TASoS, which expresses the SoS architecture in terms of a constraint satisfaction problem that can be automatically analyzed for an initial set of properties. The feedback obtained in this analysis can be used for subsequent refinements or revisions of the architecture description. A software tool named SoSy was also developed to support the Ark method as it automates the generation of intermediate models and concrete architectures, thus concealing the use of constraint solvers during SoS design and development. The method and its accompanying tool were applied to model a SoS for urban river monitoring in which the feasibility of candidate abstract architectures is investigated. By formalizing and automating the required steps for SoS architectural synthesis, Ark contributes for adopting formal methods in the design of SoS architectures, which is a necessary step for obtaining higher reliability levels.
Keywords: Software architecture, Architectural Description, Systems-of-systems, SoS.

Resumo

Sistemas-de-sistemas (SoS) englobam sistemas diversos e independentes que cooperam entre si para executar uma ação combinada que supera suas competências individuais. Em paralelo, descrições arquiteturais são artefatos que expressam arquiteturas de software, desempenhando no contexto de SoS um importante papel na promoção da interoperabilidade entre constituintes ao facilitar a comunicação entre interessados e apoiar atividades de inspeção e análise desde o início de seu ciclo de vida. O principal problema abordado nessa tese consiste na falta de descrições arquiteturais adequadas para SoS que estão sendo desenvolvidos sem um devido cuidado à sua arquitetura de software. Uma vez que os sistemas constituintes não são necessariamente conhecidos em tempo de projeto devido à natureza evolucionária dos SoS, a descrição arquitetural precisa definir em tempo de projeto quais coalisões entre sistemas constituintes são possíveis em tempo de execução. Como muitos desses sistemas são desenvolvidos para o domínio crítico de segurança, medidas adicionais precisam ser adotadas para garantir a correção e completude da descrição arquitetural. Visando tratar esse problema, esse projeto de doutorado emprega SosADL, uma linguagem formal criada especialmente para o domínio de SoS que permite expressar arquiteturas de software como associações dinâmicas entre sistemas independentes em que as interações devem ser mediadas para desempenhar uma ação conjunta. Em particular, é proposto um novo método formal, denominado Ark, para sistematizar os passos necessários na síntese de arquiteturas concretas aderentes a essa descrição. Para isso, o método cria um modelo formal intermediário, denominado TASoS, que expressa a arquitetura do SoS em termos de um problema de satisfatibilidade de restrições, possibilitando desse modo a verificação automática de um conjunto inicial de propriedades. O resultado obtido por essa análise pode ser utilizado em refinamentos e revisões subsequentes da descrição arquitetural. Uma ferramenta de apoio denominada SoSy também foi desenvolvida para automatizar a geração de modelos intermediários e arquiteturas concretas, ocultando o uso de solucionadores de restrições no projeto e desenvolvimento de SoS. O método e sua ferramenta foram aplicados em um modelo de SoS para monitoramento de rios em áreas urbanas em que a viabilidade de arquiteturas abstratas foi investigada. Ao formalizar e automatizar os passos necessários para a síntese arquitetural de SoS, é possível adotar métodos formais no projeto arquitetural de SoS, que são necessários para alcançar níveis maiores de confiabilidade.

Palavras-chave: Arquitetura de Software, Descrição Arquitetural, Sistemas-de-sistemas, SoS.

Résumé

Les systèmes-de-systèmes (SoS) interconnectent plusieurs systèmes indépendants qui travaillent ensemble pour exécuter une action conjointe dépassant leurs compétences individuelles. Par ailleurs, les descriptions architecturales sont des artefacts qui décrivent des architectures logicielles jouant dans le contexte SoS un rôle important dans la promotion de l'interaction des éléments constituants tout en favorisant la communication parmi les intéressés et en soutenant les activités d'inspection et d'analyse dès le début de leur cycle de vie. Le principal problème traité dans cette thèse est le manque de descriptions architecturales adéquates pour les SoS qui sont développés sans l'attention nécessaire à leur architecture logicielle. Puisque les systèmes constituants ne sont pas forcément connus pendant la conception du projet à cause du développement évolutionnaire des SoS , la description architecturale doit définir à la conception même du projet quelles coalitions entre les systèmes constituants seront possibles pendant son exécution. En outre, comme plusieurs de ces systèmes sont développés pour le domaine critique de sécurité, des mesures supplémentaires doivent être mises en place pour garantir l'exactitude et la complétude de la description architecturale. Afin de résoudre ce problème, nous nous servons du SosADL, un langage formel créé spécialement pour le domaine SoS et qui permet de décrire les architectures logicielles comme des associations dynamiques entre systèmes indépendants où les interactions doivent être coordonnées pour réaliser une action combinée. Notamment, une nouvelle méthode formelle, nommée Ark, est proposée pour systématiser les étapes nécessaires dans la synthèse d'architectures concrètes obéissant à cette description. Dans ce dessein, cette méthode crée un modèle formel intermédiaire, nommé TASoS, qui décrit l'architecture du SoS en tant que problème de satisfaisabilité de restrictions, rendant ainsi possible la vérification automatique d'un ensemble initial de propriétés. Le résultat obtenu par cette analyse peut s'utiliser en raffinements et révisions ultérieurs de la description architecturale. Un outil logiciel nommé SoSy a été aussi développé pour automatiser la génération de modèles intermédiaires et d'architectures concrètes, en cachant l'utilisation de solveurs de contraintes dans le projet de SoS. Particulièrement, cet outil intègre un environnement de développement plus important et complet pour le projet de SoS . Cette méthode et son outil ont été appliqués dans un modèle de SoS de surveillance de rivières urbaines où la faisabilité d'architectures abstraites a été étudiée. En formalisant et en automatisant les étapes requises pour la synthèse architecturale de SoS , Ark contribue à l'adoption de méthodes formelles dans le projet d'architectures SoS , ce qui est nécessaire pour atteindre des niveaux plus élevés de fiabilité.

Mots-clés: Architecture logicielle, Description architecturale, Systèmes de systèmes, SoS.

List of Figures

List of source codes

List of Tables

List of abbreviations and acronyms

AADL .... Architecture Analysis and Design Language
ADL ..... Architectural Description Language
ASR ..... Architecturally Significant Requirement
CML ..... COMPASS Modeling Language
CNF ..... Conjunctive Normal Form
CP ..... Constraint Programming
CPS ..... Cyber-Physical System
DEVS .... Discrete Event System Specification
DSL ..... Domain Specific Modeling Languages
DSML ... Domain Specific Modeling Languages
EMF ..... Eclipse Modeling Framework
FSA ..... File System Access
IDE ..... Integrated Development Environment
M2M ..... Model-to-Model
M2T ..... Model-to-Text
MDD ..... Model-Driven Development
OCL ..... Object-Constraint Language
OMG ..... Object Management Group
OWL ..... Web Ontology Language
RUP ..... Rational Unified Process
SAT ..... Satisfiability Problem
SoS ..... Systems-of-Systems
SPEM .... Software and Systems Process Engineering Meta-Model Specification
SysML ... Systems Modeling Language
T2M ..... Text-to-Model
UML ..... Unified Modeling Language
URI ..... Uniform Resource Identifier
URM ..... Urban River Monitoring
VDM ..... Vienna Development Method
WSMO ... Web Service Modeling Ontology

Contents

Chapter

1

Introduction

Software architecture is acknowledged as a particular discipline of software engineering since the late 1990s when it became central to realize managed approaches for the development of increasingly complex software systems (KRUCHTEN; OBBINK; STAFFORD, 2006). The ISO/IEC/IEEE 42010 (2011) standard defines a software architecture as the “fundamental organization of a system, embodied in its components, their relationships to each other and to the environment, and the principles governing its design and evolution over time." Hence, software architectures must be able to capture the structure of a software system, such as the hierarchical relationship between its elements, as well as its behavior, encompassing interactions that are exchanged at run-time among its elements and environment. To offer a comprehensive view about software systems, software architectures omit implementation details and focus on the architectural decisions, i.e., choices taken during the design of software systems that can largely impact their perception (ALLEN, 1997). As a result, different designs for software architectures can be investigated aiming to promote specific quality attributes, e.g., performance, safety, maintainability, and longevity (CLEMENTS, 1996; BUSCHMANN *et al.*, 1996; BASS; CLEMENTS; KAZMAN, 2012; AVGERIOU; STAL; HILLIARD, 2013).

As the set of tangible artifacts that describe a software architecture, the architecture description (or architectural representation) provides the means for assessing software architecture properties, sharing architectural knowledge, and preventing software systems decay (GARLAN; SHAW, 1993; KRUCHTEN, 1995; CLEMENTS, 1996; KRUCHTEN, 2009). In particular, architecture descriptions can be used as input for automated simulation, generation, and analysis tools, and as basis for evaluating software architectures (ISO/IEC/IEEE 42010, 2011; MENS; MAGEE; RUMPE, 2010; KRÜGER *et al.*, 2002; OBBINK *et al.*, 2002; LEDECZI; KARSAI; BAPTY, 2000; OQUENDO, 2004; ALLEN *et al.*, 2002; IACOBUCCI; MAVRIS, 2011). Since architecture descriptions contribute for a clearer understanding of software architectures, they can also support choosing among different design alternatives (GARLAN; SHAW, 1993). Aiming to disseminate best practices for creating architectural descriptions, the ISO/IEC/IEEE 42010

(2011) standard establishes the main elements of such an artifact, which includes concerns, architecture views, viewpoints, and rationale. Indeed, architecture descriptions address concerns of a specific group of stakeholders (KRUCHTEN, 1995; CLEMENTS, 1996; ISO/IEC/IEEE-42010, 2011). These concerns or interests can be related to desired properties or requirements for the software system, such as functionality, security, cost, performance, or interoperability (ISO/IEC/IEEE 42010, 2011). Adhering to the separation of concerns principle, the standard recommends creating multiple views, each of which framing one or more distinct concerns about the software architecture. An architecture view conforms to a particular architectural viewpoint, which defines conventions, such as languages, notations, methods, or techniques, for the creation of corresponding views (ISO/IEC/IEEE 42010, 2011). Several studies investigate architecture viewpoints in the description of software architectures (BASS; CLEMENTS; KAZMAN, 2012; CLEMENTS *et al.*, 2011; ROZANSKI; WOODS, 2005; ISO/IEC/IEEE 42010, 2011). The ISO/IEC/IEEE 42010 (2011) standard does not constrain which views have to be selected for describing software architectures, thus specific architecture frameworks[1] have been established, such as The Open Group Architecture Framework (TOGAF) (Open Group, 2009), Zachman's framework (ZACHMAN, 1987), "4+1" Views (KRUCHTEN, 1995), and "Views and Beyond" (CLEMENTS *et al.*, 2011). Finally, the documentation of architecture decisions is also acknowledged as an important part of architecture descriptions (CLEMENTS *et al.*, 2011; BABAR, 2009; ISO/IEC/IEEE 42010, 2011; FARENHORST; BOER, 2009; KRUCHTEN, 2009). By reporting justifications for architectural decisions (e.g., reasoning that led to a particular decision, or trade-offs between design choices), the documentation of software architectures also contributes for disseminating architectural knowledge.

An Architectural Description Language (ADL), which is any form of expression used in architecture descriptions, provides one or more model kinds[2] for framing a particular set of concerns (CLEMENTS *et al.*, 2011; ISO/IEC/IEEE 42010, 2011). Differently from implementation languages, ADLs provide tailored abstractions for the main elements which are part of a software architecture, namely: *components*, which are units of computation representing functional elements of the system, such as packages, classes, systems, and sub-systems; *connectors*, representing interconnections that enable the communication among components; and architectural *configurations*, representing the way in which components and connectors can be arranged together, e.g., showing which components are communicating with each other by means of connectors or the hierarchical relationship that exists among elements of the software architecture (MEDVIDOVIC; TAYLOR, 2000). To further characterize components and connectors, ADLs support the description of interfaces, which define interaction points exposed by these elements to the environment, as well as high-level behavior models and constraints. Desired features that should be supported by ADLs encompass description of functional and

[1] An architecture framework establishes a common practice for creating, interpreting, and analyzing architecture descriptions of a particular domain or stakeholder group (ISO/IEC/IEEE 42010, 2011).

[2] A model kind defines the conventions for a type of modeling, e.g. diagrams, charts, and formalisms (ISO/IEC/-IEEE 42010, 2011).

non-functional properties, formal semantics for precision and automated analysis, and both graphical and textual representations for easing the communication between stakeholders of architecture descriptions (LAGO *et al.*, 2015).

As a particular class of software systems, Systems-of-Systems (SoS) refer to systems formed by independent systems (also referred to as constituent systems) (MAIER, 1998; NIELSEN *et al.*, 2015). The term software-intensive SoS[3] has also been used to distinguish SoS that use software for tailoring constituent systems to their particular environments or needs (BOEHM, 2011). One of such needs concerns the formation of temporary alliances among constituent systems, also referred to as *coalitions*, which enables their combined action (OQUENDO, 2016c). There are five key characteristics that distinguish SoS from traditionally complex, monolithic systems (BOEHM *et al.*, 2004): (i) coalitions produce an emergent behavior that cannot be provided by any constituent alone; (ii) constituents retain their operational independence since they can still operate even when detached from the SoS; (iii) constituents also retain their managerial independence since they can be developed, maintained, and evolved independently from the SoS; (iv) coalitions are evolutionarily developed by continuously changing in response to different environments and needs; and (v) constituents can be geographically distributed as they can only exchange information with each other.

There are several examples of SoS in practice. The Global Earth Observation Systems-of-Systems (GEOSS) (SHIBASAKI; PEARLMAN, 2009) combines several independent systems collecting wind, water, and land observations for creating a comprehensive panorama that can ultimately support disaster reduction, weather monitor and forecasting, efficient use of the land, among others. Other examples of SoS are the US ballistic missile defense system, which encompasses sea-based, land-based, and space-based legacy and new missile defense programs for performing battle management functions (e.g., monitoring missile trajectories since launch until termination) (HOLLON; DAGLI, 2007), medical systems encompassing diagnosis, treatment, and patient management systems (HATA *et al.*, 2007), and air traffic control systems encompassing aircraft, satellites, and control tower systems (WILBER, 2007). In this scenario, embedded systems[4] can also be acknowledged as SoS once they evolve and become complex software systems that are geographically distributed and can change their structure and functionality (HAVERKORT, 2013). The term Cyber-Physical System (CPS) can also be used to designate a communication network among embedded systems that may exchange energy as well as information with each other (BROY; CENGARLE; GEISBERGER, 2012). In this scenario, software is increasingly used as the means for tailoring complex software systems to new scenarios, such as part of a larger SoS. For instance, the automotive industry uses

[3] For the sake of simplicity, SiSoS and SoS are used interchangeably throughout the text.

[4] In contrast to general-purpose computing systems, embedded systems are dedicated to the execution of a particular set of tasks (WOLF, 2008), such as software systems deployed in aircraft, vehicles, smart-phones, among others. Moreover, embedded systems can interact with the physical environment by means of sensors that collect information about their surroundings and actuators that enable to perform a physical action (MARWEDEL, 2010).

software for processing complex sensory data, improving reliability, safety, and efficiency, as well as decreasing pollution (HAVERKORT, 2013). Besides the ability for adaptation, learning, communication, and reconfiguration, these systems must satisfy strict quality attributes, such as safety, security, and efficiency, to succeed. Thereby, there is an increasing necessity of software development approaches that can guarantee the quality of such systems (LIGGESMEYER; TRAPP, 2009).

1.1 Problem statement

The design, development, and maintenance of SoS faces several challenges due to their intrinsic characteristics (BOEHM; BROWN; TURNER, 2005). For instance, the formation of new configurations at run-time is a challenge that arises from the evolutionary development of SoS. To accomplish their missions in face of acquisition or loss of constituent systems, SoS software architectures must be able to form different coalitions. In this sense, SoS software architectures represent a shift from software architectures of single software systems, which can be determined early in their life cycle and remain relatively stable (RHODES, 2004; OQUENDO, 2016c). As a consequence, traditional perspectives on system design and control that assume architectures to be determined early in systems life cycle are not suitable for SoS (ULIERU; DOURSAT, 2011). Aiming to support SoS evolution, dynamic software architectures can be investigated for promoting SoS resilience in face of uncertainty (DAGLI; KILICAY-ERGIN, 2009; NAKAGAWA *et al.*, 2013; NIELSEN *et al.*, 2015). Since SoS architecture evolution can have deeper repercussions in regards to structure and behavior, tailored support is needed for *rearchitecting*, a process in which old elements of the architecture are modified, replaced, or rebuilt and/or new elements are included (AVGERIOU; STAL; HILLIARD, 2013). In particular for safety-critical applications in which there is an eminent risk of harming people and/or the environment, or causing financial loss (KNIGHT, 2002), a rigorous rearchitecting approach must be in place to guarantee that any formed coalition complies with the original design of the SoS. To use architecture descriptions as a guide to such an approach, this thesis addresses three problems, which are further elaborated in the following sections.

1.1.1 Describe Systems-of-Systems Software Architectures

The five intrinsic characteristics of SoS make their architectural design more complex. For instance, due to constituents operational and managerial independence, constituent systems are not controlled by the SoS, implying that they have their own missions and operational life cycles and their management and evolution is carried out independently from the SoS itself. Moreover, as SoS evolve, constituents that were not necessarily known at design time can join and leave the coalition, thus imposing a new requirement to the design of SoS that must be able to identify and incorporate new constituents at run-time (OQUENDO, 2016c). In this scenario, the creation of coalitions among constituent systems should be addressed from an architectural

perspective. Abstracting from implementation details, the software architecture expresses the fundamental organization of systems. Thereby, architecture descriptions of SoS must enable to express guarantees for constituents expected behavior and/or architectural properties (FITZGERALD; BRYANS; PAYNE, 2012). The connectors that support the communication among these constituent systems can be complex architectural entities on their own, e.g., acting as mediators between independent parts and enabling the realization of combined actions (NAKAGAWA *et al.*, 2013; ISSARNY; BENNACEUR, 2013). Unlike constituents, mediators operate under the explicit control of SoS to fulfill global missions (OQUENDO, 2016c). The description of SoS configurations must capture their structure and emergent behaviors for better understanding such systems (DOGAN *et al.*, 2013). Thereby, the architectural configuration of an SoS must describe how constituents and mediators are arranged together, what interactions are exchanged among them, and what behaviors emerge from the coalition. As configurations are continuously changing, an static description of the software architecture can quickly become obsolete (SELBERG; AUSTIN, 2008). Therefore, the description of SoS requires new abstractions supporting dynamic modification of architectures and interfaces (NIELSEN *et al.*, 2015).

While several ADLs can be used for the description of software systems, expressiveness is still an issue (HILLIARD; RICE, 1998; OZKAYA; KLOUKINAS, 2013), in particular in the context of SoS (BATISTA, 2013; OQUENDO, 2016c). These languages can be classified in three broad categories (BASS; CLEMENTS; KAZMAN, 2012): (i) formal languages that present mathematically defined syntax and semantics; (ii) semi-formal languages that present well-defined syntax but lack of complete semantics; and (iii) informal languages that represent ad-hoc, graphical notations with neither defined syntax or semantics. Languages in each category can offer different mechanisms for the communication of architectural design to stakeholders, who in turn may need a specific set of skills for effectively applying and/or understanding models expressed in those languages. Formal languages, such as Wright (ALLEN, 1997) and π-ADL (OQUENDO, 2004), enable the creation of formal models that can be verified and validated against quality attributes using tailored tools. Thus, such languages are particularly interesting for ensuring architectural properties and supporting automated analysis and evolution of software systems (ZHANG; MUCCINI; LI, 2010). In turn, semi-formal languages, such as Unified Modeling Language (UML) (OMG, 2011) and Systems Modeling Language (SysML) (OMG, 2012) that have a well-known syntax, facilitate the communication among stakeholders but offer limited support for automated analysis.

The precision and rigor that can be achieved with formal models becomes essential for obtaining higher reliability in software systems (JACKSON, 2006; MANDRIOLI, 2015). In the context of SoS, precise architecture descriptions are needed for understanding emergent behaviors, evolutionary development, and independence of constituent systems (DOGAN *et al.*, 2013; NIELSEN *et al.*, 2015). Nonetheless, the design of software architectures for SoS and, in particular, their architecture description are still open topics of research in the Software Engineering field (NAKAGAWA *et al.*, 2013; BATISTA, 2013; KLEIN; VLIET, 2013; GUESSI

et al., 2015). While previous works focus on adaptation and evolution of the dynamic changes in software architectures (LEMOS *et al.*, 2013), traditional practices for architectural description from a run-time viewpoint lack expressiveness to describe SoS architectures (NIELSEN *et al.*, 2015; GUESSI; CAVALCANTE; OLIVEIRA, 2015). Hence, the first problem addressed in this thesis is summarized as follows:

Problem I

Traditional practices for the description of software architectures of single, complex systems lack tailored abstractions for the main elements of SoS.

1.1.2 Ensure Feasibility of Systems-of-Systems Software Architectures

SoS adaptation can be triggered by changes in constituent systems, emergent behaviors, or missions, which makes dynamic reconfiguration and evolution two capabilities that must be supported for guaranteeing SoS resilience. Dynamic reconfiguration is the result of unplanned changes to the architecture and/or constituents whereas evolution is the result of slow-paced, planned changes to the architecture, hence requiring intervention (NIELSEN *et al.*, 2015). In either case, the impact of modifications to the configuration of a SoS can be extensive since, in general, it is not possible to act on the constituents themselves due to their operational and managerial independence. Thus, new means that can mediate the collaboration among heterogeneous constituents and tailor these constituents to the SoS specific context are required (BOEHM, 2011). For instance, Boardman and Sauser (2006) discuss the need for a dynamic determination of connectivity which is accomplished by establishing and dissolving communication links among constituents accordingly.

A key challenge faced during SoS design and deployment concerns the validation of their architectural feasibility (BOEHM, 2011) to achieve their specified missions (SILVA; BATISTA; OQUENDO, 2015). To avoid quality degradation after reconfiguring or evolving SoS, it is important to check their architectural feasibility, making sure that resulting architectures are still able to form new coalitions that preserve desired architectural properties. Also, this verification can prevent reaching an erroneous state of the SoS architecture for which there will be no feasible coalition. Hence, the second problem addressed in this thesis is summarized as follows:

Problem II

Lack of automated support for analyzing the feasibility of SoS software architectures, which is required to prevent the formation of ill-formed coalitions throughout the entire SoS life cycle.

1.1.3 Support Synthesis of Systems-and-Systems Concrete Architectures

The continuous evolution of SoS hinders an early definition of their software architecture (SELBERG; AUSTIN, 2008). The need of adhering to best practices for developing SoS is pointed out by Valerdi, Ross and Rhodes (2007), who identify three complementary SoS models: (i) normative models, describing norms and standards for how SoS should be; (ii) descriptive models, describing how SoS are by showing deviations from the normative model; and (iii) prescriptive models, describing how one can achieve the normative model given the descriptive model. The creation of normative SoS models, which support the dissemination of best practices about SoS engineering, are needed for evaluating descriptive models and refining predictive models that will enable to improve SoS quality. As normative models, abstract architectures define at design time the baseline from which coalitions will be formed at run-time. In turn, concrete architectures can be regarded as descriptive models of such coalitions, thus detailing how their individual actions are combined. As a consequence, a number of concrete architectures can be realized from a single abstract architecture depending on which and how many constituents participate in the coalition.

Candidate concrete architectures can be verified and validated against each other before choosing the one that better suits stakeholders' interests. Nonetheless, synthesis and comparison of alternative designs can be challenging and time consuming, specially due to the different ways in which constituents can be assembled together (CHATTOPADHYAY; ROSS; RHODES, 2008). Moreover, it is not possible to assume that all concrete architectures complying with an abstract description are appropriate, i.e., they preserve all architectural properties, such as performance or safety, and external conditions, such as geographical distribution of constituents. Therefore, tailored support is needed for the synthesis of concrete architectures for a particular SoS abstract architecture. Given that abstract architectures are continuously changing, it is also important to support SoS evolution. In this scenario, a systematic approach that supports evolution of SoS can use formal models of abstract architectures as input for tools that create, analyze, and maintain concrete architectures for these systems prior, during, and after changes have taken place. Formal languages can be used for describing target architectures and then modeling change requests as refinement relations among them so that consistent models of the system can be guaranteed throughout their entire life cycle (BROY, 2010). Hence, the third and last problem addressed in this thesis is summarized as follows:

Problem III

Lack of a systematic approach supporting synthesis of SoS concrete architectures in conformance to normative architectural models.

1.2 Research Questions

Architectural descriptions play a central role in the three problems that were previously discussed. To adequately support the creation of architecture descriptions for SoS, three research questions should be answered:

RQ1. Which abstractions must be supported by ADLs for expressing the software architecture of SoS? A common understanding of what exactly are the requirements of SoS architecture descriptions is needed, mainly for facilitating organizations and partners that will develop or adapt their systems as part of a SoS (NAKAGAWA *et al.*, 2013). To answer this question, tailored abstractions that can enable architects to reason about the structure and behavior of the SoS as a whole are investigated. In particular, formal languages are desirable aiming to support the creation of traceable, consistent models that can be automatically analyzed.

RQ2. How is it possible to guarantee the feasibility of SoS software architectures? The lack of formal foundations can introduce critical gaps in SoS design, specially if an abstract architecture reaches a state from which no possible coalition can be realized. To answer this question, a formal model, which represents SoS software architectures in terms of constraints, is investigated aiming to prevent the design of incorrect coalitions. By resolving these constraints, this model can be used to confirm the feasibility of an abstract architecture by presenting a concrete architecture that satisfies its specification. This approach is in line with the research on autonomous systems architecture (KRAMER; MAGEE, 2009) where mechanisms are provided to either automatically realize a solution for which purposes, properties, and constraints of the description are satisfied, or report that such a solution does not exist.

RQ3. How is it possible to support architectural synthesis of SoS given an abstract architecture? Since the applicability of formal descriptions can be hampered by the lack of tool support (LAGO *et al.*, 2015), this question investigates a mechanism that automates the synthesis of abstract architectures for SoS. In particular, this question investigates the effectiveness of a method that adopts constraint solving techniques for the automated synthesis of concrete architectures given a normative design of the coalition.

1.3 Contributions

This thesis makes the following contributions:

- A conceptual model that expresses SoS abstract architectures in terms of constraints, which can be automatically analyzed using off-the-shelf constraint satisfaction tools;
- A method that combines formal models and constraint solving in the synthesis of concrete architectures in conformance to such a normative design; and
- A software tool that implements model transformations required by the method, hence providing automated support for the synthesis of SoS.

Tool support for SoS architectural synthesis is needed to ensure correctness of coalitions run-time reconfiguration. In particular, it contributes for mitigating the architectural feasibility risk that incurs from continuously changing the SoS architecture throughout their life cycle. In this sense, this tool will facilitate the creation, analysis, and evolution of formal models that are needed to automatically analyze architectural descriptions.

1.4 Outline

This chapter presented an overview of the context and problems that motivated the development of this doctoral research. The remaining chapters of this thesis are organized as follows. Chapter 2 presents the main concepts related to the design and description of software architectures for systems and SoS. Chapter 3 presents a method for architectural synthesis of SoS based on formal descriptions, which are detailed in Chapter 4. Chapter 5 details the implementation of a tool that supports this method and enables to employ model finding for confirming the feasibility of SoS architectures and identifying concrete architectures that comply with the abstract architecture design. The evaluation of this method on the case of a SoS for monitoring floods in urban areas is presented in Chapter 6. Finally, Chapter 7 revisits the thesis contributions and discusses perspectives for future research.

The accompanying materials of this thesis encompass: Appendix A details how to use the tool presented in Chapter 5 and provides instructions for uploading and analyzing architectural models; Appendix B presents materials that are referred to in the evaluation of the method in Chapter 6; and Appendix C lists accomplished publications throughout this doctoral research. Finally, Annex A details the ADL that is employed in this thesis for the description of abstract and concrete architectures of SoS.

Chapter

2

State of the Art on Architecture Description of Systems-of-Systems

Software architectures increase the abstraction level with which one can reason about software systems (ALLEN, 1997). Among design issues dealt within the software architecture level are the “gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives" (GARLAN; SHAW, 1993). As tangible artifacts expressing software architectures, architecture descriptions provide concrete ways for accessing systems qualities, sharing architectural knowledge, and preventing software systems decay (GARLAN; SHAW, 1993; CLEMENTS, 1996; CLEMENTS *et al.*, 2011; KRUCHTEN, 2009; ISO/IEC/IEEE 42010, 2011). To disseminate best practices regarding the content of such artifacts, the ISO/IEC/IEEE-42010 (2011) standard establishes the main elements framed in architecture descriptions and the relationships that exist among them. Still, the content of architecture descriptions can be tailored for each project, thus requiring specific guidelines for a particular domain or stakeholder community.

This chapter discusses software architectures in light of challenges that are faced when architecting Systems-of-Systems (SoS). First, Section 2.1 presents definitions for the main concepts of software architectures and, in particular, architecture descriptions. This section also details the use of formal methods and model-based practices for the design and analysis of software systems. Then, Section 2.2 introduces a generic process for the architectural design of software systems and discusses new requirements for architecting SoS. Section 2.3 introduces notations for expressing software architectures. Section 2.4 addresses a number of viewpoints that have been adopted in architectural descriptions. Section 2.5 introduces architecture decisions as a means to document architectural knowledge and discusses the main challenges for maintaining SoS. Finally, Section 2.6 summarizes the main gaps regarding architectural descriptions for SoS

and how this thesis is expected to advance the state of the art.

2.1 Basic Concepts

Throughout software systems life cycle, software architectures can play different roles, such as (BABAR, 2009): (i) a communication vehicle among stakeholders during analysis and development of software systems; (ii) a repository of architecture decisions which impact subsequent development stages; and (iii) an abstraction of software systems which can be read and examined by people and constitutes the basis for supporting reuse. Accordingly, several definitions can be found for software architectures in the literature based on these different roles. For instance, Jansen and Bosch (2005) define software architectures as the set of architecture decisions that shape software systems and that have greater influence to the quality of software systems. To facilitate the understanding of the following chapters, definitions for the main concepts related to software architectures are presented in this section.

Concrete software architecture: is a descriptive model that identifies the main elements or properties of a system which are embodied in its systems, relationships, and in the principles governing its design and evolution over time (GARLAN; SHAW, 1993; KRUCHTEN; OBBINK; STAFFORD, 2006; ISO/IEC/IEEE 42010, 2011; BASS; CLEMENTS; KAZMAN, 2012). It can also be referred to as run-time architecture. This artifact is further explained in Section 3.3.

Abstract software architecture: is a normative model of the main *types* of elements, relationships, and principles governing the design and evolution of a software system over time. It can also be referred to as design-time architecture, intentional architecture (OQUENDO, 2016e), or conceptual architecture (NIELSEN *et al.*, 2015). This artifact is further explained in Section 3.3.

Stakeholder: refers to any individual, team, or organization playing a relevant role in the software architecture process (ISO/IEC/IEEE 42010, 2011; ROZANSKI; WOODS, 2005). Examples of stakeholders are clients, users, project managers, developers, and suppliers.

Concern: is any interest or need of one or more stakeholders about the software system (ISO/IEC/IEEE 42010, 2011; ROZANSKI; WOODS, 2005). For example, concerns can be features, requirements, functionality, or qualities of the software system.

Architecturally Significant Requirement (ASR): is a quality attribute, also referred to as non-functional requirement. For instance, an ASR can specify a core feature of the system, a constraint to elements of the system, or a detail of the application environment (CHEN; BABAR; NUSEIBEH, 2013). In particular, an ASR is related to one or more components or to the system as a whole whereas requirements that are not architecturally significant are

restricted to single components (OBBINK *et al.*, 2002). The significance of a requirement can be determined in terms of time, resources, reputation, or cost, so that an (CHEN; BABAR; NUSEIBEH, 2013). Thereby, a requirement is architecturally significant if its impact to the architecture can be measured (CHEN; BABAR; NUSEIBEH, 2013).

Architecture pattern: addresses a particular recurring design problem that arises in specific design situations, and presents a well-proven generic scheme for its solution (BUSCHMANN *et al.*, 1996). This solution scheme encompasses a set of predefined subsystems, their responsibilities and relationships, and the rules and guidelines for organizing these relationships between them. Moreover, architectural patterns can be used as templates for concrete software architectures.

Architecture style: is a set of common restrictions to the form and structure of software architectures (GARLAN; PERRY, 1995; BASS; CLEMENTS; KAZMAN, 2012).

Architecture decision: refers to a design decision that affects the software architecture as a whole (GARLAN; SHAW, 1993). For example, a design decision may require the addition or modification of architecture description elements.

2.1.1 System and System-of-Systems

Systems, as defined by the ISO/IEC/IEEE 15288 (2015) standard, are man-made systems encompassing hardware, software, data, humans, processes, materials, and others. A software-intensive systems can be distinguished as a system in which software plays a central role in its design, development, deployment, and evolution (ISO/IEC/IEEE 42010, 2011). In this sense, the term system can be used to designate individual applications, subsystems, systems-of-systems, product lines, product families, whole enterprises, among others. Nonetheless, Boardman and Sauser (2006) discuss five particular dimensions that make SoS intrinsically different from systems, namely: autonomy, belonging, connectivity, diversity, and emergence. First, constituent systems, and the SoS itself, are not merely parts in that they retain their autonomy for pursuing their own purposes. Indeed, by exercising their autonomy, constituents can voluntarily decide whether to belong to the SoS, which is out of the control of the SoS architect. Since constituents are geographically distributed, they can only exchange information with each other. To effectively belong to an SoS, new means are needed for dynamically determining constituents connectivity, i.e., deploying custom communication links on demand. Finally, diversity of constituents provides SoS with rich, dynamic capabilities obtained from independent constituent systems, which have limited functionality by design. Finally, the combination of autonomy, belonging, dynamic connectivity, and diversity allows the emergence of behaviors which enable to fulfill the SoS mission. These dimensions are also discussed by Gonçalves *et al.* (2014), who adds software dominance of constituents as a particular characteristic of software-intensive SoS.

The design of SoS must address the complexity that is added by combining several independent constituent systems, thus requiring tailored processes and techniques, such as for acquisition of constituent systems (BOEHM *et al.*, 2004; GAGLIARDI; BERGEY; WOOD, 2010) and evolution of software architectures (SELBERG; AUSTIN, 2008). The evolution must be addressed by SoS software architectures, which must frequently change in unpredictable ways at run-time due to autonomy and diversity of constituent systems (VALERDI; ROSS; RHODES, 2007; SELBERG; AUSTIN, 2008; OQUENDO, 2016e). Thus, SoS architectures must address possible changes to constituents' functionality, performance, or interfaces by defining a persistent technical framework that describes how constituents collaborate with each other (DAHMANN *et al.*, 2011a). Since constituents are often out of the control of SoS architects, software architectures can solely focus on those key constituent systems and connectors that are needed for achieving its desired capability, fostering interoperability among them by documenting protocols used for data communication and/or synchronization across constituents (DAHMANN *et al.*, 2011a). In this scenario, the key research challenges raised by SoS are architectural, i.e., they concern how local interactions among constituents can be organized so that global, desired behaviors will emerge (OQUENDO, 2016e).

2.1.2 Architecture Description

As the main artifact expressing software architectures, architecture descriptions offer support to the communication, evaluation, and evolution of software systems (GARLAN; SHAW, 1993; CLEMENTS, 1996; CLEMENTS *et al.*, 2011; KRUCHTEN, 2009; MENS; MAGEE; RUMPE, 2010; ISO/IEC/IEEE 42010, 2011). The architecture description of a given software system can present one or more architecture models that depict its main parts and the relationships existing among these parts. By documenting the architecture rationale, the architecture description can support the education of future maintainers of the software architecture, thus providing valuable guidance for evolution of software systems (KRUCHTEN; OBBINK; STAFFORD, 2006). To effectively use architecture descriptions at different stages of software systems life cycle, the ISO/IEC/IEEE 42010 (2011) standard was introduced. This standard, which replaces the ANSI/IEEE 1471-2000 standard, "addresses the creation, analysis, and sustainment of architectures of software-intensive systems through the use of architecture descriptions" (ISO/IEC/IEEE 42010, 2011). Figure 1 clarifies the relationship among the main elements of architecture descriptions, namely:

Model kind: selects a set of techniques (e.g., metamodels, templates, languages, and operations) for creating architectural models from a particular viewpoint. Examples of model kinds are class diagrams, Petri nets, charts, among others;

Architecture model: it uses the modeling conventions defined by a model kind for framing a particular set of concerns;

Architecture viewpoint: it establishes the languages, notations, methods, and model kinds for creating, interpreting, and using architectural views;

Architecture view: it expresses a particular software architecture from a given architecture viewpoint. Considering this definition, an abstract architecture and a concrete architecture can be regarded as two different views of software architectures;

Architecture rationale: it adds explanations, reasoning, and implications of architecture decisions, such as considered alternatives, trade-offs, and consequences;

Correspondence: it is a relation among architecture description elements. Examples of meaningful relations among architecture description elements are composition, refinement, consistency, traceability, dependency, constraint, and obligation;

Correspondence Rule: it enforces a particular relationship between architecture description elements. A correspondence rule **holds** if it is possible to show an associated correspondence satisfying the rule whereas it is **violated** if it is possible to show an associated correspondence that does not satisfy the rule or if no associated correspondence exists.

Architecture descriptions identify the stakeholders of a software system and build architectural views for framing their concerns. Several architectural models can be created for describing each of these views. Combined, the set of views forms a cohesive, complete description of the software architecture. Then, correspondences can be established among architectural elements for capturing relations (e.g., refinement, constraint, dependency) that are enforced for ensuring the consistency among independent artifacts. Moreover, any known inconsistencies must be recorded so that they can be properly addressed. The architecture rationale is also an important part of the description since it provides the context for understanding views and documents considered alternatives and decisions that shaped the software architecture.

The ISO/IEC/IEEE 42010 (2011) standard also clarifies the requirements of complex elements, such as architecture frameworks, architecture styles, and Architecture Description Languages (ADLs), that can be created on top of these primitive constructs. According to the standard, an architecture framework establishes a common practice for creating, interpreting, analyzing, and using architecture descriptions for a particular domain or stakeholder community and it can be characterized by viewpoints, model kinds, stakeholders, and concerns that it selects for creating architecture descriptions. In turn, an architecture style can be considered as an architecture model in terms of this standard as it selects model kinds and concerns that are characteristic of a particular class of software architectures. Finally, an ADL, which is defined as any form of expression used in architecture descriptions, can be characterized in terms of selected model kinds, concerns, stakeholders, and correspondence rules.

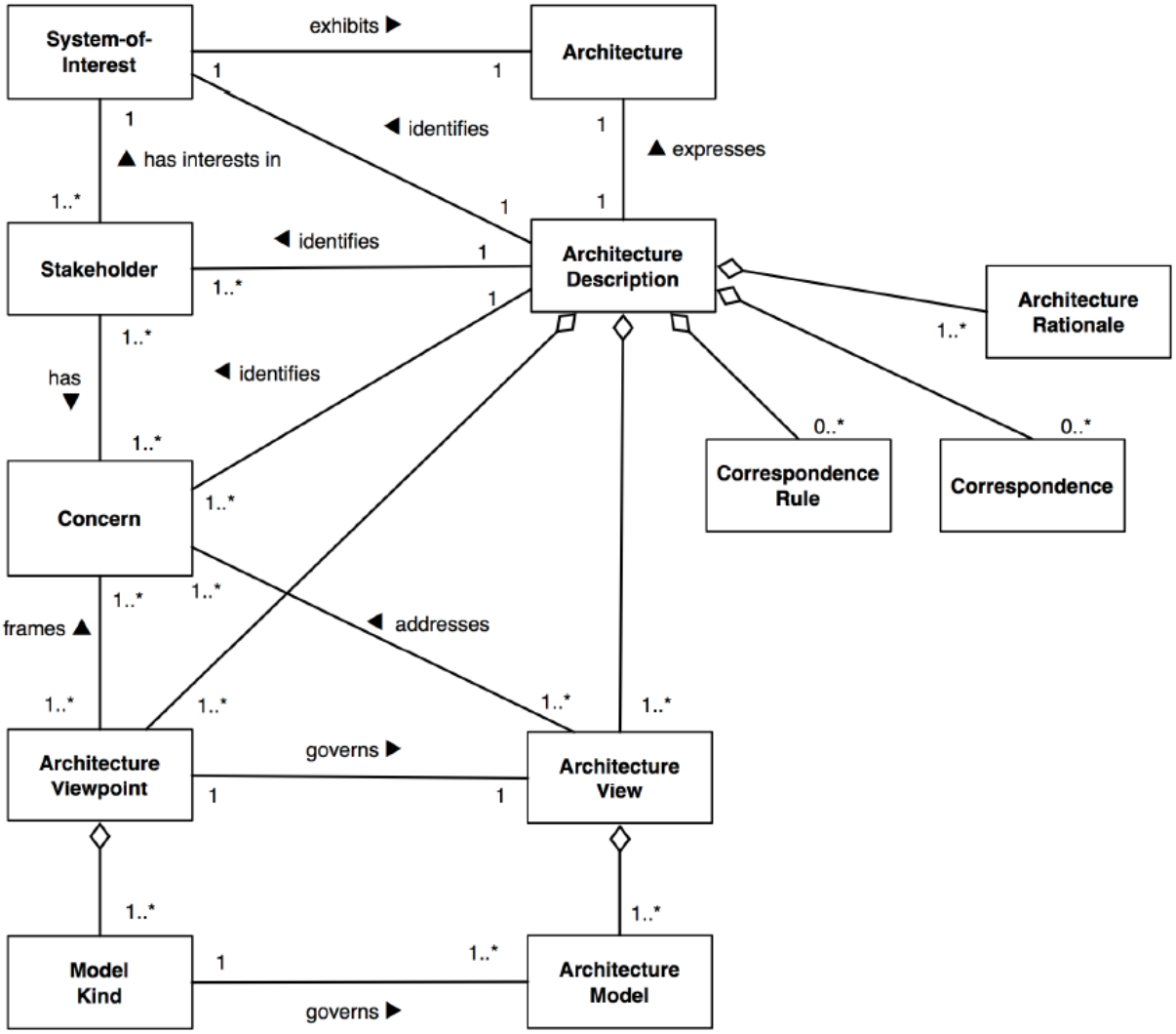


Figure 1 – Architecture description conceptual model
Source: ISO/IEC/IEEE 42010 (2011).

2.1.3 Formal Methods

A formal method is traditionally defined as a mathematical technique employed in the development of software and hardware systems, including methods for specification, execution, proof, model checking, refinement, and test generation (WOODCOCK *et al.*, 2009). Any method based on discrete symbolic systems that enables precise specification and analysis can be perceived as a formal method, encompassing formal notations (such as UML state diagram or xUML (Executable UML) (MELLOR; BALCER, 2002) and Domain Specific Modeling Languages (DSL) besides formal logics (JASPAN *et al.*, 2009). While the impact of formal methods on time and cost is undetermined[1], the impact of these methods on quality is widely acknowledged (WOODCOCK *et al.*, 2009). In particular for safety and security critical systems, precision and rigor achieved with formal methods seem to be unavoidable for obtaining higher

[1] Practitioners interviewed in a survey by Woodcock *et al.* (2009) regarded formal methods as more beneficial than harmful to time and cost although no quantitative evidence was available to confirm their perception.

reliability[2] (MANDRIOLI, 2015). The use of formal methods has also been linked to increased productivity (by supporting automated generation of source code) and decreased maintenance (by correcting subtle errors early in the development and improving designers understanding) (WOODCOCK *et al.*, 2009).

Aiming to counter-balance costs associated with the introduction of new technologies in the development of software systems, formal methods can be employed with different levels of intensity, focusing on a limited part of the system or step of the design process (WOODCOCK *et al.*, 2009; JASPAN *et al.*, 2009). One can use formal methods for pointing out generic errors, e.g. using statistical analysis to detect trends in code that could be related to anomalies. Formal methods can also be used for checking assertions at run-time (CLARKE; ROSENBLUM, 2006). In this scenario, *lightweight formal methods* emerge as a way to partially employ formal methods in the design of software systems (WOODCOCK *et al.*, 2009). Jones (BOWEN *et al.*, 1996, p. 20-21) advocates in favor of the use of formal methods early on a project's life cycle for producing formal documents that can later be inspected and, if needed, performing formal proofs for steps of (portions of) a system. In turn, Jackson and Wing (BOWEN *et al.*, 1996, p. 21-22) suggest partial application of formal methods for reducing costs. For instance, Alloy (JACKSON, 2012) is a lightweight formal method supporting the adoption of model checking to a limited analysis scope. Advances in SAT solvers have enabled to apply the method for solving partial instances (TORLAK; DENNIS, 2006) and higher-order problems (which contain $\exists and \forall$ quantifiers) (MILICEVIC *et al.*, 2015).

Constraint Programming (CP) aims at solving problems, e.g. as graph colouring, n-queens, and time scheduling, that can be expressed in terms of constraints (PETKE, 2015). In parallel, constraint satisfaction is a branch of artificial intelligence concerned with finding an assignment of variables that satisfy a Boolean formula (PETKE, 2015). Let $\varphi(x_1,\ldots,x_n)$ be a Boolean formula. Variables x_1, ..., x_n are arranged within several clauses using the logical operators depicted in Table 1 and each of these variables can be assigned a value of 0 (False) or 1 (True). The formula is satisfied if all clauses are satisfied and unsatisfied if at least one clause is unsatisfied after all variables have been assigned. Optionally, it is also possible to express all Boolean formulas in their equivalent Conjunctive Normal Form (CNF), i.e., as a conjunction of clauses that are a disjunction of variables, such as

$$\varphi(x_1,x_2,x_3) = (\bar{x_1} \vee x_3) \wedge (x_2 \vee x_3) \wedge (\bar{x_2} \vee x_3)$$

for variables x_1, x_2, and x_3. In this form, a clause is satisfied if at least one variable is True and unsatisfied if all variables are False. Thereby, only one clause needs to be unsatisfied to render the entire formula unsatisfied.

[2] The ISO/IEC 25010:2011 (2011) standard defines reliability as "the degree to which the software product can maintain a specified level of performance when used under specified conditions." Sub-characteristics of reliability are availability, fault tolerance, recoverability, and reliability compliance.

Table 1 – Boolean logic operators

Operator	Description
$\bar{x_n}$	Negation (read NOT x_n)
$x_n \wedge x_m$	Conjunction (read x_n AND x_m must be true)
$x_n \vee x_m$	Disjunction (read either x_n OR x_m must be true)

The problem of determining if there is any assignment to the Boolean variables that satisfies the formula or proving that such an assignment does not exist is also referred to as Satisfiability Problem (SAT), which is the first known NP-complete problem (COOK, 1971). Thereby, all known algorithms for this problem face state space explosion issue as the number of variables exponentially increases in the worst-case scenario. Nonetheless, improvements on algorithms and technologies have increased the efficiency of SAT-solvers. These algorithms can be distinguished between: (i) incomplete algorithms (e.g., local search, genetic algorithms) which either prove a formula unsatisfiable or satisfiable, or (ii) complete algorithms (e.g., resolution, recursive learning, conflict-driven clause learning) which prove a formula to be both unsatisfiable and satisfiable.

In this scenario, advances in tools and algorithms have contributed for better understanding and employing formal methods in practice, such as model checking, software verification, and artificial intelligence (WOODCOCK *et al.*, 2009; JASPAN *et al.*, 2009). Constraint solving has been applied in Search-Based Software Engineering (SBSE) for the definition of a search problem that investigates optimal or near optimal solutions by using a fitness function that distinguishes between better and worse solutions (HARMAN; MANSOURI; ZHANG, 2012). Model checking has also been applied to software architectures in the execution of an exhaustive and automatic verification that confirms the presence of desired properties, such as safety[3], liveness[4], and compatibility[5](ZHANG; MUCCINI; LI, 2010). In this scenario, formal languages supporting the specification of software systems, including ADLs, enable to create executable and unambiguous models that play an important role in guaranteeing the correctness and soundness of software systems, often supporting code generation. In turn, support for code generation from formal specifications can have a positive impact in productivity provided that sufficient tools are available (WOODCOCK *et al.*, 2009).

2.2 Architecting Software Systems

The ISO/IEC/IEEE 42010 (2011) standard defines *architecting* as the "process of conceiving, defining, expressing, documenting, communicating, certifying proper implementation of, maintaining and improving an architecture throughout systems' life cycle". There are many ways

[3] Safety properties verify whether a specific event never occurs (ZHANG; MUCCINI; LI, 2010).
[4] Liveness properties verify whether a specific event eventually occurs (ZHANG; MUCCINI; LI, 2010).
[5] Compatibility properties verify whether components or connectors can collaborate to achieve the desired interactions (ZHANG; MUCCINI; LI, 2010).

in which software architectures can be created. For instance, traditional software development processes, such as the Rational Unified Process (RUP) (KRUCHTEN, 2003) that encompasses inception, elaboration, construction, and transition phases, the architecture is built iteratively during the elaboration phase, resulting in executable architectural prototypes that can be used for validating the architectural design. Even agile methods, such as Scrum (SCHWABER; BEEDLE, 2001), that favor speed at the expense of having a consolidated design upfront, software architectures support scaling up agile practices to larger system scope, team size, and projects length, as well as minimizing the risk of unpredictable issues by employing smaller incremental steps and engaging stakeholders in the development process (NORD; OZKAYA; KRUCHTEN, 2014). Indeed, the combination of architecture and agile practices, such as incremental development, design patterns, frequent architecture analysis and improvement, and prototyping, results in a development process that guarantees agility and sustainability for continuous software delivery (KOONTZ; NORD, 2012).

Hofmeister *et al.* (2007) propose a general model of architectural design activities depicted in Figure 2. In this model, the process of realizing a suitable software architecture encompasses analyzing the problem that it must solve, synthesizing potential solutions for that problem, and evaluating alternative solutions until reaching a satisfiable design. The architectural analysis uses the context (i.e., environment) and architectural concerns to formulate the set of Architecturally Significant Requirements (ASRs) in terms of desired systems properties that must be fulfilled by the architecture. As distinguishing characteristic, an ASR is related to one or more components or the system as a whole whereas requirements that are not architecturally significant are restricted to single components (OBBINK *et al.*, 2002). The architectural synthesis builds upon these ASRs to outline potential solutions satisfying these requirements. Then, the architectural evaluation validates or invalidates these candidate solutions against ASRs until one of them is chosen as the validated architecture. Each of these activities can be repeated until a validated architecture is obtained.

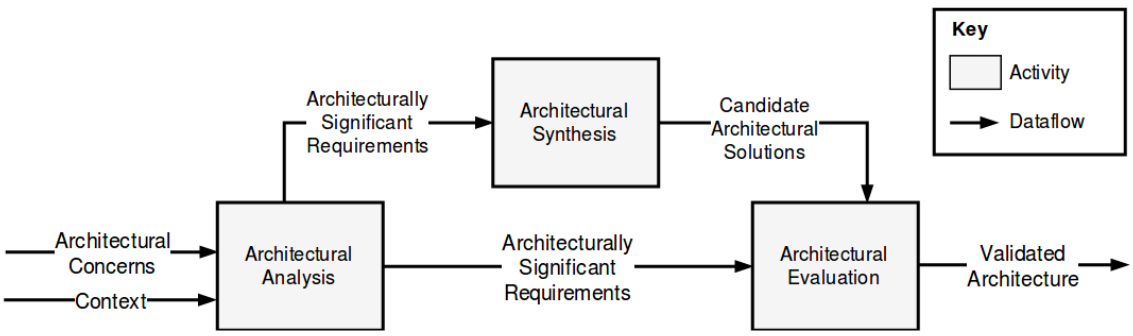


Figure 2 – General model of architectural design activities

Source: Hofmeister *et al.* (2007).

2.2.1 Architecting Systems-of-Systems

The software architecture of an SoS encompasses the structure of constituent systems, the relationships that exist among them, and the principles and guidelines governing its design and evolution over time (GAGLIARDI; BERGEY; WOOD, 2010). Descriptions of SoS architectures detail functionality performed by key constituent systems, data and control flow, externally visible properties and interfaces of constituents (e.g., behaviors, dependencies, and use of shared resources), relationships among organizational entities and constituents as well as rationale and governance policies that apply to the SoS, including for instance guidelines for acquisition, termination, and replacement of constituents (GAGLIARDI; BERGEY; WOOD, 2010).

As SoS architectures establish a baseline for developing constituents and shared infrastructure and distributing work, they are key artifacts of SoS engineering processes (DAHMANN *et al.*, 2011a). The process defined by the Office of the Deputy Under Secretary of Defense for Acquisition and Technology, Systems and Software Engineering (2008) identifies seven key activities for engineering SoS: (i) translating capability objectives; (ii) understanding systems; (iii) assessing requirements and solution options; (iv) monitoring change; (v) developing and evolving SoS architectures; (vi) assessing performance; and (vii) orchestrating upgrades. By rearranging these activities in a timeline, Dahmann *et al.* (2011b) show that this process can be viewed as a wave model (shown in Figure 3), where "Initiate SoS" encompasses activities (i) and (ii), "Conduct SoS Analysis" encompasses activities (i) to (iv), "Develop SoS Architecture" corresponds to activity (v), "Plan SoS Update" corresponds to activity (vi), and "Implement SoS Update" corresponds to activity (vii). In this timeline, normal process flows are depicted by arrows, which optionally have embedded circles indicating that back-and-forth iterations between activities are possible. This workflow highlights some inherent characteristics of SoS, such as: SoS are incrementally developed by evolution iterations; SoS are continuously analyzed against changing contexts and requirements; the development of SoS is constantly receiving inputs from the external environment; the SoS architecture is incrementally developed and can evolve between iterations; external events determine the speed with which new iterations are needed but every iteration delivers a working SoS.

Aiming to support the development of SoS, Maier (1998) defines four main architecting principles: (i) stable intermediate forms: "complex systems will develop and evolve within an overall architecture much more rapidly if there are stable intermediate forms than if there are not"; (ii) policy triage: "the triage: Let the dying die. Ignore those who will recover on their own. And treat only those who would die without help"; (iii) interface design: "the greatest leverage in system architecting is at the interfaces. The greatest dangers are also at the interfaces"; (iv) ensuring cooperation: "if a system requires voluntary collaboration, the mechanism and incentives for that collaboration must be designed in." Hollon and Dagli (2007) apply these principles for developing the ballistic missile defense system, which combines several legacy systems, focusing in particular to the interface design that enabled the collaboration among

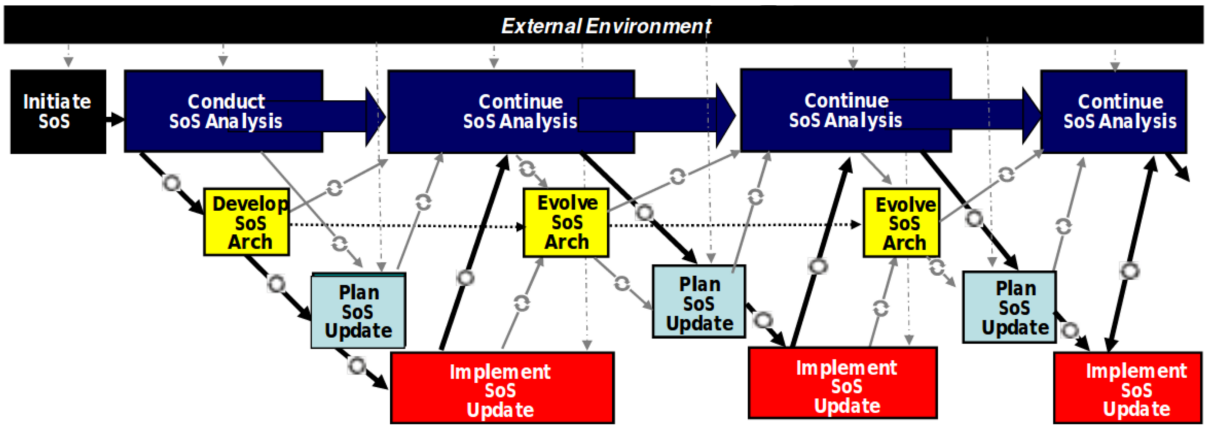


Figure 3 – Wave model for engineering SoS
Source: Dahmann *et al.* (2011b).

constituents.

According to this fourth principle, SoS architects must design efficient ways for constituents to collaborate with each other (BOARDMAN; SAUSER, 2006). Thus, on top of challenges related to scope, complexity, and distribution, the design of SoS is complicated by enabling connectivity among autonomous constituent systems. For instance, mature constituent systems can resist to make changes for interacting with the SoS (Office of the Deputy Under Secretary of Defense for Acquisition and Technology, Systems and Software Engineering, 2008). In this case, SoS architects can use a gateway or encapsulate the constituent system so that required changes to the actual constituents are minimized. Moreover, architecture styles can be used to foster connectivity among constituent systems, such as net-centric architectures, layered architectures, and agent-based architectures (DAGLI; KILICAY-ERGIN, 2009; KOONTZ; NORD, 2012). In particular, Loiret *et al.* (2011) establish a reference model for developing Component-Based Systems of Systems (CBSoS) in which a front-end component is responsible for loading and checking heterogeneous descriptors used for specifying the CBSoS and instantiating the appropriate models. This approach is particularly interesting for SoS since it is not bound to a specific ADL or implementation language.

2.2.2 Model-driven Development

Models increase the abstraction level with which one can reason about complex systems. Different kinds of abstraction levels can be employed for creating such models, thus determining their purpose and the types of supported analysis (KRAMER, 2007). Models can be useful when designing, maintaining, and evolving software systems since each one of these models focus on a particular aspect of the software system, e.g., structure, behavior, workflow, platform, among others. These models can be either static, in which time does not play a role in the model (e.g., data, topology, or logic), or dynamic, in which time plays a key role (e.g., interactions, events, or state) (BRAMBILLA; CABOT; WIMMER, 2012). In this scenario, source code is the ultimate

model of a software system, which can be expressed in programming languages such as C, Java, or Python, and depicts the systems' behavior when executed (FRANCE; RUMPE, 2007). Modern software engineering practices are increasingly relying on different types of models to gain confidence on the design and quality of software systems. These models can be classified between (FRANCE; RUMPE, 2007): (i) development models that represent the software system at a higher abstraction level than source code (e.g., requirements, architectural, implementation, and deployment models), and (ii) run-time models that represent the executing software system.

In Model-Driven Development (MDD) processes, models are first class entities that represent software systems at different abstraction levels. Models can be expressed in Domain Specific Modeling Languages (DSML), which are either graphic or textual notations providing tailored abstractions for concepts of the problem domain (ROYER; ARBOLEDA, 2012). The definition of a DSML encompasses: a concrete syntax, which defines the notation for creating models; an abstract syntax, which defines the vocabulary of domain concepts, e.g., elements supported by the language, relationships between them, and constraints for creating valid models; mappings between abstract and concrete syntax, e.g., stereotypes in classes of a UML Class Diagram that refer to concepts defined by the abstract syntax; and a semantic domain, which specifies how to create well-formed models using domain concepts (ROYER; ARBOLEDA, 2012). A model can also be used to define the abstract syntax, thus defining a metamodel for this DSML.

Model transformations are software artifacts that specify an algorithm for obtaining one model from another (ROYER; ARBOLEDA, 2012). Transformations are a key activity of MDD processes as they successively create models for representing software systems at different abstraction layers, thus helping to bridge the gap between problem and solution domains (FRANCE; RUMPE, 2007). Figure 4 illustrates a generic model transformation scenario between one source model and one target model. The model transformation, which is defined at the metamodel level of the source and target models, is composed of several *transformation rules*, which are functions or procedures implementing a transformation step (ROYER; ARBOLEDA, 2012). An engine can automatically execute these steps for obtaining the target model from the source model. Operational transformations, such as model refinement, abstraction, refactoring, composition, decomposition, and translation, receive the source set of models and produce the target set of models, e.g., a transformation of UML models into Alloy (FRANCE; RUMPE, 2007). Alternatively, synchronization transformations are used for updating the target set of models with changes made to the source set of models (FRANCE; RUMPE, 2007).

According to Czarnecki and Helsen (2006), model transformations can be classified as Model-to-Model (M2M), which transform source models into target models, or as Model-to-Text (M2T), which transform models into source code. Royer and Arboleda (2012) also classify transformations as Text-to-Model (T2M), which can extract models from source code, thus enabling reverse engineering processes. Alternatively, France and Bieman (2001) classify model

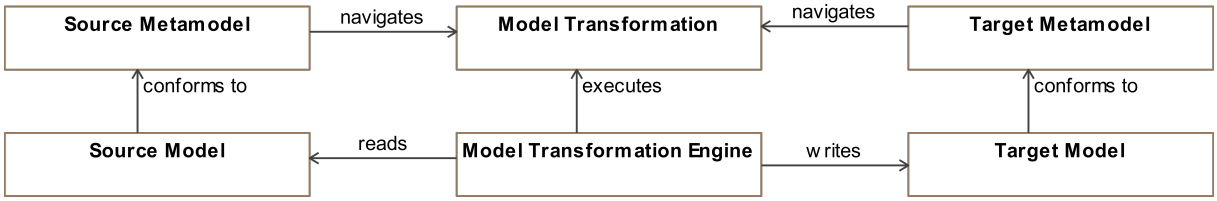


Figure 4 – A generic model transformation scenario
Source: Royer and Arboleda (2012).

transformations as vertical, which are defined between models at different abstraction levels, or horizontal, which are defined between models at the same abstraction level. Vertical transformations can be further distinguished by direction as refinement (i.e., by decreasing the abstraction level) or abstraction (i.e., by increasing the abstraction level) whereas horizontal transformations can be further classified by purpose as migration (i.e., transforming models conforming to one metamodel into another), merge (i.e., unifying separate models), or identification (i.e., selecting specific elements of the source model).

The development of software systems based on models gains popularity with the maturity of modeling tools and languages. For instance, modern generation tools can use models to automatically generate executable source code, thus reducing development time and effort. The semantic consistency between models at different abstraction layers is enforced in MDD processes by the correct-by-construction philosophy (BALASUBRAMANIAN *et al.*, 2006). By employing formal languages in the description of source and target models, it is possible to implement change requests as model refinements, thus enabling consistent updates and modifications to the model (BROY, 2010).

Several variations exist for MDD, with some particularities to each of them (BRAMBILLA; CABOT; WIMMER, 2012): Model-Driven Architecture (MDA) is a branch of MDD based on Object Management Group (OMG) standards. Among other OMG standards, it relies on the Meta Object Facility (MOF)[6] to define the abstract syntax of modeling languages and UML to represent models; Model-Driven Engineering (MDE) applies MDD to the whole spectrum of activities encompassed in software engineering processes, such as evolution, maintenance, and retirement. In this sense, MDE defines approaches that create and systematically transform abstract models of software systems into concrete implementations, i.e., executable source code (FRANCE; RUMPE, 2007).

In the context of SoS, model-driven processes can take advantage of executable models, which provide valuable information for the analysis and construction of SoS architectures (DAGLI; KILICAY-ERGIN, 2009). Graciano Neto *et al.* (2014)[7] investigate MDD practices in the context of SoS and report on a range of approaches that have been using models for documenting the software architecture, generating source code, testing, and representing requirements. Models

[6] MOF, <http://www.omg.org/mof/>
[7] This literature review included 12 studies from 2002 to 2013 reporting on the use of MDD practices in SoS.

that have been created in this domain use semi-formal notations, such as UML and SysML, as well as formal notations, such as Object-Constraint Language (OCL), Discrete Event System Specification (DEVS), and Architecture Analysis and Design Language (AADL). Among tools supporting these approaches, the authors identified open-source initiatives, e.g., extensions to the Eclipse Modeling Framework (EMF)[8] that automate model transformations and code generation, as well as proprietary tools, e.g., MATLab[9]. Nonetheless, the authors identified few approaches exploring MDD for modeling and generating source code for SoS.

2.3 Architecture Description Languages

The definition of a language encompasses three core elements (BRAMBILLA; CABOT; WIMMER, 2012): (i) *abstract syntax*, which can be defined in terms of a grammar (for textual languages) or metamodel (for modeling languages), for creating syntactically correct structures; (ii) *concrete syntax*, which defines the representation (encoding) of structures in terms of textual and graphical notations; and (iii) *semantics*, which convey the meaning of created structures. Originally, ADLs were mainly regarded as formal notations that were similar to programming languages but offered higher level constructs (such as platforms, systems, components, and connectors) for expressing software architectures (SHAW; GARLAN, 1994; CLEMENTS, 1996; MEDVIDOVIC; TAYLOR, 2000). By using these constructs, an ADL would facilitate expressing the creation, refinement, and validation of software systems' elements at the software architecture abstraction level (CLEMENTS, 1996). Nonetheless, the ISO/IEC/IEEE 42010 (2011) standard broadened this definition by classifying "any form of expression used in architecture descriptions" as an ADL.

Shaw and Garlan (1994) describe several desired features of notations for expressing software systems at their software architecture abstraction level. For instance, an ADL must support software systems description in terms of compositions of independent architectural components, which can be packaged as styles, patterns, and correspondences and thus can be reused in different contexts. As several models can be created for describing software architectures at different abstraction levels, an ADL must support a consistent translation among them. Moreover, an ADL must support different kinds of analysis over architecture descriptions, thus enabling its use at different stages of development (e.g., design, validation, or evolution). Since then, additional features have been elicited. For instance, while deemed not mandatory, tool support is increasingly desired for an automated analysis of architecture descriptions (CLEMENTS, 1996; MEDVIDOVIC; TAYLOR, 2000; LAGO *et al.*, 2015). An ADL must also support expressing different structures (or topologies) in which a particular concern is framed, such as understandability, compositionality, refinement, heterogeneity, scalability, and dynamism (MEDVIDOVIC; TAYLOR, 2000). Another desired feature concerns the description

[8] EMF, <http://www.eclipse.org/modeling/emf/>
[9] MATLab, <https://www.mathworks.com/>

of multiple viewpoints (HILLIARD; RICE, 1998). Besides describing software architectures from a run-time viewpoint, i.e., in terms of how components and connectors are arranged together, modern ADLs are also required to describe software architectures from a dynamic viewpoint, i.e., in terms of how the architecture can evolve by means of adding, deleting, reconfiguring, or modifying elements at run-time based on data input, or mobile viewpoint, i.e., in terms of how the architecture can logically move during the execution of the system (OQUENDO, 2004). The description of complex systems can also be facilitated by providing assorted mechanisms for expressing quantification. For instance, first-order logic quantifiers $\forall$ (for all) and $\exists$ (exists/some) can be used for describing global properties, thus increasing the ADL expressiveness, whereas constraints can be added for quantifying relations or attributes in architectural models (HILLIARD; RICE, 1998).

While more than 100[10] languages can be regarded as ADLs, only a fraction of these are widely employed. Aiming to classify and compare different ADLs, Medvidovic and Taylor (2000) define a framework of required and desired features for describing software architectures, such as support for expressing constraints and non-functional properties. In particular, they define *components*, *connectors*, and architecture *configurations* (i.e., topologies) as the high level building blocks that can be used for describing a software architecture: (i) components are units of computation that represent functional elements of a system; (ii) connectors represent the interconnections enabling the communication among components; and (iii) configurations specify the pattern and/or structure in which components and connectors are assembled. In light of the ISO/IEC/IEEE 42010 (2011) standard, ADLs can also be distinguished by underlying formalism, i.e., formal, semi-formal, and informal notations (BASS; CLEMENTS; KAZMAN, 2012).

Formal notations have well-defined syntax and semantics, such as Rapide (LUCKHAM, 1996), Wright (ALLEN, 1997), and π-ADL (OQUENDO, 2004). For instance, π-ADL was created to support the dynamic nature of software architectures by focusing on the description of architectures from both structural and behavioral viewpoints and evolving these viewpoints together over time. Since π-ADL is based on the higher-order typed π-calculus (SANGIORGI, 1992), it is also Turing-complete. Solely, π-calculus is not suitable as an ADL since it does not provide architecture-centric constructs which would support the description of architectural structures (OQUENDO, 2004). The main advantage of using formal ADLs is the possibility of formally representing the software architecture. Moreover, formal architecture descriptions are also more likely to be maintained and followed than an informal one as it can be more readily consulted and treated as authoritative and it can be more easily transferred to other projects as a core asset (CLEMENTS, 1996). As formal models provide a rigorous, accurate, and unambiguous description of the software architecture (MEDVIDOVIC; TAYLOR, 2000), they can also be used for verifying and validating the architecture against selected properties and

[10] Index of current ADLs, <http://www.di.univaq.it/malavolta/al/>

quality characteristics. As a result, formal descriptions offer an important resource in the making of architecture decisions (CLEMENTS, 1996). On the other hand, since formal languages can be difficult for non-technical stakeholders to understand, they require specialized tools and training.

Semi-formal notations have well-defined syntax but lack complete semantics, such as UML (Unified Modeling Language) (OMG, 2011), SysML (Systems Modeling Language) (OMG, 2012), and AADL (Architecture Analysis and Design Language) (SAE International, 2011). Firstly, UML is an Object Management Group (OMG) standard for specifying, visualizing, and documenting software systems. Improvements made since its version 2.0 have encouraged its use in architectural descriptions (IVERS *et al.*, 2004; LAGO *et al.*, 2015). However, UML still lacks a first-order construct for connectors (OZKAYA; KLOUKINAS, 2013). Second, SysML was developed to fill in the semantic gap between systems, software, and other engineering disciplines. Also an OMG standard, SysML has been introduced as an evolution to UML and it reuses a subset of UML 2.0. For example, SysML introduces a Requirements Diagram that can express hierarchy among requirements, which guarantees traceability among them, and a Parametric Diagram that can describe constraints for system property values (e.g., performance, confidence, or weight), which can be useful for integrating specifications and models of the system into engineering analysis models. Finally, AADL describes architectures in terms of distinct components and their interactions aiming to support model-based analysis and specification of complex real-time embedded systems (FEILER; GLUCH; HUDAK, 2006).

Informal notations presented neither defined syntax or semantics for describing software architectures. Therefore, informal models are solely created to provide a high abstraction picture of the software architecture (CLEMENTS *et al.*, 2011). For instance, several informal models can be drafted and later discarded during the design of software architectures. To be part of the architecture documentation, informal models must be accompanied by textual explanations that help to convey architects' intentions, thus hindering reuse and automated analysis of such descriptions (CLEMENTS *et al.*, 2011). Another disadvantage of using informal notations concerns the lack of tool support, which makes models difficult to be kept up-to-date and prone to ambiguous interpretations (CLEMENTS, 1996).

In this scenario, the selection of a semi-formal notation can be seen as a compromise between formal and informal notations as they combine the rigor of formal languages to the understandability of natural languages. However, semi-formal languages often favor higher understandability at the cost of lowered accuracy. For instance, semi-formal languages may conceal mismatches among architectural models that could be propagated to subsequent stages of the development life cycle (MENS; MAGEE; RUMPE, 2010). Thus, semi-formal languages are not adequate for describing software architectures of critical domains since they cannot enforce correctness and consistency of an architecture description (PERSEIL; PAUTET, 2010). Aiming to fill this gap, formal notations are being used in conjunction to semi-formal ones. For example, π-ADL provides an UML Profile for supporting UML as a concrete modeling notation. Thus, an

UML modelling tool can be used for creating semi-formal models while still taking advantage of π-ADL formal foundations for verifying and modeling transformations (OQUENDO, 2006). On the other hand, semi-formal ADLs have also been enhanced with formal foundations. For example, DesyreML (FERRARI *et al.*, 2012) is a SySML profile for the formal description of heterogeneous embedded systems.

Ozkaya and Kloukinas (2013) add to discussion on the evaluation of formal and semi-formal notations by evaluating ADLs in respect to their formal semantics, usability (which is subdivided into easy-to-use formal behavioural specifications and support for user-defined, complex connectors), and realizability. In particular, their evaluation uses as criteria the availability of support for performing formal analysis on architecture models, level of usability of the notations for creating large and complex architectures, and level of protection against the creation of models that cannot be realized. In their study, realizability is a major issue in several formal notations, such as Wright and Rapide, as they allow the creation of invalid models.

To avoid the downside of selecting a single formalism, the research on ontology gains relevance in the Software Engineering area as a possible integrator for different notations and technologies (W3C, 2006). An ontology captures knowledge about some domain of interest for promoting a homogeneous understanding. Different notations can be used for expressing an ontology, ranging from precise, descriptive statements (i.e., informal ontology) to formal models (i.e., formal ontology) (W3C, 2012). A formal ontology, expressed in OWL, represents the domain of interest in terms of classes (which establish the requirements for membership to a class), individuals (which are instances of a class), and properties (which depict relationships among classes, individuals, or literals) (HORRIDGE, 2011). Then, such formal ontology can be automatically processed by computer programs that automatically verify its consistency (i.e., validate the ontology semantics) and infer implicit knowledge (e.g., compute implicit membership of individuals to classes).

2.3.1 Notations in Systems-of-Systems Description

The literature on SoS architecture description has been using varied notations for describing software architectures, including formal, semi-formal, and informal notations as well as combinations among these (GUESSI *et al.*, 2015). Formal notations used or suggested for describing SoS software architecture comprise: COMPASS Modeling Language (CML) (WOODCOCK *et al.*, 2012), Capability Focused Modeling Language (CFML) (IACOBUCCI; MAVRIS, 2011), Finite State Machine (FSM), Message Sequence Charts (MSC) assertions (COOK; DRUSINKSY; SHING, 2007), Petri-nets/colored Petri-nets, fuzzy models, Web Ontology Language (OWL) (W3C, 2012), Web Service Modeling Ontology (WSMO), Vienna Development Method (VDM) specification language (FITZGERALD *et al.*, 2005), and description logic. CML is specifically designed for SoS modeling and analysis. CFML is a declarative notation for describing and evaluating SoS architecture alternatives based on mission dependent metrics, such as probability

of mission success and time to completion. MSC assertions is a formal language which extends the UML message sequence diagram by superimposing UML state-charts in which it is possible to distinguish among mandatory and optional events. Petri-nets and colored Petri-nets have been employed for generating executable SoS models. VDM is a formal specification language for defining interfaces and concrete system specifications.

There is an increasing concern for providing SoS stakeholders with a shared vocabulary, encoded as formal and informal ontology, that can promote interoperability and understandability. For example, Henrie and Delaney (2005) extend the definition of common terms used in traditional systems engineering to the SoS domain for establishing a terminology framework that could assist engineers in understanding, discussing, and designing complex systems. In turn, Nikolic and Dijkema (2007) use OWL for creating a language that specifies the communication among the agents of an agent-based architecture. By requiring all agents to conform to this ontology, a high level modularity can be achieved. In the same perspective, Moschoglou *et al.* (2012) use WSMO, which provides a richer conceptual model and a formal language for semantically describing aspects of web services, publishing constituents' system capabilities, and enabling the automatic discovery, selection, and composition of web services at run-time.

Semi-formal notations have also been applied for describing SoS architectures, in particular: UML, SysML, and the Unified Profile for DoDAF and MoDAF (UPDM) (HAUSE, 2010). The first two languages are well-known OMG standards for the description of software systems. The latter is an UML profile supporting consistent, standardized descriptions of DoDAF (version 1.5) and MoDAF (version 1.2) architectures using UML-based tools as well as a standard for interchanging between these two frameworks. In spite of its popularity, there are concerns related to UML and SysML expressiveness for SoS architectures, as they may not be sufficient for unambiguously and precisely specifying interfaces. Moreover, semi-formal ADLs offer limited support for automatic analysis, which hampers their applicability in subsequent activities of SoS design.

The combination of formal and semi-formal notations has also been suggested for SoS architectural description. For example, Dagli and Kilicay-Ergin (2009) suggest using of UML or SysML for modeling the SoS' static structure, such as user requirements and complex relationships between constituents, whereas Petri-nets or colored Petri-nets are used for building executable models of the SoS. Payne *et al.* (2012) combine SysML and VDM for specifying interfaces of constituent systems. Similarly, Bryans *et al.* (2013) use SysML and CML for specifying interfaces of constituent systems. The model expressed in CML is a collection of process definitions, which encapsulates states and operations expressed in VDM and interact with the environment via synchronous communications. Thereby, formal interface descriptions enable event ordering specifications by means of pre- and post-conditions for operations.

2.3.2 Features of Architecture Description Languages for Systems-of-Systems

The particular nature of SoS, e.g., the independence of constituent systems, their evolutionary development, and emergent behavior, raises additional requirements for their adequate description. For instance, the ways in which constituents interact with each other determine the behaviors of the SoS. Thereby, SoS architecture descriptions should enable reasoning about how these interactions can be organized to enable SoS-wide behaviors to emerge (OQUENDO, 2016c). In this sense, architecture descriptions can help detecting and analyzing emergent behaviors so that undesired behaviors and risks can be mitigated as early as possible. However, there is no consensus on which notations can be adopted or which features are required for expressing SoS software architectures. As a consequence, in spite of the many ADLs that could be employed for expressing SoS software architectures, existing languages often lack the proper set of features for framing the particular concerns of SoS (BATISTA, 2013; GUESSI; CAVALCANTE; OLIVEIRA, 2015; OQUENDO, 2016c).

Nielsen *et al.* (2015) discuss some of the main concerns that must be framed by notations expressing SoS, including: (i) description of well-founded interfaces; (ii) description of autonomous behavior and capabilities and responsibilities of systems; (iii) description of rely/guarantee policies for independently modeling constituents; and (iv) description of desired or undesired behaviors that can be observed by means of model simulations. As SoS are continually evolving, notations must provide well-defined semantics, which will enable the analysis as well as extension of core elements of these notations for representing concerns that were not initially known at design time. Considering that the expressiveness of an ADL is improved by having specific constructs for describing the main aspects of the software architecture (HILLIARD; RICE, 1998), ADLs for SoS should respectively replace the notions of components, connectors, and configurations by the ones of constituents, mediators, and coalitions (GUESSI; CAVALCANTE; OLIVEIRA, 2015). Figure 5 illustrates features that have been included, tailored, or reused from Medvidovic and Taylor (2000) framework of ADLs characteristics for expressing architectures of SoS.

Constituent is the abstraction for a complex entity, e.g., a system or another SoS, that may participate in the SoS. It can be partially specified at design time in terms of an abstract type, which characterizes the main aspects of a constituent system that it makes visible to other constituents and the SoS, so that at run-time concrete constituents complying with this specification can be attached to the SoS. As an independent entity, a constituent can have its own mission, which it may continue to fulfill in spite of its participation in the SoS. The semantics of a constituent is specified by a behavioral model, such as specific sequences over exchanged messages, that are visible at the SoS level.

Mediator is the abstraction for a complex communication channel that enables to exchange information among constituents. In particular, this abstraction is more complex than the

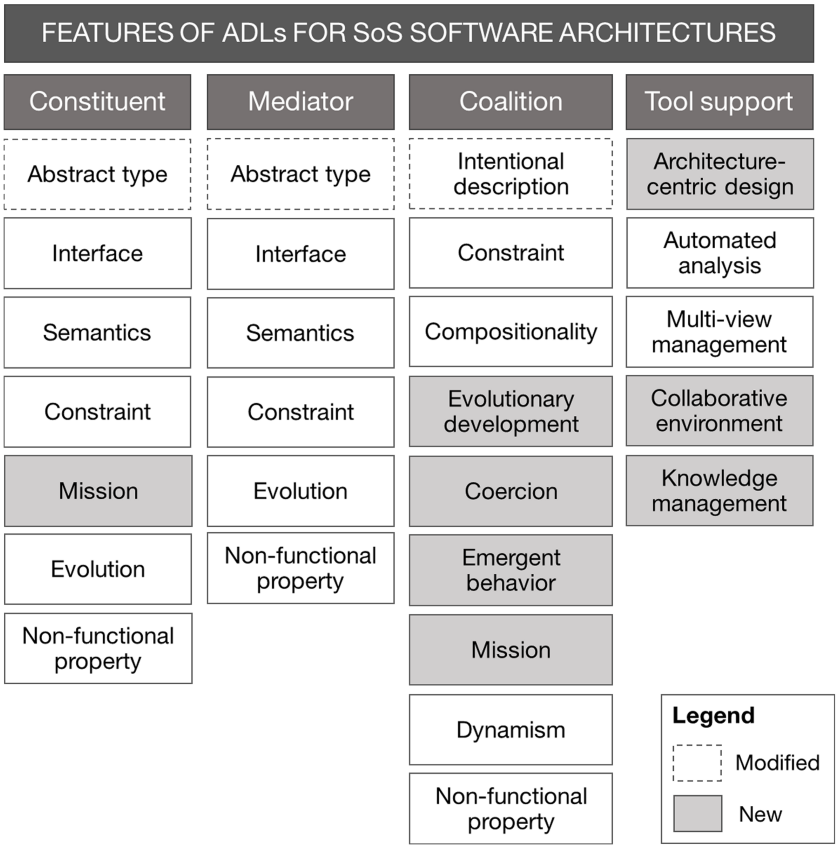


Figure 5 – Features of ADLs for SoS software architectures

Source: Guessi, Cavalcante and Oliveira (2015).

one of connector since mediators accumulate different roles in the architecture (e.g., translating, storing, processing, and coordinating constituents actions). It can be partially specified at design time in terms of an abstract type, which details its tasks and commitments, so that it can be dynamically deployed at run-time as needed. In this sense, the existence of mediators is subjected to the control of the SoS. The semantics of this element clarifies the sequence in which messages will be exchanged and processed.

Coalition is the abstraction representing the configuration of constituents and mediators. The coalition has a global mission, which can be fulfilled solely by the collaboration among its independent constituents. Because the coalition is continuously changing, it is difficult to explicitly determine at design time how constituents can be arranged together. Therefore, this new framework updates the feature of configuration type with the notion of an *intentional description*, which defines a normative model of coalitions that can be formed at run-time. To achieve this goal, it is essential that an ADL also supports expressing formal constraints, which can be automatically evaluated. As in traditional ADLs, the compositionality feature is related to the ability of representing coalitions at different abstraction levels (MEDVIDOVIC; TAYLOR, 2000). For instance, an entire SoS can be simultaneously viewed at a higher abstraction level as a black-box that only shows its main interfaces and external behavior and at a lower level as a network of constituents and mediators.

Tool support is also added to this set of features as it contributes for the usability and dissemination of ADLs. Thereby, this feature encompasses several mechanisms, such as syntax validators, customizable templates, and code generators that can make creating, debugging, and executing models a straightforward task. In regards to SoS software architectures, tailored tools should be able to support (GUESSI; CAVALCANTE; OLIVEIRA, 2015): (i) architecture-centric design, which ultimately obtain source code from consecutive model refinements; (ii) automated analysis for verifying, validating, and simulating SoS architecture descriptions; (iii) multi-view management for visualizing SoS architectures according to different viewpoints or abstraction levels; (iv) collaborative environment for sharing with geographically distributed teams; and (v) *knowledge management* for recording and tracking important design decisions.

Comparison between UML, SysML, and X-UNITY

Three ADLs that have been used for describing SoS architectures, namely UML, SysML, and X-UNITY, are evaluated in light of the features discussed in the previous section (GUESSI; CAVALCANTE; OLIVEIRA, 2015). The first two are well-known semi-formal notations whereas the latter is a formal language. SysML is evaluated in conjunction with CML, a formal language that complements the former with the ability to specify contracts. X-UNITY is a formal notation focused on the description of the interface of constituents and SoS. Despite the existence of several profiles to both UML and SysML, this evaluation is limited to the core concepts of these languages. Moreover, since none of these notations provide run-time support, this evaluation is restricted to the expressiveness of these ADLs at design time.

UML provides 14 diagram types in its 2.4.1 version. These diagrams are grouped under structural diagrams (e.g., class diagram, composite structure diagram, deployment diagram) and behavioral diagrams (e.g., timing diagram, sequence diagram, state machine diagram). From a static viewpoint, several UML constructs could be employed for realizing constituents, mediators, and coalitions, such as classes, packages, components, association classes, and composite structures. Table 2, for instance, lists some diagram types that can be employed for each of the features elicited for the description of SoS architectures. For the sake of simplicity, the class diagram can be used for expressing both constituents and mediators. Abstract types of constituents and mediators can be distinguished using stereotypes. In turn, a state machine diagram or sequence diagram can be attached to the specification of both constituents and mediators for detailing their semantics. Constraints on attributes, associations, and operations of constituents and mediators can be expressed in the Object Constraint Language (OCL)[11]. Evolution is currently not supported in UML because structural diagrams are static.

A coalition expressed in a composite structure diagram can explicitly specify constituents and mediators as parts. Ports and interfaces of the coalition can be internally relayed to any of its parts by means of associations. Moreover, bindings between constituents and mediators

[11] Object Constraint Language (OCL), <http://www.omg.org/spec/OCL/2.4>.

Table 2 – Design time features for describing SoS architectures in UML

Element	Feature	Diagram
Constituent	Abstract type	Class diagram
	Interface	Class diagram
	Semantics	State machine or sequence diagram
	Constraint	OCL expressions
	Mission	Not supported
	Evolution	Not supported
	Non-functional property	UML profile
Mediator	Abstract type	Class diagram
	Interface	Class diagram
	Semantics	State machine or sequence diagram
	Constraint	OCL expressions
	Evolution	Not supported
	Non-functional property	UML profile
Coalition	Intentional description	Composite structure diagram
	Constraint	Not supported
	Compositionality	Composite structure diagram
	Evolutionary	Not supported
	Coercion	Not supported
	Emergent behavior	State machine or sequence diagram
	Mission	Not supported
	Dynamism	Not supported
	Non-functional property	UML profile

Source: Guessi, Cavalcante and Oliveira (2015).

can be expressed as association links. The use of the same name for ports and interfaces is one way of implicitly defining bindings that may occur in a coalition but this feature requires compatible tools. Another downside of using this approach is that it does not support the definition of constraints, such as the ones that express valid bindings between constituents and mediators. A state machine diagram or sequence diagram can also be used to express the emergent behavior of the coalition in terms of the collaborations that take place among its constituents. Regarding compositionality, the class diagram and the composite structure diagram can support the description of SoS from different abstraction levels. The dynamism of a coalition is related to the capability of an SoS to rearrange its own structure. Since this is mainly a run-time feature, it is currently not possible to express it in UML. The description of missions and different levels of coercion is currently not supported. Extensions to core elements of the UML can be investigated to describe non-functional properties.

As UML is a well established standard for modeling software systems, several tools currently support it. For instance, Papyrus Modeling Tool[12] is an open-source graphical editing tool provided as an extension of Eclipse[13]. Papyrus does not support automated analysis,

[12] Papyrus Modeling Tool, <http://www.eclipse.org/modeling/mdt/papyrus>.
[13] Eclipse IDE, <http://www.eclipse.org>.

collaboration, and knowledge management. Even though this tool does not completely support architecture-centric design, it is important to mention that UML does. Therefore, despite the limitations observed in this tool, new features such as for syntax validation and code generation can be integrated due to facilities supported by the development environment.

SysML defines nine diagram types which are grouped under behavior diagrams, structure diagrams, and a requirements diagram (FRIEDENTHAL; MOORE; STEINER, 2015). As a UML profile, SysML extends UML with: (i) a requirements diagram specifying textual requirements and their relationships to each other and to other models; (ii) a block construct defining system's structure and properties; (iii) an activity diagram that modifies the one available in UML to support continuous behavior; (iv) a constraint block construct defining constraints in terms of equations and their parameters; and (v) ports and information flows extending the UML structural model. Constituents and mediators can be individually represented in block definition diagrams and coalitions in internal block diagrams, which are a modification to UML's composite structure diagram. Therefore, a coalition can be expressed in terms of parts[14] and interconnections among them. The support for expressing constraints is improved in this notation since they can be captured in a separate parametric model, which promotes the separation of concerns in the architecture description, and then evaluated by custom analysis tools. The requirements diagram can be adapted represent missions and different levels of coercion. There are several tools supporting SysML, such as Papyrus.

CML is a textual language supporting SoS design. A CML model can be derived from SysML block definition or state machine diagrams, thus enabling to describe the combined semantics of constituent systems as synchronous events. This notation supports the definition of interface contracts for determining the collaboration among constituents. A contract expresses behavioral properties as well as specific policies and constraints that are necessary for coordinating constituents in an SoS. As a formal language, CML supports different types of analysis, such as model checking, testing, and simulation (COLEMAN *et al.*, 2012). In particular, it enables to observe behaviors that emerge from constituents' interaction. Mediators can be described by channels that bind constituents together, which can further characterized by properties, constraints as well as behavioral protocols. There are also explicit mechanisms for expressing mobile channels and mobile processes. However, the intentional description feature is only partially supported since it is not possible to describe abstract mediator types. The level of coercion can be expressed in CML by specifying if an SoS may interfere in the actions performed by any given constituent. It is also possible to frame SoS dynamic behavior by expressing post-conditions (i.e., the SoS state after performing a given operation) for operations. The main tool supporting CML is Symphony[15], an open-source tool based on Eclipse. Symphony collaborates with external proprietary tools for automatically generating CML models from SysML and automating

[14] A *part* refers to a particular use of a block. It may present unique behaviors, properties, and constraints that only apply to its particular usage (FRIEDENTHAL; MOORE; STEINER, 2015).

[15] Symphony IDE, <http://symphonytool.org>

analysis.

X-UNITY is a textual language that addresses the description of interfaces of constituents and SoS. In particular, constituents in a X-UNITY description present internal variables (called context) and exposed variables that are visible to other constituents. Thereby, this language addresses the visibility and compatibility problems that can emerge among interfaces of constituents and SoS when they are attached to an existing coalition. However, there is no tool supporting this language.

Table 3 – Design time features for realizing SoS architectural description in UML, SysML, and X-UNITY

		ADL			
Feature		*UML*	*SysML*	*SysML + CML*	*X-UNITY*
Constituent	Abstract type	●	●	●	●
	Interface	●	●	●	○
	Semantics	●	●	●	-
	Constraint	○	●	●	●
	Mission	-	○	-	-
	Evolution	-	-	-	-
	Non-functional property	○	●	●	-
Mediator	Abstract type	○	○	-	-
	Interface	●	●	-	-
	Semantics	●	●	●	-
	Constraint	○	●	●	-
	Evolution	-	-	-	-
	Non-functional property	○	●	●	-
Coalition	Intentional description	○	○	○	●
	Constraint	-	●	●	●
	Compositionality	●	●	●	●
	Evolutionary development	-	-	○	-
	Coercion	-	-	●	○
	Emergent behavior	○	○	●	-
	Mission	-	○	-	-
	Dynamism	-	-	●	-
	Non-functional property	○	●	●	-
Tool Support	Architecture-centric design	○	○	○	-
	Automated analysis	-	○	●	-
	Multi-view management	○	○	●	-
	Collaborative environment	-	-	●	-
	Knowledge management	-	-	-	-

Source: Guessi, Cavalcante and Oliveira (2015).

Key: ● feature is supported, ○ feature is partially supported, and - feature is not supported

Table 3 summarizes features of the framework that are supported (i.e., the notation

enables to frame the feature in the description), partially supported (i.e., the notation indirectly supports the feature), or not supported (i.e., it is currently not possible to frame this feature in the description) by these notations. This chart shows, for instance, that all four notations support a limited subset of the features that are required for expressing the software architecture of SoS (GUESSI; CAVALCANTE; OLIVEIRA, 2015). In particular, only X-UNITY supports an intentional description of the coalition even though this description is limited to the interface. With exception of the combined use of SysML and CML, the tool support available for the remaining notations is limited or nonexistent. Thereby, among evaluated alternatives, the combination of SysML and CML seems more adequate for representing the architecture of SoS. However, it still lacks run-time support and explicit abstractions that would enable to describe complex mediator types as well as dynamic connectivity among constituents.

2.3.3 SosADL: a Formal Language for Describing Systems-of-Systems

Aiming to overcome limitations commonly found on traditional ADLs, such as lack of support for dynamically changing structures, the Systems-of-Systems Architectural Description Language (SosADL) has been introduced for formally describing the architecture of software-intensive SoS (OQUENDO, 2016c). In SosADL, SoS architectures are represented in abstract terms since concrete systems that will participate in the SoS are not necessarily known at design-time. Then, this abstract architecture can represent a number of concrete architectures, which depict run-time coalitions for the actual constituents of the SoS. The abstract description of a SoS architecture in SosADL encompasses abstract specifications of potential constituent systems, mediators, and their architecture configuration. Thereby, the main architectural elements of the language are the one of system to represent constituents, the one of mediator to represent connectors among constituents, and the one of coalition to represent on-the-fly configurations among constituents and mediators (OQUENDO, 2016c).

The abstract architecture of a SoS encompasses (OQUENDO, 2016c): (i) constituent *systems*, which are architectural elements defined by intention (in terms of abstract systems) and incorporated at run-time; (ii) *mediators*, which are architectural elements defined by intention (in terms of abstract mediators) and created at run-time by the SoS to achieve a goal[16], thus fostering emergent behaviors; and (iii) *coalitions*, which are architectural configurations of mediated constituent systems, partially defined (in terms of abstract systems, abstract mediators, and policies for their on-the-fly configuration) at design time and created at run-time to fulfill a SoS mission. Thereby, SosADL is the first formal language supporting the description of SoS evolutionary behavior. As the internal structure and behavior of constituents can be concealed from the SoS, SosADL enables to express assumptions and guarantees on their internal behavior in terms of expected input or output. The description of mediators, which are controlled by the SoS, can express guarantees about their concrete behavior. Protocols can be attached to ports for

[16] A goal is part of an encompassing mission (SILVA; BATISTA; OQUENDO, 2015).

enforcing a predefined sequence of interactions exchanged with the environment. Conditions for the synthesis of mediators as well as assumptions and guarantees over expected mediating behaviors can also be expressed. Moreover, an intentional description of the coalition is supported by expressing constraints on the expected form of concrete coalitions. Finally, the language supports run-time features by means of *ask* and *tell* actions. The first action allows mediators to check which constraints hold in the environment while the latter allows constituents to update the set of constraints in the environment. Table 4 summarizes the main architectural elements supported by this language. Appendix A presents additional details on SosADL syntax and semantics.

Table 4 – Architectural elements supported by the SosADL language

Architecture Element	Description
System	Represents a constituent in terms of gates (interfaces) exposed to the environment and an internal behavior to fulfill its mission. Gates can group connections and enforce assume-guarantee assertions between: (i) assumptions, i.e., properties that must be satisfied by the environment for the system to deliver its expected behavior; and (ii) guarantees, i.e., properties enforced by the system itself when the assumptions are satisfied.
Mediator	Represents possible connectors among constituents in terms of duties (interfaces) exposed to the environment and an internal behavior. Duties and gates can establish a commitment when a gate fulfills the obligations defined by the duty and the duty satisfies the assumptions of the gate. Moreover, interactions between the two are governed by a specific protocol.
Coalition	Represents architecture configurations in terms of constraints that identify and unify constituents by means of mediators created by the SoS itself. Coalitions can be assembled or disassembled on-the-fly in different ways or with different systems.
Environment	Represents the individual surroundings on which constituents operate. Systems can perceive their environment and act on them through gates.

Source: Adapted from Oquendo (2016c).

The SosADL language, which shares some of the foundations of the well known π-ADL (OQUENDO, 2004), is based on a novel process calculus of the family of the π-Calculus, named π-Calculus for SoS (OQUENDO, 2016d). This novel formalism was developed to encompass two additional paradigms, namely concurrent constraints for modeling locality of processes and inferred bindings for modeling mediation. Processes are modeled as behavioral constraints that only have partial information about the state of the environment, i.e., they have local knowledge. In turn, interactions are modeled as binding constraints, which have been tailored for the particular needs of SoS, namely: (i) support for exogenous bindings between channels as interactions are externally driven by the SoS and not by constituents themselves; (ii) support for constrained bindings between channels as interactions are limited by the local context of the constituent; (iii) support for intentional bindings between channels as interactions are dynamically decided;

and (iv) support for mediated bindings between channels as interactions are deployed on-the-fly by the SoS itself.

Finally, the language is also supported by a development tool suite, named SosADE (OQUENDO *et al.*, 2016), which extends the Eclipse Modeling Framework (EMF) with a tailored environment for SoS design based on SosADL. The tool suite itself is implemented in Xtext (BETTINI, 2016), which provides several custom features for the environment, such as syntax coloring, semantic coloring, error checking, formatting, debugging, structure view, quick fix help, among others.

2.4 Architectural Viewpoints

Viewpoints (and views) are an approach for structuring the architectural description based on the principle of separation of concerns (ROZANSKI; WOODS, 2005). Thus, each viewpoint focuses on a particular concern or set of concerns for describing the software architecture. The need of several architectural viewpoints for describing a software architecture reflects the several concerns that may exist. Moreover, it is easier to communicate the software architecture to the several stakeholders that may exist by using separate viewpoints. In this context, several architecture frameworks, which guide the construction of architecture descriptions for a specific domain or stakeholder community, have been proposed in the literature (ZACHMAN, 1987; KRUCHTEN, 1995; CLEMENTS *et al.*, 2011; HEESCH; AVGERIOU; HILLIARD, 2012). The 4+1 Views (KRUCHTEN, 1995) and the Views and Beyond (CLEMENTS *et al.*, 2011) frameworks are further detailed in this section since they are broadly referenced in the literature.

Kruchten (1995) proposes five viewpoints in the 4+1 Views framework for describing software architectures. The *logical viewpoint* shows a decomposition of the system, considering abstraction, encapsulation, and inheritance for enabling analysis and identification of common elements throughout the architecture. The *development viewpoint* describes the static organization of the software, e.g., packages, libraries, or subsystems, in the development environment. Moreover, this viewpoint lists the rules governing the development architecture, such as partitioning, grouping, and visibility. The information presented in the development viewpoint assists in several development and management activities, such as allocating work to development teams, monitoring work progress, and reasoning about software reuse. The *process viewpoint* captures concurrency and synchronization by mapping elements of the logical viewpoint into processes, i.e., groups of tasks that form an executable unit. For example, the replication of processes can enhance processing load distribution and availability. The *physical viewpoint* describes how the software can be allocated to the hardware. Moreover, different configurations could be used for testing or deployment. Finally, the *scenarios viewpoint* illustrates how the elements in previous viewpoints relate to each other. As a result, the scenarios viewpoint plays a special role in illustrating sequences of interactions that can be useful for both test planning and

construction of the architecture. While the viewpoints can be described at different abstraction levels, with each level potentially framing different concerns, each viewpoint of the 4+1 Views framework is targeted for a particular audience. For instance, the logical viewpoint is intended for end-users as it presents the high-level functions of the system. Conversely, the development viewpoint is intended for developers and managers as concrete details about the system are exposed. Even though the 4+1 Views framework has great relevance, precise guidelines for tailoring the framework are still missing. Moreover, it is not clear how evaluation could be undertaken, such as manual or automated evaluation of the architecture.

Clements *et al.* (2011) propose the Views and Beyond framework. In particular, the authors introduce the term viewtype as a reference to a set of similar viewpoints. The *modular viewtype* describes the system decomposition as a set of implementation units (e.g., classes, subsystems, systems, and layers). This viewtype can focus on the decomposition of the system into code units, the uses relations that exist among these units, the relationship among these units (such as generalization), or the organization of these units into layers. The *components and connectors viewtype* expresses runtime behavior whereas a component (e.g., object, process, or collection of components) is one of the principal processing units of the executing system and a connector (e.g., pipes, repositories, and sockets) is an interaction mechanism for the components. The *allocation viewtype* describes how software can be mapped to elements of the environment (e.g., hardware, file systems, or development teams). This viewtype can focus on the deployment of processes into processing nodes, communication channels, memory stores, and data stores; the implementation of modules in a development infrastructure; and the work assignment of modules to human development teams. It is also possible to find information that transcend a given viewtype. Therefore, architects using this framework can find among each viewtype the information required for their system and then complement this information with documentation that applies to more than one viewtype. Table 5 summarizes the viewpoints, stakeholders, and concerns of the 4+1 Views and Views and Beyond architecture frameworks.

2.4.1 Architectural Viewpoints for Systems-of-Systems

The heterogeneity, dynamism, and large scale which are inherent to modern SoS require architecture frameworks that can cover a larger diversity and number of concerns and create architecture descriptions that can fulfill different tasks throughout software systems life cycle. The Department of Defense Architecture Framework (DoDAF) (DOD, 2010) is one of such frameworks. In its version 2.0, DoDAF provides a common communication language for designing and operating SoS of the military domain. The framework applies net-centric concepts to meet the needs of next generation SoS developed by DoD. In particular, it focuses on promoting the collection, storage, and maintenance of data that will enable to make efficient and effective decisions. Architects can select views from a set of eight viewpoints, namely: (i) all viewpoint describes global aspects of the architecture context that crosscut other views; (ii) capability

Table 5 – Summary of classic architecture frameworks in the literature

Framework	Viewpoint	Stakeholder	Concern
4+1 Views (KRUCHTEN, 1995)	Logical Viewpoint	End-user	Functionality
	Development Viewpoint	Developer, Manager	Organization, reuse, portability, product-line
	Process Viewpoint	System integrator	Performance, availability, software fault-tolerance, integrity, scalability
	Physical Viewpoint	System designer	Scalability, performance, availability
	Scenarios Viewpoint	End-user, Developer	Understandability
Views and Beyond (CLEMENTS *et al.*, 2011)	Modular Viewtype	End-user, Developer, Manager	Functionality, responsability, reuse, and portability
	Component and Connector Viewtype	Developer	Performance, reliability, availability, integrity, fault-tolerance
	Allocation Viewtype	Manager	Performance, security, reliability

viewpoint frames capability requirements, delivery timing, and deployed capability; (iii) data and information viewpoint frames data relationships and alignment structures that are consumed by requirements, processes, systems, and services; (iv) operational viewpoint describes the operational scenarios, activities, and requirements supporting capabilities; (v) project viewpoint describes the relation between operational and capability requirements and current projects; (vi) services viewpoint describes solutions in which performers, activities, services, and their exchanges are arranged to fulfill operational and capability functions; (vii) standards viewpoint describes policies, standards, guidelines, constraints, and forecasts applied to operational or capability requirements, processes, systems, and services; and (viii) systems viewpoint describes solutions in which legacy or independent systems, their composition, interactions, and context are arranged to fulfill operational and capability functions. Viewpoints (i), (iii), (v), and (vii) are traversal to the other four viewpoints. In particular, the framework enforces the documentation of relationships and design decisions so that its artifacts can be used for explaining the requirements and proposed solutions to stakeholders.

The European Space Architectural Framework (ESA-AF) (GIANNI *et al.*, 2011) is another example of architecture framework for SoS. In particular, ESA-AF provides an architectural methodology for the representation of interface specifications, data policies, security requirements, and financial regulations tailored for the European Space-based SoS. Another example is SoSE (SoS Engineering) (BUTTERFIELD *et al.*, 2009), an architecture framework that develops a functional architecture model for an SoS which results in system specification and capability descriptions. In common, these three architecture frameworks use different abstraction levels and employ formal or a combination of formal and semi-formal notations in the creation of architecture models.

2.5 Architecture Rationale

Software architectures abstract from implementation details to focus on the design issues that have greater impact to the quality of software systems. In this scenario, the software architecture can be acknowledged as the work product of subsequent decisions balancing quality and functionality of software systems (JANSEN; BOSCH, 2005). The description of architectural decisions also prevents knowledge vaporization (KRUCHTEN, 2009; FARENHORST; BOER, 2009) and enables to instruct future users and developers of the software architecture, thus ensuring its sustainability (CLEMENTS *et al.*, 2011). Since the architecture description is the main artifact for sharing expertise and best practices, documented decisions play a central part of their content (KRUCHTEN, 1995; KRUCHTEN, 2009; CLEMENTS *et al.*, 2011; ISO/IEC/IEEE-42010, 2011). For instance, documented decisions may describe consequences, justifications, and trade-offs leading to a particular design, e.g., known limitations of a style or pattern, considered standards impacting the operation and/or management of the system, as well as rules and/or constraints for attaching new components to the system. During evolution, documented architecture decisions can help software architects in making new decisions or removing obsolete ones to satisfy changing requirements (JANSEN; BOSCH, 2005). Clements *et al.* (2011) points out some indicators of relevant architecture decisions: (i) it required a significant amount of time to be taken; (ii) it is critical for a particular requirement of the system; (iii) it covers an intricate topic/concern/context for the system; (iv) it has a widespread effect that will be difficult to undo; and, (v) it solves a problem whose solution is not obvious.

To document architecture decisions, implicit and tacit knowledge must be encoded into explicit and documented knowledge. Several approaches have been proposed for encoding architecture decisions. These approaches can be classified among (FARENHORST; BOER, 2009): (i) pattern-centric, which aims at establishing a shared vocabulary of reusable, abstract solutions; (ii) dynamism-centric, which uses ADLs to codify explicit architectural knowledge into models that can be accessed during run-time; (iii) requirements-centric, which intends to bridge the gap between architecture and requirements; and (iv) decision-centric, which prevents knowledge vaporization by capturing the reasoning leading to the software architecture. For instance, Kruchten, Capilla and Dueñas (2009) follow a requirement-centric approach when they propose the use of architecture knowledge as a new crosscutting viewpoint which overlaps the information of the other viewpoints in the 4+1 Views framework. This new viewpoint shows the underlying design rationale and the motivation of selecting a particular concrete design option. This approach encompasses: (i) deciding what information about the design decision will be present and how it will be represented; (ii) relating the design decision to the requirements motivating it; (iii) managing design decisions state and hierarchy; (iv) linking the design decisions to the resulting architecture; and (v) sharing the decisions using communication and documentation mechanisms. Furthermore, different ADLs, models, or tools can be used for realizing this approach.

2.6 Final Remarks

This chapter introduced the basic concepts of the Software Architecture area as well as the current state-of-the-art on architecture description of SoS. As an architecture description provides the basis for evaluating and analyzing software architectures (BASS; CLEMENTS; KAZMAN, 2012), the main part of this chapter was dedicated to the elements that compose such an artifact. The completeness of an architecture description must be evaluated to guarantee that it contains all required data (BASS; CLEMENTS; KAZMAN, 2012). In addition, architecture descriptions can be evaluated in regards to their fit for purpose. For example, to verify that an architecture description communicates the architectural design to all stakeholders, an evaluation can check the readability and effectiveness of the description (CLEMENTS *et al.*, 2011). Active design reviews (PARNAS; WEISS, 1987; CLEMENTS, 2000) can also be used to scan the architecture description for defects in documentation and engage reviewers to also exercise the software architecture design, e.g. by thinking how portions of the architecture could be implemented. The main goal of these evaluations is to reveal false assumptions, inconsistencies, omissions, and other weaknesses that limit or jeopardize the use of the architecture descriptions in subsequent stages of the development life cycle(CLEMENTS *et al.*, 2011).

The architecture description of SoS must be correct and complete to support the analysis and evaluation of coalitions. Due to their inherent characteristics, traditional languages used for describing software systems lack expressiveness for the description of SoS. SosADL, which is presented in this chapter, is the first formal language with tailored building blocks for the architectural description of SoS. While this language supports a normative description of coalitions that can be formed at run-time, further support is needed to verify the feasibility of this design and realize descriptive models that represent actual coalitions derived from it. To address this limitation, the next chapter introduces Ark, a method based on constraint solving for the synthesis of concrete architectures that comply with a SosADL description of the SoS. Thereby, the method can automatically verify if key architectural properties are preserved when moving from SoS abstract architectures to concrete architectures.

Chapter

3

Formal Method for Architectural Synthesis of Systems-of-Systems

Software architectures can establish the blueprint from which coalitions can be formed and reconfigured at run-time. In addition to the identification of user needs, SoS requirements must define which functions are required to accomplish a specific mission and how these functions should be performed (DAHMANN *et al.*, 2011a). The software architecture for a SoS should explain how independent constituent systems can collaborate with each other, functions that are supported by each system, as well as relationships, data flow, and communication protocols that govern the interactions between systems (DAHMANN *et al.*, 2011a). In this scenario, SoS architectural synthesis encompasses creating an architecture solution that meets SoS requirements (GONÇALVES; OQUENDO; NAKAGAWA, 2015). Central to this task are architecture descriptions, which express a SoS from two abstraction levels:

i) an *abstract architecture* provides a normative model for types of coalitions that could be formed among constituents and mediators at run-time. In particular, the architectural configuration is specified by constraints, which abstractly represents a family of run-time coalitions, and
ii) a *concrete architecture* provides a descriptive model for a run-time coalition that could exist at run-time, explicitly showing the interactions that take place among constituents to accomplish the SoS mission. Thereby, it represents a singular operational coalition.

An abstract architecture is feasible if it supports the realization of at least one candidate concrete architecture. In turn, a concrete architecture is correct if it represents a valid arrangement among constituents and mediators that preserves desired architectural properties. In this scenario, a rigorous approach must be in place to perform the architectural synthesis of SoS aiming to guarantee both the feasibility of the abstract architecture and the correctness of all coalitions that can be formed from it, enhancing SoS trustworthiness. Formal languages play an important role

in this approach as they enable the creation of precise, traceable models for abstract and concrete architectures that can be used as input to specialized tools.

This chapter presents Ark, a method supporting SoS architectural synthesis that aims at preventing the creation of ill-formed concrete architectures. To achieve this goal, the method expresses both the abstract architecture of an SoS as well as the rules governing the formation of coalitions in terms of a Constraint Satisfaction Problem (CSP). Thereby, all solutions can be considered as correct candidate concrete architectures. The remaining of this chapter is organized as follows. Before presenting Ark, Section 3.1 reviews the concepts of formal methods and also how this method adheres to best practices for software systems architectural description. Afterwards, Section 3.2 details each step required for this method, including which are artifacts are used and what is the expected outcome. Then, Section 3.3 further describes these artifacts. Finally, Section 3.4 concludes this chapter with final remarks on Ark.

3.1 Foundations of Ark

To perform its mission, which is accomplished by emergent behaviors, an SoS requires specific means for fostering constituents' collaboration (MAIER, 1998; BOARDMAN; SAUSER, 2006). Whereas an abstract architecture describes types of elements that can take part of an SoS, a concrete architecture shows actual coalitions that are formed among constituents at runtime. Therefore, a single abstract architecture can represent a number of concrete architectures depending on which connectivity mechanisms are available. The architectural synthesis of SoS must check if an abstract architecture provides mechanisms for establishing connections among constituents and, also, if these mechanisms are adequate for arranging constituents into a cohesive whole, i.e., a coalition that preserves the architectural properties defined by an abstract architecture.

This chapter presents Ark, a method that employs model finding and model transformations for bridging the gap between abstract and concrete architectures. This method expands on Hofmeister *et al.* (2007) general architecting process by adding specific activities for the synthesis of SoS. In particular, the method defines coalitions as a set of constraints, i.e., descriptive specifications expressed as formulas or Boolean expressions (JACKSON, 2012). As formal notations are employed to describe both abstract and concrete SoS architectures, tailored tools can be developed to automatically analyze and confirm the correctness of created architecture models.

In adherence to the ISO/IEC/IEEE 42010 (2011) standard recommendations, this method identifies: (i) main architectural elements of SoS; (ii) viewpoints for describing SoS architectures, namely an abstract viewpoint and a concrete viewpoint; and (iii) model kinds, for expressing the architectural models that compose the SoS architecture description. In particular, the method adopts three model kinds: a conceptual model (TASoS) that provides the basic theory for

representing SoS architectures in terms of a constraint satisfaction problem (described in detail in Chapter 4); a formal language (SosADL), which offers the basic elements to describe abstract types of systems, mediators, and coalitions; and a metamodel of the instance generated by the constraint solver (Solution), which represents a concrete architecture as sets of elements (also described in detail in Chapter 4). Figure 6 shows a fragment of the ISO/IEC/IEEE 42010 (2011) standard conceptual model for architecture description that is tailored for SoS description.

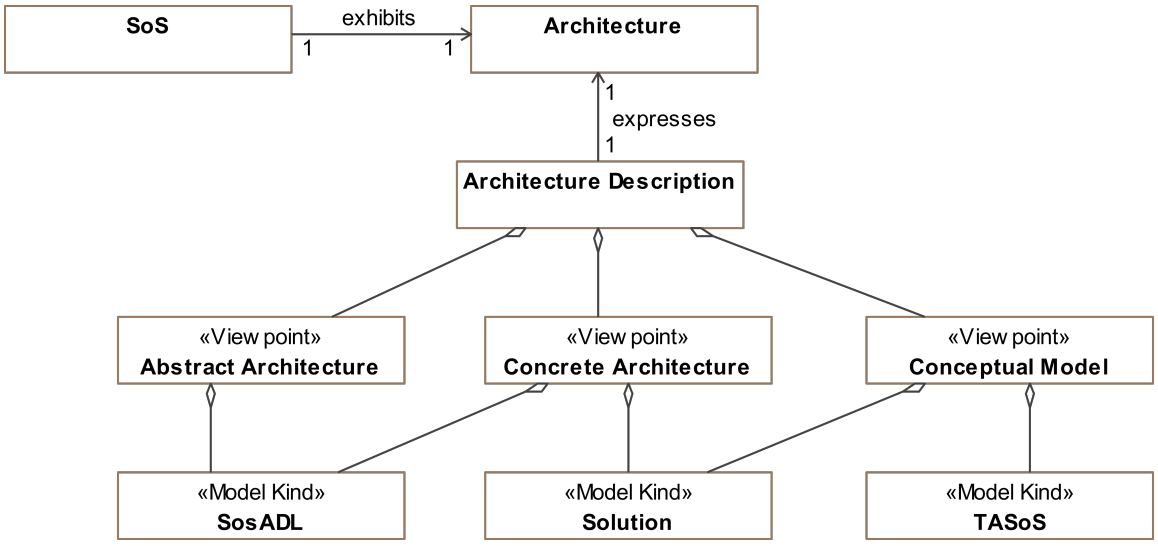


Figure 6 – Instance of ISO/IEC/IEEE 42010 (2011) standard for SoS description

3.2 Structure of Ark

Different forces play a role in the architectural synthesis of SoS abstract and concrete architectures. Abstract architectures are impacted by SoS evolutionary development, which can add types of constituents, mediators, and behaviours that were not initially envisioned, whereas concrete architectures are also impacted by SoS dynamic reconfiguration. While the abstract architecture evolves, it also changes what type of concrete architecture can be realized at run-time. Moreover, the concrete architecture can be reconfigured due to environmental conditions (e.g., lost of connection) or constituents independent management or operation (e.g., incorporation or loss of constituent systems), resulting in different arrangements among constituents and mediators.

In this scenario, the Ark method adopts constraint satisfaction techniques to support the synthesis of SoS concrete architectures that comply with a given abstract architecture. To accomplish this task, this method consumes and produces three main architecture models: (i) *abstract architectures*, which are high-level architectural models of a SoS describing its abstract types of constituents, mediators, and coalitions; (ii) *conceptual models*, which offer an alternative representation for the SoS abstract architecture in terms of constraints; and (iii) *concrete architectures*, which are detailed models of a SoS architecture describing its actual

constituents, mediators, and configurations. These three artifacts are intrinsically related to each other. Conceptual models frame the same architectural elements described in the abstract architecture. Constraint solving tools can be used for automatically analyzing this conceptual model, whose solutions represent alternative concrete architectures in which the purposes, properties, and constraints of the abstract specification are satisfied. Figure 7 illustrates the four steps encompassed in this method, also outlined in Table 6. Following, each step is further explained.

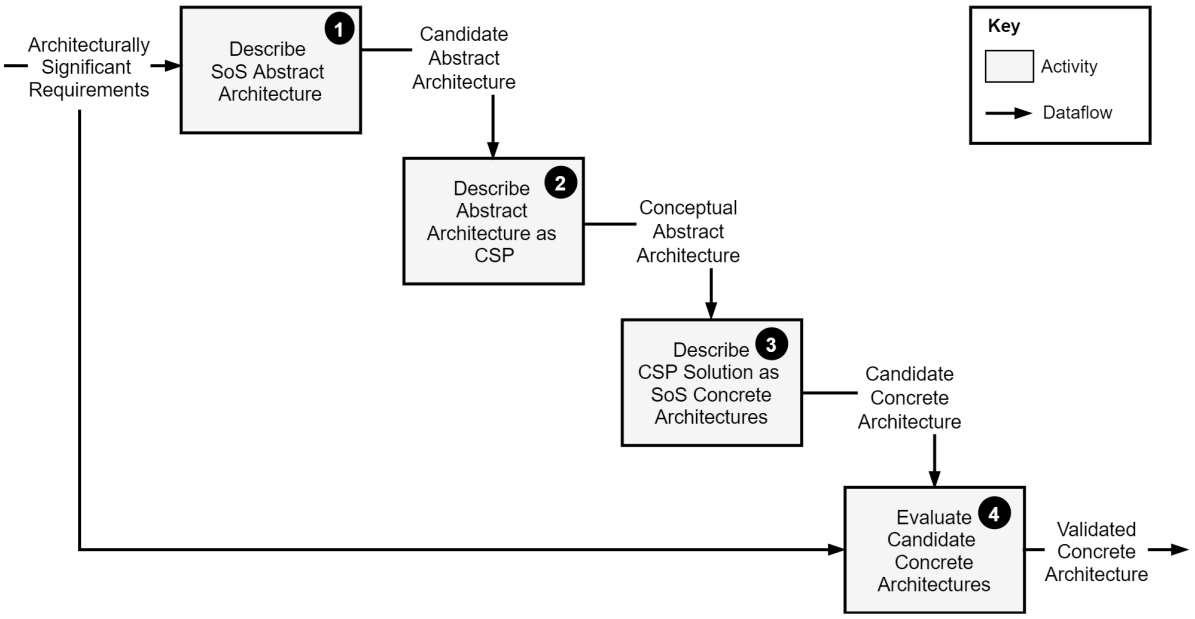


Figure 7 – General model of Ark for SoS architectural synthesis

3.2.1 Step 1: Describe System-of-Systems Abstract Architecture

The first step concerns understanding the SoS mission creating an **abstract architecture** that describes key constituent systems and available communication capabilities embodied in mediators supporting their collaboration. The description of abstract types of constituents defines properties, interfaces, and constraints that must hold in the environment to enable their participation in a coalition. Due to constituents independence, this abstract description is focused on relevant aspects of constituents for the SoS mission. In turn, the description of abstract types of mediators defines properties and interfaces that it exposes to the environment for establishing communication links among constituents. This abstract description also contains abstract types of coalitions, i.e. mediated architectural configurations, defining a normative design for the way in which communication links can be instantiated among constituents. This design may support, for instance, only a layered or client-server architecture configuration among constituents. To create such a description of the SoS, architects consider Architecturally Significant Requirements (ASRs) identified during architectural analysis and elaborate a candidate abstract architecture that satisfies these requirements. As main output, architects obtain a candidate abstract architecture

Table 6 – Outline of activities in the Ark method

Step	Description	Input	Output
1	Architects describe the SoS in terms of an abstract description, which comprises types of systems and mediators that can participate in the architecture, and types of coalitions that can be formed at run-time.	ASRs for constituents, mediators, coalitions, and/or missions.	Description of an abstract architecture.
2	Architects translate an abstract architecture in terms of a constraint satisfaction problem that is recognized as a valid model by the constraint solver	Description of an abstract architecture.	Conceptual model of the abstract architecture.
3	Architects find solution for the constraint satisfaction problem that represent a concrete architecture that satisfy the properties of the abstract description.	Conceptual model of the abstract architecture.	Solution of constraint satisfaction problem corresponding to a candidate concrete architecture.
4	Architects transform solutions into concrete architectures and evaluate whether they correspond to desired instances of the abstract architecture.	Solution of constraint satisfaction problem.	Concrete architectures that preserve desired architectural properties.

that outlines the general structure and behavior of potential coalitions. This step is repeated whenever ASRs change.

Input: As mandatory input, this step receives one or more ASRs, related to constituents, mediators, coalitions, and/or missions.

Output: This step returns a candidate abstract architecture of the SoS, which describes abstract types of constituents, mediators, and coalitions.

3.2.2 Step 2: Describe Abstract Architecture as a Constraint Satisfaction Problem

The second step concerns expressing candidate abstract architectures obtained in the previous step as an instance module of TASoS, a **conceptual model** that can be automatically analyzed using constraint solving tools. This conceptual model frames abstract types of constituents, mediators, and coalitions as well as relevant relations that exist among them. In particular, this conceptual model is expressed in terms of constraints, describing rules that relate a specific set of concrete architectures to a given type of coalition or enforcing a particular connectivity requirement between constituent systems. Thereby, a SoS conceptual model defines communication links that can be established between constituent systems that can participate in the coalition. To create this conceptual model, architects map architectural elements of the SoS abstract architecture in TASoS by extending abstract definitions of model and creating custom

connectivity constraints on their relationship. Moreover, architects can add constraints that rule out families of concrete architectures, hence limiting architectural configurations that can be realized at run-time. If the resulting instance module cannot be recognized as a valid problem by the constraint solver, the architect can decide whether to go back to Step 1 for correcting the abstract architecture or to select another candidate abstract architecture.

Input: As mandatory input, this step receives a candidate abstract architecture of the SoS.

Output: This step returns a conceptual model of the SoS abstract architecture.

3.2.3 Step 3: Find Solution for Constraint Satisfaction Problem

To synthesize concrete architectures for the SoS, the third step of the method initially checks the feasibility of the abstract architecture. The feasibility can be validated if there is any assignment of values to the constraints that evaluates the SoS conceptual model true. In this case, solutions for this conceptual model represent **concrete architectures** of the SoS that adhere to the abstract coalition type defined by the abstract architecture. Since a conceptual model is already expressed in a format that can be recognized by off-the-shelf constraint solvers, this evaluation can be automatically performed. In particular, model finding technique is used to search for a solution under a predefined scope, that constrain the analysis space and, consequently, the size of the solutions that can be discovered. Alternatively, different scope sizes can be used to evaluate different scenarios. If no solution exists for this conceptual model under this selected scope, the architect can try to run the analysis again for a larger scope or return to Step 1 and select an alternative abstract architecture for the SoS. Optionally, the SoS architect can also instruct the constraint solver to check the validity of assumptions regarding a particular architectural property that the conceptual model is expected to satisfy. If an assumptions fails, the constraint solver produces a counter-example, which represents a concrete architecture of the SoS that violates this assumption. Otherwise, the architect can infer that the assumption holds for that particular scope and repeat the analysis for a larger scope. As a result, the architect gains confidence in the abstract architecture specification and may proceed to the evaluation of concrete architectures that satisfy these assumptions.

Input: As mandatory input, this step receives a conceptual model that expresses SoS abstract architectures in terms of constraints.

Output: Provided that the conceptual model is valid and that it describes a feasible abstract architecture, this step returns a solution that corresponds to a candidate concrete architecture for the abstract coalition type.

3.2.4 Step 4: Evaluate Candidate Concrete Architectures

The fourth step of the method concerns the evaluation of candidate concrete architectures obtained in the previous step against ASRs and its abstract description. First, the solution returned

by the constraint solver must be expressed in a notation that SoS architects are more familiar with. Then, architects can review the artifacts related to different coalitions for confirming that the concrete architecture is in-line with what they have originally envisioned. If not successful, SoS architects can decide whether to change abstract coalition types (i.e., return to Step 1) or to add new constraints to the conceptual model (i.e., return to Step 2) for ruling out ill-formed coalitions. Moreover, alternative concrete architectures can be compared with each other aiming to elect one of them as the validated concrete architecture, which is then used as supporting evidence to architecture reviews, e.g. as a coalition prototype for which the presence of desired properties, such as performance, dependability, among others, can be evaluated, or as part of feasibility studies that document ASRs met by the design and elaborate on architectural concepts or technical approaches that can be used for realizing them (OBBINK *et al.*, 2002). Validated concrete architectures can also be used for simulating the emergence of new behaviors that can only be observed when constituents interact with each other. This step can be repeated whenever SoS architects identify a new concrete architecture for the SoS.

Input: As mandatory input, this step receives a solution for the constraint satisfaction problem representing the SoS architecture obtained in the previous step.

Output: This step produces one or more concrete architectures that hold desired architectural properties as well as adheres to the abstract architecture of the SoS.

3.3 Artifacts

The architecture description of SoS must employ notations with well defined semantics for effectively using these models in their analysis (NIELSEN *et al.*, 2015). In this scenario, two formal notations were selected to express architectural models in Ark: SosADL (OQUENDO, 2016c), an ADL that has been specifically created for describing SoS architectures, and TASoS, a conceptual model implemented in Alloy (JACKSON, 2012) for the representation of SoS architectures in terms of constraints. Table 7 summarizes the main elements of architectural models expressed for each notation. The content and purpose of each of these artifacts is further explained in the following sections.

3.3.1 Abstract Architecture

An abstract architecture defines the main concepts or properties of a SoS embodied in abstract types of elements, relationships, and in the principles governing its design and evolution over time. The abstract architecture can be perceived as an open structure from which several concrete architectures can be synthesized. Nielsen *et al.* (2015) refer to an abstract architecture as a conceptual description of an envisaged SoS. Since concrete systems that will actually participate in the coalition are not necessarily known at design time, the abstract architecture defines abstract types of constituents, which can be identified and incorporated to the coalition at

Table 7 – Notations for the description of abstract architectures, concrete architectures, and conceptual models

Artifact	Notation	Language Elements
Abstract Architecture	SosADL	*System* is an abstract type of system that can participate in the SoS. It can engage or disengage from the SoS at run-time by their own decision. *Mediator* is an abstract type of mediator that realizes the interaction between systems. It can be dynamically created, modified, and destroyed by the SoS. *Sos* is an abstract type of coalition that combines constituents and mediators into a cohesive whole. The Sos configuration is defined by intention, i.e., it defines policies for selecting and binding declared constituents using declared mediators that can be created by the SoS itself.
Conceptual Model	Alloy	*Signature* is a set of objects. *Relation* is a function relating objects of different sets. It can also be understood as a field of a signature. *Fact* is a constraint that is assumed to hold. *Assertion* is a constraint that is intended to hold but whose satisfaction must be checked.
Concrete Architecture	SosADL	*System* is a concrete system that participates in the SoS. It can enter or leave the coalition without the control of the SoS. *Mediator* is a concrete mediator that realizes the interaction between systems. *Sos* is an alliance between concrete constituents and mediators. The Sos configuration explicitly defines the topology of this alliance which is realized by concrete bindings among its participants.

run-time. Thereby, abstract architectures support a normative model of SoS without having to constrain which will be its actual constituent systems. Architectural descriptions of SoS from an abstract viewpoint frame which are the types of mediators that can be dynamically realized to support constituents' interaction. As such, architects can use the abstract architecture to focus on the policies governing the interactions among constituents that can foster emergent behaviors. Moreover, the architecture description is completed by defining constraints for arranging these elements into cohesive coalitions so that different ones can be realized at run-time with different elements and/or different configurations to fulfill the SoS mission.

The main elements of an abstract architecture expressed in SosADL encompass abstract types of *systems*, *mediators*, and *coalitions*, which can be used (and reused) across several SoS. The system is an abstraction for a complex, independent entity (e.g., system or SoS) that may participate in the SoS, mediator is an abstraction for a complex, dependent connection that supports the communication between constituent systems, and coalition is an abstraction for the intentional, dependent architectural configuration that can be created based on abstract types of constituents and mediators, constraining which sort of configurations can be formed

at run-time (GUESSI; CAVALCANTE; OLIVEIRA, 2015). The description of abstract types of constituents, mediators, and coalitions is enriched by the specification of interfaces, which are predefined interaction points exposed by these architectural elements to the environment supporting the definition of a high-level behavior models for the SoS. Source code 1[1] illustrates the abstract coalition type of an SoS composed of clients (system abstraction) and exactly two servers (mediator abstraction), defined in lines 8-10. The coalition type has a gate and a connection which receives an integer value as input (lines 4-6). Constituents of this coalition may be arranged according to the policy described in lines 12-19, which states that all clients in a coalition must be linked to exactly one server.

Source code 1 – Example of abstract coalition type expressed in SosADL

```
1 with ClientServerDefinitions
2 sos ServersXor is {
3   architecture Coalition() is  {
4     gate unusedGate is {
5       connection unusedConnection is in{t}
6     } guarantee { protocol allowAll is {repeat{anyaction}}}
7     behavior main is compose {
8       server1 is server
9       server2 is server
10      clients is sequence {client}
11    } binding {
12      forall { c in clients suchthat
13      (
14      unify one {c::rr::req} to one {server1::rr::req}
15      and unify one {server1::rr::ack} to one {c::rr::ack}
16      ) xor (
17      unify one {c::rr::req} to one {server2::rr::req}
18      and unify one {server2::rr::ack} to one {c::rr::ack}
19      )
20 }    } } }
```

3.3.2 Conceptual Model

The SoS conceptual model expresses the abstract architecture in terms of constraints. In particular, this model formally specifies SoS architectural elements as concepts and intentional bindings as constraints between these concepts. A constraint can for instance define that any concrete architecture must have all constituents connected by mediators, hence preventing disconnected topologies. Thereby, this model enables the specification of global constraints that

[1] This source code was elaborated by the Jérémy Buisson and Jean Quilbeuf, members of the ArchWare research group.

will prevent an entire group of ill-formed concrete coalitions and promote the trustworthiness of run-time SoS architectures. Moreover, the description of abstract architectures as a constraint satisfaction problem enables using off-the-shelf constraint solvers in the automated analysis of concrete architectures that adhere to this specification. Source code 2 illustrates an excerpt of the constraint problem expressed in Alloy that corresponds to the abstract coalition type presented in Source code 1. This description imports `tasos` library (line 1), the conceptual model of SoS architectures. Chapter 4 presents TASoS and details how to express SosADL architectural elements in Alloy. Additional libraries can be imported by this model (line 3), preserving the structure of the original SosADL project in that it is based on. To create an instance module of TASoS, architects must extend existing definitions with new elements that will frame a particular SoS, e.g., `ServersXor`, `unusedGate`, and `unusedConnection`, which are respectively defined in lines 6, 7, and 8. The behavior of the architecture is used for customizing definitions introduced by this SoS, e.g., the signature `server1` extends the abstract signature `server` which is defined in the `ClientServerDefinitions` library (line 12). Also, architects translate intentional bindings of the abstract architecture into constraints (lines 18-39). Finally, the model can have one or more execution commands, such as the one shown in line 43, that either attempt to find an instance within a predefined analysis scope or produce a counterexample for a property defined as an assertion.

Source code 2 – Excerpt of abstract coalition expressed in terms of constraints

```
1 open tasos
2 open util/ordering[tasos/Architecture] as A0
3 open ClientServerDefinitions
4
5 -- Architecture(s) Declaration(s)
6 sig ServersXor extends Sos{}
7 sig Coalition_unusedGate extends Gate{}
8 sig Coalition_unusedGate_unusedConnection extends Inward{}
9
10 ...
11 -- constraints about constituents instances in the coalition
12 one sig server1 extends server{}
13 one sig server2 extends server{}
14 sig clients extends client{}
15
16 -- abstract unifications that can exist in the architecture
17 sig Coalition extends Architecture{}{
18   all c: clients |
19     (((unify[client_rr_req&(c.~owner),
20         server_rr_req&(server1.~owner)] and
21       unify[server_rr_ack&(server1.~owner),
```

```
22          client_rr_ack&(c.~owner)])
23      ) or
24      ((unify[client_rr_req&(c.~owner),
25          server_rr_req&(server2.~owner)] and
26        unify[server_rr_ack&(server2.~owner),
27          client_rr_ack&(c.~owner)])
28      )
29      and not (
30        ((unify[client_rr_req&(c.~owner),
31            server_rr_req&(server1.~owner)] and
32          unify[server_rr_ack&(server1.~owner),
33            client_rr_ack&(c.~owner)])
34        ) and
35        ((unify[client_rr_req&(c.~owner),
36            server_rr_req&(server2.~owner)] and
37          unify[server_rr_ack&(server2.~owner),
38            client_rr_ack&(c.~owner)])
39        )
40      ))
41  }
42  ...
43  run {instanceOfCoalition} for 3 but 3 Architecture, 8
      ArchitecturalElement, 7 Port, 14 Connection, 32 Unification, 3
      Relay
```

3.3.3 Concrete Architecture

Differently from an abstract architecture, a concrete architecture explicitly defines the architectural configuration of coalitions, identifying concrete constituents and establishing interconnections that will enable information exchange among them. It can also be referred to as a run-time software architecture of a particular SoS since all of its constituent systems are known. Kenley *et al.* (2014) refer to concrete architectures as allocated architectures, which represent candidate solutions. A candidate solution represents assorted distribution of functionality among constituents, hence potentially showing varying levels of quality. In fact, each concrete architecture is unique in the way that constituent systems interact with each other, depending on which communication links have been instantiated (MAIER, 1998). To be feasible, an abstract architecture must have at least one possible concrete architecture adhering to its specification. A single abstract architecture can designate a number of concrete architectures that comply with its specification. New concrete architectures should be realized when constituents are incorporated to the coalition, which can require changes to current constituent systems and/or associated mediating elements as well as new instances of communication links among them. To be correct,

all concrete architectures must comply with the policies defined by an abstract architecture and satisfy the constraints defined by its corresponding conceptual model.

To facilitate the analysis of concrete architectures by architects, Ark also selects SosADL for expressing concrete architectures. The main elements of such description are the same ones of the abstract architecture, i.e., *systems*, *mediators*, and *coalitions*. Nonetheless, the description of a coalition is different in that: it concretely specifies constituents and mediators that participate in the SoS, and it explicitly defines the communication links that exist among systems and mediators within the coalition. Thus, the concrete instance of a coalition specifies the architectural configuration of a SoS in the traditional way that ADLs usually specify configurations instead of defining policies as in abstract architectures. Source code 4 illustrates a candidate concrete coalition expressed in SosADL for the abstract coalition type presented in Source code 1. In this instance, there are exactly two servers and two clients (identified as clients1 and clients2) and each client communicates with only one server.

Source code 3 – Excerpt of constraint solver solution

```
1 ----INSTANCE----
2 ...
3 tasos/Connection={...}
4 tasos/Connection<:owner={...}
5 tasos/Connection<:hasDatatype={...}
6 tasos/Inward={...}
7 this/Coalition_unusedGate_unusedConnection={...}
8 ClientServerDefinitions/client_rr_ack={...}
9 ClientServerDefinitions/server_rr_req={...}
10 tasos/Outward={...}
11 ClientServerDefinitions/client_rr_req={...}
12 ClientServerDefinitions/server_rr_ack={...}
13 tasos/Port={...}
14 tasos/Gate={...}
15 this/Coalition_unusedGate={...}
16 ClientServerDefinitions/client_rr={...}
17 tasos/Duty={...}
18 ClientServerDefinitions/server_rr={...}
19 tasos/ArchitecturalElement={ServersXor$0, clients$0, clients$1,
       clients$2, clients$3, server1$0, server2$0}
20 tasos/ArchitecturalElement<:hasPort={...}
21 tasos/Sos={ServersXor$0}
22 tasos/Sos<:arch={...}
23 this/ServersXor={ServersXor$0}
24 tasos/Mediator={server1$0, server2$0}
25 ClientServerDefinitions/server={server1$0, server2$0}
```

```
this/server1={...}
this/server2={...}
tasos/System={clients$0, clients$1, clients$2, clients$3}
ClientServerDefinitions/client={clients$0, clients$1, clients$2,
    clients$3}
this/clients={clients$0, clients$1, clients$2, clients$3}
...
```

Source code 4 – Example of concrete coalition expressed in SosADL

```
with A1_ServersXor_library
sos ServersXor0 is {
  architecture Coalition0() is {
    gate unusedGate0 is {
      connection unusedConnection0 is in{RangeType0}
    }
    guarantee {
      protocol allowAll is {
      repeat {
      anyaction
    } } }
    behavior main is compose {
      server10 is server10
      server20 is server20
      clients0 is clients0
      clients1 is clients1
    }
    binding {
      unify one {server20::rr0::ack0} to one {clients0::rr1::ack1}
    and
      unify one {server10::rr1::ack1} to one {clients1::rr0::ack0}
    and
      unify one {clients1::rr0::req0} to one {server10::rr1::req1}
    and
      unify one {clients0::rr1::req1} to one {server20::rr0::req0}
}   } }
```

3.4 Final Remarks

The formal representation of SoS architectures plays a central role in the synthesis and analysis of concrete architectures since it enables to verify the presence of key architectural

properties, hence validating the architectural design. To confirm the feasibility of SoS architectures, this chapter presented Ark, a method for synthesis of SoS concrete architectures. Ark is intended to support normative models of SoS architectures in which only abstract types of constituents, mediators, and coalitions are specified at design time. This method is composed of four steps: (i) description of SoS abstract architectures; (ii) translation of SoS abstract architecture to constraints; (iii) search of solutions for the resulting constraint satisfaction problem; and (iv) evaluation of solutions obtained by the constraint solver as candidate concrete architectures. In particular, Ark prevents the realization of ill-formed concrete architectures by formalizing SoS architectures in terms of a constraint satisfaction problem, enabling to use constraint solving technologies in the realization of concrete architectures that are correct-by-construction.

Ark is further explained in the following chapters. First, Chapter 4 TASoS, a theory for modeling SoS architectures in terms of a constraint satisfaction problem. Inspired by elements of the SosADL notation, this model is the basis for the verification of abstract architectures, whose feasibility must be confirmed. Chapter 5 details SoSy, a tool developed for assisting architects in tasks comprised by the Ark method.

Chapter

4

A Theory for Software Architectures of Systems-of-Systems

New abstractions are needed for describing and reasoning about SoS dynamic architectures and interfaces (NIELSEN *et al.*, 2015). In particular, abstractions should support the description of independent constituent systems whose interactions are mediated towards accomplishing a global mission. The SosADL introduces the concept of intentional bindings as an abstraction for dynamic connectivity, which is the capability associated with the on demand realization of connections enabling the collaboration among concrete constituents (BOARDMAN; SAUSER, 2006). In particular, bindings are expressed as relations that identify which constituents' connections fulfill the commitments of which mediators. In this scenario, the feasibility of an abstract architecture implies that at least one concrete coalition adhering to its abstract description exists. While synthesizing concrete coalitions from an abstract architecture, a number of different configurations can be obtained as new concrete systems are identified and attached to the coalition. However, it is not possible to assume that all coalitions are correct. While the abstract architecture evolves, it is important to preserve its dynamic connectivity so that it does not reach a state for which there cannot be concrete architectures adhering to the specification.

To check the feasibility of abstract architectures and help to uncover hidden flaws in the design of SoS, this chapter presents a conceptual model for SoS architectures, named TASoS. In particular, this model employs Alloy (JACKSON, 2012), a formal method expressing the fundamental concepts related to software systems behavior (JACKSON, 2015), for modeling the main architectural elements framed by SosADL, namely constituents, mediators, and coalitions, and expressing connectivity as constraints. Thereby, this model is able to characterize SoS concrete architectures in terms of a satisfiability problem so that constraint solvers can be used for automatically determining the feasibility of an intentional description. In this scenario, constraint solvers will play a key role in the synthesis of SoS concrete architectures by producing

a valid coalition that satisfies the abstract architecture, if any exists within the analyzed space.

The remaining of this chapter is organized as follows. First, Section 4.1 introduces the main concepts of the Alloy language that are needed for understanding this conceptual model. The TASoS model encompasses two separate modules: (i) a core module, which defines concepts for the main architectural elements supported by the abstract syntax of SosADL (presented in Section 4.2); and (ii) an instance module, which is a customization of the core module for the particular elements described by the SoS under analysis. Thereby, the core module can be reused across SoS models whereas instance modules must be created for each SoS. Section 4.3 describes tailored functions that could facilitate the creation of instances of the core module. The mapping between the SosADL abstract syntax and TASoS is discussed in Section 4.4. Finally, Section 4.5 concludes this chapter with final remarks on the use of formal methods and, in particular, constraint solvers in the context of software architectures.

4.1 The Alloy Language

Alloy is a formal language that expresses software systems in terms of constraints, i.e., formal statements about the software system which are also referred to as formulas or Boolean expressions (JACKSON, 2012). This language combines quantifiers from first-order logic with operators of relational calculus for expressing the behavior of software systems. Instead of defining test cases (as in testing), the analysis performed by Alloy is based on checking the validity of a property (i.e., a constraint). To guarantee efficiency, the analysis performed by the language (i.e., model checking and model-finding) is complete under a predefined execution scope, i.e., the analyzed space (also referred to as solution space) is finite. Therefore, the scope must be carefully selected so that a sufficiently large space is considered for analysis. Nonetheless, a small scope still yields a sizable solution space (in the order of 10^8 clauses) which is sufficiently large for discovering problems in small instances of the model (JACKSON, 2002). In fact, the analysis performed by the language is grounded on the premise that even small instances of a model can illustrate flaws, which arise from incorrectly handling shapes (JACKSON, 2012). Furthermore, the power of this analysis is greater than one that could be achieved by specifying the problem in Java, which would lack support for generating arbitrary samples, performing exhaustive checking on test cases, and visualizing the results.

Atoms are the primitive entities of an Alloy model presenting three inherent properties: (i) indivisible; (ii) immutable; and (iii) uninterpreted. A *signature* defines an atom that behaves as a set, i.e., they have a containment relationship to other atoms. It is also possible to define relations among atoms. Relations can also be defined among atoms of different sets and they can be defined as unary, binary, and ternary relations. Constraints that always hold in the model can be declared inside a signature declaration or as part of a *fact*. Because Alloy is a declarative language, the order in which facts are declared is not relevant since all constraints are analyzed together. The

language supports the declaration of predicates and functions which define operations over signatures. A predicate defines one or more constraints that when executed instruct the analyzer to produce an instance of the model satisfying the predicate. A function defines an expression whose evaluation yields a subset of atoms and, thus, can be used for improving modularity and readability of models. Differently from a predicate, it is also possible to declare assertions, i.e., constraints that are intended to be valid in the model. The execution of an assertion instructs the analyzer to produce a counterexample, i.e., an instance of the model that violates the property. Modules can be created for sharing generic declarations and constraints among models, thus facilitating reuse. Table 8 summarizes supported commands for the creation of conceptual models in Alloy.

Table 8 – Commands of operators and quantifiers supported by Alloy

Constants	Operators		Constraints	
	Set	Relational	Logical	Quantification
none *empty set* univ *universal set* iden *identity*	+ *union* & *intersection* - *difference* in *subset* = *equality*	-> *product* . *join* [] *join* ~ *transpose* ^ *transitive closure* * *reflexive-transitive closure* <: *domain restriction* :> *range restriction* ++ *override*	not ! *negation* and && *conjunction* or || *disjunction* implies => *implication* if <=> *bi-implication*	all *universal* no *empty* lone *empty or singleton* one *singleton* some *not empty*

The Alloy Analyzer is an open-source tool supporting Alloy (JACKSON, 2012). This tool can automatically translate an Alloy formula (relational) into a Boolean formula, thus enabling to invoke an off-the-shelf SAT-solver, such as the SAT4J[1] or KodKod (TORLAK; DENNIS, 2006), for performing the analysis. However, the tool has two limitations: it lacks an API for the creation, manipulation, and analysis of Alloy formulas, and it lacks support for partial instances, i.e., a part of the problem's solution that is known a *priori*. An API is needed to enable the use of a constraint solver as a back-end analyzer for modeling tools that accept the Alloy language. Besides addressing these two limitations, KodKod can replace the Alloy Analyzer because it accepts a subset of the Alloy language and, hence, can support the creation, manipulation, and analysis of Alloy models. KodKod is also more efficient than the Alloy Analyzer since it employs a mechanism for sharing sub-formulas and sub-expressions which contributes for the performance of SAT solvers. To execute an Alloy model, a fixed scope (i.e., bound) must be assigned to the analysis. While the Alloy Analyzer requires an integer to be assigned to each

[1] SAT4J, <http://www.sat4j.org/>.

signature in the model, which bounds the maximum number of atoms that can exist for each set, KodKod requires a formula to be bounded by a relational constant, i.e., a fixed set of tuples drawn from the universe of atoms. In particular, KodKod assigns an upper bound (i.e., tuples that a variable *may* contain) as well as a lower bound (i.e., tuples that a variable *must* contain), thus enabling to describe partial solutions, which enhance the efficiency of the solver by further constraining the solution space.

To illustrate the expressiveness of Alloy for describing conceptual models, Figure 8a shows an example of a complete Alloy model about the view and viewpoint concept as defined by the ISO/IEC/IEEE 42010 (2011) standard. The standard specifies that each viewpoint governs exactly one view and frames some stakeholders' concerns. In this example, *governs* is a binary relation (its atoms can be tuples of type $Viewpoint \rightarrow View$) and *frames* is a ternary relation (its atoms can be tuples of type $Viewpoint \rightarrow Stakeholder \rightarrow Concern$). Quantifiers can be used for qualifying relations, such as the keyword `some` that in the case of the *frames* relation of the `Stakeholder`[2] signature constrains each stakeholder to be related to at least one concern. Then, the fact body adds constraints without which signatures and relations would be allowed to potentially contain all possible tuples. For instance, the constraint in lines 15-17 defines that the set of concerns framed by a particular viewpoint is always equal to the set of concerns framed by the view that it governs. The empty body of the predicate `show` instructs the Alloy Analyzer to return any instances of this model that can be found within the scope of three atoms per signature, with exception of `Viewpoint` that can have at most two atoms. The execution of this specification by the constraint solver confirms that this is a syntactically correct and sound model since it was able to find at least one solution for the problem within the informed scope. In particular, this solution can be alternatively presented to the user as text or a graphic model. Figure 8b shows one of the model's solutions in which there are two viewpoints, each of them governing their own view and framing one of the two concerns that a given stakeholder has on the described system.

4.2 Conceptual Model of Systems-of-Systems Architecture

To capture the main concepts framed by SoS architecture descriptions, the TASoS model takes inspiration on the SosADL abstract syntax. The reasons for the selection of this language are threefold: (i) it has explicit constructs for abstract types of constituents, mediators, and coalitions, including a mathematical foundation that enables to express dynamic connectivity in terms of constraints; (ii) it supports the description of SoS abstract architectures from a dynamic as well as static viewpoint; and (iii) it is supported by a tool that can be extended to work with different modeling and implementation languages, such as UML, Java, and C, as well as used in

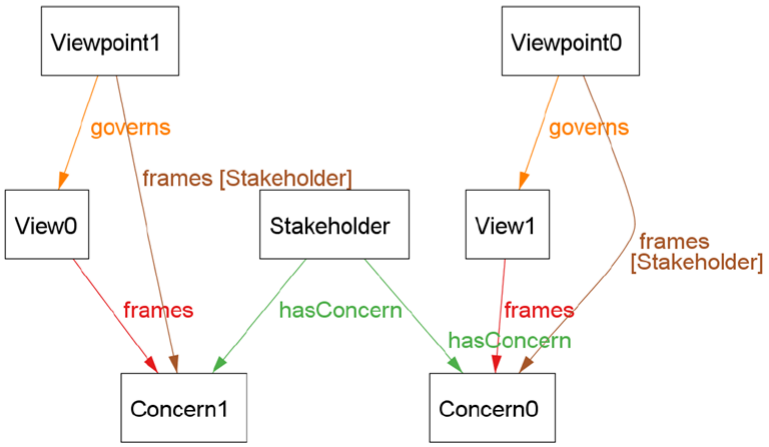
[2] Identifiers and commands are case sensitive in Alloy.

Figure 8 – Example of a complete Alloy specification on the relation among views, viewpoints, concerns, and stakeholders

```
1: module basic
2: sig Concern {}
3: sig View {
4:   frames:some Concern
5: }
6: sig Stakeholder {
7:   hasConcern:some Concern
8: }
9: sig Viewpoint {
10:   governs:one View,
11:   frames:Stakeholder->some Concern
12: }
13: fact {
14:  all v:View|one p:Viewpoint{v in p.governs}
15:  all v:View,p:Viewpoint|
16:    v in p.governs implies
17:      v.frames = ~(p.frames).Stakeholder
18:  all s:Stakeholder,p:Viewpoint|
19:    let cons=p.frames|
20:      s in cons.Concern implies
21:        s.cons in s.hasConcern
22: }
23: pred show{}
24: run show for 3 but 2 Viewpoint
```

(a) Alloy specification

```
1 ---INSTANCE---
2 integers={}
3 univ={Concern$0, Concern$1, Stakeholder$0, View$0, View$1,
    Viewpoint$0, Viewpoint$1}
4 Int={}
5 seq/Int={}
6 String={}
7 none={}
8 this/Concern={Concern$0, Concern$1}
9 this/View={View$0, View$1}
10 this/View<:frames={View$0->Concern$1, View$1->Concern$0}
11 this/Stakeholder={Stakeholder$0}
12 this/Stakeholder<:hasConcern={Stakeholder$0->Concern$0,
    Stakeholder$0->Concern$1}
13 this/Viewpoint={Viewpoint$0, Viewpoint$1}
14 this/Viewpoint<:governs={Viewpoint$0->View$1, Viewpoint$1->
    View$0}
15 this/Viewpoint<:frames={Viewpoint$0->Stakeholder$0->Concern$0
    Viewpoint$1->Stakeholder$0->Concern$1}
```



(b) Alternative representations of a solution

conjunction with version control tools. Table 9 summarizes how SosADL features support the distinguishing characteristics of a SoS as defined by Boardman and Sauser (2006).

Table 9 – SosADL support for SoS characteristics

Characteristic	SosADL features
Autonomy to operate without the control of another entity	Architects can describe at design-time abstract types of constituents that will only be realized later on at run-time. Each constituent is described in terms of external gates (interfaces) that are exposed to the SoS environment and an internal behavior to fulfill its own mission. Moreover, the participation of the constituent in the coalition is not controlled by the SoS and it only occurs if there exists a duty that satisfies the assumptions of its gates.
Belonging to voluntarily contribute for the SoS mission	Architects can express assumptions for the participation of constituents in the coalition, which describe conditions for their permanence in the SoS that must be fulfilled by the environment. Moreover, the language supports on-the-fly evolution of the architecture by enabling to identify and incorporate different concrete constituents.
Connectivity to dynamically establish connections among constituents	Architects can describe obligations of duties that must be fulfilled by gates and assumptions of gates that must be satisfied by duties, thus supporting dynamic creation of commitments between constituents and mediators that participate in the coalition. Moreover, architects can describe physical properties about constituents and mediators, thus enabling to take location as a condition for constituents interaction.
Diversity to create new capabilities from constituents inherited limited functionality	Architects can describe several abstract types of constituents and mediators, which can then be arranged in different coalitions for taking advantage of SoS diversity.
Emergence of behaviors by exercising the connectivity, autonomy, and belonging of constituents	Connections can be dynamically created from abstract commitments between gates of constituents and duties of mediators if the obligations defined by the duty are fulfilled by the gate and the assumptions of the gate are satisfied by the duty. Thus, coalitions can be composed in different ways or with different systems, thereby providing the necessary conditions for new behaviors to emerge.

Source: Adapted from Oquendo (2016c).

The conceptual model for SoS architectures, which is named TASoS, defines 20 signatures and 14 relations listed in Table 10 and covers the architecture elements supported by SosADL, such as systems, mediators, and coalitions. Signatures may inherit the relations of higher level signatures by extending its prior definition and, optionally, adding more relations. For the sake of simplicity, inherited relations are omitted in this chart. Atoms of an extension are treated as atoms of the higher level signature in that they must also satisfy the constraints that apply to the higher level signature. Atoms of a higher level signature with only one extension must necessarily pertain to that extension. On the other hand, if there are more extensions, atoms can pertain to only one of them. For instance, `System`, `Mediator`, and `SoS` are all extensions to the higher level signature `ArchitecturalElement` and, therefore, atoms of this higher level

signature necessarily pertain to either `System`, `Mediator`, or `SoS`. In the case where the conceptual model should allow atoms of a higher level signature to pertain to more than one of its extensions, the set operator `in` is used instead of `extends` in the signature definition instruction. Following, the main signatures of TASoS are detailed. The source code of the conceptual model is presented in its entirety in Appendix A.

Table 10 – Signatures and relations defined by TASoS

Signature	Relation
ArchitecturalElement	hasPort
System (ArchitecturalElement*)	
Mediator (ArchitecturalElement*)	
Sos (ArchitecturalElement*)	arch
Port	hasConnection
Gate (Port*)	
Duty (Port*)	
Connection	hasDatatype, owner
Outward (Connection*)	
Inward (Connection*)	
Unification	src, dest
Relay	relayCon
Datatype	
IntegerType (Datatype*)	
NamedType (Datatype*)	
RangeType (Datatype*)	
SequenceType (Datatype*)	
TupleType (Datatype*)	complexType
Architecture	contain, unifiedAs, bindings
Topology	inTopology, path

Key: The signature *extends* the one marked with *. This implies that the current signature inherits the relations of the extended one and that atoms of sister signatures do not overlap with each other.

`ArchitecturalElement` is the signature for entity blocks in SosADL. As previously mentioned, this signature is further extended by the ones of `System`, `Mediator`, and `SoS`, which are separated in TASoS because they play a different role in the coalition and, thus, are subjected to different constraints. Source code 5 presents an excerpt of constraints referring to `ArchitecturalElement` or one of its extensions. Atoms of the higher level signature have a binary relation named *hasPort* of type $ArchitecturalElement \rightarrow Port$ that specifies the ownership of ports[3], which in turn may contain one or more connections (constraint 14). Additional constraints are defined to refine this relation for each extension, i.e., mediators can own duties (constraint 11) and systems and SoS can own gates (constraints 12 and 13, respectively). In particular, the `SoS` signature extends the one of `ArchitecturalElement` with a binary relation *arch* that has atoms of type $Architecture \rightarrow Relay$ for internally relaying a set of connections to the elements of a particular coalition, which is supported by SosADL. The evolving nature

[3] The `Port` signature is extended by the ones of `Gate` and `Duty`.

of systems, mediators, and SoS is also supported by allowing a set of ports owned by a given element to change.

Source code 5 – Excerpt of constraints on architectural elements in TASoS

```
1: //#11 All Ports of mediators are duties
2: Mediator.hasPort in Duty
3: //#12 All Ports of constituents are gates
4: System.hasPort in Gate
5: //#13 All Ports of sos are gates
6: Sos.hasPort in Gate
7: //#14 All Ports owned by a given architectural element must have at
       least one connection
8: all p: Port | some p.~hasPort implies
9:   some p.hasConnection
```

An interaction represents a physical or logical channel that enables to exchange information between architectural elements. In SosADL, interactions take place at the connection level of interfacing architectural elements. In TASoS, two independent signatures can be used for modeling interactions, namely `Unification` and `Relay`. Source code 6 presents an excerpt of constraints that apply to interactions. The first signature is used for modeling interactions between systems and mediators within coalitions and has two binary relations for mapping its domain (*src*, which is of type $Unification \rightarrow Connection$) and image (*dest*, which is of type $Unification \rightarrow Connection$). To convey the direction of a unification, atoms of the domain must be of type $Unification \rightarrow Outward$ (constraint 17.1) and atoms of image must be of type $Unification \rightarrow Inward$ (constraint 17.2). Moreover, a unification can only exist between connections of systems and mediators (constraint 17.3-5). To prevent two or more unifications over the same pair of connections, which should not be allowed in concrete architectures, there cannot be two unifications to the same pair of domain and image connection (constraint 18). In turn, the latter signature models the interaction that exists between a connection owned by the SoS and a connection of a system or mediator. In particular, the ternary relation named *relayCon* of type $Relay \rightarrow Connection \rightarrow Connection$ can only map connections owned by a SoS to connections of elements that are presently in the coalition (constraint 8). Moreover, connections of a SoS can be relayed only once (constraint 9) to connections of the same type (constraint 10) in any given coalition.

Source code 6 – Excerpt of constraints on interactions in TASoS

```
1: //#8 Relay interactions require some element in the coalition
2: all r: Relay, c1,c2: Connection |
3:   c1->c2 in r.relayCon implies {
4:     all a: Architecture, s: Sos | a->r in s.arch implies
5:       c2.owner in a.contain and c1.owner=s
```

```
6: }
7: //#9 All connections of an Sos can be relayed to at most connection
       per Relay sig
8: all disj c,c': Connection, r: Relay |
9:   c->c' in r.relayCon implies
10:     {no c'':Connection | c'!=c'' and c'->c'' in r.relayCon}
11: //#10 A connection can be relayed to another of the same type
12: all disj c1,c2: Connection, r: Relay |
13:   c1->c2 in r.relayCon implies
14:     (c1+c2) in Inward or (c1+c2) in Outward
15: all u: Unification {
16:   //#17.1 The domain of an unification is an outward connection
17:   u.src in Outward
18:   //#17.2 The image of an unification is an inward connection
19:   u.dest in Inward
20:   //#17.3-5 Unifications are allowed between elements of System and
        Mediator
21:   (u.src.owner+u.dest.owner) in (System+Mediator)
22:   u.src.owner in System implies u.dest.owner in Mediator
23:   u.src.owner in Mediator implies u.dest.owner in System
24: }
25: //#18 Two unifications are equal if they relate the same pair of
       connections
26: all x, y: Unification |
27:   (x.src = y.src) and (x.dest = y.dest) implies x=y
```

Another core signature of TASoS is `Architecture`, which combines elements of abstract and concrete architectures for expressing possible run-time coalitions for a SoS. This concept is defined in terms of three relations, namely: (i) *contains*, a binary relation of type $Architecture \rightarrow (System + Mediator)$ relating a particular architecture to the set of systems and mediators that are presently engaged in a coalition; (ii) *bindings*, a binary relation of type $Architecture \rightarrow Unification$ listing intentional bindings between systems and mediators that have been defined in the abstract architecture; and (iii) *unifiedAs*, a binary relation of type $Architecture \rightarrow Topology$ describing different ways (i.e., candidate configurations) in which systems and mediators within the coalition can be arranged given the available types of bindings. The `Topology` signature, which is introduced in TASoS for representing the explicit configuration of concrete architectures and it is further explained in the next section. Source code 7 shows an excerpt of constraints in which `Architecture` plays an important role. For instance, any two architectures are considered to be equal if they share the same set of topologies (constraint 28). Systems that participate in any given architecture must engage in at least one unification of candidate concrete architectures (constraint 31). All unifications of a system in the coalition must appear in candidate concrete architectures (constraint 32). Moreover, connections of any given gate only can be unified to connections of a single duty (constraint 34).

Source code 7 – Excerpt of constraints on architectures in TASoS

```
1: //#28 Two architectures should be equal if they have the same set of
     topologies
2: all a,a': Architecture|
3:   (a.unifiedAs).elems = (a'.unifiedAs).elems implies a=a'
4: //#31 All systems that participate in an architecture must engage in
     at least one unification in all candidate topologies
5: all e: System, a: Architecture, t: Topology | e in a.contain
6:   and t in (a.unifiedAs).elems implies
7:     some participatesInTopology[e,t]
8: //#32 All unifications in the set of bindings must be used in the
     topology.
9: all u: Unification, a: Architecture, e: System |
10:   u in a.bindings and e in a.contain and e in isUnifiedTo[u] implies
11:    all t: Topology | t in (a.unifiedAs).elems implies
12:      u in path[t].Unification
13: //#34 Connections of the same gate can only be unified to connections
     of the same duty in a given Architecture. As a consequence, two
     unifications may share the exact same ports or none at all.
14: all g: Gate, a: Architecture, t: Topology |
15:   let Suni = unificationsOfPort[g,t] |
16:     g.~hasPort in (a.contain) and t in (a.unifiedAs).elems
17:       implies one dutiesOfUnification[Suni]
```

4.2.1 The Topology Concept

The architectural configuration is abstractly expressed in SosADL in terms of policies. To automatically resolve concrete configurations adhering to such an abstract description, TASoS introduces the `Topology` signature for representing explicit configurations that comply with the policies described in the abstract architecture. Atoms of this signature have two relations, namely: (i) *inTopology*, a binary relation of type $Topology \rightarrow Unification$ which contains the set of intentional bindings that can be realized between abstract types of systems and mediators; and (ii) *path*, a ternary relation of type $Topology \rightarrow Unification \rightarrow Unification$ which connects atoms of `Unification` in order to compose a network of systems and mediators. Thereby, this signature merges the description of an abstract architecture with the one of a concrete architecture in that the first relation lists abstract types of bindings that can take place among systems and mediators at run-time (which is taken from the abstract description) and the latter lists connected networks that can be created on top of these bindings (which results in the concrete description). Source code 8 shows an excerpt of constraints that apply to `Topology`. For instance, only unifications listed by the *inTopology* relation can be used by *path* (constraint 25). Moreover, the path of all possible topologies for architectures without any systems must be necessarily empty (constraint 29). A mediator that participates in a topology engages in at least

one unification in the topology (constraint 30). Finally, the connectivity of the path is guaranteed by constraining related unifications to share at least one system or mediator (constraint 33).

Source code 8 – Excerpt of constraints on topologies in TASoS

```
1: //#25 A topology may only use the unifications that exist in the set
      of possible bindings and connect elements that participate in the
       architecture
2: all t: Topology, a: Architecture |
3:   t in (a.unifiedAs).elems implies t.inTopology in a.bindings and
4:     (isUnifiedTo[t.inTopology]) = (a.contain)
5: //#29 The relations inTopology and path must be empty if no system
      participates in the coalition
6: all t: Topology, a: Architecture | t in (a.unifiedAs).elems implies
     {{no t.path} iff {no a.contain&System}}
7: //#30 For all topologies of an architecture, a mediator engages in at
       least one unification
8: all t: Topology, m: Mediator |
9:   let Suni = participatesInTopology[m,t] | some Suni implies
10:     m in owner[Suni.src] or m in owner[Suni.dest]
11: //#33 The path of a topology connects unifications that originate or
      end in the same architectural element
12: all t: Topology, a: Architecture | {some a.contain} and
13:   t in (a.unifiedAs).elems implies {
14:     all u,v: Unification | u->v in t.path implies
15:       some isUnifiedTo[u]&isUnifiedTo[v]
16: }
```

4.2.2 Verification of the Architecture

Differently from facts, which denote constraints that always hold, assertions define constraints whose satisfiability must be evaluated. In fact, the execution of an assertion instructs the constraint solver to find a counterexample that violates the property. If no counterexample is found within the predefined analysis scope, the user can infer that the assertion holds and repeat the analysis for larger scope. Thereby, if subsequent analyses still cannot produce a counterexample, the architect gains confidence on the correctness of the model. The main purpose of assertions in TASoS is to verify the soundness of the model by stating constraints that are expected to hold or fail in the context of SoS abstract and concrete architectures. Table 11 describes all assertions included in the model, which cover different aspects of the architecture of SoS. These assertions can be used in the analysis of abstract architectures, e.g., in the verification of completeness of the set of constraints that describe a particular SoS. In particular, assertions can be used to find families of configurations that violate a particular property so that new policies might be added to the abstract architecture for preventing them to occur. As a result, the

correctness and completeness of the abstract architecture of a SoS is enhanced.

Table 11 – Summary of assertions included in TASoS

Assertion	Description
A1 architectureCantBeEmpty	Checks the feasibility of architectures without elements. This assertion is expected to fail.
A2 pathCantBeEmpty	Checks if a not empty architecture can have no path in topology. This assertion is expected to hold since elements of a coalition must necessarily participate in at least one unification.
A3 samePortUnification	Checks the feasibility of a topology for an architecture in which there are more than one unification over two ports. This assertion is expected to fail. Otherwise, it might indicate that the model is overconstrained.
A4 allBindingsAreUsed	Checks the feasibility of an architecture for which some of its bindings do not appear in at least one of its topologies. This assertion is expected to fail.
A5 oneMediator	Checks the correctness of a description by investigating if it allows any topology to have more than one mediator. This assertion is expected to fail. Otherwise, it might indicate that the model is overconstrained.
A6 oneMediatorPerSystem	Checks the correctness of a description by investigating if it allows a system to interact with more than one mediator. The outcome of this assertion varies depending on particular binding policies of each SoS. If it is expected to fail and does not, it might indicate that the model is underconstrained.
A7 cantModifyConstituents	Checks the changeability of systems in an architecture. This assertion is expected to fail.
A8 cantModifyBindings	Checks the changeability of set of bindings of an architecture. This assertion is expected to fail.

Source code 9 shows an excerpt of these assertions. The first assertion, named `architecture-CantBeEmpty`, evaluates the feasibility of architectures without elements. When executed, the command `check` in line 5 instructs the constraint solver to find a counterexample that shows an architecture without elements. This assertion is expected to fail since the *contain* relation of `Architecture` is only defined as a set of systems and mediators. To prevent scenarios in which coalitions have no elements, one could change the definition of the *contain* relation to `some` systems and mediators. But, since the feasibility of empty coalitions is needed for validating the outcome of operations (e.g., loss of systems) on the architecture, no constraint was added or changed in the model. The second assertion, named `pathCantBeEmpty`, evaluates the feasibility of architectures that are not empty but have candidate configurations that are. This assertion is expected to hold due to constraint 31 (shown in Source code 7), which states that all elements of an architecture must participate in at least one unifications in its *path* relation. Finally, the third assertion, named `samePortUnification`, evaluates the feasibility of having more than one unification between two ports. Even though there is a constraint that prevents a connection

to participate in more than unification in the same path, this assertion is expected to fail since there are not any constraints preventing this scenario. Thus, if this assertion holds, the model of the architecture might be overconstrained. It is also worth mentioning that the execution of this assertion under the small analysis scope of 5 atoms per signature can produce an instance of the architecture that violates the property and is also easier to understand than a large instance of the model would be.

Source code 9 – Excerpt of assertions in TASoS

```
1: //A#1 Checks whether an architecture can have no elements.
2: assert architectureCantBeEmpty {
3:   no a: Architecture | no a.contain
4: }
5: check architectureCantBeEmpty for 4 but 2 Architecture, 3 Topology -- If this assertion fails, an architecture is allowed to have no elements
6: // A#2 Checks if a topology can have no unifications.
7: assert pathCantBeEmpty {
8:   no a:Architecture, t: Topology | t in (a.unifiedAs).elems and some a.contain and {no path[t].Unification}
9: }
10: check pathCantBeEmpty for 4 but 2 Architecture, 3 Topology -- If this assertion fails, the model might need corrections
11: // A#3 Checks if it is possible to have more than one unification over the exact same pair of ports
12: assert samePortUnification {
13:   no a: Architecture, t: Topology, disj u,v: Unification |
14:     t in a.unifiedAs.elems and
15:     u in inTopology[t] and
16:     v in inTopology[t] and
17:     #(portsOfUnification[u] & portsOfUnification[v])>1
18: }
19: check samePortUnification for 5 -- If this assertion holds, the model might be overconstrained
```

4.3 Predicates for the Customization of TASoS

As mentioned in the beginning of this chapter, the TASoS model encompasses a core module, which has been presented in the previous section, and an instance model, which has to be created for each particular SoS under analysis. To create such an instance model, one needs to extend the abstract signatures in TASoS with elements derived from the SoS abstract architecture. To facilitate the creation of such an instance, the core module defines the predicates listed in Table 12, which encompass one or more constraints that apply to the atoms passed as parameters.

In particular, if the predicate is called within a fact statement, it is also treated as constraints that always hold. Thereby, these predicates can further refine the problem by adding new constraints to the instance module. Source code 10 shows an excerpt of predicates for the customization of TASoS. For instance, the first predicate, named `attributeConToPort`, customizes which atoms of `Connection` pertain to the *hasConnection* relation of `Port`. The second predicate, named `attributeArchToSos`, attributes a given atom of `Architecture` to a particular atom of `SoS`. Finally, the third predicate, named `unify`, attributes a pair of atoms of `Connection` to the relations *src* and *dest* of an atom of `Unification`. Thereby, this predicate can customize the atoms of `Unification` based on intentional bindings defined by the abstract architecture.

Source code 10 – Excerpt of predicates in TASoS for creating instances of the core module

```
1: pred attributeConToPort [sCon: set Connection, p: Port]{
2:   p.hasConnection in sCon
3: }
4: pred attributeArchToSos[a: Architecture, s: Sos]{
5:   getArch[s] in a
6: }
7: pred unify [cFirst, cSecond: Connection] {
8:   cFirst in Outward and cSecond in Inward implies
9:     {some u: Unification | u.src in cFirst and u.dest in cSecond}
     else
10:  cFirst in Inward and cSecond in Outward implies
11:    {some u: Unification | u.src in cSecond and u.dest in cFirst}
12: }
```

Table 12 – Predicates for customization of TASoS

Predicate	Parameter
attributeConToPort	set, element
attributePortToAE	set, element
attributeDatatypeToCon	element, element
attributeRelay	element, element
attributeArchToSoS	element, element
attributeNary	set, element
unify	element, element

Once an instance module has been created, constraint solvers can be employed for automatically synthesizing configurations that adhere to the description of the SoS abstract architecture. To do so, the architect must instruct the solver to search for concrete architectures within a predefined analysis scope. Source code 11 shows an excerpt of the core module that instructs the solver to search for one or more architectures that comply with the constraints described in the predicate named `case`. This predicate describes a scenario where there must be two distinct mediators participating in such a concrete architecture. Based on the core set of

constraints defined in TASoS, this predicate describes a feasible setting and, hence, should return at least one concrete architecture adhering to this description. Since the investigation is limited to a small analysis space, this scope must be carefully selected so it is sufficiently large that it finds an instance of the model (in other words, a solution of the problem) and yet small that it returns such an instance within an acceptable time frame. The abstract architecture is considered to be feasible if at least one concrete architecture is found to satisfy the complete set of constraints, i.e., the core and instance modules of TASoS. But, in the case where the evaluated scenario cannot return a solution for the abstract architecture, the architect can decide between broaden the scope of the analysis or check the correctness of the instance module by using assertions that have been previously described or new ones that can be created for checking a particular SoS instance.

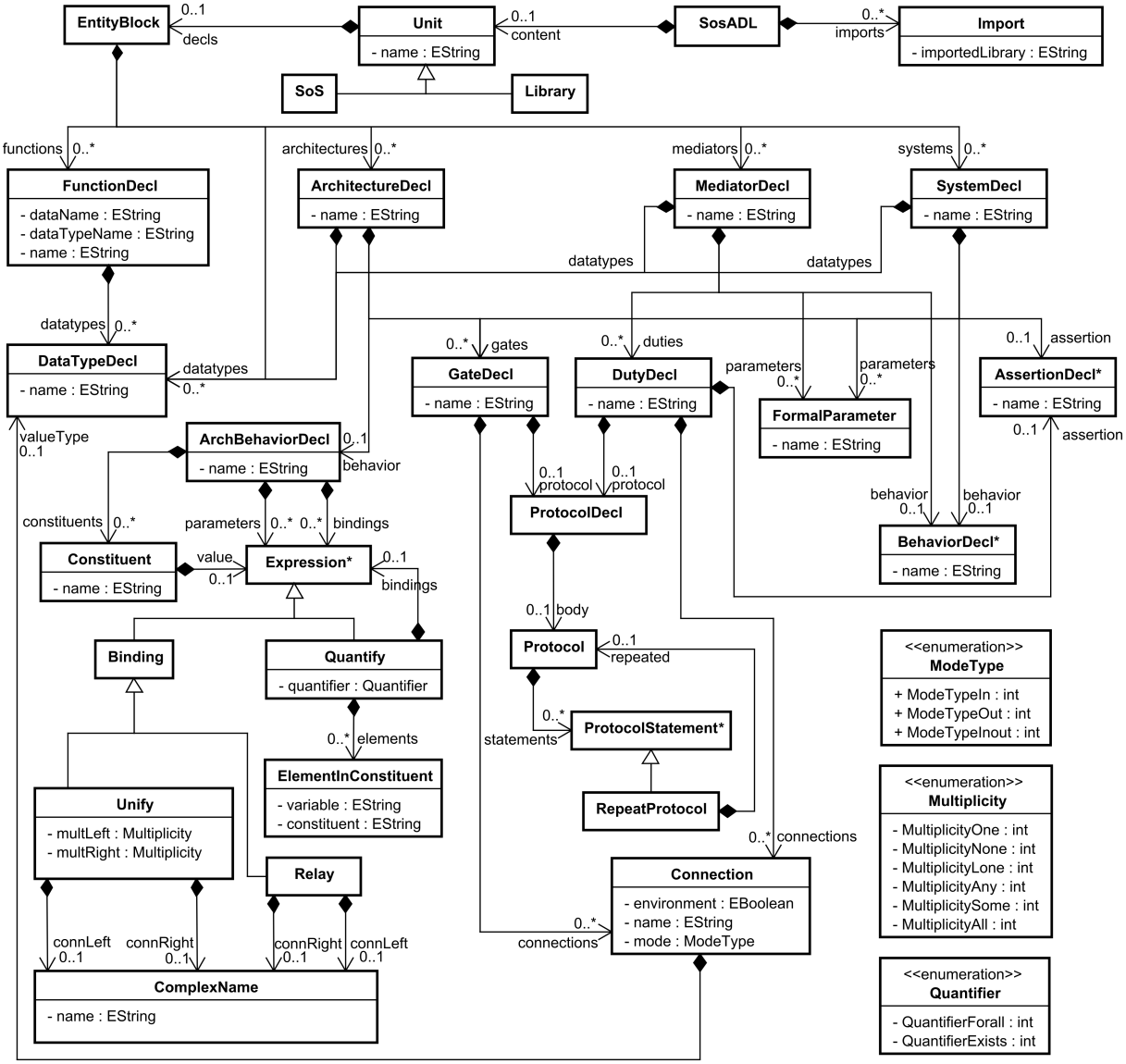
Source code 11 – Excerpt of a predicate checking the feasibility of a scenario in TASoS

```
1: pred case {
2:     some disj m,m': Mediator, a: Architecture |
3:         some participatesIn[m,a] and
4:         some participatesIn[m',a]
5: }
6: run Overview {case} for 6 but 2 Architecture, 2 Topology, 10 Port, 4
      System, 10 Connection, 10 Unification
```

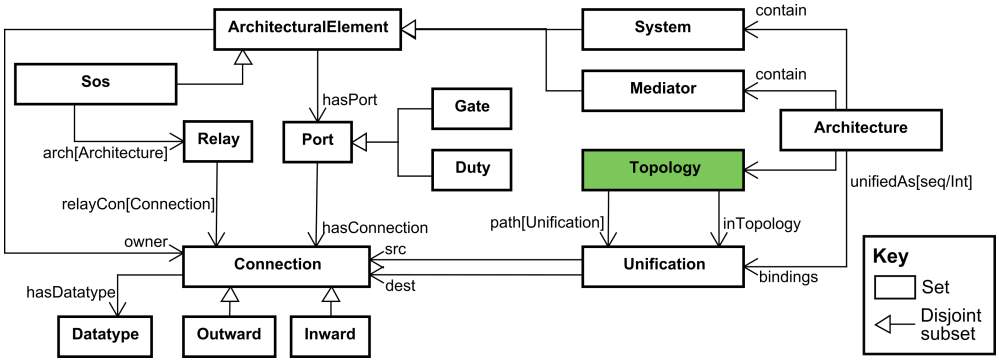
4.4 Mapping between SosADL and TASoS

As previously discussed, the conceptual model for SoS architectures implemented in Alloy takes inspiration on the abstract syntax of SosADL. Figure 9 on page 72 shows an excerpt of the metamodels for SosADL and TASoS. Despite many similarities between the two metamodels, some of the concepts framed by TASoS play a different role from their namesakes in SosADL. Moreover, not all elements that appear in the SosADL metamodel are explicitly framed by TASoS. For instance, SosADL has a `SosADL` element as the overarching construct of the architecture description. While there is no explicit concept for this element in TASoS, an instance module may be regarded as such since it encompasses the set of constraints that make up for the corresponding architecture. Also regarding the `SosADL` element, it can import several `Library` elements and it can be composed of at most one `Unit` element, which is either a `SoS` or `Library` element. In turn, TASoS supports referencing external libraries as modules, which are not signatures but a mechanism provided by the Alloy language for referencing external files, and `SoS` elements can be modeled as extensions to the `SoS` signature. The `Unit` element is defined in SosADL by at most one `Entity Block`, which can be composed of an assorted group of `System`, `Mediator`, `Architecture`, `Function`, or `Datatype` elements. Apart from `Function`, all other elements can be modeled as extensions to existing signatures in TASoS. In particular, the `Architecture` element is framed by two separate signatures, namely: (i) `SoS`,

Figure 9 – Elements of architecture descriptions for SoS. While (b) is based on (a), (b) introduces a new element (colored green) to explicitly describe the configuration of concrete architectures.



(a) Excerpt of elements in SosADL abstract syntax. Asterisked elements are partially shown.



(b) Excerpt of signatures in TASoS

which models the structure of this element, e.g., its interfaces; and (ii) `Architecture`, which models its behavior and configuration. Table 13 presents a detailed comparison between the elements of SosADL and their corresponding signatures in TASoS.

Table 13 – Mapping between SosADL and TASoS. Extended elements are depicted within parenthesis.

SosADL Element	TASoS Signature	Description
SosADL	-	A SosADL element imports any set of Library and comprises at most one Unit, which is either a Library or a SoS element. TASoS does not have a signature for this element but an *instance module* can be regarded as one of such since it also describes a SoS or a Library element.
Library (Unit)	-	As a Unit element, a Library has at most one Entity Block element, which defines a collection of assorted System, Mediator, Architecture, Function, and Datatype elements. TASoS does not have a signature for this element but references to external libraries are possible as a *module*, which is an importing mechanism provided by the Alloy language.
SoS (Unit)	SoS (Architectural Element)	As a Unit element, a SoS has at most one Entity Block element, which defines a collection of assorted System, Mediator, Architecture, Function, and Datatype elements. Moreover, a SoS element can also define its own set of Gate and Connection elements. This element is framed in TASoS by the SoS signature, which extends the Architecture Element signature with an additional ternary relation named *arch* of type $SoS \rightarrow Architecture \rightarrow Relay$ describing connections of the SoS that are internally relayed to participants of the coalition, which is modeled by the Architecture signature.
System (Entity Block)	System (Architectural Element)	A System extends the Entity Block element with its own set of formal parameters, datatypes, and gates. It also has an internal behavior and assertions, which must be fulfilled by Mediators. In TASoS, the System signature extends the one of Architectural Element and constrains the *hasPort* relation to only have atoms of type $System \rightarrow Gate$, thus describing which gates are owned by a particular system.
Mediator (Entity Block)	Mediator (Architectural Element)	A Mediator extends the Entity Block element with its own set of formal parameters, datatypes, and duties. Differently from systems, a Mediator has only an internal behavior. In TASoS, the Mediator signature extends the one of Architecture Element and constrains the *hasPort* relation to only have atoms of type $Mediator \rightarrow Duty$, thus describing which duties are owned by a particular mediator.
Architecture (Entity Block)	SoS (Architectural Element), Architecture	Similarly to the System element, an Architecture extends the Entity Block with its own set of formal parameters, datatypes, and gates. In addition, this element contains an architecture behavior and assertions. In TASoS, the SoS signature can also have its own set of gates. In particular, the *arch* relation of this signature is used for associating a particular configuration (captured by the Architecture signature) to a particular set of connections that are internally relayed to some of its constituents.
Function (Entity Block)	-	A Function element defines an operation over a Datatype element described by the SosADL model. This element is not mapped to TASoS.

Continued on next page

Table 13 – *Continued from previous page*

SosADL	TASoS	Description
Datatype (Entity Block)	Datatype, Inward, Outward (Connection)	A Datatype element is extended by the one of IntegerType, NamedType, TupleType, SequenceType, RangeType, and ConnectionType. The difference between TupleType and SequenceType is that elements of the latter must necessarily be of the same Datatype. The first five elements are mapped in TASoS by namesake signatures. However, TASoS does not have constraints in which the aforementioned signatures play a determinant role, such as one forbidding unifications that do not match a particular datatype. On the other hand, the ConnectionType element must be specified for every Connection in the model as either in, out, or inout. In TASoS, this element is mapped by signatures Inward and Outward, which are extensions of the Connection signature and play a determinant role in several constraints of the model, such as the ones for customizing unifications.
Gate and Duty	Gate and Duty (Port)	A Gate and Duty are distinct elements representing an interface. Besides the declaration of connections, a duty also declares assertions that once satisfied will guarantee the operation of the mediator. In TASoS, these elements are represented by namesake signatures, which can customize the *hasConnection* relation for expressing connection ownership.
Expression	Unification, Relay	An Expression element describes constraints over the SoS description. Several elements extend Expression, such as Binding, Any, UnobservableValue, CallExpression, among others. The TASoS model is mainly concerned with the description of Binding, which is specialized by elements Relay and Unify, the latter being required for expressing intentional bindings. These elements are captured by namesake signatures, which can be customized for a particular SoS.
Protocol	-	A Protocol element describes a logical order for the analysis of expressions (such as choose, repeat, for, and if-then-else) or execution of actions (such as receive, send, assert, and done). This element is not mapped to TASoS.
Behavior	-	A System and Mediator elements can have an internal behavior that describes a logical order for the action of these elements, such as the sequence of interactions exchanged with the environment. This element is not mapped to TASoS.
Connection	Inward, Outward (Connection)	A Connection element describes an interaction point that a Gate or Duty element exposes to their environment. An environment connection is similar to a regular connection, hence they are both mapped to the Connection signature in TASoS. Moreover, Inward and Outward signatures extend the one of Connection, thus enabling to distinguish them based on their mode.
Architecture Behavior	Architecture, Topology	An Architecture Behavior element has three main parts: *constituents* that are allowed to participate at run-time, *parameters* that it can receive from the environment, and *bindings* that it can replicate for connecting these constituents. In TASoS, the first relation is mapped to the *contain* relation of the Architecture signature, the second relation is not mapped, and the third is mapped to the *bindings* relation, which associates atoms of Unification with a particular Architecture. Moreover, to describe different ways in which concrete architectures can be assembled, the *unifiedAs* relation is added to associate a given Architecture signature to one or more topologies.

This table shows that the conceptual model for SoS architectures defined by TASoS frames several of the elements defined in the abstract syntax of SosADL. In particular, this conceptual model has explicit signatures supporting the description of abstract and concrete architecture for SoS, such as coalitions, intentional bindings, and configurations. Still, some elements of SosADL are not mapped in the conceptual, such as the behavior of systems and mediators, functions, and protocols. Expressions, on the other hand, are partially supported. TASoS maps Unify and Relay expressions to namesake signatures, thus enabling the description of constraints related to interactions that take place within a coalition. Finally, the model can be refined with additional constraints for the customization of existing signatures, hence resulting in an instance module of TASoS that complements the core module introduced in this chapter.

4.5 Final Remarks

This chapter presented TASoS, a theory for the specification of SoS architectures based on formal constraints. To describe SoS, this model takes inspiration on the main architectural elements supported by SosADL, in particular coalitions, configurations, and intentional bindings. By using Alloy as the underlying model-finder tool built on SAT solvers, the feasibility and correctness of abstract architectures can be automatically analyzed. Therefore, if the solver does not find a concrete architecture complying with such an abstract description, architects can refine the set of policies governing the connectivity among constituents and run the solver to check the satisfiability of a new abstract coalition type. Moreover, architects can instruct the constraint solver to analyze the correctness of user-specified properties that are expected to hold in any architectural configuration of the SoS. To create such a model for representing a particular SoS, architects must extend the abstract signatures defined in TASoS with custom elements and add new constraints for the relation among these elements. In the following chapter, an automated mechanism is presented to support the creation of such an instance model for TASoS and automate the steps for the architectural synthesis of SoS based on constraint solving.

Chapter

5

Automated Synthesis of Systems-of-Systems Architectures

This chapter presents a software tool named **SoSy** that supports the synthesis of concrete architectures given an abstract description of a coalition type expressed in SosADL. This tool is provided as an extension to SosADE (Architectural Framework for Systems-of-Systems Design) (OQUENDO *et al.*, 2016), an integrated environment for the design, validation, and simulation of SoS based on SosADL. In particular, SoSy supports the translation of abstract architectures as constraints and, if feasible, the translation of the constraint solver as concrete architectures. These two translations are accomplished by two transformations: (i) a Model-to-Text (M2T) transformation from SosADL to TASoS, in which instance modules can be dynamically created for abstract architectures; and (ii) a Text-to-Model (T2M) transformation from the solution returned by the constraint solver to SosADL, in which concrete instances found by the tool are represented in SosADL. Thereby, SoSy is able to conceal the use of constraint solvers in a process for the architectural synthesis of SoS based in Ark.

The remainder of this chapter is organized as follows. First, Section 5.1 presents an automated model-driven process that supports the activities defined by Ark and how this process was implemented to complement an integrated environment for the design of SoS. Section 5.3 discusses the results of a functional evaluation of this tool. Finally, Section 5.4 concludes with final remarks on the use of an integrated environment for the design of SoS. Appendix A complements this chapter with additional guidance for installing and executing SoSy.

5.1 A Model-Driven Process Supporting Ark

To perform the tasks recommended by Ark, architects must create at least three models of the SoS architecture. First, architects manually create an abstract architecture for the SoS under analysis using SosADL. Then, to express the abstract architecture as a set of constraints,

architects translate the first artifact as an instance module of TASoS. If the new model corresponds to a correct instance of TASoS, architects can automatically analyze this artifact using constraint solvers. Finally, architects might need to translate the solution found by the constraint solver, which is provided as either text or graphic models, back into a format that is more familiar to them so that they can proceed to the analysis of the concrete architecture. In this scenario, the tasks required by Ark can be time consuming and prone to error. Aiming to mitigate these risks, this chapter presents SoSy, a tool that automates model transformations among SosADL, TASoS, and the constraint solver solution. In particular, this tool implements two model transformations (illustrated in Figure 10): (i) an Xtend[1] class named `SosADL2Alloy-Generator`, which implements a M2T transformation from SosADL to TASoS; and (ii) a Java method named `Solution2SosADLGenerator`, which implements a T2M transformation from solutions of the constraint solver to SosADL. This figure also shows that each transformation is defined at different abstraction levels, i.e., the former is defined at the metamodel level between SosADL and TASoS whereas the latter is defined at their instance level.

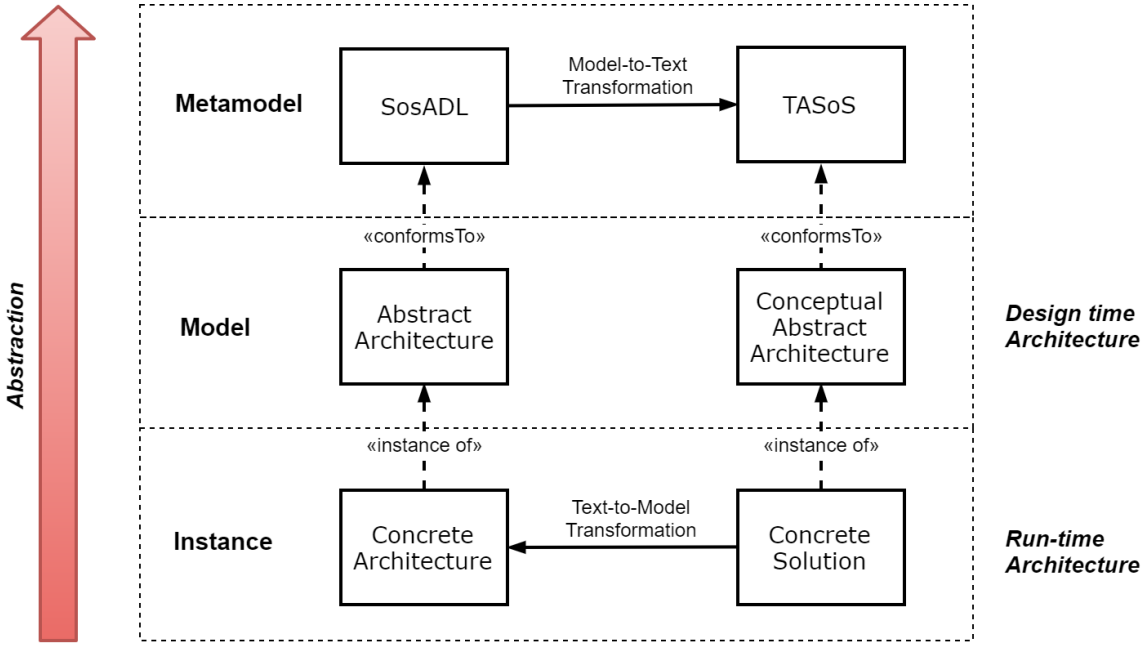


Figure 10 – Relationship between abstract and concrete architecture models in SosADL and TASoS

The tool implements a model-driven process illustrated in Figure 11 that takes inspiration on steps of the Ark method. This process is represented in SPEM(OMG, 2008), an OMG standard for the specification of software methods and processes. In particular, this figure describes the sequence in which *tasks* (i.e., work that has a particular purpose) and *activities* (i.e., group of related tasks) must be carried out by architects. The creation of an abstract architecture triggers an automatic transformation of this model as an instance of TASoS. In parallel, a Java[2] class that calls the constraint solver over the generated Alloy artifact is also generated. Afterwards, the

[1] Xtend, <http://www.eclipse.org/xtend/> (last accessed 07/22/2017)
[2] Java, <http://www.java.com> (last accessed 07/22/2017)

architect executes the main method of this class for analyzing the generated Alloy artifact. If the analyzed model is a valid architecture, the solver returns one or more instances of this model which can then be automatically translated into concrete architectures expressed in SosADL. Otherwise, architects can decide whether to increase the analysis scope (if the model is correct) or revise the abstract architecture (if the model is incorrect), triggering an update of related artifacts in the project. The process is concluded by an evaluation of candidate concrete architectures that is expected to terminate with a valid concrete architecture of the SoS.

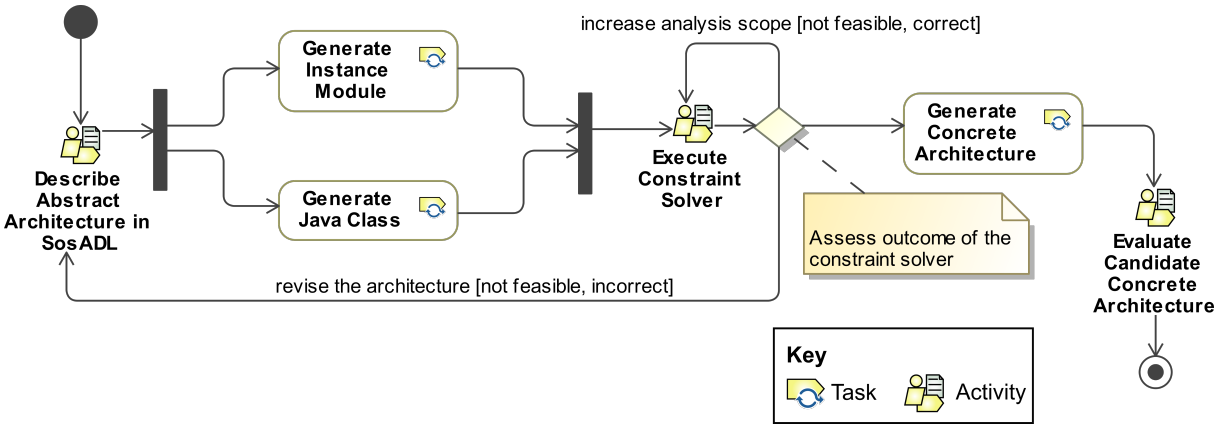


Figure 11 – Workflow of a model-driven process for the synthesis of concrete architectures

In particular, SoSy is developed as an extension to a comprehensive environment supporting the design of SoS named SosADE (OQUENDO *et al.*, 2016). In particular, this environment is itself an extension to Eclipse[3] that supports SosADL as modeling notation. This tool is developed in Xtext, a framework that supports the development of customized run-time and Integrated Development Environment (IDE) features for the use of DSL (Domain-Specific Languages) in the design of software systems. To achieve this purpose, the framework adopts the programming language Xtend, which is similar to Java and offers additional facilities for the implementation of customized tool support for DSLs. Moreover, as an Eclipse modeling environment, SosADE offer several facilities that support the use formal methods in practice, including (WOODCOCK *et al.*, 2009): multi-language support, cross-platform support, version control, and assistance for teams working on shared projects. Thereby, architects can leverage the facilities provided by SosADE in the creation and analysis of SosADL models that are required by the Ark method.

Figure 12 shows an activity detail diagram for the synthesis of concrete architectures as part of SosADE. The architect interacts with the tool in three instances: (i) to create an abstract description for the SoS, (ii) to execute the Java class that calls the constraint solver; and (iii) to evaluate candidate concrete architectures returned by the solver. Hence, intermediary tasks are automatically handled by SoSy, which implements transformations from architectural models to conceptual models and from the solution to this conceptual model back to architectural models. In particular, the analysis of the conceptual model is automatically handled by the constraint

[3] Eclipse IDE, <https://eclipse.org/mars/> (last accessed 07/22/2017)

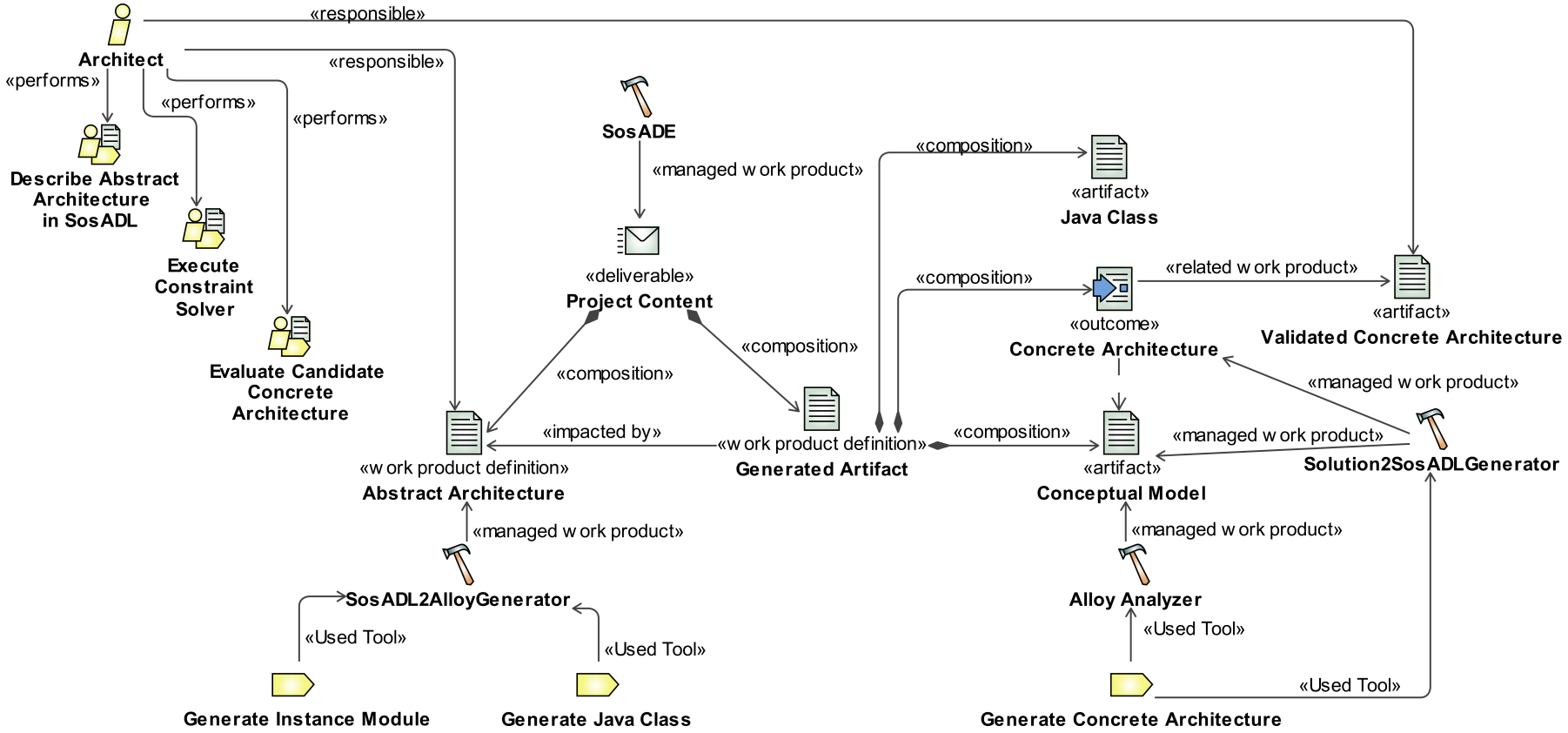


Figure 12 – Activity detail diagram for the synthesis of concrete architectures in SosADE

solver, which in this figure is represented by the Alloy Analyzer. As intermediary models are automatically managed by SoSy, in particular by two separate transformations, the architect is responsible for managing initial and final work products, namely abstract architecture and validated concrete architecture descriptions. Additional details about the installation and use of this tool are presented in Appendix A. Following, the implementation of the two transformations supported by the tool are further explained.

5.1.1 Transformation from SosADL to TASoS

The first transformation is implemented in SoSy as an Xtend class, which is named `SosADL2AlloyGenerator`. Table 14 lists the classes that compose this generator and explains their role in the creation of instance modules of TASoS and Java classes. The main class of this generator is `SosADL2AlloyGenerator`, which extends the SosADE class `SosADLPrettyPrinterGenerator` by changing the way in which elements of a SosADL model are printed in an instance of TASoS. Since this is the main class of this generator, it is discussed in more detail.

The complete list of methods in the `SosADL2AlloyGenerator` class is included in Appendix A. The main method of this class is *doGenerate*, which receives as input an element of type `Resource`, which is the link for a persistent document. Each resource is identified by a Uniform Resource Identifier (URI), which determines its location so that it can be reached by other methods. This method also receives an element of type `fsa`, which grants file system access for the creation and exclusion of files in the project, enabling to dynamically update its content as the abstract architecture changes. Two different types of files are created by the execution of this generator: (i) instance module of TASoS that describes a SosADL model in terms of a set of constraints; and (ii) Java class that calls the constraint solver on these instance modules. The remaining methods in this class are responsible for transforming each element of a SosADL model and translating them into elements of an instance of TASoS. To realize this transformation, these methods implement templates, a special type of expression delimited by triple single quotes (’’’) for string concatenation. The text within quotations is copied "as-is" to an instance module. Within a template, guillemets (« ») indicate a placeholder for values or expressions that can be dynamically evaluated and replaced by their *toString* representation. The method illustrated in Source code 12 compiles a SosADL object and creates the header of both a library and architecture instance modules of TASoS. Then, compilation of library and architecture are delegated to their corresponding methods.

Throughout the execution of this transformation, three global variables are accessed to customize generated instance modules. The variable *runargs* of type `AlloyBound` (shown in Source code 13) defines the quantity of elements for each signature in an architecture instance. This data is needed to specify a custom command that instructs the solver to search for a solution within a predefined scope. As previously mentioned, the analysis scope must be carefully selected so that it comprehends at least one feasible solution but it is still small enough so that the efficiency

Class	Description
SosADL2AlloyGenerator	Main class of the generator. It is responsible for calling the methods that transform SosADL models in the project folder into Alloy instance modules of TASoS and Java classes.
AlloyFunction	Defines the content of a function. In particular, defines the datatype name as a signature of TASoS. It stores the function declaration as it appears in the SosADL model so that it can be copied to concrete architectures.
AlloyListOfFunctions	Defines a list of functions declared in SosADL models.
AlloyType	Defines a superclass for datatypes that are declared in SosADL models. It is extended by the classes AlloyIntType, AlloyBoolType, AlloyRangeType, AlloyTupleType, AlloySequenceType, and AlloyNamedType which provide their own implementations for the methods toString and equals.
AlloyBound	Defines the content of a custom analysis scope for an instance module. In particular, it can scale up the number of allowed elements for signatures in an Alloy model based on the number of constituents instances in the coalition.
AlloyConnection	Defines the content of a connection. In particular, connections of mode inout are translated as two separate connections in an instance module.
AlloyPredicate	Defines how TASoS predicates are to be printed in instance modules.
AlloyPort	Defines the content of a port. In particular, it copies the content of guarantees for ports declared in the abstract architecture to related ports in the concrete architecture.
AlloySystem	Defines the content of a system, mediator, or architecture. In particular, it copies the content of the behavior of a system or mediator declared in the abstract architecture to related elements in the concrete architecture.
AlloyLibrary	Defines the content of a SoS or a Library unit.
ConcreteSolution	Defines the location and content of a template Java class that will call the constraint solver on a particular Alloy file in the project folder.

Table 14 – Classes in the `SosADL2AlloyGenerator` file

of the constraint solver is not impacted. In turn, the *sigs* variable is used to relate the signatures of constituents that participate in the analyzed scenario with their corresponding abstract signatures in the TASoS model. Finally, the *elems* variable is used to support the compilation of SosADL unifications by documenting extensions of the signature connection.

Source code 12 – Excerpt of method that compiles SosADL objects

```
1 override def compile(SosADL s)
2   '''
3   open tasos
4   open util/ordering[tasos/Architecture] as A0
```

```
5    «FOR i : s.imports»
6    «i.compile»
7    «ENDFOR»
8
9    «IF s.content instanceof Library»
10   «(s.content as Library).compile»
11   «ELSEIF s.content instanceof SoS»
12   «(s.content as SoS).compile»
13   «ENDIF»
14   '''
```

Source code 13 – Excerpt of the `AlloyBound` class

```
1 class AlloyBound {
2   private LinkedHashMap<String,Integer> args
3   private LinkedHashMap<String,List<Integer>> listAE
4   private Integer size //Default is 3 elements per signature
5   private Integer elem
6   private Integer arch
7   private Integer port
8   private Integer con
9   private Integer uni
10  private Integer relay
11 }
```

The customization of instance modules is also facilitated by the implementation of the predicates defined in Table 12. The class `AlloyPredicate` defines templates that can be dynamically evaluated during the transformation of a SosADL model. As some predicates can be attributed to a set of related signatures (as extensions to the same high level signature) to a particular element, these templates may vary depending on which sets of parameters are passed. Table 15 identifies all uses of predicates in methods of the generator. For instance, the method that compiles `ArchitectureDecl` creates an object of `AlloyPredicate` set with predicate `attributeArchToSoS`, which associates the current architecture with the SoS that is being described. Source code 14 shows an excerpt of the `AlloyPredicate` class that implements this predicate.

Source code 14 – Excerpt of the `AlloyPredicate` class

```
1 class AlloyPredicate {
2     var String predicate
3     var String relation
4     var int srcSize //1 if element, 0 if set
5     var int destSize //1 if element, 0 if set
```

```
6   def setPredicate(String pred){
7         this.predicate = pred
8         switch pred {
9           ...
10          case 'attributeArchToSos':{
11            this.relation = ""
12            srcSize = 1
13            destSize = 1
14          }
15        }
16  }
17  def toString(List<String> src, List<String> dest){
18        var String output = ""
19      ...
20      //If both src and dest have exactly one element
21      if (srcSize == 1 && destSize == 1){
22        srcParams = '''«src.get(0)»'''
23        destParams ='''«dest.get(0)»'''
24      }
25      if (src.size > 0 && dest.size > 0 ) {
26        output = '''
27        «predicate»[«srcParams», «destParams»]
28        '''
29      }...
30  }
31 }
```

Table 15 – Use of predicates for the customization of instances of TASoS

Predicate	Parameter	Used By (Method)
attributeConToPort	set, element	Gate, Duty
attributePortToAE	set, element	Gate, Duty
attributeDatatypeToCon	element, element	Gate, Duty
attributeRelay	element, element	Relay
attributeArchToSoS	element, element	ArchitectureDecl
attributeNary	set, element	DatatypeDecl
unify	element, element	Unify

5.1.2 Transformation from Solution to SosADL

Once the first transformation is successfully concluded, the architect can execute the generated Java class that will use a constraint solver to automatically analyze its corresponding architecture instance module. Each generated Java class is derived from a template defined by

the *ConcreteSolution* class that is referenced in the previous `SosADL2AlloyGenerator` class for the description of library and architecture models for a particular abstract architecture. In particular, this class encompasses custom methods derived from the abstract declaration of systems, mediators, and SoS that are responsible for printing the content of these elements in the format of a SosADL model. Thereby, the definition of this class as a template supports transferring data that are not processed by intermediary models between an abstract architecture and its concrete description, such as the declaration of behavior, protocol, and guarantee.

Provided that an analyzed abstract architecture is feasible, i.e., there are concrete architectures satisfying its set of constraints, the execution of this Java class will return a solution to the instance model which represents one or more concrete architectures for the coalition. Since this solution is originally presented to the user in a format other than SosADL, a second transformation is required so that architects can more easily understand the outcome of this analysis. This transformation is implemented by a method named `Solution2SosADLGenerator` that is provided as part of the generated Java class and defines the main structure of library and architecture models in SosADL based on elements provided by the solution. Because it is defined as a template, the content of this method can be customized for each architecture, thus showing custom names for ports and connections as well as updated behaviors, guarantees, and protocols. An excerpt of this method is shown in Source code 15. The result of this transformation is a set of new SosADL models which correspond to library and SoS concrete models.

Source code 15 – Excerpt of library template in `Solution2SosADLGenerator` method

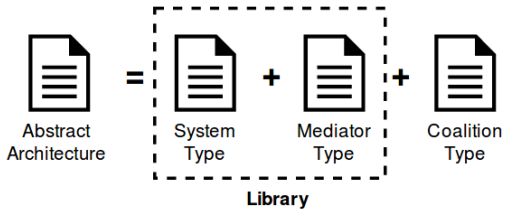
```
1  public static void Solution2SosADLGenerator (String answer, String
       path) throws Exception {
2    String library = "";
3    File solutionLibrary = new File(path.concat("_library.sosadl"));
4      ...
5    //Declaration of datatypes
6    LinkedHashMap<String,String> functions = new LinkedHashMap<
       String,String>();
7    «FOR elem: libs.entrySet»
8      «FOR t : elem.getValue.types.entrySet»
9      for (String dt : elems.get("Datatype")){
10       ...
11     }
12     «ENDFOR»
13   «ENDFOR»
14   «FOR elem: abstractions.entrySet»
15     «FOR t : elem.getValue.typesMap.entrySet»
16     for (String dt : elems.get("Datatype")){
17       String aux = dt.substring(dt.lastIndexOf("/")+1);
18       «IF !t.value.functions.empty»
```

```
19          ...
20          «ELSE»
21          library += level(1,"datatype "+aux+" is «t.value»");
22          «ENDIF»
23          ...
24        }
25      «ENDFOR»
26    «ENDFOR»
27    //Concrete declaration of systems
28    for (String sys : elems.get("System")){
29      «FOR elem: abstractions.entrySet»
30        «IF elem.getValue.type==0»
31        if (sys.contains("«elem.getKey»")){
32          library+=«elem.getKey»Decl(sys);
33        }
34        «ENDIF»
35      «ENDFOR»
36    }
37    //Concrete declaration of mediators
38    for (String med : elems.get("Mediator")){
39      «FOR elem: abstractions.entrySet»
40        «IF elem.getValue.type==1»
41        if (med.contains("«elem.getKey»")){
42          library+=«elem.getKey»Decl(med);
43        }
44        «ENDIF»
45      «ENDFOR»
46    }
47    library = updateDatatypes(library,functions);
48    String libDecl = newBlock("library «nameClass»_library is",0,
      library);
49    ...
50  }
```

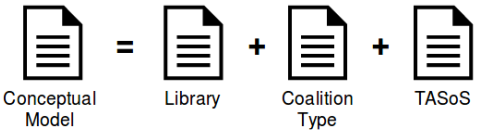
5.2 Project Content

As shown in Figure 12, the content of a project in SoSy comprises artifacts related to abstract architectures as well as generated artifacts, which comprise Java classes, conceptual models, and concrete architectures. All artifacts created by SoSy can be accessed by the user from the project folder perspective in Eclipse. Figure 13a illustrates the artifacts of an abstract architecture description in the project folder, namely: (i) one or more abstract system types;

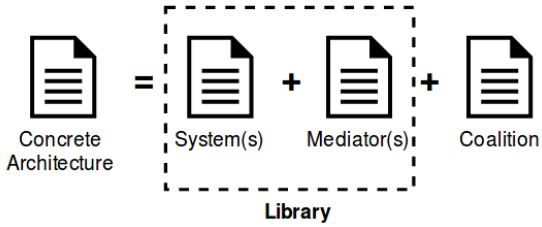
(ii) one or more abstract mediator types; and (iii) at least one coalition type, which identifies potential participants for the SoS and establishes the intentional bindings (i.e., policies) for arranging them into a cohesive whole. Abstract types of systems and mediators are arranged into library files, which can be reuse across different SoS. In Figure 13b, a conceptual model that is generated for an abstract architecture is composed of: (i) one or more library instance modules, which can be reused and/or shared among different instance modules; (ii) a coalition type, which is an architecture instance module of TASoS that represents the architecture behavior defined by the abstract architecture in terms of constraints; and (iii) TASoS, which is the Alloy module that defines the main elements expressed in a SosADL description. This organization promotes reuse and readability of the resulting project since details about a particular architecture are not interwoven with constraints related to SosADL building blocks. The solution for a conceptual model represents one or more concrete instances of the abstract architecture for which all constraints hold. Therefore, the outcome of the constraint solver can be transformed into concrete architectures expressed in SosADL, which contain (Figure 13c): (i) one or more concrete systems; (ii) one or more concrete mediators; and (iii) one or more candidate coalitions, which depict a feasible architectural configuration for these elements. In particular, a validated concrete architecture is selected among concrete architectures realized by this process.



(a) Artifacts of an abstract architecture expressed in SosADL



(b) Artifacts of a conceptual model expressed in Alloy



(c) Artifacts of a concrete architecture expressed in SosADL

Figure 13 – Description of SosADL project content

5.3 Suitability of Tool Support

The suitability of SoSy is demonstrated using the project of a client-server abstract architecture (previously presented in Section 3.3.1 on page 49) as an illustrative scenario. This project is originally composed of four SosADL models: one library model, which declares the abstract types of client and server, and three SoS models, which declare abstract coalition

types. For the sake of completeness, an excerpt of the declaration of clients and servers is shown in Source code 16. According to this description, a client is an abstract type of system that has one public gate named `rr`. The interactions taking place on this gate follow a specific protocol which is omitted from this excerpt. Moreover, the behavior of this element specifies that for each value that it sends through its `req` connection, it receives a response through its `ack` connection. In turn, a server is an abstract type of mediator that has one public duty named `rr`. This duty follows a specific interaction protocol provided that a particular property holds in the environment. The internal behavior of a server defines that for each request that it receives through its `req` connection, it shall send as response the processed value of this request through its `ack` connection. Following, different abstract architectures are outlined, each of which defines their own set of policies for assembling constituents within coalitions.

Source code 16 – Excerpt of library model with client-server abstract types expressed in SosADL

```
1 library ClientServerDefinitions is {
2   datatype t is integer{0..0} {
3     function (self: t) :: f(): t is { ... }
4   }
5   system client() is {
6     gate rr is {
7       connection req is out { t }
8       connection ack is in { t }
9     } guarantee { ... }
10    behavior main is {
11      repeat {
12        value msg: t = 0
13        via rr::req send msg
14        via rr::ack receive v
15  } } }
16  mediator server() is {
17    duty rr is {
18      connection req is in { t }
19      connection ack is out { t }
20    }
21    assume { ... }
22    guarantee { ... }
23    behavior main is {
24      repeat {
25        via rr::req receive v
26        via rr::ack send v::f()
27    } } }
28 }
```

The first coalition type, referred to as “Exactly Two Servers” and shown in Source code 1 on page 51, defines a client-server architecture that can have any number of clients but exactly two servers. The configuration of this architecture is constrained by a single policy in which each client necessarily sends and receives messages from either one of these servers. The second coalition type is specified as compositions of any number of clients and servers. The configuration of this coalition is constrained by two policies: each client must necessarily send and receive a message from a server, and also its interactions are constrained to a single server, hence its name “Single Server” architecture. Source code 17 shows an excerpt of its abstract description. Finally, the third coalition type describes an architecture that can has any number of clients and server provided. In particular, its configuration is constrained by a single policy that states that each client is necessarily connected to the same server for both request and acknowledgement interactions. Source code 18 shows an excerpt of this coalition type.

Source code 17 – Excerpt of coalition type for client-server architectures expressed in SosADL

```
1 with ClientServerDefinitions
2 sos ClientToSingleServer is {
3   architecture Coalition() is  {
4     gate unusedGate is {...}
5     behavior main is compose {
6       servers is sequence { server }
7       clients is sequence { client }
8     } binding {
9       // every client is connected to at least one server
10      forall { c in clients suchthat
11        exists { s in servers  suchthat
12           unify one { c::rr::req } to one { s::rr::req }
13           and unify one { s::rr::ack } to one { c::rr::ack }
14      } }
15      and
16      // every client talks to at most one server
17      forall { c in clients suchthat
18        forall { s1 in servers suchthat
19          forall { s2 in servers suchthat (
20            ( // c talks to s1
21            unify one { c::rr::req } to one { s1::rr::req }
22           or unify one { s1::rr::ack } to one { c::rr::ack }
23            )
24            and
25            ( // c talks to s2
26            unify one { c::rr::req } to one { s2::rr::req }
27           or unify one { s2::rr::ack } to one { c::rr::ack }
```

```
28                )
29              ) implies s1 = s2
30      }    }    } } }
31 }
```

Source code 18 – Excerpt of coalition type for well-formed client-server architectures expressed in SosADL

```
1 with ClientServerDefinitions
2 sos ClientToSingleServer is {
3   architecture Coalition() is  {
4     gate unusedGate is { ...
5     } guarantee { ... }
6     behavior main is compose {
7       servers is sequence { server }
8       clients is sequence { client }
9     } binding {
10      forall { c in clients suchthat
11        forall { s in servers  suchthat
12          // c-s connected via req  <=> c-s connected via ack
13          unify one { c::rr::req } to one { s::rr::req }
14            = unify one { s::rr::ack } to one { c::rr::ack }
15    } } }   }
16 }
```

The suitability of SoSy is determined by its efficiency in terms of elapsed time for the automated synthesis of concrete architectures for each of the aforementioned abstract coalition types. In particular, the tool is expected to: (i) automatically generate valid instance modules of TASoS from abstract descriptions in SosADL; (ii) automatically generate valid Java classes that delegate the search for a solution to a constraint solver; (iii) automatically find a solution that satisfies the constraint satisfaction problem specified by an instance module within the analyzed scope; and (iv) automatically generate one or more concrete architectures in SosADL from the previous solution. A criteria is satisfied if the tool manages these previous tasks without requiring external changes to automatically generated artifacts, e.g., modifying the analysis scope or manually specifying predicates in Alloy models. Table 16 details the outcome of this evaluation. As shown in this table, the tool is able to successfully generate instance modules and Java classes for the three coalition types. The user can manually execute these Java classes to execute the constraint solver and synthesize candidate concrete architectures adhering to the specification of each coalition, such as the one presented in Source code 4 on page 55 for the abstract coalition type “Exactly Two Servers". The table also shows the total running time of this tool, which discounts user interactions. Moreover, it is possible to notice that both transformations are quickly accomplished by the tool.

Criteria	Artifact		
	Single Server	Exactly Two Servers	Well-formed Architecture
Generates conceptual model	✓	✓	✓
Generates Java class	✓	✓	✓
Finds solution	✓	✓	✓
Generates concrete architecture(s)	✓	✓	✓
Elapsed Time (ms)			
Transformation 1 to TASoS	31	29	50
Transformation 1 to Java	2	1	1
Transformation 2	28	37	68
Solver (SAT4J)	7491	19466	9465
Total running time	7552	19533	9584

Table 16 – Overview of SoSy performance

5.4 Final Remarks

In the previous chapter, a constraint based model was presented to support checking the correctness and feasibility of an abstract architecture by finding concrete architectures that adhere to its specification. Nonetheless, the translation of concrete architectures into a constraint satisfaction problem is not trivial, specially as this technology requires one to be familiar with a declarative programming paradigm. To fully take advantage of formal methods in the design of SoS whilst diminishing the risks incurred by adopting a new technology, specialized tools are needed to support architects in the creation of such artifacts. In this scenario, a new tool was presented in this chapter to support activities that are required by Ark. This tool is provided as part of a larger tool set that allows architects to design software architectures for SoS in SosADL. In particular, the tool automates the transformations between SosADL and TASoS and between the output of constraint solvers and SosADL so that the use of constraint solving tools is concealed from the user who can work directly with artifacts expressed in SosADL. In the next chapter, this tool is demonstrated in a case study where it is used to check the feasibility of an abstract architecture for the case of a urban river monitoring SoS.

Chapter

6

Evaluation on Architectural Synthesis of System-of-Systems

One way of automatically verifying if there is any concrete architecture satisfying the set of properties defined by an abstract architecture is through the description of coalitions in terms of a set of constraints. Thereby, off-the-shelf constraint solvers can be employed to search for instances of a family of architectural configurations. To systematize the steps of this process, a new method named Ark was developed, which employs formal languages and model transformations. A custom tool named SoSy supporting this method is provided to automatically describe abstract descriptions of SoS as a Constraint Satisfaction Problem (CSP) and synthesize concrete architectures adhering to such descriptions. In particular, this chapter reports the results of a quasi-experiment on the effectiveness of Ark for the synthesis of SoS architectures.

The remaining of this chapter is organized as follows. Section 6.1 presents the main characteristics and emergent behaviors of a Urban River Monitoring SoS (URMSoS). Section 6.2 details the context and motivation for this experiment. Section 6.3 presents its planning, describing hypothesis, validity concerns, instrumentation, and analyses procedures. Section 6.4 details the execution of Ark steps supported by SoSy and discusses the outcome of these steps. Section 6.5 evaluates the effectiveness of this method. Moreover, this section discusses related work, including alternative approaches for the synthesis of concrete architectures for SoS. Finally, Section Section 6.7 presents final remarks on the results of this evaluation.

6.1 Urban River Monitoring System-of-Systems

Emergency management and response is a relevant application domain for complex SoS (NIELSEN *et al.*, 2015). The design of such SoS often requires the coordination among heterogeneous constituent systems (e.g., surveillance, weather forecast, and flood monitoring systems) and protocols (e.g., police, firefighters, traffic management, first aid and rescue teams).

An Urban River Monitoring SoS plays a key role in obtaining precise, real-time data that supports authorities' timely and organized response. By collecting data about current speed and level of rivers crossing urban areas, the existence of such SoS is key for organizing and taking action in case of flash floods. To achieve this goal, information provided by independent, heterogeneous sensors (e.g., water level, current, and pollutant sensors) are combined. This information can be used for determining imminent flooded zones and organizing rescue and defense teams response. Ultimately, the effectiveness of such SoS has also a relevant impact on costs incurred by flood events, which can be greatly diminished by sending out warnings in advance (ROURE, 2005).

The Urban River Monitoring (URM) is one of such systems that have been placed on the Monjolinho River in São Carlos, Brazil (HUGHES *et al.*, 2011; DEGROSSI *et al.*, 2014). This system is based on a wireless sensor network (WSN) composed of multiple sensor motes[1], which are spread in flood-prone areas near the riverbed and monitor the river condition. Sensor motes can provide pressure and/or ultrasound sensors to respectively gauge the depth and the average current speed. These networks have been widely used in river monitoring and warning systems as they support distributed data collection and provide wide area coverage as a result of independent communication capabilities that are embedded in sensor motes, such as WiFi, ZigBee (IEEE 802.15.4), GPRS, or Bluetooth (HUGHES *et al.*, 2009). To determine the water level, data collected from these sensors are forwarded to a gateway station, which has dedicated resources for processing, integrating, and publishing this information. The gateway station can transform raw data collected by the sensor for determining the relative height reached by the water level and feed this information to a web-based tool that supports decision-making processes. In particular, if a gateway station cannot be reached by individual communication capabilities of a sensor, the sensors network transmits these data to their neighboring motes until the gateway station is reached.

Since this system operates in a highly dynamic environment, its architectural configuration must be continuously changed for ensuring: (i) efficiency in the use of the available resources, mainly in terms of power consumption and communication; (ii) resiliency in case of temporary unavailability of motes during operation; (iii) accuracy in flood detection; and (iv) autonomy in adapting to dynamic environmental conditions while minimizing manual intervention. Thereby, the Urban River Monitoring presents the five characteristics of an SoS (MAIER, 1998). Each sensor mote operates in a way that is independent of other sensor motes, as they belong to different city councils and have different missions, e.g. pollution control or water supply. Each one has its own management strategy for transmission vs. energy consumption and acts under the authority of the different city councils. New sensor motes may be installed by the different councils as well as existing ones may be changed or deactivated without any control from the system. Finally, the sensor motes, coordinated by the gateway, make emerge the behavior of flood detection. This behavior is collectively achieved by the distributed, independent sensors

[1] Sensor motes are tiny hardware/software platforms equipped with embedded CPU, wireless networking capabilities, and simple sensors

working together with the gateway station rather than being provided by any of them working in isolation. In particular, the URM can be considered as *collaborative SoS*.

6.2 Experiment Scoping

The objective of this experiment is to investigate the effectiveness of the Ark method and its tool support for analyzing an abstract architecture that is originally described in SosADL. As previously discussed in Chapter 3, the analysis performed in Alloy is complete within a limited analysis scope. Thus, the Ark method returns concrete architectures for the SoS provided that one of such instances exist within the analyzed scope. Still, an architect may wish to increase the analysis scope to find alternative concrete architectures for the SoS or decrease the scope to improve the efficiency of the solver. As all constraint solvers face the state explosion problem, increasing the analysis scope has a direct impact to the running time of the tool. In this scenario, this experiment is motivated by a need to understand the trade-off that exists between analyzing a broader scope and realizing concrete architectures within feasible amount of time. To ensure the performance of the constraint solver, it is important that architects understand how different scope sizes impact the concrete architectures realized by the method so that it is possible to optimize computational resources and advise them against or in favor of changing the generated analysis scope.

Object of study. The object of study is the Ark method and its support for the transformation of SosADL models into a constraint satisfaction problem expressed in Alloy.

Purpose. The purpose of the experiment is to evaluate the effectiveness of the method based on the analysis scope that is assigned to the constraint solver. In this scenario, the study will help to understand whether an assigned scope is sufficient for discovering any concrete architectures that comply with a given abstract architecture.

Perspective. The perspective is from the point of view of the author, who would like to minimize required computational resources for the synthesis of concrete architectures by means of a constraint solver.

Quality focus. The main effect investigated in this experiment is the tool's performance measured in terms of elapsed time for automatically performing the steps of the Ark method. In particular, the study focuses on productivity (i.e., number of lines of code of generated concrete architecture) and existence of at least one concrete architecture within the analyzed scope.

Context. The experiment is run within the context of a Urban River Monitoring SoS. The analysis is based on a SosADL description for the abstract architecture of this SoS that is composed of one library model and one SoS model. The subject runs this model in the SoSy tool to obtain a representation of the abstract architecture as a set of constraints in Alloy. Using the Alloy Analyzer, several test cases are performed over this description, each of which considers a custom

analysis scope for the architecture instance module that is automatically generated by the tool. The evaluation is performed by one subject, who operates the Alloy Analyzer with the purpose of finding one or more concrete architectures that comply with this model. The subject is familiar with both SosADL and Alloy languages, particularly with the TASoS model, as well as the SoSy tool. In this scenario, the experiment can be characterized as a "single object study", which is run on a single subject and a single object. Moreover, the lack of randomization of subjects and objects characterizes this experimental setting as a quasi-experiment (WOHLIN *et al.*, 2012). The scope of this quasi-experiment can be summarized as follows:

Scope: Analysis of *Ark and its tool support, SoSy,* for the purpose of *evaluation* with respect to *effectiveness* from the point of view of the *SoS researcher* in the context of *a Urban River Monitoring SoS*.

6.3 Experiment Planning

Context Selection. The quasi-experiment is developed in the context of a Urban River Monitoring SoS. The experiment is run off-line (not in an industrial software development), being conducted by the author, and it is focused on a singular architecture and its corresponding abstract description in SosADL. The quasi-experiment addresses the architectural synthesis of concrete architectures and its relation to the analyzed scope.

Hypothesis Formulation. The quasi-experiment aims at understanding two potential implications of an analysis scope to the architectural synthesis of SoS with the Ark method: (i) a larger scope faces the state explosion problem, hence the method is expected to present lower reliability due to failure of returning a concrete architecture within a feasible time frame; and (ii) a larger scope may hold an architectural instance that is larger than the one that would exist in a smaller scope, hence the method is expected to present a higher productivity. Based on this description, it is possible to formally state these hypothesis and the measures that are needed to evaluate them:

1. There is no difference in the reliability of the method based on the selection of different analysis scope for the synthesis of concrete architectures.

   H_0: The reliability of the method is independent of analysis scope.

   H_1: The reliability of the method changes with analysis scope.

   Measures: analysis scope and satisfiability of abstract architecture.

2. There is no difference in the productivity of the method based on the selection of different analysis scope for the synthesis of concrete architectures.

   H_0: The productivity of the method is independent of analysis scope.

   H_1: The productivity of the method changes with analysis scope.

Measures: analysis scope, productivity (LOC/running time), and satisfiability of abstract architecture.

Based on these hypothesis, the following data must be collected: (i) analysis scope: measured by the quantity of elements assigned to each signature in the execution of the conceptual model (ordinal scale); (ii) SoS description: measured as LOC of the abstract architecture (ratio scale); (iii) productivity: measured as LOC of the concrete architecture divided by running time (ratio scale); (iv) running time: measured as combined running time of the constraint solver and the Solution2SosADL generator (ratio scale); and (v) satisfiability of abstract architecture: measured by satisfiable, undetermined, or unsatisfiable (nominal scale). In particular, the measure for LOC is determined by a line counter program.

Variables Selection. The quasi-experiment has control over independent variables, or input variables. In turn, dependent variables, also referred to as response or output variables, have their values observed throughout the quasi-experiment to determine the effect of the independent variable on them. In this scenario, the independent variables in this quasi-experiment are analysis scope and SoS description, which is then transformed into a constraint satisfaction problem. Dependent variables are productivity, running time, and satisfiability of abstract architecture. Table 17 further details these variables. Since the SoS description is an independent variable with fixed level (the experiment is run on a single object case), the analysis scope is the single treatment investigated in this quasi-experiment.

Experiment Design. Neither the object of this experiment or its subject have been selected by chance. The subject of this experiment is a PhD student with experience on architectural descriptions and ADLs (in particular, UML, SysML, and SosADL).

1. The first part of this quasi-experiment evaluates the correlation between the analysis scope and reliability of the method.

2. The second part of this quasi-experiment evaluates the correlation between the analysis scope and productivity of the tool and transforming the model back into a SosADL description.

Instrumentation. The object of this experiment is the abstract architecture description of URM is described in SosADL. This description is uploaded in the SoSy tool, which automatically generates an instance module of TASoS for SosADL model and a Java class for executing the SoS model. Once the Java class is generated, its main method is modified to measure the running time of the constraint solver and the transformation from the solution returned by the solver to a SosADL model. This operation is repeated for each different scope that varies the maximum number of systems, mediators, and relay connections in a coalition (the scope of other signatures is dynamically determined). The Java class is then executed several times and the satisfiability

Table 17 – Description of experiment variables

Name	Type of variable	Class	Entity	Type of attribute	Scale type	Unit	Range or scale points	Counting rule
Analysis scope (treatment)	independent	Process	Constraint solving	Internal	Interval	Quantity of elements in category	$0 < n < 10$	Quantity of elements in system, mediator, and relay signatures (quantity of unifications, ports and connections are derived from the previous ones)
SoS description	independent	Product	Abstract architecture	Internal	Ratio	LOC	$t >= 0$	LOC of abstract architecture expressed in SosADL
Productivity	dependent	Resource	Concrete architecture	External	Ratio	LOC per ms	$t >= 0$	LOC of generated concrete architecture expressed in SosADL per running time
Running time	dependent	Process	Transformation	External	Ratio	ms	$0 < t < 3 \times 10^5 ms(5min)$	Combined elapsed time for the execution of the constraint solver execution and the Solution2SosADL generator
Satisfiability of abstract architecture	dependent	Process	Constraint solving	Internal	Nominal	Category	Satisfiable, undetermined, unsatisfiable	Satisfiable if the solver finds a solution within the scope. Unsatisfiable if it terminates execution within 5 minutes without finding a solution. Undetermined if it does not terminates execution within 5 minutes.

Key: Type of variable: independent or dependent; Class: product, process, or resource; Entity: instance of class;
Type of attribute: internal, external;Scale type: nominal, ordinal, interval, ratio, among others

of the instance module representing the abstract architecture and productivity of the tool are recorded. The generated concrete architectures are kept in the project folder and their size is calculated by a line counter tool. The materials used in this experiment are organized into two datasets: Dataset 1 contains the input for the Ark method, which is the source code for the library model and the SoS model of the URM described in SosADL; and Dataset 2 contains the output of the SoSy tool, which includes generated instance modules of TASoS as intermediary models and solutions found by the constraint solver as concrete architectures, and experiment materials, i.e., the instrumented Java class and a set of concrete architectures.

Validity Evaluation. Four levels of validity threats were identified for this quasi-experiment. Following, each of these threats are discussed in detail.

- *Internal validity* is an inherent risk of quasi-experiments since treatments are not assigned to subjects and objects by chance. Furthermore, there is only one object and one subject in this experiment due to stage of development of the tool set (SosADE) and lack of subjects with required skills. Still, this risk is ameliorated since the experiment is focused on evaluating the constraint solving activity of the method and, thus, the subject plays a minimal role to start the execution of the test and record its outcome.
- *External Validity* is also a concern for this experiment since results reflect a single object case and particularly only one abstract coalition type. This risk is mitigated by selecting as object a system that has the fundamental characteristics of an SoS, as previously discussed. It is also possible to expand this experiment for the comparison of the reliability for different constraint solvers and/or coalition types.
- *Construct validity* is related to the threat that selected measurements are not appropriate for the entities. To mitigate this threat, the experiment selects objective measurements, such as LOC and analysis scope, which do not depend on the context. To mitigate this risk for subjective variables, such as running time, each test is replicated twice.
- *Conclusion Validity* is a threat in this experiment since only one abstract architecture for the SoS is investigated. However, for the purposes of this experiment, i.e. investigate the effect of analysis scope to the productivity and reliability of the method, it is important to select a feasible abstract architecture, for which the constraint solver is expected to eventually produce a concrete architecture. This threat is further mitigated by presenting a detailed protocol that can be used for future replications and expansions of this experiment.

6.4 Experiment Operation

This section details step-by-step of the synthesis of a concrete architecture for the URM given its abstract description in SosADL. First, the abstract architecture of the SoS is detailed in Step 1. Then, the transformation of this description into instances modules of TASoS is described in Step 2. Step 3 describes how this instance module is run with different analyses scope by

instrumenting the generated Java class. Finally, Step 4 discusses solutions that have been found for this model.

6.4.1 Step 1: Describe URM Abstract Architecture

The first step of the method concerns the description of an abstract architecture for the SoS under analysis. To perform this task, the subject uses the SosADE tool to create SosADL models for URM. The SosADL description of this SoS comprises two abstract types of systems, namely sensor and gateway. These systems can collaborate with each other by means of a mediator, named transmitter, which represents wireless connections. Then, an abstract coalition type that describes families of architectural configurations that can be realized from such elements is also specified for URM. Following, the description of these abstract types is further elaborated.

A *gateway* is an abstract type of system that requests observations to sensors and publishes collected data. The specification of this system, shown in Source code 19, encompasses one gate named `alerting` that has three connections: (i) `request` receives queries from other systems about current condition of the river; (ii) `measure` handles observations returned by other systems; and (iii) `alert` sends out warning messages when an internal parameter of the gateway is violated by environmental conditions. All connections have an associated direction (inward or outward) and data type, which is limited to an abstract integer type named `t` for the sake of simplicity. For describing the behavior, the subject specifies a sequence in which the system is expected to interact with the environment. The main behavior of the gateway is depicted in lines 20-28. It begins by receiving an observation from another systems via its `measure` connection. Then, the gateway evaluates if this observation surpasses the local depth threshold (line 24). If so, it publishes a warning message via its `alert` connection of the same gate. These actions can be repeated indefinitely.

Source code 19 – Excerpt of abstract gateway type for URM in SosADL

```
3   system gateway() is {
4       gate alerting is {
5           connection measure is in {t}
6           connection request is in {t}
7           connection alert is out {t}
8       } guarantee {
9           protocol alertingpact is {
10              repeat {
11                  via request receive any
12                  repeat {
13                      via measure receive any
14                      repeat { anyaction }
15                  }
16                  via alert send any
```

```
17                  }
18              }
19          }
20          behavior main is {
21              value depththreshold : t = 3
22              repeat {
23                  via alerting::measure receive v
24                  if (v > depththreshold) then {
25                      via alerting::alert send v
26                  }
27              }
28          }
29      }
```

A *sensor* is also an abstract type of system that collects observations from the environment. The specification of this system, shown in Source code 20, encompasses two gates: (i) `measuring`, that comprises one environment connection named `sense` that reads observations and one connection named `measure` that handles these observations over to a neighboring system; and (ii) `passing`, that has two connections `pass` and `measure` for just handling observations over to other systems. The specification of these gates is complemented by a guarantee protocol, which describes assumptions (i.e., properties) that must be fulfilled by the environment. For instance, the `measuring` gate guarantees that while the sensor is operational (i.e., if all assumptions that a sensor makes about the environment hold), it will continually receive observations from the environment via its connection `sense` and transmit these data via its connection `measure`. The main behavior of this system is declared in lines 54-64. The sequence of interactions is formed by a choice, the sensor either makes an observation via its `sense` connection of the `measurement` gate and sends it to one of its neighboring systems via the `measure` connection of the same gate, or it receives an observation via its `pass` connection of the passing and forwards it to one of its neighboring systems via the `measure` connection of the same gate.

Source code 20 – Excerpt of abstract sensor type for URM in SosADL

```
30      system sensor() is {
31          gate measuring is {
32              environment connection sense is in {t}
33              connection measure is out {t}
34          } guarantee {
35              protocol measuringpact is {
36                  repeat {
37                      via sense receive observation
38                      repeat { anyaction }
39                      via measure send observation
```

```
40                  }
41              }
42          }
43          gate passing is {
44              connection pass is in {t}
45              connection measure is out {t}
46          } guarantee {
47              protocol passingpact is {
48                  repeat {
49                      via pass receive data
50                      via measure send data
51                  }
52              }
53          }
54          behavior main is {
55              repeat {
56                  choose {
57                      via measuring::sense receive observation
58                      via measuring::measure send observation
59                  } or {
60                      via passing::pass receive data
61                      via passing::measure send data
62                  }
63              }
64          }
65      }
```

A *transmitter* is an abstract type of mediator that forwards observations from sensors to a gateway, either directly or by means of other sensors. The specification of this mediator, shown in Source code 21, comprehends one duty, named `transmitting`. This duty has two connections named `fromSensors` and `toGateway` that handle the transmission of observations from sensors to a gateway. A duty can also declare assumptions that must be fulfilled by constituents in the environment. In this case, this duty requires that the distance between the source and target of this communication link do not exceed a predefined range threshold. If this property holds, then the abstract description of mediator can guarantee that the protocol declared in `transmittingpact` is also fulfilled. This protocol defines that all observations received via the `fromSensors` connection are forwarded via its `toGateway` connection. Then, transmitting behavior of this mediator states that this behavior is continuously repeated, with no processing in between.

Source code 21 – Excerpt of abstract mediator type for URM in SosADL

```
66      mediator transmitter() is {
```

```
67          duty transmitting is {
68              connection fromSensors is in {t}
69              connection toGateway is out {t}
70          } assume {
71              property inrange is {
72                  repeat {anyaction}
73              }
74          } guarantee {
75              protocol transmittingpact is {
76                  repeat {
77                      via fromSensors receive measure
78                      via toGateway send measure
79                  }
80              }
81          }
82          behavior transmitting is {
83              repeat {
84                  via transmitting::fromSensors receive measure
85                  via transmitting::toGateway send measure
86              }
87          }
88      }
```

The aforementioned types compose a single SosADL library, named urmLibrary. To complement this description, the subject creates an abstract coalition type for URM built on top of these definitions. Source code 22 shows the abstract architecture named simple for URM. The *serving* gate of this architecture enables the communication of the Flood Monitoring coalition with other systems and SoS. This gate declares two connections named `request` and `alert` that handles external queries about the conditions of the river and publish warning messages in case one of the sensors has identified a flood event. The main behavior of this coalition is named main. Differently from the behavior of systems and mediators, the behavior of architectures contains: a **compose** statement, which establishes that constituents are allowed to participate in the coalition; and, a **binding** statement, which defines policies for organizing systems and mediators into a cohesive whole (lines 16-39). The simple type of this coalition is composed of any number of sensors and transmitters but has only one gateway, named gateway1. Furthermore, any instance of this abstract coalition type may only exercise the unifications defined in the binding statements.

Source code 22 – Excerpt of abstract coalition type for URM in SosADL

```
1 with urmLibrary
2 sos FloodMonitoring is {
```

```
3     architecture simple() is {
4         gate serving is {
5             connection request is in {t}
6             connection alert is out {t}
7         } guarantee {
8             protocol servingpact is {
9                 repeat {
10                    via request receive location
11                    repeat { anyaction }
12                    via alert send data
13                }
14            }
15        }
16        behavior main is compose {
17            sensors is sequence {sensor}
18            transmitters is sequence {transmitter}
19            gateway1 is gateway
20        } binding {
21            forall { t in transmitters suchthat
22             forall { s1 in sensors suchthat
23              forall { s2 in sensors suchthat (
24                // t receives from s
25                ( unify one { s1::measuring::measure } to
26                      one { t::transmitting::fromSensors }
27                or unify one { s1::passing::measure } to
28                      one { t::transmitting::fromSensors }
29                // and sends to s or g
30                ) and (
31                 unify one { t::transmitting::toGateway } to
32                      one { gateway1::alerting::measure} xor
33                 unify one { t::transmitting::toGateway } to
34                      one { s2::passing::pass} )
35               )
36              }
37             }
38            }
39        }
40    }
41 }
```

6.4.2 Step 2: Describe URM as a Constraint Satisfaction Problem

The second step of Ark requires the transformation of the URM abstract architecture described in SosADL into a constraint satisfaction problem described in Alloy. To accomplish this task, the subject must extend abstract signatures in TASoS with elements of URM that were defined in the abstract architecture. In SosADE, this task can be automatically performed by the SoSy tool, which is running on the background. Therefore, instance modules of TASoS are dynamically created by the environment while the subject is creating SosADL models. Source code 23 shows an excerpt of the generated Alloy model for the simple coalition abstract type. The compose statement of the abstract architecture is mapped by the constraints in lines 30-32. Accordingly, the binding statement is mapped by the constraints in lines 39-50. SoSy also generates a default scope to run the analysis, which is shown in line 65.

Source code 23 – Architecture instance module of TASoS for URM

```
1 open tasos
2 open util/ordering[tasos/Architecture] as A0
3 open urmLibrary
4 /** Architecture(s) Declaration(s)
5 */
6 sig FloodMonitoring extends Sos{}
7 sig simple_serving extends Gate{}
8 sig simple_serving_request extends Inward{}
9 sig simple_serving_alert extends Outward{}
10 -- constraints about the structure of the architectural element
11 fact {
12 attributePortToAE[simple_serving,FloodMonitoring]
13 all c:FloodMonitoring |
14     one p1:simple_serving |
15         p1 in c.hasPort
16 attributeConToPort[simple_serving_request+simple_serving_alert,
     simple_serving]
17 all c:simple_serving |
18     one p1:simple_serving_request, p2:simple_serving_alert |
19         p1 in c.hasConnection and p2 in c.hasConnection
20 attributeDatatypeToCon[t,simple_serving_request]
21 all c:simple_serving_request |
22     one p1:t |
23         p1 in c.hasDatatype
24 attributeDatatypeToCon[t,simple_serving_alert]
25 all c:simple_serving_alert |
26     one p1:t |
27         p1 in c.hasDatatype
```

```
28 }
29 -- constraints about constituents instances in the coalition
30 sig sensors extends sensor{}
31 sig transmitters extends transmitter{}
32 one sig gateway1 extends gateway{}
33 -- abstract unifications that can exist in the architecture
34 sig simple extends Architecture{}{
35 all t: transmitters |
36  all s1: sensors |
37    all s2: sensors |
38     ((
39     unify[sensor_measuring_measure&(s1.~owner),
      transmitter_transmitting_fromSensors&(t.~owner)] and //u1
40     unify[transmitter_transmitting_toGateway&(t.~owner),
      gateway_alerting_measure&(gateway1.~owner)] //u3
41     ) or (
42     unify[sensor_measuring_measure&(s1.~owner),
      transmitter_transmitting_fromSensors&(t.~owner)] and //u1
43     unify[transmitter_transmitting_toGateway&(t.~owner),
      sensor_passing_pass&(s2.~owner)] //u4
44     ) or (
45     unify[sensor_passing_measure&(s1.~owner),
      transmitter_transmitting_fromSensors&(t.~owner)] and //u2
46     unify[transmitter_transmitting_toGateway&(t.~owner),
      gateway_alerting_measure&(gateway1.~owner)] //u3
47     ) or (
48     unify[sensor_passing_measure&(s1.~owner),
      transmitter_transmitting_fromSensors&(t.~owner)] and //u2
49     unify[transmitter_transmitting_toGateway&(t.~owner),
      sensor_passing_pass&(s2.~owner)] //u4
50     ))
51 }
52 -- constraints about the architecture
53 fact {
54     attributeArchToSos[simple,FloodMonitoring]
55 }
56 -- scenario(s) that will be checked
57 fact instanceOfsimple{
58     some a: simple {
59         sensors in a.contain
60         transmitters in a.contain
61         gateway1 in a.contain
```

```
62      }
63 }
64 -- command line to find one solution
65 run case0 {some sensor} for 3 but 1 Architecture, 4 System, 4
       Mediator, 1 Sos, 7 Port, 16 Connection, 16 Unification, 1 Relay
```

6.4.3 Step 3: Find Solution for Constraint Satisfaction Problem

The third step of Ark comprises the automated analysis of the generated instance model by the constraint solver. To perform this task, the subject executes the main method of the generated Java class. For the default analysis scope, the subject is presented with a solution that shows a concrete coalition of this model. Thus, SoSy automatically transforms this solution into its corresponding SosADL model. Source code 24 shows an excerpt of a concrete architecture found in this analysis.

Source code 24 – Excerpt of concrete architecture for URM that is compliant with abstract coalition type

```
1 with urm_library
2 sos FloodMonitoring0 is {
3   architecture simple0() is {
4     gate serving0 is {
5       connection request0 is in{RangeType0}
6       connection alert0 is out{RangeType0}
7     } guarantee {
8           protocol servingpact is {
9             repeat {
10                    via request0 receive location
11                    repeat {anyaction}
12                    via alert0 send data
13              }
14            }
15        }
16      behavior main is compose {
17        gateway10 is gateway10
18        urmLibrary/gateway0 is urmLibrary/gateway0
19        transmitters0 is transmitters0
20      } binding {
21        unify one {transmitters0::transmitting0::toGateway0} to
22            one {urmLibrary/gateway0::alerting0::request0} and
23        unify one {gateway10::alerting1::alert1} to
24            one {transmitters0::transmitting0::fromSensors0}
25      }
```

```
26     }
27 }
```

Since this model is satisfiable, the generated Java class was instrumented to repeatedly run this analysis with a different scope each time. Appendix B further details which materials were used to automatically run this quasi-experiment. The scope of each test case is a combination of all possible variations withing the interval defined in Table 18, which adds up to 396 test cases. For the sake of simplicity, signature `Relay` was fixed with value 1. In particular, the scope assigned to signature `ArchitecturalElement` covers all elements of types `System`, `Sos`, and `Mediator`, including their extensions, if any of such exists in the analyzed model. For each test case, the running time and response of the constraint solver were recorded. Also, if the model was satisfiable, the running time of the `Solution2SosADL` generator was also recorded. In particular, only the first solution returned by the solver is translated into a SosADL model. Thereby, even if an analysis scope contains more than one solution, only the first solution returned by the solver is considered for the determination of productivity.

Table 18 – Variation of analyzed scope

	Signature			
Variation	Architectural Element	Port	Connection	Unification
Min	3	4	8	8
Max	7	8	11	11

6.4.4 Step 4: Evaluate Candidate Concrete Architectures for URM

Finally, in the fourth and final step of Ark, the subject can evaluate generated concrete architectures in order to check the suitability of this concrete architecture in regards to one's original intent when designing the abstract coalition type in the first step. For instance, the subject can use generated concrete architectures as input for simulations of this SoS. Then, based on the outcome of this analysis, the subject may want to modify the bindings described in the abstract architecture to prevent the formation of a particular kind of coalition or accept this abstract architecture, hence proceeding to next stage of development. Despite its relevance, this activity is out of the scope of this quasi-experiment.

6.5 Analysis and Interpretation of Results

From all 396 test cases, only one (n. 108) did not terminate correctly due to lack of memory. For all other cases, the constraint solver was able to produce a response under the predefined time limit of five minutes. Moreover, the evaluated coalition type was considered as unsatisfiable in 237 cases (i.e., 60%) and as satisfiable in 158 cases (i.e, 40%). Raw data collected in this quasi-experiment is presented in Appendix B. Following, the results of the

analysis performed over these data are discussed. First, Table 19 summarizes the descriptive statistics for dependable variables running time of solver, running time of transformation, and productivity (LOC/min). In particular, productivity is only calculated for satisfiable test cases since unsatisfiable or undetermined test cases generate no solution. The running time of the constraint solver shows great variation among test cases that have been evaluated, although none of them has taken longer than one minute to complete. Analyzing satisfiable and unsatisfiable test cases separately, it is possible to observe that the solver was at least three times faster to evaluate the latter (running time between 754ms minimum and 6861ms maximum) than the former (running time between 2591ms minimum and 42910ms maximum). Overall, the transformation of the solution back into SosADL is efficient once a solution for the architecture is found.

Table 19 – Descriptive statistics of dependent variables

	Solver (ms)	Transformation (ms)	Productivity (LOC/min)
Quantity	395	158	158
Min	754	30	37
Max	42910	5706	680
Mean	3954	109	322
Median	3578	43	309

Figure 14 shows the distribution of test cases for type of result (satisfiable or unsatisfiable) and maximum quantity of elements per signature. This analysis shows that only the scope of connections has a potential relation with the outcome of the solver, i.e., practically all test cases that assigned 10 or 11 connections to this signature evaluated the model as satisfiable. Indeed, there is only one test case that had 10 or 11 connections but it was evaluated as unsatisfiable. All other signatures, however, seem not to interfere with the outcome of the constraint solver.

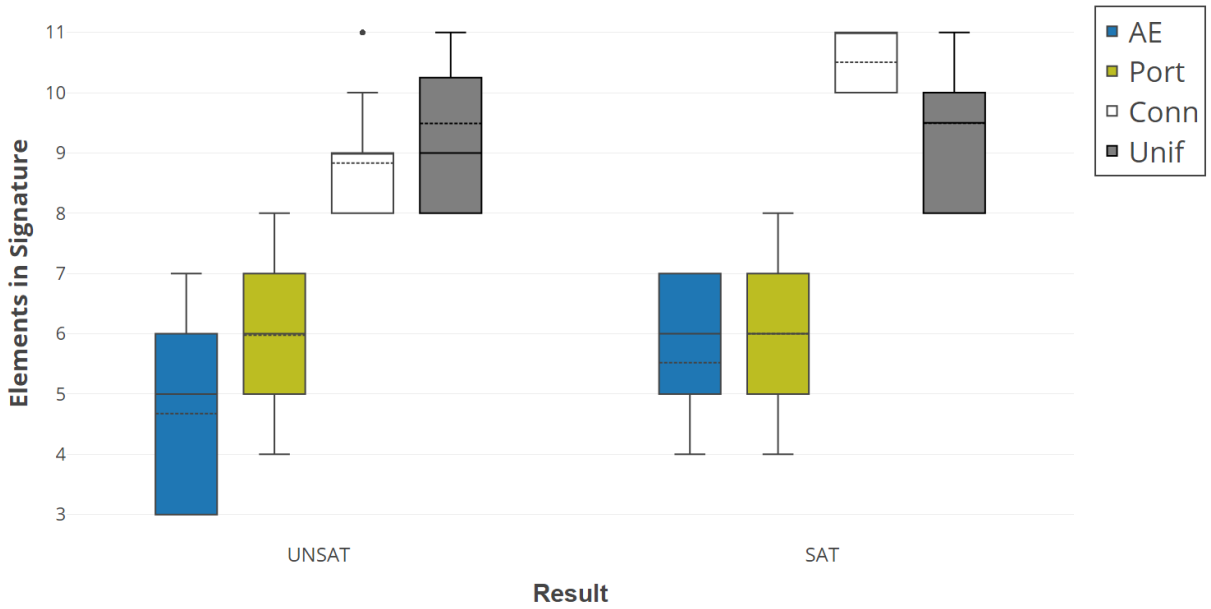


Figure 14 – Distribution of signature per satisfiability

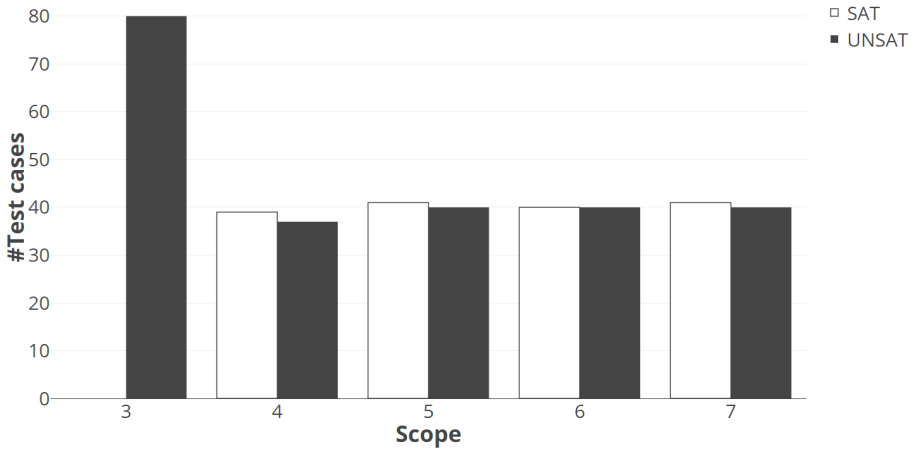
To evaluate the impact of a particular scope to the reliability of the method, the distribution of test results was analyzed in regards to the maximum number of elements per signature, shown in Figure 15. By selecting a scope that is too small, the constraint solver can fail to return an instance of the analyzed abstract coalition type, which is observed in Figure 15a and Figure 15b for a scope of three elements and a scope of eight or nine elements, respectively. A scope that has less than four elements for `ArchitecturalElement` is always unfeasible because this top level signature encompasses the ones of `Sos`, `System`, and `Mediator` and the model defines four elements of this type, namely `sensor`, `gateway`, `transmitter`, and `FloodMonitoring`. Hence, an analysis scope must have at least four elements of `ArchitecturalElement`. The fact that a particular scope for a signature presents mixed results also indicates that it is not determinant for the outcome. Conversely, by selecting a scope that is too large, the constraint solver can fail to return an instance within a feasible amount time. Nonetheless, it was not possible to further investigate this issue since all test cases finished under the time limit.

The productivity evaluation of SoSy is measured as the ratio between LOC of generated SosADL architectures and combined running time of the constraint solver and the `Solution2SosADL` transformation. The following analysis is focused on test cases that returned a solution, since the productivity of an unsatisfied test case is always zero (i.e., no architecture is generated). Figure 16 shows how different scopes for the signatures `ArchitecturalElement`, `Port`, and `Connection`, `Unification` are distributed in respect to productivity. The greatest productivity variation is observed in Figure 16a for the `ArchitecturalElement` signature, whose values are more scattered. It is important to point out that raw data collected in this experiment (presented in Appendix B on page 151) shows the same LOC for all satisfied test cases. This is due the fact that the solver automatically returns the smallest solution within the analyzed scope. Considering the design of this experiment, it follows that the productivity of a (combined) small scope is greater than the one of a (combined) large scope as it takes less time to perform an exhaustive analysis of the solution space. However, a larger scope might contain more than one solution, which are not taken into account in this measure. Therefore, it is not possible to conclude that the productivity of the tool is always better with a smaller scope, since only the first solution returned by the solver is transformed into SosADL.

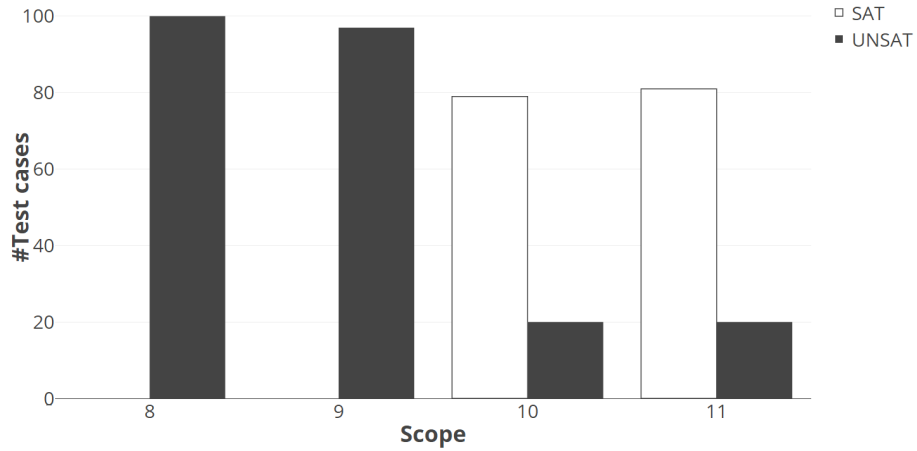
6.6 Related Work

The method presented in this thesis is related to works that investigate constraint solving in the context of software architectures. Kezniki *et al.* (2014) use Alloy for automatically conceiving a connector instance configuration which determines how available elements can be composed together for realizing the architecture. Their approach takes into account non-functional properties, which describe structured/enumerated features, and roles for connectors. Their work, however, does not take into account the concept of coalition, which is covered in the approach developed in this thesis. Heyman, Scandariato and Joosen (2010) propose a metamodel

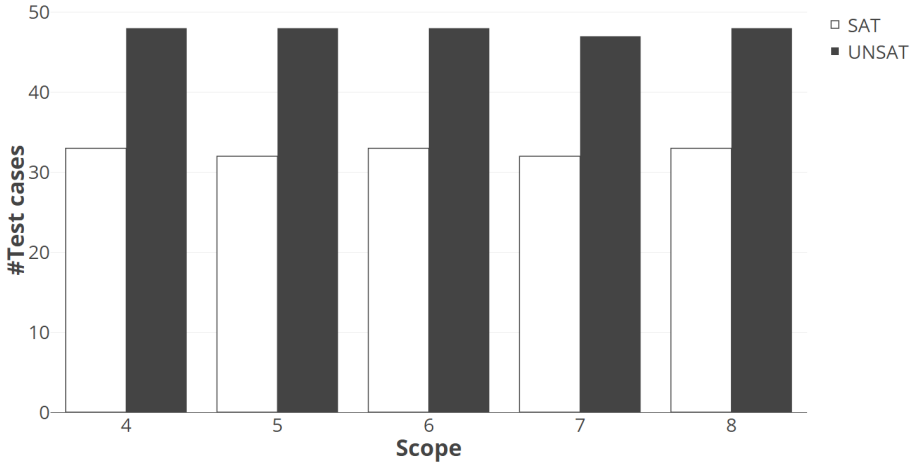
Figure 15 – Distribution of test result per signature



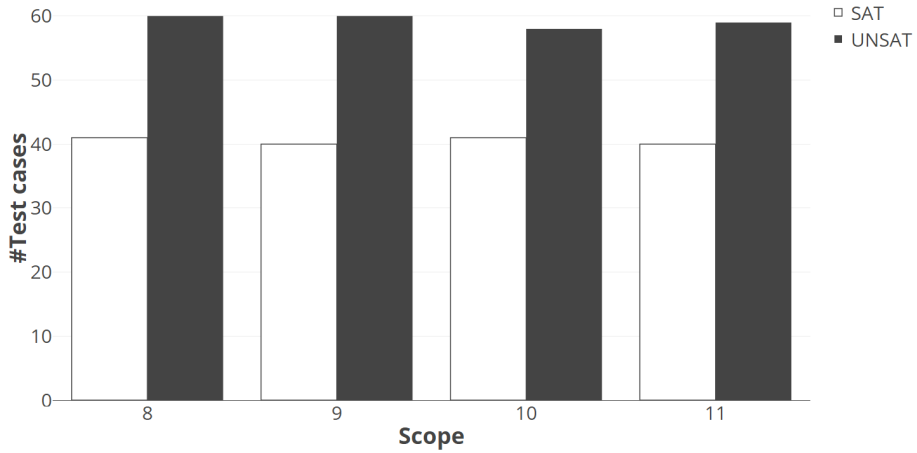
(a) Results for architectural element



(b) Results for connection

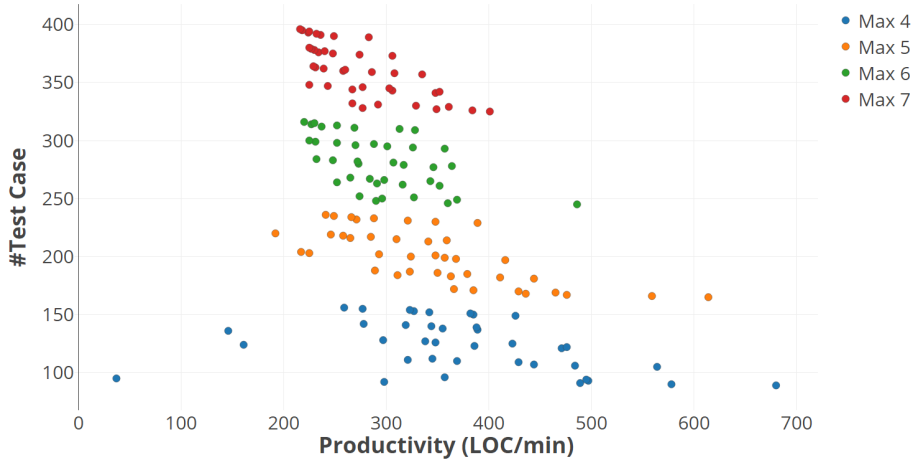


(c) Results for port

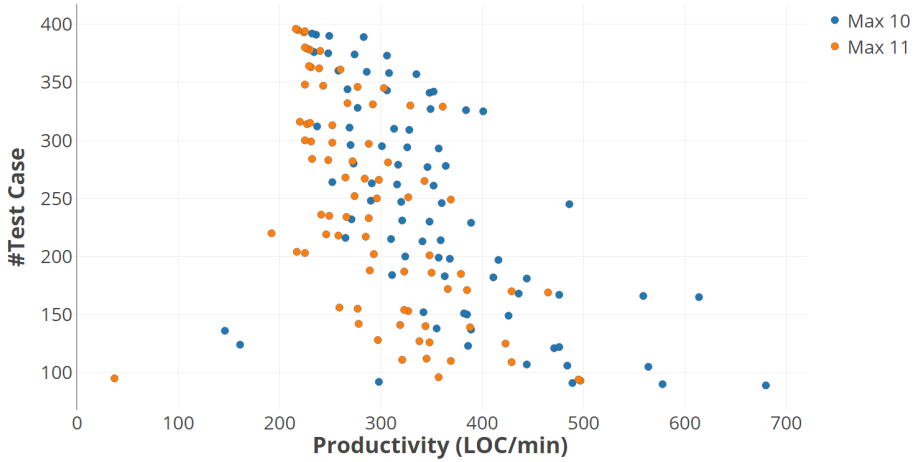


(d) Results for unification

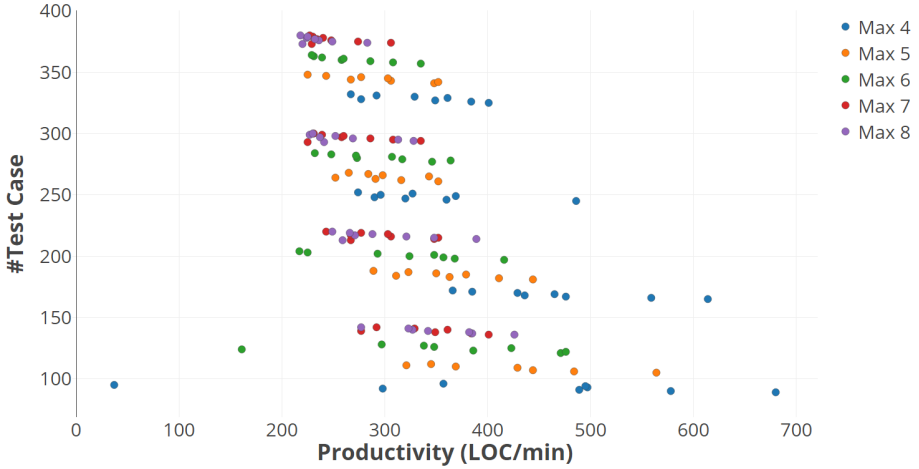
Figure 16 – Distribution of test productivity per signature



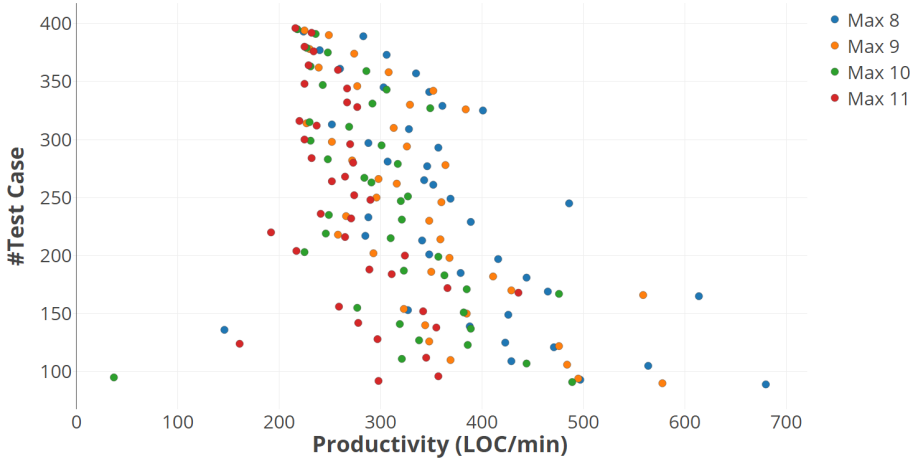
(a) Productivity relative to architectural element



(b) Productivity relative to connection



(c) Productivity relative to port



(d) Productivity relative to unification

of software architectures in terms of constraints, focusing on security aspect of these software architectures. Their metamodel is derived from the “4+1" Viewpoints (KRUCHTEN, 1995) realized with UML. Despite the relevance of UML for describing software systems, UML lacks important features for describing all relevant aspects of SoS, such as abstract types for mediators, coalitions, and dynamic properties (GUESSI; CAVALCANTE; OLIVEIRA, 2015).

The design and analysis of SoS is also the subject of international projects, such as COMPASS[2] (Comprehensive Modelling for Advanced Systems of Systems) and DANSE (Designing for Adaptability and evolutioN in System of systems Engineering)[3]. DANSE employs SysML to the description of executable architectures that can be analyzed against interface contracts. The approach is aimed at supporting the generation of new architectures as well as generation of subsequent ones by means of transformation steps from its evolution. COMPASS develops a formal approach and applies CML (discussed in Table 20) for enriching the specification of systems and interfaces with contracts. One disadvantage of this approach is that an automatic transformation of SysML into CML can produce large, unreadable descriptions. Kenley *et al.* (2014) defines a process for synthesizing SoS architectures. The authors refer to three different models of the SoS: a functional architecture, that establishes a flow for functions execution that facilitates to achieve a mission; a physical architecture, that defines a set of physical capabilities (e.g., sensors, databases, and communication links); and an allocated architecture, which attributes functional capabilities to physical components. According to this description, the purpose of functional and physical architectures is closer to the meaning of abstract architectures whilst an allocated architecture is closer to concrete architectures. The authors use a dynamics mode for the dynamic behavior of allocated architectures as input for an executable model that simulates its behavior. These executable models are implemented using Discrete Agent Framework (DAF)(MOUR *et al.*, 2013), which is based on MATLAB. The task of creating allocated architectures is automated by a model builder developed in DAF that replaces explicit definitions of arrangements and interconnections by a physical network, which defines available point-to-point links, and an agent data path, which selects which constituents are to be connected. Assumptions and the link allocation algorithm are employed to tailor architectures according to architects preferences, e.g., by choosing faster physical links in place of the shortest path. Then, different techniques can be used, including UML activity diagrams for modeling the dynamics model and transforming then in Petri nets to create an executable model. Table 20 summarizes the characteristics of each related approach.

The method is also related to works that investigate the main concepts that are covered by a SoS architecture. Gonçalves *et al.* (2014) present a conceptual model of SoS that extends the ISO/IEC/IEEE 42010 (2011) standard with notions that are derived from the literature on SoS. In particular, their conceptual model defines purposes, stakeholders, and systems (i.e., constituents and the SoS itself) related to SoS. In turn, TASoS is a theory for describing SoS architectures

[2] COMPASS, <www.compass-research.eu>
[3] DANSE, <www.danse-ip.eu>

Table 20 – Comparison of model-based approaches for SoS architectural synthesis

Citation	Abstract Architecture	Concrete Architecture	Technique	Tool Support
Ark	●	●	SosADL, Alloy	SoSy (Eclipse)
COMPASS	●	●	SysML, CML	Symphony (Eclipse)
DANSE	○	●	SysML	Tool-Net
Kenley *et al.* (2014)	○	●	UML, Petri net	DAF (MATLAB)

●: supported, ○: partially supported, -: not available

as constraints. Besides covering the architectural elements presented by SosADL language, the model is implemented Alloy, hence enabling its automated analysis with constraint solving tools.

The method is also related to works that investigate formal architectural models for SoS. Baldwin and Sauser (2009) use set theory for mathematically representing SoS characteristics by means of systems, goals, and actions. Autonomy is individually defined for each system as the cardinality of the set of actions that it contributes for achieving the SoS goals. Belonging is defined as the ratio between actions that a system contributes for the SoS goal and its autonomy. Still, systems can only participate in the SoS if their belonging is greater than a threshold, which is inversely proportional to their contributed value. Connectivity is enabled if two systems share at least one connector. Moreover, the connection is broken if any system contributes below the belonging threshold.Using agent-based modeling, they simulated if it is possible to dynamically create coalitions given constituents' autonomy, belonging, and connectivity. At the architectural level, however, their model lacks abstractions for describing interfaces, coalitions, and architectural configurations, which would enable to specify particular architectural styles for their interaction. Khlif *et al.* (2014) focus on the decomposition and refinement of an SoS architecture expressed in SySML. In particular, their approach does not cover the topology of the architecture, i.e., how constituents and mediators are connected together, and possible modifications to the software architecture. In this scenario, SoSy enables to use Ark to automatically derive formal models that can be analyzed in constraint solving tools and analyze alternatice concrete architectures that can be realized from it.

6.7 Final Remarks

This chapter presented the results of a quasi-experiment that evaluates the effectiveness of Ark and its accompanying tool. This experiment showed that the tool can provide architects with a quick feedback on the soundness and correctness of the abstract architecture description. In particular, architects can use the tool to visualize potential instances of this description and, hence, decide in favor of refining or changing the policies that govern the formation of coalitions at run-time. The study also investigated the reliability and productivity of the method considering different scopes for analysis. This evaluation shows that a scope must be sufficiently large to

encompass all possible extensions of a given signature. While it was not possible to conclude about the productivity of the tool, the study shows that it can automatically generate SosADL models that adhere with an abstract description. Motivated by these results, further studies can be conducted to investigate different scenarios and case objects.

Chapter

7

Conclusion

SoS are evolutionarily developed to achieve missions through emergent behaviors. In this context, a precise architectural description is important to better understand the particularities of SoS (DOGAN *et al.*, 2013), as well as enhance their perceived quality (PARNAS, 2010). However, a complete description of SoS architecture is often not possible since concrete systems that actually participate in a coalition could be not known in advance and, as a consequence, architectural configurations might change in unpredictable ways at run-time. While some studies address the architectural description of SoS (LANE; BOHN, 2012), some challenges still remain. For instance, well known ADLs that have been used in the description of SoS (GUESSI *et al.*, 2015), such as UML and SysML, lack expressiveness for describing structural and behavior changes (GUESSI; CAVALCANTE; OLIVEIRA, 2015; MALAVOLTA *et al.*, 2013). Therefore, this thesis applies a formal ADL to create partial descriptions (referred to as abstract architectures) of the SoS and its constituent systems. To guarantee the correctness and soundness of abstract architectures that are being designed for SoS, additional means must be put in place to verify that all possible concrete configurations that can be realized from it strictly conform to its original intent.

This thesis contributed in this direction, supporting the synthesis of SoS based on partial descriptions. Deliverables of this thesis encompass a method, supported by tailored techniques and tools, for the automated analysis of a normative description of the architecture of a SoS. In particular, this method takes advantage of constraint solving to perform an exhaustive, yet limited search for concrete architectures that adhere to such an abstract description. Thereby, it can be used to automatically analyze a comprehensive set of families of architectural configurations that can be realized at run-time. The remaining of this chapter is organized as follows. First, Section 7.1 revisits the main contributions of the thesis. Then, Section 7.2 discusses limitations of the method and techniques, detailing how these limitations could be overcome in future research. Finally, Section 7.3 presents possible extensions to this thesis that can further the contributions of this research.

7.1 Revisiting the Thesis Contributions

This thesis makes the following contributions:

Definition of a theory for software architectures of SoS: we elaborated a conceptual model, named TASoS, for the elements framed in an abstract description of the SoS architecture. This conceptual model, presented in Chapter 4, maps most of the architectural elements of SosADL, a formal language presented in Section 2.3.3. In particular, this conceptual model supports the specification of coalitions, intentional bindings, and explicit configurations of SoS architectures in terms of constraints, i.e. Boolean expressions, that can be automatically analyzed by constraint solver tools. As a result, this formal model can be used to verify the correctness and completeness of a partial description of the SoS as well as obtain instances of this model for which all properties have been satisfied.

Definition of a method for the synthesis of concrete architectures: we elaborated a method, named Ark, to support the architectural synthesis of SoS based on abstract descriptions (presented in Chapter 3). This method details activities and artifacts that are required for realizing concrete architectures that comply with an abstract description. In particular, it comprises the specialization of the aforementioned conceptual model for the SoS under analysis. By incorporating constraint solving, this method will perform an exhaustive investigation of the solution space in order to produce a concrete architecture for which all constraints of the abstract architecture are satisfied. The method enables to verify the correctness and completeness of policies that constrain dynamic connectivity in abstract architectures.

Development of tool support: to facilitate the execution of the activities required by Ark, we developed a tool, named SoSy, that automates the main model transformations encompassed in the method (presented in Chapter 5). This tool is provided as part of a larger tool set that is tailored for the design and analysis of SoS in SosADL. In particular, the tool enables to automatically create an instance of the aforementioned conceptual model for the particular SoS under analysis and generate concrete architectures for this SoS based on solutions that were returned by the constraint solver. Thus, the tool supports the creation of intermediary and final artifacts required by Ark for the synthesis of concrete architectures that are correct-by-construction.

Experimental evaluation of the method: we evaluated the effectiveness of Ark and its accompanying tool, reported in Chapter 6. This quasi-experiment investigated the efficiency of Ark for checking the feasibility of concrete architectures for a urban river monitoring SoS described in SosADL considering different analyses scope. Results of this investigation indicate that there is a minimum scope for two of the four signatures analyzed in the

scope range. The tool can also improve the productivity of architects by providing a quick feedback for the feasibility of the architecture

The main advantage of the method and its accompanying tool is the automation of feasibility checks during the design of SoS abstract architectures. In other words, it allows to determine the existence of any concrete architecture that complies with a partial description without requiring an explicit description of its architectural configuration. Furthermore, the provided tool automates the transformations among abstract architectures, conceptual models, and concrete architectures, which can possibly reduce learning requirements for the adoption of this new technique. Thereby, the achievements of this thesis can contribute to the areas of Software Architecture and SoS, advancing the state of the art on architectural design and analysis of SoS based on formal methods.

7.2 Limitations and Future Work

This section describes limitations of this thesis and how these can be addressed in future research.

Expand the experiment: the evaluation of the effectiveness of Ark and its tool was based on one object case and one subject. We plan to expand this evaluation considering more object case. In particular, the experiment can consider several abstract architectures for each object, so as to obtain more data for the productivity of the method. Moreover, the experiment can investigate the impact of a larger analysis scope to memory consumption.

Overcome limitations in the experiment: the evaluation reported in Chapter 6 was unable to explore where the analyzed scope is unable to find a solution within the assigned time frame. We plan to repeat the previous experiment, considering a larger portion of the solution space. This new evaluation can also expand the current measure for productivity, taking into account generated source code of all possible solutions found by the constraint solver within the analyzed scope.

Evaluate Ark in a case study: the experiment reported in this thesis focuses on the perspective of one subject that executes generated Java classes with different analyses scope. In future work, we plan to perform a case study to investigate the perspective of multiple subjects that use Ark as part of their regular activities for architecting SoS in SosADL. This study can compare, for instance, how much time and effort incurs from revising an abstract architecture with and without the support of Ark and its accompanying tool.

Evaluate concrete architectures: although Ark contains a step for the evaluation of generated architecture for SoS, this activity is out of the scope of the experiment reported in Chapter 6.

In future work, we intend to individually assess alternative concrete architectures realized with the support of SoSy.

7.3 Possible Extensions

Several opportunities of research emerge to further the achievements of this thesis. Following, three of these opportunities are described.

Improve the description of bindings in SosADL: as SoSy automatically generates an instance of TASoS for a SosADL model, it translates intentional bindings of an abstract architecture as a set of constraints. Nonetheless, the way in which constraints can be nested in SosADL may impact the performance of the constraint solver. Therefore, a first extension to this thesis encompasses the investigation of how bindings can be described in SosADL to improve the efficiency of the constraint solver.

Integrate SoSy with an approach for SoS simulation: the outcome generated by the SoSy tool can be used as input to an approach that simulates the behavior of coalitions formed at run-time. Hence, a second extension to this thesis concerns the integration of SoSy to such an approach. For instance, SoSy can be modified to customize generated concrete architectures, e.g. passing specific values to be evaluated in a simulation.

Define concrete architectures as partial instances: the theory for SoS presented in this thesis is based on abstract architecture descriptions. To check the feasibility of concrete architectures requires a new theory that accepts the description of partial instances, in which part of the solution is provided as input to the constraint solver. Hence, future research focused on resolving partial instances of a concrete architecture can complement the Ark method by enabling to verify the feasibility of concrete architectures as well.

References

ALLEN, R.; VESTAL, S.; CORNHILL, D.; LEWIS, B. Using an architecture description language for quantitative analysis of real-time systems. In: **3rd International Workshop on Software and Performance (WOSP' 2002)**. Rome, Italy: ACM New York, NY, USA, 2002. p. 203–210. Cited on page 1.

ALLEN, R. J. **A Formal Approach to Software Architecture**. Phd Thesis (PhD Thesis) — Carnegie Mellon University, 1997. Cited 4 times on pages 1, 5, 11, and 25.

AVGERIOU, P.; STAL, M.; HILLIARD, R. Architecture Sustainability. **IEEE Software**, p. 41–44, 2013. Cited 2 times on pages 1 and 4.

BABAR, M. A. Supporting the Software Architecture Process with Knowledge Management. In: BABAR, M. A.; DINGSØYR, T.; LAGO, P.; VLIET, H. van (Ed.). **Software Architecture Knowledge Management**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 69–86. ISBN 978-3-642-02374-3. Cited 2 times on pages 2 and 12.

BALASUBRAMANIAN, K.; GOKHALE, A.; KARSAI, G.; SZTIPANOVITS, J.; NEEMA., S. Developing applications using model-driven design environments. **Computer**, v. 39, n. 2, p. 33–40, Feb. 2006. Cited on page 23.

BALDWIN, W. C.; SAUSER, B. Modeling the characteristics of system of system. In: **IEEE International Conference on System of Systems Engineering (SoSE)**. Albuquerque, USA: , 2009. p. 1–6. Cited on page 114.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. : Addison-Wesley, 2012. 528 p. Cited 7 times on pages 1, 2, 5, 12, 13, 25, and 41.

BATISTA, T. Challenges for SoS architecture description. In: **1st International Workshop on Software Engineering for Systems-of-Systems (SESoS)**. Montpellier, France: ACM New York, NY, USA, 2013. p. 35–37. Cited 3 times on pages 5, 6, and 29.

BETTINI, L. **Implementing Domain-Specific Languages with Xtext and Xtend**. : Packt Publishing Ltd, 2016. 426 p. Cited on page 37.

BOARDMAN, J.; SAUSER, B. System of Systems - the meaning of of. In: **1st IEEE/SMC International Conference on System of Systems Engineering (SoSE)**. Los Angeles, USA: IEEE, 2006. p. 1–6. Cited 6 times on pages 6, 13, 21, 44, 57, and 61.

BOEHM, B. Some Future Software Engineering Opportunities and Challenges. In: NANZ, S. (Ed.). **The Future of Software Engineering**. : Springer, 2011. p. 1–32. Cited 2 times on pages 3 and 6.

BOEHM, B.; BROWN, W.; BASILI, V.; TURNER, R. Spiral Acquisition of Software-Intensive Systems-of-Systems. **Crosstalk**, p. 4–9, May 2004. Cited 2 times on pages 3 and 14.

BOEHM, B.; BROWN, W.; TURNER, R. Spiral Development of Software-intensive systems of systems. In: **ICSE '05**. St. Louis, USA: , 2005. p. 706–707. Cited on page 4.

BOWEN, J.; BUTLER, R.; DILL, D.; GLASS, R.; GRIES, D.; HALL, A. An invitation to formal methods. **IEEE Computer**, v. 29, n. 4, p. 16–30, Apr. 1996. ISSN 0018-9162. Cited on page 17.

BRAMBILLA, M.; CABOT, J.; WIMMER, M. **Model-Driven Software Engineering in Practice**. 1st. ed. : Morgan & Claypool Publishers, 2012. ISBN 1608458822, 9781608458820. Cited 3 times on pages 21, 23, and 24.

BROY, M. Seamless Method- and Model-based Software and Systems Engineering. In: NANZ, S. (Ed.). **The Future of Software Engineering**. : Springer Nature, 2010. p. 33–47. Cited 2 times on pages 7 and 23.

BROY, M.; CENGARLE, M. V.; GEISBERGER, E. Cyber-Physical Systems: Imminent Challenges. In: **17th Monterey Workshop**. Oxford, UK: , 2012. p. 1–28 (LNCS. 7539). Cited on page 3.

BRYANS, J.; PAYNE, R.; HOLT, J.; PERRY, S. Semi-formal and formal interface specification for system of systems architecture. In: **6th IEEE International Systems Conference (SysCon' 2013)**. 2013. p. 612–619. Cited on page 28.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. **Pattern-oriented Software Architecture: A System of Patterns**. : John Wiley & Sons, 1996. Cited 2 times on pages 1 and 13.

M. L. Butterfield, H. F. Krikorian, A. D. Shivananda and J. A. Gula. **Architecture model developing method for system-of-system**. 2009. US2009018806-A1; US7979247-B2. Cited on page 39.

CHATTOPADHYAY, D.; ROSS, A. M.; RHODES, D. H. A Framework for Tradespace Exploration of Systems of Systems. In: **Conference on Systems Engineering Research (CSER)**. Los Angeles, USA: , 2008. p. 1–13. Cited on page 7.

CHEN, L.; BABAR, M. A.; NUSEIBEH, B. Characterizing Architecturally Significant Requirements. **IEEE Software**, Institute of Electrical and Electronics Engineers (IEEE), v. 30, n. 2, p. 38–45, Mar. 2013. Cited 2 times on pages 12 and 13.

CLARKE, L. A.; ROSENBLUM, D. S. A historical perspective on runtime assertion checking in software development. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 31, n. 3, p. 25–37, May 2006. ISSN 0163-5948. Available: <http://doi.acm.org/10.1145/1127878.1127900>. Cited on page 17.

CLEMENTS, P. A survey of architecture description languages. In: **8th Internation Workshop on Software Specification and Design**. Germany: , 1996. p. 1–10. Cited 7 times on pages 1, 2, 11, 14, 24, 25, and 26.

____. **Active Reviews for Intermediate Designs**. 2000. Available: <http://resources.sei.cmu.edu/asset_files/TechnicalNote/2000_004_001_13685.pdf>. Cited on page 41.

CLEMENTS, P.; BACHMANN, F.; BASS, L.; GARLAN, D.; IVERS, J.; LITTLE, R.; MERSON, P.; NORD, R.; STAFFORD, J. **Documenting Software Architectures: Views and Beyond**. 2. ed. : Addison-Wesley, 2011. 512 p. Cited 9 times on pages 2, 11, 14, 26, 37, 38, 39, 40, and 41.

COLEMAN, J. W.; MALMOS, A. K.; LARSEN, P. G.; PELESKA, J.; HAINS, R.; ANDREWS, Z.; PAYNE, R.; FOSTER, S.; MIYAZAWA, A.; BERTOLINIK, C.; DIDIER, A. COMPASS tool vision for a system of systems collaborative development environment. In: **7th International Conference on System of Systems Engineering (SoSE)**. Genoa, Italy: IEEE, 2012. p. 451–456. Cited on page 33.

COOK, S. A. The Complexity of Theorem Proving Procedures. In: **3rd Annual ACM Symposium on Theory of computing (STOC)**. : ACM Press, 1971. Cited on page 18.

COOK, T. S.; DRUSINKSY, D.; SHING, M.-T. Specification, Validation and Run-time Monitoring of SOA Based System-of-Systems Temporal Behaviors. In: **2nd International Conference on System of Systems Engineering (SoSE)**. San Antonio, USA: IEEE, 2007. p. 1–6. Cited on page 27.

CZARNECKI, K.; HELSEN, S. Feature-based survey of model transformation approaches. **IBM Systems Journal**, v. 25, n. 3, p. 621–645, 2006. Cited on page 22.

DAGLI, C. H.; KILICAY-ERGIN, N. System of systems architecting. In: JAMSHIDI, M. (Ed.). **System of Systems Engineering**. : John Wiley & Sons, Inc., 2009. p. 77–100. Cited 4 times on pages 4, 21, 23, and 28.

DAHMANN, J.; REBOVICH, G.; LANE, J. A.; LOWRY, R. System Engineering Artifacts for SoS. **IEEE Aerospace and Electronic Systems Magazine**, Institute of Electrical and Electronics Engineers (IEEE), v. 26, n. 1, p. 22–28, Jan. 2011. Cited 3 times on pages 14, 20, and 43.

DAHMANN, J.; REBOVICH, G.; LOWRY, R.; LANE, J. A.; BALDWIN, K. An Implementers' View of Systems for Systems of Systems. In: **IEEE International Systems Conference (SysCon)**. 2011. p. 212–217. Cited 2 times on pages 20 and 21.

DEGROSSI, L. C.; ALBUQUERQUE, J. P.; FAVA, M. C.; MENDIONDO, E. M. Flood Citizen Observatory: A crowdsourcing-based approach for flood risk management in Brazil. In: **26th International Conference on Software Engineering and Knowledge Engineering (SEKE)**. Vancouver, Canada: Knowledge Systems Institute Graduate School, 2014. p. 570–575. Cited on page 94.

DOD. **DoD Architecture Framework v.2.02**. 2010. [*On-line*], *World Wide Web*. Available: <http://dodcio.defense.gov/Library/DoD-Architecture-Framework/>. Cited on page 38.

DOGAN, H.; NCUBE, C.; LIM, S. L.; HENSHAW, M.; SIEMIENIUCH, C.; SINCLAIR, M.; BAROT, V.; HENSON, S.; JAMSHIDI, M.; DELAURENTIS, D. Economic and Societal Significance of the Systems of Systems Research Agenda. In: **SMC '13**. Manchester, UK: , 2013. p. 1715–1720. Cited 2 times on pages 5 and 117.

FARENHORST, R.; BOER, R. C. de. Knowledge Magament in Software Architecture: State of the Art. In: BABAR, M. A.; DINGSØYR, T.; LAGO, P.; VLIET, H. van (Ed.). **Software Architecture Knowledge Management Theory and Practice**. : Springer, 2009. p. 21–38. Cited 2 times on pages 2 and 40.

FEILER, P. H.; GLUCH, D. P.; HUDAK, J. J. **The Architecture Analysis & Design Language (AADL): An Introduction**. 2006. Available: <http://www.sei.cmu.edu/reports/06tn011.pdf>. Cited on page 26.

FERRARI, A.; MANGERUCA, L.; FERRANTE, O.; MIGNOGNA, A. DesyreML: A SysML Profile for Heterogeneous Embedded System. In: **Embedded Real Time Software Systems (ERTS)**. Toulose, France: , 2012. p. 1–12. Cited on page 27.

FITZGERALD, J.; BRYANS, J.; PAYNE, R. A formal model-based approach to engineering systems-of-systems. In: **Collaborative Networks in the Internet of Services**. : Springer, 2012, (IFIP Advances in Information and Communication Technology, v. 380). p. 53–62. Cited on page 5.

FITZGERALD, J.; LARSEN, P. G.; MUKHERJEE, P.; PLAT, N.; VERHOEF, M. **Validated Designs For Object-oriented Systems**. Santa Clara, CA, USA: Springer-Verlag TELOS, 2005. Cited on page 27.

FRANCE, R.; BIEMAN, J. Multi-view software evolution: a UML-based framework for evolving object-oriented software. In: **IEEE International Conference on Software Maintenance (ICSM)**. Florence, Italy: , 2001. p. 386–395. Cited on page 22.

FRANCE, R.; RUMPE, B. Model-driven Development of Complex Software: A Research Roadmap. In: **Workshop on the Future of Software Engineering (FOSE) at International Conference on Software Engineering (ICSE)**. Minneapolis, MN, United States: IEEE Computer Society, 2007. p. 37–54. Cited 2 times on pages 22 and 23.

FRIEDENTHAL, S.; MOORE, A.; STEINER, R. **A practical guide to SysML: The Systems Modeling Language**. 3. ed. USA: Morgan Kaufmann, 2015. Cited on page 33.

GAGLIARDI, M.; BERGEY, J.; WOOD, B. **System of Systems (SoS) Architecture Centric Acquisition**. 2010. [*On-line*], *World Wide Web*. Available: <https://resources.sei.cmu.edu/asset_files/Presentation/2010_017_001_53032.pdf>. Cited 2 times on pages 14 and 20.

GARLAN, D.; PERRY, D. Introduction to the special issue on software architecture. **IEEE Transactions on Software Engineering**, v. 21, n. 4, p. 269–274, Apr. 1995. Cited on page 13.

GARLAN, D.; SHAW, M. An Introduction to Software Architecture. In: AMBRIOLA, V.; TORTORA, G. (Ed.). **Advances in Software Engineering and Knowledge Engineering**. : World Scientific Publishing Company, 1993. v. 1. Cited 5 times on pages 1, 11, 12, 13, and 14.

GIANNI, D.; LINDMAN, N.; FUCHS, J.; SUZIC, R. Introducing the european space agency architectural framework for space-based systems of systems engineering. In: **2nd International Conference on Complex Systems Design and Management (CSDM)**. Paris, France: , 2011. p. 335–346. Cited on page 39.

GONÇALVES, 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)**. San Diego, USA: IEEE, 2014. p. 1605–1610. Cited 2 times on pages 13 and 113.

GONÇALVES, M. B.; OQUENDO, F.; NAKAGAWA, E. Y. A Meta-Process to Construct Software Architectures for System of Systems. In: **30th ACM Symposium on Applied Computing (SAC)**. Salamanca, ES: , 2015. p. 1411–1416. Cited on page 43.

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: **European Conference on Software Architecture Workshops**. Vienna, Austria: , 2014. p. 1–8. Cited on page 23.

GUESSI, M.; CAVALCANTE, E.; OLIVEIRA, L. B. R. Characterizing Architecture Description Languages for Software-Intensive Systems-of-Systems. In: **3rd International Workshop on Software Engineering for Systems-of-Systems (SESoS) at 37th International Conference on Software Enginnering (ICSE)**. Florence, Italy: , 2015. p. 12–18. Cited 10 times on pages 6, 29, 30, 31, 32, 34, 35, 51, 113, and 117.

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: **Proceedings of the 30th ACM/SIGAPP Symposium on Applied Computing**. New York, NY, USA: , 2015. To appear. Cited 4 times on pages 5, 6, 27, and 117.

HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. Search-based software engineering: Trends, techniques and applications. **ACM Computing Surveys**, v. 45, n. 1, p. 11:1–11:61, 2012. Cited on page 18.

HATA, Y.; KAMOZAKI, Y.; SAWAYAMA, T.; TANIGUCHI, K.; NAKAJIMA, H. A heart pulse monitoring system by air pressure and ultrasonic sensor systems. In: **2nd International System of Systems Engineering Conference (SoSE' 2007)**. 2007. p. 1–5. Cited on page 3.

HAUSE, M. The Unified Profile for DoDAF/MODAF (UPDM) enabling systems of systems on many levels. In: **3rd IEEE International Systems Conference (SysCon)**. San Diego, USA: IEEE, 2010. p. 426–431. Cited on page 28.

HAVERKORT, B. R. The Dependable Systems-of-Systems Design Challenge. **Computer**, p. 62–65, Sep./Oct. 2013. Cited 2 times on pages 3 and 4.

HEESCH, U. van; AVGERIOU, P.; HILLIARD, R. A documentation framework for architecture decisions. **Journal of Systems and Software**, Elsevier BV, v. 85, n. 4, p. 795–820, Apr. 2012. Cited on page 37.

HENRIE, M.; DELANEY, E. Towards a common system of systems vocabulary. In: **IEEE International Conference on Systems, Man, and Cybernetics (SMC)**. Hawaii, USA: , 2005. v. 3, p. 2732–2737. Cited on page 28.

HEYMAN, T.; SCANDARIATO, R.; JOOSEN, W. Security in context: analysis and refinement of software architectures. In: **34th Annual IEEE Conference on Computers, Software and Applications (COMPSAC)**. Seoul, South Korea: , 2010. p. 161–170. Cited on page 110.

HILLIARD, R.; RICE, T. Expressiveness in Architecture Description Languages. In: **3rd international workshop on Software architecture (ISAW)**. Orlando, Florida: Association for Computing Machinery (ACM), 1998. p. 65–68. Cited 3 times on pages 5, 25, and 29.

HOFMEISTER, C.; KRUCHTEN, P.; NORD, R. L.; OBBINK, H.; RAN, A.; AMERICA, P. A general model of software architecture design from five industrial approaches. **Journal of Systems and Software**, v. 80, p. 106–126, 2007. Cited 2 times on pages 19 and 44.

HOLLON, H.; DAGLI, C. The US Ballistic Missile Defense System: A case study in architecting systems-of-systems. In: **INCOSE International Symposium**. 2007. v. 17, n. 1, p. 274–281. Cited 2 times on pages 3 and 20.

HORRIDGE, M. **A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools**. 2011. Available: <http://mowl-power.cs.man.ac.uk/protegeowltutorial/resources/ProtegeOWLTutorialP4_v1_3.pdf>. Cited on page 27.

HUGHES, D.; THOELEN, K.; HORRÉ, W.; MATTHYS, N.; Del Cid, J.; MICHIELS, S.; HUYGENS, C.; JOOSEN, W. LooCI: A loosely-coupled component infrastructure for networked embedded systems. In: **7th International Conference on Advances in Mobile Computing and Multimedia**. Kuala Lumpur, Malaysia: ACM, 2009. p. 195–203. Cited on page 94.

HUGHES, D.; UEYAMA, J.; MENDIONDO, E.; MATTHYS, N.; HORRÉ, W.; MICHIELS, S.; HUYGENS, C.; JOOSEN, W.; MAN, K. L.; GUAN, S.-U. A middleware platform to support river monitoring using wireless sensor networks. **Journal of the Brazilian Computer Society**, v. 17, n. 2, p. 85–102, June 2011. Cited on page 94.

IACOBUCCI, J.; MAVRIS, D. A method for the generation and evaluation of architecture alternatives on the cloud. In: **6th International Conference on System of Systems Engineering (SoSE)**. Albuquerque, USA: IEEE, 2011. p. 137–142. Cited 2 times on pages 1 and 27.

ISO/IEC 25010:2011. **ISO/IEC Systems and software engineering — systems and software quality requirements and evaluation (SQuaRE) — system and software quality models**. 2011. Cited on page 17.

ISO/IEC/IEEE 15288. **ISO/IEC/IEEE 15288, Systems and software engineering — System life cycle processes**. 2015. Cited on page 13.

ISO/IEC/IEEE 42010. **ISO/IEC/IEEE 42010:2011 International Standard for Systems and Software Engineering – Architectural description**. 2011. Cited 17 times on pages 17, 1, 2, 11, 12, 13, 14, 15, 16, 18, 24, 25, 40, 44, 45, 60, and 113.

ISSARNY, V.; BENNACEUR, A. Composing Distributed Systems: Overcoming the Interoperability Challenge. In: E., G.; R., H.; S., B. F.; M., B. M. (Ed.). **Formal Methods for Components and Objects (FMCO)**. Bertinoro, Italy: Springer Berlin Heidelberg, 2013. p. 168–196 (LNCS v. 7866). Cited on page 5.

IVERS, J.; CLEMENTS, P.; GARLAN, D.; NORD, R.; SCHMERL, B.; SILVA, J. R. O. **Documenting Component and Connector Views with UML 2.0**. 2004. Available: <http://www.sei.cmu.edu/reports/04tr008.pdf>. Cited on page 26.

JACKSON, D. Alloy: A Lightweight Object Modelling Notation. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, Association for Computing Machinery (ACM), v. 11, n. 2, p. 256–290, apr 2002. Cited on page 58.

____. Dependable Software by Design. **Scientific American**, p. 69–75, 2006. Cited on page 5.

____. **Software Abstractions**. Rev. : MIT University Press Group Ltd, 2012. ISBN 0262017156. Cited 6 times on pages 17, 44, 49, 57, 58, and 59.

____. Towards a Theory of Conceptual Design for Software. In: **ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)**. Pittsburgh,USA: Association for Computing Machinery (ACM), 2015. p. 282–296. Cited on page 57.

JANSEN, A.; BOSCH, J. Software Architecture as a Set of Architectural Design Decisions. In: **5th Working IEEE/IFIP Conference on Software Architecture (WICSA)**. Pittsburgh, USA: IEEE, 2005. p. 1–10. Cited 2 times on pages 12 and 40.

JASPAN, C.; KEELING, M.; MACCHERONE, L.; ZENAROSA, G. L.; SHAW, M. Software Mythbusters Explore Formal Methods. **IEEE Software**, Institute of Electrical and Electronics Engineers (IEEE), v. 26, n. 6, p. 60–63, Nov. 2009. Cited 3 times on pages 16, 17, and 18.

KENLEY, C. R.; DANNENHOFFER, T. M.; WOOD, P. C.; DELAURENTIS, D. A. Synthesizing and Specifying Architectures for System of Systems. **INCOSE International Symposium**, Wiley-Blackwell, v. 24, n. 1, p. 94–107, Jul. 2014. Cited 3 times on pages 53, 113, and 114.

KEZNIKI, J.; BUREŠ, T.; PLÁŠIL, F.; HNĚTYNKA, P. Automated resolution of connector architectures using constraint solving (ARCAS method). **Software Systems Modeling**, v. 13, p. 843–872, 2014. Cited on page 110.

KHLIF, I.; KACEM, M. H.; KACEM, A. H.; DRIRA, K. A multi-scale modelling perspective for SoS architectures. In: **ECSAW' 2014**. Vienna, Austria: , 2014. p. 1–5. Cited on page 114.

KLEIN, J.; VLIET, H. van. A systematic review of system-of-systems architecture research. In: **9th International ACM Sigsoft Conference on the Quality of Software Architectures (QoSA)**. Vancouver, Canada: ACM Press, 2013. p. 13–21. Cited 2 times on pages 5 and 6.

KNIGHT, J. Safety critical systems: challenges and directions. In: **24th International Conference on Software Engineering (ICSE)**. Orlando, Florida, United States: Institute of Electrical and Electronics Engineers (IEEE), 2002. p. 547–550. Cited on page 4.

KOONTZ, R. J.; NORD, R. L. Architecting for Sustainable Software Delivery. **CrossTalk**, p. 14–19, May/Jun. 2012. ISSN 2160-1593. Cited 2 times on pages 19 and 21.

KRAMER, J. Is abstraction the key to computing? **Communications of the ACM**, v. 50, n. 4, p. 36–42, 2007. Cited on page 21.

KRAMER, J.; MAGEE, J. A Rigorous Architectural Approach to Adaptive Software Engineering. **Journal of Computer Science and Technology**, v. 24, n. 2, p. 183–188, 2009. Cited on page 8.

KRUCHTEN, P. Architectural Blueprints - The 4+1 View Model of Software Architecture. **IEEE Software**, IEEE Software, v. 12, n. 6, p. 42–50, 1995. Cited 6 times on pages 1, 2, 37, 39, 40, and 113.

____. **The Rational Unified Process an Introduction**. 3. ed. : Addison-Wesley Professional, 2003. ISBN 978-0321197702. Cited on page 19.

____. Documentation of Software Architecture from a Knowledge Management Perspective – Design Representation. In: BABAR, M. A.; DINGSØYR, T.; LAGO, P.; VLIET, H. van (Ed.). **Software Architecture Knowledge Management Theory and Practice**. : Springer, 2009. p. 39–57. Cited 5 times on pages 1, 2, 11, 14, and 40.

KRUCHTEN, P.; CAPILLA, R.; DUEÑAS, J. C. The Decision View's Role in Software Architecture Process. **IEEE Software**, Institute of Electrical and Electronics Engineers (IEEE), v. 26, n. 2, p. 36–42, Mar./Apr. 2009. Cited on page 40.

KRUCHTEN, P.; OBBINK, H.; STAFFORD, J. The past, present, and future for software architecture. **IEEE Software**, Los Alamitos, CA, USA, v. 23, n. 2, p. 22–30, 2006. ISSN 0740-7459. Cited 3 times on pages 1, 12, and 14.

KRÜGER, I.; PRENNINGER, W.; SANDNER, R.; BROY, M. From scenarios to hierarchical broadcasting software architectures using uml-rt. **International Journal of Software Engineering and Knowledge Engineering**, v. 12, n. 2, p. 155–174, 2002. Cited on page 1.

LAGO, P.; MALAVOLTA, I.; MUCCINI, H.; PELLICCIONE, P.; TANG, A. The road ahead for architectural languages. **IEEE Software**, v. 32, n. 1, p. 98–105, Jan./Feb. 2015. Cited 4 times on pages 3, 8, 24, and 26.

LANE, J. A.; BOHN, T. Using SysML Modeling To Understand and Evolve Systems of Systems. **Systems Engineering**, p. 87–98, 2012. Cited on page 117.

LEDECZI, A.; KARSAI, G.; BAPTY, T. Synthesis of self-adaptive software. In: **IEEE Aerospace Conference Proceedings (AERO 2000)**. Big Sky Montana, USA: , 2000. v. 4, p. 501–507. ISSN 1095-323X. Cited on page 1.

LEMOS, R.; GIESE, H.; MÜLLER, H. A.; SHAW, M.; ANDERSSON, J.; LITOIU, M.; SCHMERL, B. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In: **Software Engineering for Self-Adaptive Systems II**. 2013. p. 1–32 (LNCS 7475). Cited on page 6.

LIGGESMEYER, P.; TRAPP, M. Trends in Embedded Software Engineering. **IEEE Software**, Institute of Electrical and Electronics Engineers (IEEE), v. 26, n. 3, p. 19–25, May 2009. Cited on page 4.

LOIRET, F.; ROUVOY, R.; SEINTURIER, L.; MERLE, P. Software engineering of component-based systems-of-systems: A reference framework. In: **Component-Based Software Engineering at the Federated Events on Software Architecture (CBSE/CompArch' 2011)**. Boulder, USA: , 2011. p. 61–65. Cited on page 21.

LUCKHAM, D. C. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. In: **Partial Order Methods Workshop (POMIV)**. New Jersey, USA: , 1996. (DIMACS, v. 29), p. 1–25. Cited on page 25.

MAIER, M. W. Architecting principles for systems-of-systems. **Systems Engineering**, v. 1, n. 4, p. 267–284, 1998. Cited 5 times on pages 3, 20, 44, 53, and 94.

MALAVOLTA, I.; LAGO, P.; MUCCINI, H.; PELLICCIONE, P.; TANG, A. What Industry Needs from Architectural Languages: A Survey. **IEEE Transactions on Software Engineering**, v. 39, n. 6, p. 869–891, 2013. Cited on page 117.

MANDRIOLI, D. On the heroism of really pursuing formal methods. In: **IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering (FormaliSE) at 37th International Conference on Software Engineering (ICSE)**. Florence, IT: IEEE, 2015. p. 1–5. Cited 2 times on pages 5 and 17.

MARWEDEL, P. **Embedded System Design**. 2. ed. Springer-Verlag GmbH, 2010. ISBN 9400702566. Available: <http://www.ebook.de/de/product/13051624/peter_marwedel_embedded_system_design.html>. Cited on page 3.

MEDVIDOVIC, N.; TAYLOR, R. N. A classification and comparison framework for software architecture description languages. **IEEE Transactions on Software Engineering**, v. 26, n. 1, p. 70–93, Jan. 2000. Cited 5 times on pages 2, 24, 25, 29, and 30.

MELLOR, S. J.; BALCER, M. J. **Executable UML**. Pearson Education (US), 2002. ISBN 0201748045. Available: <http://www.ebook.de/de/product/3585980/stephen_j_mellor_marc_j_balcer_executable_uml.html>. Cited on page 16.

MENS, T.; MAGEE, J.; RUMPE, B. Evolving Software Architecture Descriptions of Critical Systems. **Computer**, v. 43, p. 42–48, Maio 2010. Cited 3 times on pages 1, 14, and 26.

MILICEVIC, A.; NEAR, J. P.; KANG, E.; JACKSON, D. Alloy*: A General-Purpose Higher-Order Relational Constraint Solver (ICSE). In: **37th IEEE International Conference on Software Engineering**. Florence, Italy: , 2015. p. 609–619. Cited on page 17.

MOSCHOGLOU, G.; EVELEIGH, T.; HOLZER, T.; SARKANI, S. A semantic mediation framework for architecting federated ubiquitous systems. In: **7th International Conference on System of Systems Engineering (SoSE)**. Genova, Italy: , 2012. p. 485–490. Cited on page 28.

MOUR, A.; KENLEY, C. R.; DAVENDRALINGAM, N.; DELAURENTIS, D. Agent-based Modeling for Systems of Systems. In: **23rd International Symposium of the International Council of Systems Engineering (INCOSE)**. 2013. v. 23, p. 973–987. Cited on page 113.

NAKAGAWA, E. Y.; GONÇALVES; GUESSI, M.; OLIVEIRA, L. B. R.; OQUENDO, F. The state of the art and future perspectives in systems of systems software architectures. In: **1st International Workshop on Software Engineering for Systems-of-Systems (SESoS)**. Montpellier, France: , 2013. p. 13–20. Cited 4 times on pages 4, 5, 6, and 8.

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 Comput. Surv.**, v. 48, n. 2, p. 1–41, 2015. Cited 9 times on pages 3, 4, 5, 6, 12, 29, 49, 57, and 93.

NIKOLIC, I.; DIJKEMA, G. Framework for Understanding and Shaping Systems of Systems The case of industry and infrastructure development in seaport regions. In: **2nd International Conference on System of Systems Engineering (SoSE)**. San Antonio, USA: , 2007. p. 1–6. Cited on page 28.

NORD, R. L.; OZKAYA, I.; KRUCHTEN, P. Agile in Distress: Architecture to the Rescue. In: T., D.; N.B., M.; R., T.; S., C.; C., G.; K., P. (Ed.). **Agile Methods. Large-Scale Development, Refactoring, Testing, and Estimation. XP 2014**. : Springer International Publishing, 2014. (Lecture Notes in Business Information Processing, v. 199), p. 43–57. Cited on page 19.

OBBINK, H.; KRUCHTEN, P.; KOZACZYNSKI, W.; HILLIARD, R.; RAN, A.; POSTEMA, H.; LUTZ, D.; KAZMAN, R.; TRACZ, W.; KAHANE, E. **Software Architecture Review and Assessment (SARA) Report**. 2002. Available: <https://pkruchten.files.wordpress.com/2011/09/sarav1.pdf>. Cited 4 times on pages 1, 13, 19, and 49.

Office of the Deputy Under Secretary of Defense for Acquisition and Technology, Systems and Software Engineering. **Systems Engineering Guide for Systems of Systems** . Washington, DC, United States: , 2008. [*On-line*], *World Wide Web*. Available: <http://www.acq.osd.mil/se/docs/SE-Guide-for-SoS.pdf>. Cited 2 times on pages 20 and 21.

OMG. **Software and Systems Process Engineering Meta-Model Specification v2.0**. USA, 2008. [On-line]. <http://www.omg.org/spec/SPEM/2.0/>. Cited on page 78.

____. **Unified Modeling Language v2.4.1**. USA, 2011. [On-line]. <http://www.omg.org/spec/UML/2.4.1/>. Cited 2 times on pages 5 and 26.

____. **Systems Modeling Language v1.3**. 2012. [On-line]. <http://www.omgsysml.org/>. Cited 2 times on pages 5 and 26.

Open Group. **The Open Group's Architecture Framework (TOGAF)**. 2009. [*On-line*], *World Wide Web*. Available: <http://www.togaf.info/>. Cited on page 2.

OQUENDO, F. π-ADL: An Architecture Description Language based on the Higher Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures. **ACM Software Engineering Notes**, v. 29, n. 3, p. 15–28, 2004. Cited 4 times on pages 1, 5, 25, and 36.

____. π-Method: a model-driven formal method for architecture-centric software engineering. **ACM SIGSOFT Software Engineering Notes**, Association for Computing Machinery (ACM), v. 31, n. 3, p. 1–13, may 2006. Cited on page 27.

____. Case Study on Formally Describing the Architecture of a Software-intensive System-of-Systems with SosADL. In: **IEEE International Conference on Systems, Man, and Cybernetics (SMC)**. Budapest, Hungary: Institute of Electrical and Electronics Engineers (IEEE), 2016. p. 2260–2266. Cited on page 167.

____. Formally describing the architectural behavior of software-intensive systems-of-systems with SosADL. In: **21st International Conference on Engineering of Complex Computer Systems (ICECCS)**. : Institute of Electrical and Electronics Engineers (IEEE), 2016. p. 13–22. Cited on page 167.

____. Formally describing the software architecture of systems-of-systems with SosADL. In: **11th System of Systems Engineering Conference (SoSE)**. Kongsberg, Norway: Institute of Electrical and Electronics Engineers (IEEE), 2016. p. 1–6. Cited 9 times on pages 3, 4, 5, 29, 35, 36, 49, 62, and 167.

____. π-calculus for sos: A foundation for formally describing software-intensive systems-of-systems. In: **11th System of Systems Engineering Conference (SoSE)**. Kongsberg, Norway: Institute of Electrical and Electronics Engineers (IEEE), 2016. p. 1–6. Cited 2 times on pages 36 and 167.

____. Software Architecture Challenges and Emerging Research in Software-Intensive Systems-of-Systems. In: B., T.; U., Z.; A., B. (Ed.). **10th European Conference on Software Architecture (ECSA)**. Copenhagen, Denmark: Springer, 2016. p. 3–21 (LNCS v. 9839). Cited 2 times on pages 12 and 14.

OQUENDO, F.; BUISSON, J.; LEROUX, E.; MOGUéROU, G.; QUILBEUF, J. SoS ADL for Formal Architecture Description and Analysis of Software-intensive Systems-of-Systems. Presentation at the Colloquium on Software-intensive Systems-of-Systems at ECSA. 2016. Cited 3 times on pages 37, 77, and 79.

OZKAYA, M.; KLOUKINAS, C. Are We There Yet? Analyzing Architecture Description Languages for Formal Analysis, Usability, and Realizability. In: **39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**. Santander, Spain: Institute of Electrical and Electronics Engineers (IEEE), 2013. p. 1–8. Cited 3 times on pages 5, 26, and 27.

PARNAS, D.; WEISS, D. M. Active Design Reviews: Principle and Practices. **Journal of Systems and Software**, Elsevier BV, v. 7, n. 4, p. 259–265, 12 1987. Cited on page 41.

PARNAS, D. L. Precise Documentation: The Key to Better Software. In: NANZ, S. (Ed.). **The Future of Software Engineering**. : Springer, 2010. p. 125–148. Cited on page 117.

PAYNE, R.; BRYANS, J.; FITZGERALD, J.; RIDDLE, S. Interface specification for system-of-systems architectures. In: **7th International Conference on System of Systems Engineering (SoSE' 2012)**. Genova, Italy: , 2012. p. 567–572. Cited on page 28.

PERSEIL, I.; PAUTET, L. Formal methods integration in software engineering. **Innovations in Systems and Software Engineering**, Springer Nature, v. 6, n. 1-2, p. 5–11, feb 2010. Cited on page 26.

PETKE, J. **Bridging Constraint Satisfaction and Boolean Satisfiability**. : Springer International Publishing, 2015. Cited on page 17.

RHODES, D. Evolving Systems Engineering for Innovative Product and Systems Development. In: **Massachusetts Institute of Technology (MIT) Systems Design and Management Alumni Conference**. 2004. Cited on page 4.

ROURE, D. D. Floodnet: A new flood warning system. **Ingenia**, v. 23, p. 50–51, June 2005. Cited on page 94.

ROYER, J.-C.; ARBOLEDA, H. Model-Driven Engineering. In: **Model-Driven and Software Product Line Engineering**. : Wiley-ISTE, 2012. ISBN 1848214278. Cited 2 times on pages 22 and 23.

ROZANSKI, N.; WOODS, E. **Software Systems Architecture: Working with stakeholders using viewpoints and perspectives**. : Addison-Wesley, 2005. Cited 3 times on pages 2, 12, and 37.

SAE International. **Architecture Analysis & Design Language**. 2011. [*On-line*], *World Wide Web*. Available: <http://www.aadl.info>. Cited on page 26.

SANGIORGI, D. **Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms**. Phd Thesis (PhD Thesis) — University of Edinburgh, UK, 1992. Available: <http://hdl.handle.net/1842/6569>. Cited on page 25.

SCHWABER, K.; BEEDLE, M. **Agile Software Development with SCRUM**. 1. ed. : Prentice Hall, 2001. ISBN 0130676349. Cited on page 19.

SELBERG, S. A.; AUSTIN, M. A. Toward an Evolutionary System of Systems Architecture. In: **INCOSE International Symposium**. Utrecht, The Netherlands: , 2008. v. 18, n. 1, p. 1065–1078. Cited 3 times on pages 5, 7, and 14.

SHAW, M.; GARLAN, D. **Characteristics of Higher-Level Languages for Software Architecture**. 1994. Available: <http://www.sei.cmu.edu/reports/94tr023.pdf>. Cited on page 24.

SHIBASAKI, R.; PEARLMAN, J. S. Systems of Systems Engineering of GEOSS. In: JAMSHIDI, M. (Ed.). **System of Systems Engineering: Innovations for the 21st Century**. : Wiley, 2009. Cited on page 3.

SILVA, E.; BATISTA, T.; OQUENDO, F. A mission-oriented approach for designing system-of-systems. In: **10th System of Systems Engineering Conference (SoSE)**. : IEEE, 2015. Cited 2 times on pages 6 and 35.

TORLAK, E.; DENNIS, G. Kodkod for Alloy Users. In: **First Alloy Workshop colocated with 14th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)**. Portland, Oregon, United States: , 2006. p. 100–104. Cited 2 times on pages 17 and 59.

ULIERU, M.; DOURSAT, R. Emergent engineering: a radical paradigm shift. **International Journal of Autonomous and Adaptive Communications Systems**, Inderscience Publishers, v. 4, n. 1, p. 39, 2011. Cited on page 4.

VALERDI, R.; ROSS, A. M.; RHODES, D. H. A Framework for Evolving System of Systems Engineering. **Crosstalk**, v. 20, n. 10, p. 28–30, Oct. 2007. ISSN 2160-1593. Cited 2 times on pages 7 and 14.

W3C. **Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering**. 2006. [*On-line*], *World Wide Web*. Available: <http://www.w3.org/2001/sw/BestPractices/SE/ODA/060211/>. Cited on page 27.

____. **OWL 2 Web Ontology Language Primer**. 2012. [*On-line*], *World Wide Web*. Available: <http://www.w3.org/TR/2012/REC-owl2-primer-20121211/>. Cited on page 27.

WILBER, F. R. A system of systems approach to e-enabling the commercial airline applications from an airframer's perspective. In: **Keynote presentation, IEEE System of Systems Engineering Conference (SoSE'2007)**. 2007. Cited on page 3.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in Software Engineering**. : Springer Nature, 2012. Cited on page 96.

WOLF, W. **Computers as Components: Principles of Embedded Computing System Design**. 2. ed. : Morgan Kaufmann, 2008. ISBN 0123743974. Cited on page 3.

WOODCOCK, J.; CAVALCANTI, A.; FITZGERALD, J.; LARSEN, P.; MIYAZAWA, A.; PERRY., S. Features of CML: a Formal Modelling Language for Systems of Systems. In: **7th Internation Conference on System of Systems Engineering (SoSE)**. Genova, Italy: IEEE, 2012. p. 1–6. Cited on page 27.

WOODCOCK, J.; LARSEN, P. G.; BICARREGUI, J.; FITZGERALD, J. Formal Methods: Practice and Experience. **ACM Computing Surveys**, v. 41, n. 4, p. 1–40, 2009. Cited 4 times on pages 16, 17, 18, and 79.

ZACHMAN, J. A. A framework for information system architecture. **IBM Systems Journal**, v. 26, n. 3, p. 276–292, 1987. Cited 2 times on pages 2 and 37.

ZHANG, P.; MUCCINI, H.; LI, B. A classification and comparison of model checking software architecture techniques. **Journal of Systems and Software**, v. 83, n. 5, p. 723–744, 2010. Cited 2 times on pages 5 and 18.

APPENDIX

A

Usage Guidelines of SoSy

This appendix details how one can install and use SoSy, a tool that provides automated support for the Ark method within a larger framework for the design of SoS architectures in SosADL. First, Section A.1 explains how to download and install SosADE, an integrated environment for designing and simulating SoS based on SosADL. In particular, it describes how SoSy can be used from within a SosADE instance of the Eclipse environment.

A.1 Installation of SosADE

SosADE requires Java 8 or higher installed and runs on a Windows 7 ou Linux operational systems. The tool is provided as an Eclipse Mars 2[1] plugin. The user can create a new Java project or upload an existing SosADL project in the environment that will recognize the `.sosadl` extension in files and execute the `SosADL2AlloyGenerator` on the background, hence automatically starting the transformation of SosADL models into instances of TASoS and related Java classes. The user should follow these steps to start SosADE and obtain instances of concrete coalitions that adhere to a SosADL abstract architecture:

- Download and execute the SosADE plugin for Eclipse at <https://goo.gl/RgqR9K>.
- Upload or create a new project.
- Upload or create one or more `.sosadl` files in your project. To generate an executable Java class, it is important that the project contains at least one SoS unit to translate into a TASoS instance module.
- Right click the main project folder > Build path > Configure Build Path > Add external JAR > select `alloy4.2_2015-02-22.jar` (this JAR is automatically added by SoSy in the generated folder of the project, usually named src-gen).
- Right click the generated folder > Build path > Use as source folder.

[1] Eclipse Mars, <https://eclipse.org/mars/>.

- Run the generated `.java` file in the generated folder as a Java application. The console will show ”Translation to KodKod and Solution: (ms)” after the constraint solver terminates its execution. The tool will also prompt a new window that shows the graphical representation of the solution automatically generated by the solver.
- Right click the generated folder and select the option Refresh. Any concrete instance that has been found by the constraint solver will appear as a separate file in the generated folder. In particular, generated files are named after the abstract description with which they are related.

The graphical interface of SosADE is illustrated in Figure 17 using the ”Simple-ClientServer” project as example. This figure shows that the tool created an Alloy (`.als` extension) file in the generated folder named ”src-gen” for each SosADL (`.sosadl` extension) file in the main project folder named ”src”. In addition, SoSy also automatically created a Java class for running each SoS unit in the project folder. Besides these files, the generated folder also contains a copy of the TASoS model (`tasos.als`) and a custom theme for visualizing the instance found by the solver (`minisosrun.thm`).

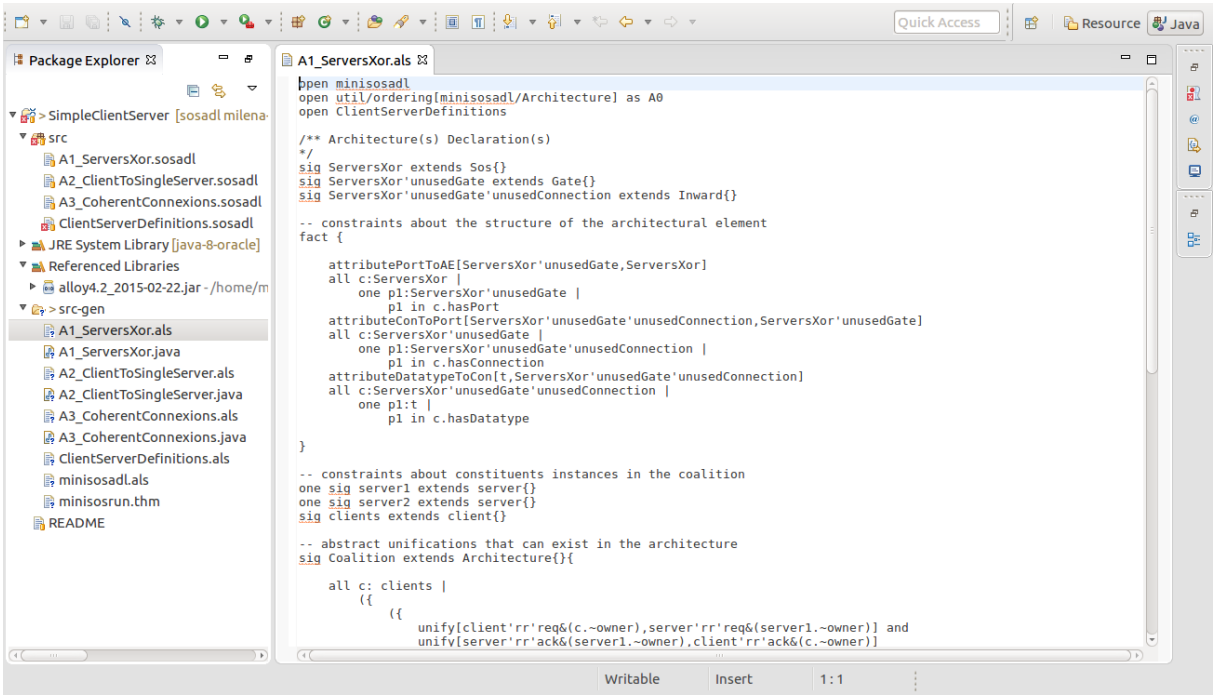


Figure 17 – User interface of SoSy

Methods of the `SosADL2AlloyGenerator`

As discussed in Chapter 5, the `SosADL2AlloyGenerator` is the main class of SoSy for translating a SosADL model into an instance of TASoS. Table 21 summarizes the list of methods encompassed by this class.

Table 21 – Methods in the `SosADL2AlloyGenerator` class

Method	Parameter	Description
doGenerate	Resource, fsa	Main method that is responsible for generating all files in the project folder.
compile	SosADL	Creates the header of an instance module file.
compile	Import	Adds a reference to an imported library in the header of an instance module file.
compile	Library	Creates the main body of a library instance module, which consists of constraints on abstract types of systems and mediators.
compile	SoS	Creates the main body of an architecture instance module. Transforms an architecture declaration into an extension to the `SoS` signature of TASoS. Also, it defines a scenario that must be checked (which lists constituents of the architecture) and the analysis scope for the execution of a constraint solver, which is given by the *runargs* global variable.
compile	EntityBlock	Defines the structure of a library instance module, which is composed of the declaration of an entity block as a list of optional datatypes and functions followed by signatures and constraints related to systems and mediators.
compile	DatatypeDecl	Transforms the declaration of a SosADL datatype as a signature in Alloy.
compile	FunctionDecl, AlloyType	Copies the declaration of a function to the description of concrete architectures. However, it is not part of an instance module file.
compile	FormalParameter	Copies the declaration of parameters passed to an entity block to the description of concrete architectures. However, it is not part of an instance module file.
compile	SystemDecl	Transforms a system declaration into an extension to the `System` signature of TASoS and calls the method that compiles gates. The behavior and assertion elements are not part of an instance module but they are copied to the description of concrete architectures. The method also adds a new type of system to the *runargs* global variable.
compile	MediatorDecl	Transforms a mediator declaration into an extension to the `Mediator` signature of TASoS and calls the method that compiles duties. The behavior, assumption, and assertion elements are not part of an instance module but they are copied to the description of concrete architectures. The method also adds a new type of mediator to the *runargs* global variable.
compile	ArchitectureDecl, AlloyLibrary	Transforms an architecture declaration into an extension to the Mediator signature of TASoS and calls the method that compiles duties. The behavior, assumption, and assertion elements are not part of an instance module but they are copied to the description of concrete architectures. The method also adds a new type of architecture to the *runargs* global variable.
compile	GateDecl, String, AlloySystem	Transforms a gate declaration into an extension of the `Gate` signature of TASoS. It also attributes connections to gates and datatypes to these connections. The assertion block is not part of an instance module but it is copied to the description of concrete architectures. It adds new type of gate to the *runargs* global variable.

Continued on next page

Table 21 – *Continued from previous page*

Method	Parameter	Description
compile	DutyDecl, AlloySystem	Transforms a duty declaration into an extension of the `Duty` signature of TASoS. It also attributes connections to duties and datatypes to these connections. The assertion and protocol elements are not part of an instance module but they are copied to the description of concrete architectures. The method also adds a new type of duty to the *runargs* global variable.
compile	Protocol	Copies the declaration of protocol statements, which are not part of an instance module file.
compile	Action	Copies the declaration of an action, which is not part of an instance module file.
compile	ProtocolAction	Copies the declaration of a protocol action, which is not part of an instance module file.
compile	AssertionDecl	Copies the declaration of an assertion, which is not part of an instance module file.
compile	IfThenElseProtocol	Copies the declaration of a conditional protocol, which is not part of an instance module file.
compile	Behavior	Copies the declaration of behavior statements, which are not part of an instance module file.
compile	ArchBehaviorDecl, String	Transforms the behavior of an architecture into a set of constraints on constituents of a coalition and unifications among them. It also clears the state of the *elems* and *sigs* global variables, which hold the set of constituents for a particular architecture instance module.
compile	Constituent	Transforms the constituents of a particular architecture behavior declaration into extensions of their recently created abstractions for the `System` signature of TASoS. It also updates the *runargs* global variable, scaling up the minimum quantity of elements allowed for that particular abstract signature.
compile	Quantify	Translates a SosADL quantifier to its equivalent quantification operator in Alloy. In particular, `forall` is translated as `all` and `exists` as `some`.
compile	Relay	Adds constraints on connections of a SoS that are internally delegated to systems or mediators that are members of the coalition.
compile	ElementInConstituent	It updates the *elems* global variable with a new constituent of the coalition.
compile	BinaryExpression	Translates a binary expression as a constraint on the Unification signature in the architecture instance module. In particular, it modifies the operator of a SosADL binary expression with its equivalent logical operator in Alloy. The logical operators `and`, `not`, and `implies` exists in both languages.
compile	UnaryExpression	Translates a unary expression as a constraint on the `Unification` signature in the architecture instance module.
compile	Unify	Transforms a binding expression into a set of constraints on the elements of the `Unification` signature of TASoS.
compile	Datatype	Transforms a declared datatype in SosADL as an extension of their corresponding abstract datatype signature of TASoS.
generateClass	ConcreteSolution	Creates a Java class in the project folder that can call the constraint solver on the new architecture instance module that was created.

TASoS

Source code 25 presents the source code for the conceptual model describing SoS architectures in Alloy that is discussed in more detail in Chapter 4.

Source code 25 – TASoS source code

```
1 module tasos
2 open util/ordering[Architecture] as A0
3 abstract sig Datatype {}
4 abstract sig IntegerType, NamedType, SequenceType, RangeType
    extends Datatype{}
5 abstract sig TupleType extends Datatype{
6   complexType: set Datatype
7 }
8 fact Datatypefact {
9   // #1 All Datatype must be related to one connection
10   all d: Datatype | some {d.~hasDatatype}
11 }
12 abstract sig Connection {
13   owner: one ArchitecturalElement, // #2 All Connections are owned
       by one AE
14   hasDatatype: one Datatype
15 }
16 fact ConnectionFact {
17   // #3 All Connections have exactly one Datatype and all
      Datatypes are related to some Connection
18   hasDatatype in Connection some -> one Datatype
19   // #4 All Connections map to exactly one AE
20   all c: Connection | c.owner = hasPort.hasConnection.c
21 }
22 abstract sig Inward, Outward extends Connection {} //There is no
    overlapping between in or out connections
23 abstract sig Port {
24   hasConnection : set Connection
25 }
26 fact PortFact {
27   // #5 Each Connection pertains to at most one Port so that we
      can add new connections to a port
28   hasConnection in Port lone -> set Connection
29 }
30 abstract sig Gate, Duty extends Port {} //There is no overlapping
    between gates and duties
31 abstract sig ArchitecturalElement {
```

```
32   hasPort: set Port
33 }
34 //There is no overlaping between Systems and Mediators
35 abstract sig System, Mediator extends ArchitecturalElement {}
36 abstract sig Sos extends ArchitecturalElement {
37   arch: Architecture some -> some Relay
38 }
39 abstract sig Relay {
40   relayCon:  Connection set -> set Connection
41 }
42 fact {
43   // Every Sos sig is related to one or more architectures and
      relay connections
44   all s: Sos | some a: Architecture, r: Relay | a -> r in s.arch
45   // Every Relay sig is related to one or more architectures
46   all r: Relay | some s: Sos, a: Architecture | a -> r in s.arch
47   // Two Relay sigs are the same if they share the same relay
      connections
48   all r1,r2: Relay | r1.relayCon = r2.relayCon implies r1=r2
49   // Relay unifications require some constituent in the
      architecture
50   all r: Relay, c1, c2: Connection | c1->c2 in r.relayCon implies
      {
51     all a: Architecture, s: Sos | a->r in s.arch implies c2.owner
      in a.contain and c1.owner=s
52   }
53   // All connections of an Sos engage in at most one relay
      unification per Relay sig
54   all c: Connection, s: Sos, r: Relay | c.owner = s and c in
      inRelay[r] implies {
55     all a: Architecture | a->r in s.arch implies {one c':
      Connection | c' in a.contain.hasPort.hasConnection and c->c' in
       r.relayCon}
56   }
57   // A connection participates in a Relay unification if it is of
      the same type
58   all c1,c2: Connection, r: Relay | c1->c2 in r.relayCon implies (
      c1+c2) in Inward or (c1+c2) in Outward
59 }
60 fact AEFact {
61   // #10 All Ports of mediators are duties
62   Mediator.hasPort in Duty
```

```
63   // #11 All Ports of constituents are gates
64   System.hasPort in Gate
65   // #12 All Ports of sos are gates
66   Sos.hasPort in Gate
67   // #12 All Ports owned by a given architectural element must
      have at least one connection.
68   all p: Port | some {p.~hasPort} implies some {p.hasConnection}
69   // # 13 All Systems must participate in some Architecture
70   all e: System | some c: Architecture | e in c.contain
71   // # 14 Any two sos are equivalent if they present the same arch
       -> relay relationships
72   all s1, s2: Sos | getArch[s1] = getArch[s2] implies s1 = s2
73 }
74 abstract sig Unification { //An Unification binds connections
     among architectural elements
75   src: one Connection,  //Outward
76   dest: one Connection //Inward
77 }
78 fact UnificationFact {
79   // #14 General properties over the domain and image of
     Unifications
80   all u: Unification {
81     u.src in Outward // #14.1 The domain of an unification is in
     the set of outward connections
82     u.dest in Inward // #14.2 The image of an unification is in
     the set of inward connections
83     u.src.owner != u.dest.owner // #14.3 An unification may only
     unify connections between different AEs
84     // #14.4 Unifications are allowed between elements of System
     and elements of Mediator exclusively
85     u.src.owner in System implies u.dest.owner in Mediator
86     u.src.owner in Mediator implies u.dest.owner in System
87     // #14.5 A connection can only engage in an unification if it
     is owned by some AE
88     some u.src.owner
89     some u.dest.owner
90   }
91   // #15 All Unifications are unique, there are no two
     unifications to the same in *and* out connections
92   // However, we can have unifications that share one of the
     connections
93   all x, y: Unification | (x.src = y.src) and (x.dest = y.dest)
```

```
    implies x=y
94  // #16 There is up to one unification per connection in any
    given topology
95  all t: Topology, c: Connection | let sPort = portsOfUnification[
    t.inTopology] | c in sPort.hasConnection implies {
96    lone u: Unification | u in t.inTopology and {c = u.src or c =
    u.dest}
97  }
98 }
99 sig Topology {
100   // This signature represents the architectural configuration
    of an SoS, i.e., the way
101 // in which its constituents and mediators are assembled
    together
102 inTopology: set Unification,
103 path: Unification set -> set Unification
104 }
105 fact TopologyFact {
106 // #17 A pair of unification may pertain to a given topology's
    path if they are different.
107 // This path relation is bidirectional.
108 all u,v: Unification, t: Topology | u -> v in t.path implies u
    != v and v -> u in t.path
109 // #18 All unifications in the relation path exist in the
    relation inTopology
110 all t: Topology | {(t.path).Unification + Unification.(t.path)}
    = t.inTopology
111 // #19 Two topologies are equal if they share the same pair of
    unifications. Also applies to empty topologies.
112 all t, t': Topology | t.path = t'.path implies t = t'
113 }
114 abstract sig Architecture {
115   //This signature contains the constituents of the SoS. Each
    set of constituents can
116 //be assembled in different ways. Therefore, each coalition can
    be related to different topologies
117 contain: set System,
118 bindings: set Unification,
119 unifiedAs: seq Topology
120 }
121 fact ArchitectureFact {
122 contain in Architecture some -> set System // #20 All systems
```

```
      must be related to at least one Architecture
123   bindings in Architecture some -> set Unification // #21 All
      Unification must be related to at least one Architecture
124   // #22 A topology may only use the unifications that exist in
      the set of possible bindings and connect the systems that
      participate in the architecture
125   all t: Topology, a: Architecture | t in (a.unifiedAs).elems
      implies t.inTopology in a.bindings and (isUnifiedTo[t.
      inTopology] & System) = a.contain
126   //Obs: This condition alone does not constraint that
      constituents form a connected topology.
127   // #23 A Topology is related to at least one architecture
128   // If we do not add this rule, it is possible to have a topology
       that is related to no architecture
129   all t: Topology | some a: Architecture | t in (a.unifiedAs).
      elems
130   // #24 An Architecture is related to at least one topology
131   all a: Architecture| {some (a.unifiedAs).elems}
132   // #25 Two coalitions should be equal if they have the same set
      of topologies
133    all a, a': Architecture| (a.unifiedAs).elems = (a'.unifiedAs).
      elems implies a=a'
134   // #26 The relations inTopology and path are empty if no system
      participates in the coalition
135   all t: Topology, a: Architecture | t in (a.unifiedAs).elems
      implies { {no t.path} and {no t.inTopology} iff {no a.contain}
      }
136   // #27 For all topologies of an architecture, a mediator engages
       in at least one unification
137   all t: Topology, m: Mediator | let Suni = participatesInTopology
      [m,t] | -- returns the set of unifications in t of which m
      participates
138    {m in participatesInTopology[t]} implies m in owner[Suni.src]
      or m in owner[Suni.dest]
139   // #28 All systems that participate in an architecture must be
      binded by at least one unification in all of its topologies
140   all e: System, a: Architecture, t: Topology | e in a.contain and
        t in (a.unifiedAs).elems implies {
141    some u: Unification | e in isUnifiedTo[u] and u in t.
      inTopology
142   }
143   // #29 The path of a topology connects unifications that have
```

```
      originate or end in the architectural element
144   all t: Topology, a: Architecture | {some a.contain} and t in (a.
      unifiedAs).elems implies {
145     all u,v: Unification | u->v in t.path implies {some
      isUnifiedTo[u]&isUnifiedTo[v]}
146   }
147   // #30 Connections of the same gate can only be unified to
      connections of the same duty.
148   // As a consequence, two unifications may share the exact same
      ports or none at all.
149   all g: Gate, a: Architecture, t: Topology | let Suni =
      unificationsOfPort[g,t] | g.~hasPort in a.contain and t in (a.
      unifiedAs).elems implies { one dutiesOfUnification[Suni] }
150   all a: Architecture | some s: Sos | a in getArch[s]
151 }
152 /**
153 * ASSERTIONS ABOUT THE INITIAL CONFIGURATION OF SoS
154 * This set of assertions covers structural properties of
      topologies and architectures
155 */
156 // A#1 Checks if an architecture can have no elements.
157 assert architectureCanBeEmpty {
158   no a: Architecture | no a.contain
159 }
160 check architectureCanBeEmpty for 8 but 2 Architecture, 3 Topology
      -- Because this assertion fails, the initial state of the
      architecture can have no elements
161 // A#2 Checks if a topology can have no unifications.
162 assert topologyCanBeEmpty {
163   no t: Topology | no inTopology[t]
164 }
165 check topologyCanBeEmpty for 8 but 2 Architecture, 3 Topology --
      Because this assertion fails, the initial state of the
      architecture can have an empty Topology
166 // A#3 Checks if it is possible to have more than one unification
      over the exact same pair of ports
167 assert  samePortUnification {
168   no a: Architecture, t: Topology, disj u,v: Unification | t in a.
      unifiedAs.elems and u in inTopology[t] and v in inTopology[t]
      and #(portsOfUnification[u] & portsOfUnification[v])>1
169 }
170 check samePortUnification for 8
```

```
171 // A#4 Checks if all bindings of a coalition appear in at least
      one of its topologies
172 assert allBindingsAreUsed {
173   all a: Architecture, u: Unification | u in a.bindings implies {
174     some t: Topology | t in a.unifiedAs.elems and u in inTopology[
      t]
175   }
176 }
177 check allBindingsAreUsed for 8
178 // A#5 Checks if a coalition can have more than two mediators
179 assert moreThanOneMediator {
180   no disj m,m': Mediator, t: Topology | let Smed =
      participatesInTopology[t] | m in Smed and m' in Smed
181 }
182 check moreThanOneMediator for 7 but 3 Topology, 3 Architecture, 8
      Port, 12 Connection -- should be valid when the execution bound
       is sufficiently big.
183 // A#6 Checks if a constituent can be binded to more than two
      mediators
184 assert moreThanOneMediatorPerSystem {
185 --  some t: Topology, disj m,m': Mediator | let Smed =
      participatesInTopology[t] | m in Smed and m' in Smed and some {
      isMediatedBy[m,t]&isMediatedBy[m',t]}
186   all t: Topology | let Smed = participatesInTopology[t] |
187     no disj m,m': Mediator | m in Smed and m' in Smed and {
      isMediatedBy[m,t]=isMediatedBy[m',t]}
188 }
189 check moreThanOneMediatorPerSystem for 6 but 2 Architecture, 2
      Topology, 10 Port, 4 System, 10 Connection, 10 Unification
190 /**
191 * OPERATIONS
192 */
193 // This set of assertions covers different aspects of the dynamic
      support required by SoS, such as addition, deletion,
194 // or modification of architectural elements
195 // A#7 Checks if constituents can enter/leave the architecture (
      which is indicated as a new instance of the architecture)
196 assert cantModifySystemsInArchitecture {
197   no a, a': Architecture, s: System | s in a.contain and s not in
      a'.contain
198 }
199 check cantModifySystemsInArchitecture for 8 -- This assertion
```

```
    should fail
// A#8 Checks if bindings can be added/removed from the
    Architecture
assert cantAddBindingToArchitecture {
  no u: Unification, a, a': Architecture | u in a.bindings and u
    not in a'.bindings
}
check cantAddBindingToArchitecture  for 8 -- This assumption
    should fail
/**
* EXECUTION OF THE SOS
*/
//The initial state for the architecture contains no elements or
    bindings
pred init [a: Architecture] {
  no a.contain    -- There are no elements (constituents)
  no a.bindings
}
// A safe state of the coalition is the one in which all
    topologies, the constituents engage in at least one unification
pred safe [a: Architecture] {
  all t: Topology, disj r,s: System | t in a.unifiedAs.elems and r
    in a.contain and s in a.contain and {
    some disj u, v: Unification | u in a.bindings and v in a.
    bindings and s in isUnifiedTo[u] and r in isUnifiedTo[v]
  }
}
// Following, we describe possible operations that can cause a
    change in the state of the architecture
// A single System enters the coalition
pred addSystem[s: System, a, a': Architecture]{
  //pre conditions
  s not in a.contain //s did not pertain to the previous coalition
  //postconditions
  s in a'.contain //s pertains to the new coalition
  //frame conditions: describe what does not change between pre-
    state and post-state
  noSystemChangeExcept[s,a,a']
}
// A single System is removed from the architecture
pred remSystem[s: System, a, a': Architecture]{
  //pre conditions
```

```
232   s in a.contain //s pertained to the previous coalition
233   //postconditions
234   ( a'.contain ) = ( a.contain - s ) //s does not pertain to the
       new coalition
235   //frame conditions: describe what does not change between pre-
       state and post-state
236   noSystemChangeExcept[s,a,a']
237 }
238 // This operation covers the inclusion of a new possible binding
       to the architecture
239 pred addBinding[u: Unification, a, a': Architecture]{
240   //pre conditions
241   u not in a.bindings
242   //postconditions
243   a'.bindings = a.bindings + u
244 }
245 // Apart from constituent s, all other constituents remain the
       same
246 pred noSystemChangeExcept[s: System, a, a': Architecture]{
247   all d: System - s | d in a.contain implies d in a'.contain
248 }
249 // This operation covers the exclusion of an existing binding of
       the architecture
250 pred remBinding[u: Unification, a, a': Architecture ]{
251   //pre conditions
252   u in a.bindings
253   //postconditions
254   a'.bindings = a.bindings - u
255 }
256 /**
257 /* MODEL SIMULATION
258 /**/
259 pred Trace {
260   init[A0/first]
261   all a: Architecture - A0/last | let a' = A0/next[a] |
262   {
263     (one s: System | addSystem [s,a,a'] ) or
264     (one s: System | remSystem [s,a,a'] ) or
265     (one u: Unification | addBinding[u,a,a']) or
266     (one u: Unification | remBinding[u,a,a'])
267   }
268 }
```

```
269 /**
270 /* GENERAL FUNCTIONS AND PREDICATES
271 /**/
272 // Returns the Duties connected in a set of unifications
273 fun dutiesOfUnification [s: set Unification]: set Duty {
274   (s.src + s.dest).~hasConnection & Duty
275 }
276 // Returns the Gates connected in a set of unification
277 fun gatesOfUnification [s: set Unification]: set Gate {
278   (s.src + s.dest).~hasConnection & Gate
279 }
280 fun getRelay[s: Sos]: set Relay{
281   {r: Relay | some a: Architecture | a->r in s.arch}
282 }
283 fun getArch[s:Sos]: set Architecture{
284   {a: Architecture | lone r: Relay | a->r in s.arch}
285 }
286 //Returns the set of inward connections of a particular
      architectural element
287 fun inConnection [a: ArchitecturalElement]: set Inward {
288   (a.hasPort).hasConnection & Inward
289 }
290 //Returns the set of inward connections of a particular
      architectural element
291 fun allConnection [a: ArchitecturalElement]: set Connection {
292   (a.hasPort).hasConnection
293 }
294 //Returns the set of architectural elements (i.e., systems and
      mediators) binded by one or more unifications
295 fun isUnifiedTo [u: set Unification]: set ArchitecturalElement {
296   u.src.owner + u.dest.owner
297 }
298 //Returns the set of outward connections of a particular
      architectural element
299 fun outConnection [a: ArchitecturalElement]: set Outward {
300   (a.hasPort).hasConnection & Outward
301 }
302 //Returns the ports binded by a set of unifications
303 fun portsOfUnification [s: set Unification]: set Port {
304   (s.src + s.dest).~hasConnection
305 }
306 //Returns the set of unifications in a coalition c in that a given
```

```
      architectural element participates.
307 // This function considers unifications that appear in any
      topology of c.
308 fun participatesIn[e: ArchitecturalElement,a:Architecture]: set
      Unification {
309   {u: Unification | u in inTopology[(a.unifiedAs).elems] and e in
      isUnifiedTo[u]}
310 }
311 //Returns the set of unifications in that a given architectural
      element (e.g., system or mediator) participates in topology t
      of coalition c
312 fun participatesInTopology[e:ArchitecturalElement,t:Topology]: set
       Unification {
313   {u: Unification | e in isUnifiedTo[u] and u in inTopology[t]}
314 }
315 //Returns the set of mediators that participate in a given
      topology
316 fun participatesInTopology[t:Topology]: set Mediator {
317   {m: Mediator | m in isUnifiedTo[inTopology[t]]}
318 }
319 //Returns the set of constituents that are mediated by a given
      mediator in topology t
320 fun isMediatedBy[m: Mediator, t: Topology]: set System {
321   {e: System | e in isUnifiedTo[participatesInTopology[m,t]]} --
      pIT[m,t] returns set of unifications
322 }
323 //Returns the set of mediators that mediate a given system in
      topology t
324 fun isMediatedBy[e: System, t: Topology]: set Mediator {
325   {m: Mediator | e in isUnifiedTo[participatesInTopology[m,t]]}
326 }
327 fun inRelay[r:Relay]: set Connection {
328   {c: Connection | some c2: Connection | c->c2 in r.relayCon or c2
      ->c in r.relayCon}
329 }
330 //Returns the unification that binds two connections together, if
      any
331 fun hasUnification [cFirst, cSecond: Connection]: lone Unification
      {
332   {u: Unification |
333     u.src in (cFirst & Outward) and u.dest in (cSecond & Inward)
334     or
```

```
335     u.src in (cSecond & Outward) and u.dest in (cFirst & Inward)
336   }
337 }
338 //Returns the set of datatypes of a given Tuple
339 fun inTuple[n: TupleType]: set Datatype {
340   {d: Datatype | d in n.complexType}
341 }
342 //Returns the set of unifications in a topology in which a port
      participates, if any
343 fun unificationsOfPort[p:Port,t:Topology]: set Unification {
344   {u: Unification | u in t.inTopology and p in portsOfUnification[
      u]}
345 }
346 /**
347 /* PREDICATES FOR INSTANTIATING THE GENERIC MODEL
348 /**/
349 pred attributeConToPort [sCon: set Connection, p: Port]{
350   p.hasConnection in sCon
351 }
352 pred attributePortToAE [sPor: set Port, ae: ArchitecturalElement]{
353   ae.hasPort in sPor
354 }
355 pred attributeAEToArchitecture [sAE: set ArchitecturalElement, a:
      Architecture]{
356   a.contain in (sAE & System) and participatesInTopology[a.
      unifiedAs.elems] in (sAE & Mediator)
357 }
358 pred attributeBinToArchitecture [sUni: set Unification, a:
      Architecture]{
359   a.bindings in sUni
360 }
361 pred attributeDatatypeToCon [d: Datatype, con: Connection]{
362   con.hasDatatype in d
363 }
364 pred attributeDatatypeToTuple [sData: set Datatype,d: TupleType]{
365   d.complexType in sData
366 }
367 pred attributeRelay[src: Connection, dest: Connection]{
368   all s: Sos, r: Relay | r in getRelay[s] implies some r.relayCon
      and r.relayCon in src->dest
369 }
370 pred attributeArchToSos[a: Architecture,s: Sos]{
```

```
371   getArch[s] in a
372 }
373 pred attributeNary[sData: set Datatype,t:TupleType]{
374   t.complexType in sData
375 }
376 //True if some unification unify the connections cFirst and
      cSecond -- one MUST be IN and the other MUST be OUT
377 pred unify [cFirst, cSecond: Connection] {
378   cFirst in Outward and cSecond in Inward implies {some u:
      Unification | u.src in cFirst and u.dest in cSecond}
379   else cFirst in Inward and cSecond in Outward implies {some u:
      Unification | u.src in cSecond and u.dest in cFirst}
380 }
```

APPENDIX

B

Experiment Materials

Table 22 presents raw data collected in the operation of the quasi-experiment described in Chapter 6. The assigned scope of signature Relay is fixed in the beginning of this experiment as 1. The solver column corresponds to running time of the constraint solver while the transformation column corresponds to running time of the Solution2Alloy generator, both measured in milliseconds. The constraint solver used in this experiment is the SAT4J[1]. Since the transformation is only performed when the model is satisfiable, this data is only available for a subset of test cases. Moreover, the value of productivity is calculated as the ratio between LOC of generated abstract architecture and running time of the transformation (calculated as LOC/min), hence unsatisfiable models present zero productivity. Source code for the instrumented class, test case commands, and script used for automatically operating this experiment are externally available at <https://goo.gl/RgqR9K> (accessed on 08/05/2017).

Table 22 – Raw data collected in quasi-experiment

Test	AE	Port	Connection	Unification	Solver	Result	Transf.	LOC	Productivity
Test Case	AE	Port	Conn	Unif	Solver	Result	Transf	LOC	Productivity*
1	3	4	8	8	826	0	0	0	0
2	3	4	8	9	754	0	0	0	0
3	3	4	8	10	913	0	0	0	0
4	3	4	8	11	1075	0	0	0	0
5	3	4	9	8	982	0	0	0	0
6	3	4	9	9	1014	0	0	0	0
7	3	4	9	10	1130	0	0	0	0
8	3	4	9	11	995	0	0	0	0
9	3	4	10	8	1122	0	0	0	0
10	3	4	10	9	1093	0	0	0	0
11	3	4	10	10	1176	0	0	0	0
12	3	4	10	11	1017	0	0	0	0

Continued on next page

[1] SAT4J, <http://www.sat4j.org/>

Table 22 – *Continued from previous page*

Test	AE	Port	Connection	Unification	Solver	Result	Transf.	LOC	Productivity
13	3	4	11	8	1280	0	0	0	0
14	3	4	11	9	1082	0	0	0	0
15	3	4	11	10	1341	0	0	0	0
16	3	4	11	11	1206	0	0	0	0
17	3	5	8	8	932	0	0	0	0
18	3	5	8	9	867	0	0	0	0
19	3	5	8	10	934	0	0	0	0
20	3	5	8	11	1094	0	0	0	0
21	3	5	9	8	845	0	0	0	0
22	3	5	9	9	1350	0	0	0	0
23	3	5	9	10	1435	0	0	0	0
24	3	5	9	11	1452	0	0	0	0
25	3	5	10	8	1740	0	0	0	0
26	3	5	10	9	1488	0	0	0	0
27	3	5	10	10	1351	0	0	0	0
28	3	5	10	11	1405	0	0	0	0
29	3	5	11	8	1495	0	0	0	0
30	3	5	11	9	1411	0	0	0	0
31	3	5	11	10	1611	0	0	0	0
32	3	5	11	11	2282	0	0	0	0
33	3	6	8	8	1137	0	0	0	0
34	3	6	8	9	834	0	0	0	0
35	3	6	8	10	1108	0	0	0	0
36	3	6	8	11	1208	0	0	0	0
37	3	6	9	8	1136	0	0	0	0
38	3	6	9	9	1212	0	0	0	0
39	3	6	9	10	1039	0	0	0	0
40	3	6	9	11	1132	0	0	0	0
41	3	6	10	8	1101	0	0	0	0
42	3	6	10	9	1616	0	0	0	0
43	3	6	10	10	1886	0	0	0	0
44	3	6	10	11	1517	0	0	0	0
45	3	6	11	8	1719	0	0	0	0
46	3	6	11	9	1454	0	0	0	0
47	3	6	11	10	1995	0	0	0	0
48	3	6	11	11	1827	0	0	0	0
49	3	7	8	8	803	0	0	0	0
50	3	7	8	9	1195	0	0	0	0
51	3	7	8	10	1155	0	0	0	0
52	3	7	8	11	1247	0	0	0	0
53	3	7	9	8	1270	0	0	0	0
54	3	7	9	9	977	0	0	0	0
55	3	7	9	10	1282	0	0	0	0
56	3	7	9	11	1584	0	0	0	0
57	3	7	10	8	1472	0	0	0	0

Continued on next page

Table 22 – *Continued from previous page*

Test	AE	Port	Connection	Unification	Solver	Result	Transf.	LOC	Productivity
58	3	7	10	9	1658	0	0	0	0
59	3	7	10	10	1618	0	0	0	0
60	3	7	10	11	1404	0	0	0	0
61	3	7	11	8	1895	0	0	0	0
62	3	7	11	9	1557	0	0	0	0
63	3	7	11	10	1785	0	0	0	0
64	3	7	11	11	1826	0	0	0	0
65	3	8	8	8	1091	0	0	0	0
66	3	8	8	9	1257	0	0	0	0
67	3	8	8	10	1183	0	0	0	0
68	3	8	8	11	1387	0	0	0	0
69	3	8	9	8	1549	0	0	0	0
70	3	8	9	9	1562	0	0	0	0
71	3	8	9	10	1290	0	0	0	0
72	3	8	9	11	1976	0	0	0	0
73	3	8	10	8	1698	0	0	0	0
74	3	8	10	9	1787	0	0	0	0
75	3	8	10	10	1689	0	0	0	0
76	3	8	10	11	1874	0	0	0	0
77	3	8	11	8	2083	0	0	0	0
78	3	8	11	9	1723	0	0	0	0
79	3	8	11	10	1946	0	0	0	0
80	3	8	11	11	1980	0	0	0	0
81	4	4	8	8	1244	0	0	0	0
82	4	4	8	9	1353	0	0	0	0
83	4	4	8	10	1529	0	0	0	0
84	4	4	8	11	2053	0	0	0	0
85	4	4	9	8	1780	0	0	0	0
86	4	4	9	9	1959	0	0	0	0
87	4	4	9	10	2476	0	0	0	0
88	4	4	9	11	2990	0	0	0	0
89	4	4	10	8	2591	1	57	30	680
90	4	4	10	9	3068	1	46	30	578
91	4	4	10	10	3581	1	103	30	489
92	4	4	10	11	5109	1	932	30	298
93	4	4	11	8	3567	1	54	30	497
94	4	4	11	9	3593	1	43	30	495
95	4	4	11	10	42910	1	5706	30	37
96	4	4	11	11	4992	1	47	30	357
97	4	5	8	8	1682	0	0	0	0
98	4	5	8	9	2039	0	0	0	0
99	4	5	8	10	2356	0	0	0	0
100	4	5	8	11	2579	0	0	0	0
101	4	5	9	8	2272	0	0	0	0
102	4	5	9	9	2089	0	0	0	0

Continued on next page

Table 22 – *Continued from previous page*

Test	AE	Port	Connection	Unification	Solver	Result	Transf.	LOC	Productivity
103	4	5	9	10	2526	0	0	0	0
104	4	5	9	11	2816	0	0	0	0
105	4	5	10	8	3143	1	50	30	564
106	4	5	10	9	3658	1	62	30	484
107	4	5	10	10	4012	1	46	30	444
109	4	5	11	8	4151	1	43	30	429
110	4	5	11	9	4771	1	110	30	369
111	4	5	11	10	5429	1	175	30	321
112	4	5	11	11	5172	1	43	30	345
113	4	6	8	8	1750	0	0	0	0
114	4	6	8	9	1821	0	0	0	0
115	4	6	8	10	2134	0	0	0	0
116	4	6	8	11	2608	0	0	0	0
117	4	6	9	8	2239	0	0	0	0
118	4	6	9	9	2397	0	0	0	0
119	4	6	9	10	2833	0	0	0	0
120	4	6	9	11	3000	0	0	0	0
121	4	6	10	8	3774	1	44	30	471
122	4	6	10	9	3734	1	47	30	476
123	4	6	10	10	4619	1	47	30	386
124	4	6	10	11	10562	1	584	30	161
125	4	6	11	8	4217	1	43	30	423
126	4	6	11	9	5136	1	41	30	348
127	4	6	11	10	5276	1	48	30	338
128	4	6	11	11	6015	1	43	30	297
129	4	7	8	8	2191	0	0	0	0
130	4	7	8	9	2446	0	0	0	0
131	4	7	8	10	3102	0	0	0	0
132	4	7	8	11	3270	0	0	0	0
133	4	7	9	8	2758	0	0	0	0
134	4	7	9	9	2852	0	0	0	0
135	4	7	9	11	6861	0	0	0	0
136	4	7	10	8	9338	1	2970	30	146
137	4	7	10	10	4584	1	47	30	389
138	4	7	10	11	5028	1	39	30	355
139	4	7	11	8	4590	1	45	30	388
140	4	7	11	9	5198	1	42	30	344
141	4	7	11	10	5606	1	38	30	319
142	4	7	11	11	6426	1	45	30	278
143	4	8	8	8	1971	0	0	0	0
144	4	8	8	9	2151	0	0	0	0
145	4	8	8	10	2591	0	0	0	0
146	4	8	8	11	2976	0	0	0	0
147	4	8	9	8	2742	0	0	0	0
148	4	8	9	9	3090	0	0	0	0

Continued on next page

Table 22 – *Continued from previous page*

Test	AE	Port	Connection	Unification	Solver	Result	Transf.	LOC	Productivity
149	4	8	10	8	4185	1	41	30	426
150	4	8	10	9	4638	1	40	30	385
151	4	8	10	10	4670	1	39	30	382
152	4	8	10	11	5228	1	40	30	342
153	4	8	11	8	5471	1	42	30	327
154	4	8	11	9	5531	1	40	30	323
155	4	8	11	10	6450	1	38	30	277
156	4	8	11	11	6902	1	40	30	259
157	5	4	8	8	1149	0	0	0	0
158	5	4	8	9	1869	0	0	0	0
159	5	4	8	10	1994	0	0	0	0
160	5	4	8	11	1774	0	0	0	0
161	5	4	9	8	1914	0	0	0	0
162	5	4	9	9	2499	0	0	0	0
163	5	4	9	10	2520	0	0	0	0
164	5	4	9	11	2971	0	0	0	0
165	5	4	10	8	2889	1	41	30	614
166	5	4	10	9	3180	1	39	30	559
167	5	4	10	10	3738	1	44	30	476
168	5	4	10	11	4092	1	39	30	436
169	5	4	11	8	3826	1	41	30	465
170	5	4	11	9	4156	1	35	30	429
171	5	4	11	10	4632	1	44	30	385
172	5	4	11	11	4873	1	41	30	366
173	5	5	8	8	1746	0	0	0	0
174	5	5	8	9	2117	0	0	0	0
175	5	5	8	10	2287	0	0	0	0
176	5	5	8	11	2656	0	0	0	0
177	5	5	9	8	2291	0	0	0	0
178	5	5	9	9	2370	0	0	0	0
179	5	5	9	10	2688	0	0	0	0
180	5	5	9	11	3325	0	0	0	0
181	5	5	10	8	4010	1	45	30	444
182	5	5	10	9	4341	1	37	30	411
183	5	5	10	10	4910	1	52	30	363
184	5	5	10	11	5753	1	38	30	311
185	5	5	11	8	4702	1	47	30	379
186	5	5	11	9	5096	1	42	30	350
187	5	5	11	10	5529	1	39	30	323
188	5	5	11	11	6187	1	43	30	289
189	5	6	8	8	2306	0	0	0	0
190	5	6	8	9	3429	0	0	0	0
191	5	6	8	10	3192	0	0	0	0
192	5	6	8	11	3613	0	0	0	0
193	5	6	9	8	3560	0	0	0	0

Continued on next page

Table 22 – *Continued from previous page*

Test	AE	Port	Connection	Unification	Solver	Result	Transf.	LOC	Productivity
194	5	6	9	9	3113	0	0	0	0
195	5	6	9	10	3474	0	0	0	0
196	5	6	9	11	3629	0	0	0	0
197	5	6	10	8	4276	1	46	30	416
198	5	6	10	9	4841	1	51	30	368
199	5	6	10	10	4984	1	56	30	357
200	5	6	10	11	5510	1	39	30	324
201	5	6	11	8	5130	1	46	30	348
202	5	6	11	9	6103	1	42	30	293
203	5	6	11	10	7932	1	51	30	225
204	5	6	11	11	8250	1	53	30	217
205	5	7	8	8	2904	0	0	0	0
206	5	7	8	9	2946	0	0	0	0
207	5	7	8	10	3991	0	0	0	0
208	5	7	8	11	3573	0	0	0	0
209	5	7	9	8	3083	0	0	0	0
210	5	7	9	9	3600	0	0	0	0
211	5	7	9	10	3741	0	0	0	0
212	5	7	9	11	4452	0	0	0	0
213	5	7	10	8	5234	1	45	30	341
214	5	7	10	9	4956	1	53	30	359
215	5	7	10	10	5753	1	47	30	310
216	5	7	10	11	6741	1	64	30	265
217	5	7	11	8	6269	1	56	30	285
218	5	7	11	9	6941	1	46	30	258
219	5	7	11	10	7259	1	54	30	246
220	5	7	11	11	9312	1	54	30	192
221	5	8	8	8	2896	0	0	0	0
222	5	8	8	9	3624	0	0	0	0
223	5	8	8	10	3786	0	0	0	0
224	5	8	8	11	3744	0	0	0	0
225	5	8	9	8	3556	0	0	0	0
226	5	8	9	9	3868	0	0	0	0
227	5	8	9	10	3578	0	0	0	0
228	5	8	9	11	3648	0	0	0	0
229	5	8	10	8	4595	1	35	30	389
230	5	8	10	9	5125	1	41	30	348
231	5	8	10	10	5568	1	44	30	321
232	5	8	10	11	6609	1	39	30	271
233	5	8	11	8	6204	1	41	30	288
234	5	8	11	9	6725	1	43	30	266
235	5	8	11	10	7177	1	43	30	249
236	5	8	11	11	7432	1	40	30	241
237	6	4	8	8	1473	0	0	0	0
238	6	4	8	9	1776	0	0	0	0

Continued on next page

Table 22 – *Continued from previous page*

Test	AE	Port	Connection	Unification	Solver	Result	Transf.	LOC	Productivity
239	6	4	8	10	1624	0	0	0	0
240	6	4	8	11	2209	0	0	0	0
241	6	4	9	8	2321	0	0	0	0
242	6	4	9	9	2427	0	0	0	0
243	6	4	9	10	3039	0	0	0	0
244	6	4	9	11	3317	0	0	0	0
245	6	4	10	8	3656	1	47	30	486
246	6	4	10	9	4955	1	47	30	360
247	6	4	10	10	5581	1	46	30	320
248	6	4	10	11	6154	1	52	30	290
249	6	4	11	8	4820	1	58	30	369
250	6	4	11	9	6039	1	48	30	296
251	6	4	11	10	5468	1	42	30	327
252	6	4	11	11	6515	1	47	30	274
253	6	5	8	8	3145	0	0	0	0
254	6	5	8	9	3382	0	0	0	0
255	6	5	8	10	4143	0	0	0	0
256	6	5	8	11	3557	0	0	0	0
257	6	5	9	8	3331	0	0	0	0
258	6	5	9	9	3207	0	0	0	0
259	6	5	9	10	4574	0	0	0	0
260	6	5	9	11	4808	0	0	0	0
261	6	5	10	8	5076	1	44	30	352
262	6	5	10	9	5649	1	53	30	316
263	6	5	10	10	6104	1	73	30	291
264	6	5	10	11	7104	1	45	30	252
265	6	5	11	8	5201	1	42	30	343
266	6	5	11	9	5985	1	47	30	298
267	6	5	11	10	6305	1	41	30	284
268	6	5	11	11	6750	1	38	30	265
269	6	6	8	8	2401	0	0	0	0
270	6	6	8	9	2739	0	0	0	0
271	6	6	8	10	3132	0	0	0	0
272	6	6	8	11	3572	0	0	0	0
273	6	6	9	8	2787	0	0	0	0
274	6	6	9	9	3527	0	0	0	0
275	6	6	9	10	3576	0	0	0	0
276	6	6	9	11	4310	0	0	0	0
277	6	6	10	8	5154	1	43	30	346
278	6	6	10	9	4894	1	47	30	364
279	6	6	10	10	5656	1	30	30	317
280	6	6	10	11	6547	1	39	30	273
281	6	6	11	8	5827	1	40	30	307
282	6	6	11	9	6586	1	41	30	272
283	6	6	11	10	7203	1	42	30	248

Continued on next page

Table 22 – *Continued from previous page*

Test	AE	Port	Connection	Unification	Solver	Result	Transf.	LOC	Productivity
284	6	6	11	11	7732	1	40	30	232
285	6	7	8	8	2517	0	0	0	0
286	6	7	8	9	2761	0	0	0	0
287	6	7	8	10	2968	0	0	0	0
288	6	7	8	11	3376	0	0	0	0
289	6	7	9	8	3268	0	0	0	0
290	6	7	9	9	3720	0	0	0	0
291	6	7	9	10	4012	0	0	0	0
292	6	7	9	11	4005	0	0	0	0
293	6	7	10	8	4994	1	42	30	357
294	6	7	10	9	5474	1	46	30	326
295	6	7	10	10	5939	1	44	30	301
296	6	7	10	11	6611	1	46	30	270
297	6	7	11	8	6219	1	41	30	288
298	6	7	11	9	7095	1	38	30	252
299	6	7	11	10	7747	1	37	30	231
300	6	7	11	11	7971	1	37	30	225
301	6	8	8	8	2678	0	0	0	0
302	6	8	8	9	3305	0	0	0	0
303	6	8	8	10	3408	0	0	0	0
304	6	8	8	11	3689	0	0	0	0
305	6	8	9	8	3631	0	0	0	0
306	6	8	9	9	3716	0	0	0	0
307	6	8	9	10	4116	0	0	0	0
308	6	8	9	11	4501	0	0	0	0
309	6	8	10	8	5439	1	46	30	328
310	6	8	10	9	5701	1	43	30	313
311	6	8	10	10	6639	1	42	30	269
312	6	8	10	11	7539	1	43	30	237
313	6	8	11	8	7093	1	42	30	252
314	6	8	11	9	7892	1	39	30	227
315	6	8	11	10	7798	1	41	30	230
316	6	8	11	11	8133	1	38	30	220
317	7	4	8	8	1913	0	0	0	0
318	7	4	8	9	3299	0	0	0	0
319	7	4	8	10	3377	0	0	0	0
320	7	4	8	11	3181	0	0	0	0
321	7	4	9	8	3181	0	0	0	0
322	7	4	9	9	2931	0	0	0	0
323	7	4	9	10	3436	0	0	0	0
324	7	4	9	11	3515	0	0	0	0
325	7	4	10	8	4446	1	41	30	401
326	7	4	10	9	4643	1	44	30	384
327	7	4	10	10	5123	1	41	30	349
328	7	4	10	11	6447	1	45	30	277

Continued on next page

Table 22 – *Continued from previous page*

Test	AE	Port	Connection	Unification	Solver	Result	Transf.	LOC	Productivity
329	7	4	11	8	4948	1	43	30	361
330	7	4	11	9	5431	1	40	30	329
331	7	4	11	10	6121	1	44	30	292
332	7	4	11	11	6690	1	45	30	267
333	7	5	8	8	2706	0	0	0	0
334	7	5	8	9	3110	0	0	0	0
335	7	5	8	10	3461	0	0	0	0
336	7	5	8	11	3844	0	0	0	0
337	7	5	9	8	3214	0	0	0	0
338	7	5	9	9	3538	0	0	0	0
339	7	5	9	10	3946	0	0	0	0
340	7	5	9	11	4110	0	0	0	0
341	7	5	10	8	5131	1	43	30	348
342	7	5	10	9	5063	1	50	30	352
343	7	5	10	10	5845	1	38	30	306
344	7	5	10	11	6703	1	38	30	267
345	7	5	11	8	5906	1	43	30	303
346	7	5	11	9	6468	1	40	30	277
347	7	5	11	10	7382	1	40	30	243
348	7	5	11	11	7962	1	38	30	225
349	7	6	8	8	2772	0	0	0	0
350	7	6	8	9	3354	0	0	0	0
351	7	6	8	10	3549	0	0	0	0
352	7	6	8	11	3770	0	0	0	0
353	7	6	9	8	3653	0	0	0	0
354	7	6	9	9	3934	0	0	0	0
355	7	6	9	10	3921	0	0	0	0
356	7	6	9	11	4466	0	0	0	0
357	7	6	10	8	5341	1	39	30	335
358	7	6	10	9	5802	1	40	30	308
359	7	6	10	10	6248	1	44	30	286
360	7	6	10	11	6948	1	40	30	258
361	7	6	11	8	6867	1	51	30	260
362	7	6	11	9	7476	1	42	30	239
363	7	6	11	10	7759	1	40	30	231
364	7	6	11	11	7825	1	35	30	229
365	7	7	8	8	3153	0	0	0	0
366	7	7	8	9	3408	0	0	0	0
367	7	7	8	10	3601	0	0	0	0
368	7	7	8	11	3858	0	0	0	0
369	7	7	9	8	3698	0	0	0	0
370	7	7	9	9	4057	0	0	0	0
371	7	7	9	10	4572	0	0	0	0
372	7	7	9	11	4670	0	0	0	0
373	7	7	10	8	5836	1	43	30	306

Continued on next page

Table 22 – *Continued from previous page*

Test	AE	Port	Connection	Unification	Solver	Result	Transf.	LOC	Productivity
374	7	7	10	9	6514	1	44	30	274
375	7	7	10	10	7211	1	43	30	248
376	7	7	10	11	7663	1	37	30	234
377	7	7	11	8	7454	1	37	30	240
378	7	7	11	9	7793	1	36	30	230
379	7	7	11	10	7905	1	38	30	227
380	7	7	11	11	7964	1	44	30	225
381	7	8	8	8	3325	0	0	0	0
382	7	8	8	9	3680	0	0	0	0
383	7	8	8	10	3810	0	0	0	0
384	7	8	8	11	4274	0	0	0	0
385	7	8	9	8	4103	0	0	0	0
386	7	8	9	9	4368	0	0	0	0
387	7	8	9	10	4704	0	0	0	0
388	7	8	9	11	4803	0	0	0	0
389	7	8	10	8	6327	1	43	30	283
390	7	8	10	9	7182	1	42	30	249
391	7	8	10	10	7591	1	44	30	236
392	7	8	10	11	7714	1	40	30	232
393	7	8	11	8	7995	1	40	30	224
394	7	8	11	9	7966	1	41	30	225
395	7	8	11	10	8227	1	43	30	218
396	7	8	11	11	8291	1	39	30	216

APPENDIX

C

Declaration of Original Authorship and List of Publications

C.1 Publications Derived from this Thesis

- **Guessi, M.**, Oquendo, F., and Nakagawa, E. Y. *Checking the architectural feasibility of Systems-of-Systems using formal descriptions*. p. 1-6. DOI: <10.1109/sysose.2016.7542939>.

  Event: 11th System of Systems Engineering Conference (SoSE), 2016, Kongsberg, Norway.

  Level of contribution: High – the PhD candidate is the main investigator.

- **Guessi, M.**, Cavalcante, E., Oliveira, L. B. R. *Characterizing Architecture Description Languages for Software-Intensive Systems-of-Systems*. p. 12-18.

  Event: 3rd International Workshop on Software Engineering for Systems-of-Systems (SESoS) at 37th International Conference on Software Enginnering (ICSE), 2015. Florence, Italy.

  Level of contribution: High – the PhD candidate is the main investigator.

- **Guessi, M.**, Graciano Neto, V. V., Bianchi, T., Felizardo, K. R., Oquendo, F., and Nakagawa, E. Y. *A systematic literature review on the description of software architectures for systems of systems*. p. 1433-1440. DOI: <10.1145/2695664.2695795>.

  Event: 30th ACM/SIGAPP Symposium on Applied Computing (SAC), 2015, Salamanca, Spain.

  Level of contribution: High – the PhD candidate is the main investigator.

C.2 Other Publications

- Nakagawa, E. Y., **Guessi, M.**, Oliveira, L. B. R., Oliveira, B. R. N. Síntese dos dados e apresentação dos resultados.

  Book: Felizardo, K. R.. Síntese dos dados e apresentação dos resultados. Nakagawa, E. Y.; Fabbri, S. C. P. F.; Ferrari, F. (Org.). Revisão sistemática da literatura em Engenharia de Software: teoria e prática. 1ed. Rio de Janeiro: Elsevier, 2016, p. 1-131.

  Level of contribution: Medium – the PhD candidate contributed to paper planning and writing.

- **Guessi, M.**, Moreira, D. A., Abdalla, G., Oquendo, F., and Nakagawa, E. Y. *OntolAD*. p. 1417-1433. DOI: <10.1145/2695664.2695795>.

  Event: 30th ACM/SIGAPP Symposium on Applied Computing (SAC), 2015, Salamanca, Spain.

  Level of contribution: High – the PhD candidate is the main investigator.

- Nakagawa, E. Y., **Guessi, M.**, Feitosa, D., Oquendo, F., and Maldonado, J.C. *Consolidating a Process for the Design, Representation, and Evaluation of Reference Architectures*. p. 1-10. DOI: <10.1109/wicsa.2014.25>

  Event: 11th Working IEEE/IFIP Conference on Software Architecture at European Conference on Software Architecture (WICSA/ECSA), Sydney, Australia, 2014.

  Level of contribution: Medium – the PhD candidate contributed to paper planning and writing.

- **Guessi, M.**, Oquendo, F., and Nakagawa, E. Y. *An Approach for Capturing and Documenting Architectural Decisions of Reference Architectures*. p. 162-167.

  Event: 26th International Conference on Software Engineering and Knowledge Engineering (SEKE), Vancouver, Canada, 2014.

  Level of contribution: High – the PhD candidate is the main investigator.

- Correia, T. P., **Guessi, M.**, Oliveira, L. B. R., Nakagawa, E. Y. RARep: a Reference Architecture Repository. p. 363-368.

  Event: 28th International Conference on Software Engineering and Knowledge Engineering (SEKE), Redwood, United States. 2016.

  Level of contribution: High – the PhD candidate contributed in the planning, execution, and reporting.

- Graciano Neto, V. V., **Guessi, M.**, Oliveira, L., Oquendo, F., and Nakagawa, E. Y. *Investigating the Model-Driven Development for Systems-of-Systems*. p. 1-8. DOI: <http://dx.doi.org/10.1145/2642803.2642825>

  Event: 8th European Conference on Software Architecture Workshop(ECSAW), Vienna, Austria, 2014.

  Level of contribution: Medium – the PhD candidate contributed to paper planning and writing.

- **Guessi, M.**, Oquendo, F., and Nakagawa, E. Y. *Variability Viewpoint to Describe Reference Architectures*. p. 1-6. DOI: <http://dx.doi.org/10.1145/2578128.2578238>

  Event: 3rd International Workshop on Variability in Software Architecture (VARSA) at 11th Working IEEE/IFIP Conference on Software Architecture (WICSA). Sydney, Australia, 2014.

  Level of contribution: High – the PhD candidate is the main investigator.

- Nakagawa, E. Y., Gonçalves, M., **Guessi, M.**, Oliveira, L., and Oquendo, F. *The State-of-the-Art and Future Perspectives in Systems-of-Systems Software Architectures*. p. 13-20. DOI: <http://dx.doi.org/10.1145/2489850.2489853>

  Event: 1st International Workshop on Software Engineering for Systems-of-Systems (SESoS) at the 7th European Conference on Software Architecture (ECSA), Montpellier, France, 2013.

  Level of contribution: Medium – the PhD candidate contributed to paper planning and writing.

- Toda, A. M., Valle, P. H. D., **Guessi, M.**, Rocha, R. V.; Maldonado, J.C.; Isotani, S. *Plataforma de Recursos Educacionais Abertos: Uma Arquitetura de Referência com Elementos de Gamificação*. URL: <http://seer.ufrgs.br/index.php/renote/article/view/70650>

  Journal: RENOTE. 2016. v. 14, n. 2, p.1-10. ISSN: 1679-1916. h-index: 20 (Publish or Perish).

  Level of contribution: Medium – the PhD candidate contributed to paper planning and writing.

- Santos, J. F. M., **Guessi, M.**, Galster, M., Feitosa, D., Nakagawa, E. Y. *A Checklist for Evaluation of Reference Architectures for Embedded Systems*. p. 451-454.

  Event: 25th International Conference on Software Engineering and Knowledge Engineering (SEKE). Boston, United States, 2013.

  Level of contribution: High – the PhD candidate contributed in the planning, execution, and reporting.

- Oliveira, L. B. R., **Guessi, M.**, Feitosa, D., Manteuffel, C., Galster, M., Oquendo, F., Nakagawa, E. Y. *An Investigation on Quality Models and Quality Attributes for Embedded Systems*. p. 523-528.

  Event: 8th International Conference on Software Engineering Advances (ICSEA), Venice, Italy, 2013.

  Level of contribution: Medium – the PhD candidate contributed to paper planning and writing.

- **Guessi, M.**, Oliveira, L. B. R., Garcés, L., Oquendo, F. *Towards a Formal Description of Reference Architectures for Embedded Systems*. p. 17-20. DOI: <http://dx.doi.org/10.1145/2755567.2755571>

  Event: 1st International Workshop on Exploring Component-based Techniques for Constructing Reference Architectures (CoBRA) at WICSA/ECSA, Montréal, Canada, 2015.

  Level of contribution: High – the PhD candidate is the main investigator.

- **Guessi, M.**, Oquendo, F., Nakagawa, E. Y. *An Approach for Capturing and Documenting Architectural Decisions of Reference Architectures*. p. 162-167.

  Event: 26th International Conference on Software Engineering and Knowledge Engineering (SEKE), Vancouver, Canada. 2014.

  Level of contribution: High – the PhD candidate is the main investigator.

- Abdalla, G., Damasceno, C. D. N., Guessi, M., Oquendo, F., Nakagawa, E. Y. *A Systematic Literature Review on Knowledge Representation Approaches for Systems-of-Systems*. p. 70-79. DOI: <http://dx.doi.org/10.1109/SBCARS.2015.18>

  Event: Brazilian Symposium on Components, Architectures and Reuse Software (SBCARS), Belo Horizonte, Brazil, 2015.

  Level of contribution: High – the PhD candidate contributed in the planning, execution, and reporting.

- Graciano Neto, V. V., Garcés, L., Guessi, M., Oliveira, L. B. R., Oquendo, F. *On the Equivalence between Reference Architectures and Metamodels*. p. 21-24. DOI: <http://dx.doi.org/10.1145/2755567.2755572>

  Event: 1st International Workshop on Exploring Component-based Techniques for Constructing Reference Architectures (CoBRA) at WICSA/ECSA, Montréal, Canada, 2015.

  Level of contribution: High – the PhD candidate contributed in the planning, execution, and reporting.

- Soares, D., Oliveira, B., Guessi, M., Oquendo, F., Delamaro, M. E., Nakagawa, E. Y. *Towards the Evaluation of System-of-Systems Software Architectures*. p. 53-57.

  Event: 8th Workshop de Desenvolvimento Distribuído de Software, Ecossistemas de Software e Sistemas de Sistemas (WDES), Maceió, Brazil, 2015.

  Level of contribution: Medium – the PhD candidate contributed to paper planning and writing.

ANNEX

A

The SosADL Language

This annex presents the abstract syntax for SosADL (shown in Source code 26), an ADL tailored for the description of SoS discussed in Section 2.3.3 on page 35. This grammar is implemented in Xtext[1]. Examples, justifications, and foundations of this new are discussed by Oquendo on (OQUENDO, 2016a; OQUENDO, 2016c; OQUENDO, 2016b; OQUENDO, 2016d).

Source code 26 – SosADL abstract syntax

```
1 grammar org.archware.sosadl.SosADL with org.eclipse.xtext.common.
    Terminals
2 generate sosADL 'http://www-archware.irisa.fr/sosadl/SosADL'
3
4 SosADL: (imports+=Import)* content=(NewNamedLibrary | NewSoS)
5 ;
6 Import: 'with' importedLibrary=Name
7 ;
8 NewNamedLibrary returns Unit: {Library} 'library' name=Name 'is' '
    {' decls=EntityBlock '}'
9 ;
10 NewSoS returns Unit: {SoS} 'sos' name=Name 'is' '{'
11   (decls=EntityBlock)
12   '}'
13 ;
14 EntityBlock: {EntityBlock}
15   (datatypes+=DataTypeDecl)*
16   (functions+=FunctionDecl)*
17   (systems+=SystemDecl)*
18   (mediators+=MediatorDecl)*
```

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

```
19   (architectures+=ArchitectureDecl)*
20 ;
21 SystemDecl: 'system' name=Name '(' (parameters+=FormalParameter ('
     ,' parameters+=FormalParameter)*)? ')' 'is' '{'
22   (datatypes+=DataTypeDecl)*
23   (gates+=GateDecl)+
24   behavior=BehaviorDecl
25   '}' ('guarantee' '{' assertions+=AssertionDecl+ '}')?
26 ;
27 ArchitectureDecl: 'architecture' name=Name '(' (parameters+=
     FormalParameter (',' parameters+=FormalParameter)*)? ')' 'is' '
     {'
28   (datatypes+=DataTypeDecl)*
29   (gates+=GateDecl)+
30     behavior=ArchBehaviorDecl
31   '}' ('guarantee' '{' assertions+=AssertionDecl+ '}')?
32 ;
33 MediatorDecl: 'mediator' name=Name '(' (parameters+=
     FormalParameter (',' parameters+=FormalParameter)*)? ')' 'is' '
     {'
34   (datatypes+=DataTypeDecl)*
35   (duties+=DutyDecl)+
36   behavior=BehaviorDecl
37   '}'
38   ('assume' '{' assumptions+=AssertionDecl+ '}')?
39   ('guarantee' '{' assertions+=AssertionDecl+ '}')?
40 ;
41 GateDecl:
42   'gate' name=Name 'is' '{'
43     (connections+=Connection)+
44   '}' 'guarantee' '{' protocols+=ProtocolDecl+ '}'
45 ;
46 DutyDecl:
47   'duty' name=Name 'is' '{'
48     (connections+=Connection)+
49   '}'
50   'assume' '{' assertions+=AssertionDecl+ '}'     // WAS: 'require'
       '{' assertion=AssertionDecl '}'
51   'guarantee' '{' protocols+=ProtocolDecl+ '}'    // WAS: 'assume'
     '{' protocol=ProtocolDecl '}'
52 ;
53 Connection:
```

```
54    (environment?='environment')? 'connection' name=Name 'is' mode=
       ModeType '{' valueType=DataType '}'
55  ;
56  AssertionDecl:
57    ('property'|'protocol') name=Name 'is' body=Protocol
58  ;
59  ProtocolDecl:
60      ('property'|'protocol') name=Name 'is' body=Protocol
61  ;
62  Protocol:
63    '{' (statements+=ProtocolStatement)+ '}'
64  ;
65  ProtocolStatement:
66    {ValuingProtocol} valuing=Valuing
67    | {AssertProtocol} assertion=Assert
68    | ProtocolAction
69    | {AnyAction} 'anyaction'
70    | {RepeatProtocol} 'repeat' repeated=Protocol
71    | {IfThenElseProtocol} 'if' condition=Expression 'then' ifTrue=
       Protocol ('else' ifFalse=Protocol)?
72    | {ChooseProtocol} 'choose' branches+=Protocol  ('or' branches+=
       Protocol)+
73    | {ForEachProtocol} 'foreach' variable=Name 'in' setOfValues=
       Expression repeated=Protocol
74    | {DoExprProtocol} 'do' expression=Expression
75    | {DoneProtocol} 'done'
76  ;
77  ProtocolAction:
78    'via' complexName=ComplexName suite=ProtocolActionSuite
79  ;
80  ProtocolActionSuite:
81    ({SendProtocolAction} 'send' expression=FinalExpression)
82    | ('receive' ({ReceiveAnyProtocolAction} 'any'
83              |{ReceiveProtocolAction} variable=Name))
84  ;
85  BehaviorDecl:
86    'behavior' name=Name 'is' body=Behavior
87    // WAS: 'behavior' name=Name '(' (parameters+=FormalParameter ('
       ,' parameters+=FormalParameter)*)? ')' 'is' body=Behavior
88  ;
89  Behavior:
90    '{' (statements+=BehaviorStatement)+ '}'
```

```
91 ;
92 BehaviorStatement:
93   {ValuingBehavior} valuing=Valuing
94   | {AssertBehavior} assertion=Assert
95   | Action
96   | {RepeatBehavior} 'repeat' repeated=Behavior
97   | {IfThenElseBehavior} 'if' condition=Expression 'then' ifTrue=
     Behavior ('else' ifFalse=Behavior)?
98   | {ChooseBehavior} 'choose' branches+=Behavior ('or' branches+=
     Behavior )+
99   | {ForEachBehavior} 'foreach' variable=Name 'in' setOfValues=
     Expression repeated=Behavior
100  | {DoExprBehavior} 'do' expression=Expression
101  | {DoneBehavior} 'done'
102  | {RecursiveCall} 'behavior' '(' (parameters+=Expression (','
     parameters+=Expression)*)? ')'
103  | {UnobservableBehavior} 'unobservable'
104 ;
105 Assert:
106   {TellAssertion} 'tell' name=Name 'is' '{' expression=Expression
     '}'
107   | {UntellAssertion} 'untell' name=Name
108   | {AskAssertion} 'ask' name=Name 'is' '{' expression=Expression
     '}'
109 ;
110 Action:
111   'via' complexName=ComplexName suite=ActionSuite
112 ;
113 ActionSuite:
114   {SendAction} 'send' expression=FinalExpression
115   | {ReceiveAction} 'receive' variable=Name
116 ;
117 ArchBehaviorDecl:
118     // WAS: 'behavior' name=Name '(' (parameters+=Expression (','
     parameters+=Expression)*)? ')'
119   'behavior' name=Name
120   'is' 'compose' '{' (constituents+=Constituent)+ '}'
121   'binding' '{' bindings=Expression '}'
122 ;
123 Constituent:
124   name=Name 'is' value=Expression
125 ;
```

```
126 Binding returns Expression:
127   {Relay} 'relay' connLeft=ComplexName 'to' connRight=ComplexName
128   | {Unify} 'unify' multLeft=Multiplicity '{' connLeft=ComplexName
        '}' 'to' multRight=Multiplicity '{' connRight=ComplexName '}'
129   | {Quantify} quantifier=Quantifier '{' elements+=
       ElementInConstituent (',' elements+=ElementInConstituent)* '
       suchthat' bindings=Expression '}'
130 ;
131 enum Quantifier:
132   QuantifierForall='forall' | QuantifierExists='exists'
133 ;
134 ElementInConstituent:
135   variable=Name 'in' constituent=Name
136 ;
137 enum Multiplicity:
138   MultiplicityOne='one'
139   | MultiplicityNone='none'
140   | MultiplicityLone='lone'
141   | MultiplicityAny='any'
142   | MultiplicitySome='some'
143   | MultiplicityAll='all'
144 ;
145 DataTypeDecl: 'datatype' name=Name ('is' datatype=DataType)? ('{'
       functions+=FunctionDecl+ '}')?
146 ;
147 DataType:
148   BaseType
149   | ConstructedType
150   | {NamedType} name=Name  // name of another type
151 ;
152 FunctionDecl:
153   'function' '(' data=FormalParameter ')' '::'
154   name=Name '(' (parameters+=FormalParameter (',' parameters+=
       FormalParameter)*)? ')' ':' type=DataType 'is' '{'
155       (valuing+=Valuing)*
156     'return' expression=Expression
157   '}'
158 ;
159 FormalParameter:
160   name=Name ':' type=DataType
161 ;
162 BaseType returns DataType:
```

```
163   {IntegerType} 'integer'
164 ;
165 ConstructedType returns DataType:
166   {TupleType} 'tuple' '{' fields+=FieldDecl (',' fields+=FieldDecl
       )* '}'
167   | {SequenceType} 'sequence' '{' type=DataType '}'
168   | {RangeType} 'integer' '{' vmin=Expression '..' vmax=Expression
       '}'  // range of Integer
169   | {ConnectionType} mode=ModeType '{' type=DataType '}'
170 ;
171 FieldDecl:
172   name=Name ':' type=DataType
173 ;
174 enum ModeType:
175   ModeTypeIn='in' | ModeTypeOut='out' | ModeTypeInout='inout'
176 ;
177 Name: ID ;
178 ComplexName:
179   name+=Name ('::' name+=Name)*
180 ;
181 Valuing:
182   'value' name=Name (':' type=DataType)? '=' expression=Expression
183 ;
184 Value returns Expression:
185   BaseValue
186   | ConstructedValue
187 ;
188 BaseValue returns Expression:
189   IntegerValue
190   | {Any} 'any'
191 ;
192 // IntegerValue is a natural integer (>=0). Use a UnaryExpression
      to get a negative value.
193 IntegerValue:
194   absInt=INT // INT == ('0'..'9')+ rend une valeur ecore::EInt;
195 ;
196 ConstructedValue returns Expression:
197   {Tuple} 'tuple' '{' elements+=TupleElement (',' elements+=
      TupleElement)* '}'
198   | {Sequence} 'sequence' '{' (elements+=Expression (',' elements
      +=Expression)*)? '}'
199 ;
```

```
200 TupleElement:
201   label=Name '=' value=Expression
202 ;
203 Expression:
204   BinaryExpression0
205 ;
206 BinaryExpression0 returns Expression:
207   BinaryExpression1 ({BinaryExpression.left=current} op=BinaryOp0
      right=BinaryExpression0)?
208 ;
209 BinaryExpression1 returns Expression:
210   BinaryExpression2 ({BinaryExpression.left=current} op=BinaryOp1
      right=BinaryExpression2)*
211 ;
212 BinaryExpression2 returns Expression:
213   BinaryExpression3 ({BinaryExpression.left=current} op=BinaryOp2
      right=BinaryExpression3)*
214 ;
215 BinaryExpression3 returns Expression:
216   BinaryExpression4 ({BinaryExpression.left=current} op=BinaryOp3
      right=BinaryExpression4)*
217 ;
218 BinaryExpression4 returns Expression:
219   BinaryExpression5 ({BinaryExpression.left=current} op=BinaryOp4
      right=BinaryExpression5)*
220 ;
221 BinaryExpression5 returns Expression:
222   BinaryExpression6 ({BinaryExpression.left=current} op=BinaryOp5
      right=BinaryExpression6)*
223 ;
224 BinaryExpression6 returns Expression:
225   BinaryExpression7 ({BinaryExpression.left=current} op=BinaryOp6
      right=BinaryExpression7)*
226 ;
227 BinaryExpression7 returns Expression:
228   FinalExpression ({BinaryExpression.left=current} op=BinaryOp7
      right=FinalExpression)*
229 ;
230 FinalExpression returns Expression:
231   UnaryExpression
232   | CallExpression
233   | '(' Expression ')'
```

```
234   | Binding
235 ;
236 UnaryExpression:
237   op=UnaryOp right=FinalExpression
238 ;
239 CallExpression returns Expression:
240   (
241     {IdentExpression} ident=Name
242     | {CallExpression} function=Name '(' (parameters+=Expression (
      ',' parameters+=Expression)*)? ')'
243       | LitteralExpression
244   )
245   ('::'
246     (
247       {Field.object=current} field=Name
248       | {Select.object=current} 'select' '{' variable=Name '
      suchthat' condition=Expression '}'
249       // WAS: {Map.object=current} 'map' '{' variable=Name 'to'
      expression=Expression '}'
250       | {Map.object=current} 'collect' '{' variable=Name 'suchthat
      ' expression=Expression '}'
251       | {MethodCall.object=current} method=Name '(' (parameters+=
      Expression (',' parameters+=Expression)*)? ')'
252     )
253   )*
254 ;
255 LitteralExpression returns Expression:
256   Value
257 ;
258 BinaryOp0:  'implies' ;
259 BinaryOp1:  'or' ;
260 BinaryOp2:  'xor' ;
261 BinaryOp3:  'and' ;
262 BinaryOp4:  '=' | '<>' ;
263 BinaryOp5:  '<' | '<=' | '>' | '>=' ;
264 BinaryOp6:  '+' | '-' ;
265 BinaryOp7:  '*' | '/' | 'mod' | 'div' ;
266 UnaryOp:
267   BooleanUnaryOp
268   | ArithmeticUnaryOp
269 ;
270 BooleanUnaryOp: 'not' ;
```

```
271 ArithmeticUnaryOp:   '+' | '-';
272 HiddenBooleanType returns DataType:
273   {BooleanType}
274 ;
```